

MySIM: A Spontaneous Service Integration Middleware for Pervasive Environments

Noha Ibrahim, Frédéric Le Mouël, Stéphane Frénot

▶ **To cite this version:**

Noha Ibrahim, Frédéric Le Mouël, Stéphane Frénot. MySIM: A Spontaneous Service Integration Middleware for Pervasive Environments. Proceedings of the ACM International Conference on Pervasive Services (ICPS'2009), Jul 2009, London, United Kingdom. pp.1–10, 10.1145/1568199.1568201 . inria-00395427

HAL Id: inria-00395427

https://inria.hal.science/inria-00395427

Submitted on 16 Jul 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MySIM: A Spontaneous Service Integration Middleware for Pervasive Environments

Noha Ibrahim*
Grenoble Informatics Laboratory
Grenoble
France
noha.ibrahim@imag.fr

Frédéric Le Mouël
Université de Lyon, INRIA
INSA-Lyon, CITI
F-69621, France
frederic.le-mouel@insa-lyon.fr

Stéphane Frénot
Université de Lyon, INRIA
INSA-Lyon, CITI
F-69621, France
stephane.frenot@insa-lyon.fr

ABSTRACT

A computing infrastructure where "everything is a service" offers many new system and application possibilities. Among the main challenges, however, is the issue of service integration for the application development in such heterogeneous environments. Service integration has been considered by major middleware as a user centric approach as it responds to user requests and needs. In this article, we propose a novel way to integrate services considering only their availability, the functionalities they propose and their non functional QoS properties rather than the users direct requests. We define `MySIM`, a spontaneous service integration middleware. `MySIM` integrates services spontaneously on an event based mechanism and transparently for users and applications, extending by that the environment with functionalities. We developed a prototype as a proof of concept and evaluated its efficiency over a real use case.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability—*Distributed objects*

General Terms

Design, theory

Keywords

Middleware, pervasive, service integration

1. INTRODUCTION

Building pervasive computing applications can be a tedious task if performed from scratch. The developer will need to deal with low level networking protocols to high level application context awareness. This, of course, deviates the attention of the developers to tasks that are not the purpose of the application itself. Instead, they should only concentrate on the application logic, that is, the tasks the application must perform. This is why developing middleware for pervasive computing is essential for building and managing pervasive applications. Middleware are enabling technologies for the development, execution and interaction of applications, standing between the operating systems and applications. The service-oriented architecture (SOA) middleware emerged to describe the approach of building loosely coupled distributed systems with minimal shared understanding among system components. A computing infrastructure where "everything is a service" offers many new system and application possibilities. Among the main challenges, however, is the issue of application development in such heterogeneous environments. The natural way of doing this is by performing service integration [9], either by creating services and composing them according to requirements, or adapting and reusing existing services in order to achieve a given task. In such open environment the ability of services to adapt and be composed represents the primary driving force. These two actions of composition and adaptation are only possible if services are implemented and described in interoperable languages. For that reason, service transformation is a critical step preceding any composition or adaptation action. Many middleware dealt with one or more of these functional aspects and proposed middleware for composing, adapting, replacing, and/or transforming services. However, these middleware are user centric as they obey to user or application needs and requests. Users searching for specific functionalities will trigger the service integration in order to fulfill their needs. We state that these middleware are goal-oriented as they fulfill a goal in executing a user request.

In this article, we propose `MySIM` a spontaneous service integration middleware adapted for pervasive environments. The idea behind `MySIM` is:

- to define a service integration as the combination of three functional aspects: service transformation, service composition, and service adaptation, taking place at run-time and concerning the computational and behavioral parts of services.
- to propose a middleware that is user transparent and event based. It reacts to the appearance and disappearance of services by spontaneously integrating ser-

*This article describes the work performed when the author was employed in the CITI lab, Université de Lyon, INRIA, France

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICPS'09, July 13–17, 2009, London, United Kingdom.
Copyright 2009 ACM 978-1-60558-644-1/09/07 ...$10.00.

vices into their new inhabitance in a completely transparent way for users and applications.

- to define a metric based on functional service aspects (syntactic and semantic descriptions) and non functional QoS aspects (qualitative and quantitative) in order to provide a way to compare, evaluate and find the best service configuration.

Spontaneous service integration is an interesting feature in pervasive environments, as services meet when the users encounter, and interesting service compositions, substitutions and adaptations can be generated from these meetings, even though not required at that moment by users. We need to point out that `MySIM` is also capable of integrating services in a goal-oriented way, as the major middleware do. In this article, we only develop the spontaneous service integration aspect.

One can ask why providing these new services if not required? why not just wait until the user ask for this combination and then execute the integration? The answer lies in the definition of a pervasive environment and its potential to provide continuously more to users with less work from their sides. We argue that a user would be more than satisfied to find a multitude of services at his vicinity, and can always decide whether or not to use these services. An environment gains to transparently evolve from a state to another - proposing new services and new substitutions - without disturbing the user, but still taking full advantages of what is available around the user.

Use Case 1. *All along the article, we choose to illustrate our spontaneous service integration using the following use case (cf. figure 1).*

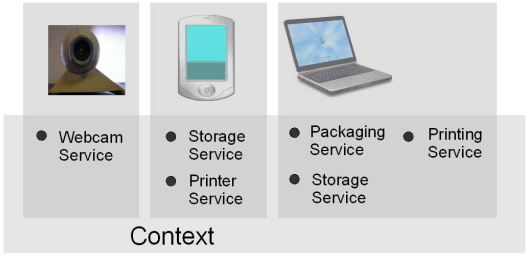


Figure 1: MyStudio use case

MyStudio use case is composed of five services - webcam, storage, printer, printing and packaging - that offer diverse functionalities with different non functional QoS properties to users and applications. The chosen use case is simple and allows to understand the key elements of our proposal. Considering these services, a spontaneous composition could be, if possible, combining two services together, for example, packaging and storage, even if not required by users. Users will have access to the two services storage via similar interfaces but the two services offer different implementations. One service allows to only store a picture in a defined location where the other one allows not only to store it but also to resize it to the user convenience. The spontaneous composition deploys a new service in the environment for the user intention. A spontaneous adaptation is also possible. If one of the printing service is unavailable, the middleware proposes the other printer service, or any other available printing service, spontaneously and transparently to the users, as the two services publish similar interfaces. For the user, these substitutions are transparent.

The rest of the article is as follows. Section 2 presents a state of the art of the three major families of service integration middleware - transformation, composition, and adaptation middleware - and highlights the lacks of these middleware in providing spontaneity. Section 3 explains and details the spontaneous service integration. Section 4 presents our `MySIM` middleware. Section 5 presents the prototype and its evaluation results. Finally, section 6 concludes and gives perspectives.

2. STATE OF THE ART

We survey in this section research efforts investigating service integration middleware for pervasive computing. We examined these middleware by paying attention to three aspects: the unified vision for the integration, the management of the non functional QoS properties of services, and the spontaneity. We divided the state of the art into three families of middleware.

Service transformation middleware tackle the problem of heterogeneity brought by pervasiveness. They propose to transform the different service technologies available in the vicinity into one common model to enable service interaction and communication. The area that tackles this problem is the Model-Driven Development [11]. It proposes a software development methodology in which software are developed not by writing code directly in implementation languages, but by constructing high level models that can be transformed into code by automated transformation engines and code generators. The slogan of MDD is "Model one, generate anywhere". The two service transformation middleware reposing on MDD principles are Perv-ML [12] and MIDAS [4].

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 of user tasks on the fly. 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. Major service composition middleware such as PERSE [10], SeGSeC [6], Broker [3] are all goal-oriented as they dynamically compose services in response to a user task. They provide their own service model and use semantic descriptions to compare and match services before composing them on the fly.

Service adaptation middleware adapt the functional and non functional behavior of a service to meet the application needs. This adaptation is done at run-time (dynamic), as pervasive applications need to cope with unpredictable changes. While an adaptive behavior implies the capability of a middleware to run in a number of different configurations, these middleware also need to dynamically perceive the characteristics of the surrounding environments and that by being context aware. Adaptation to changes might happen in middleware such in Carisma [2], and Madam [5] or in the applications such in Socam [7].

Figure 2 classifies these middleware under the three as-

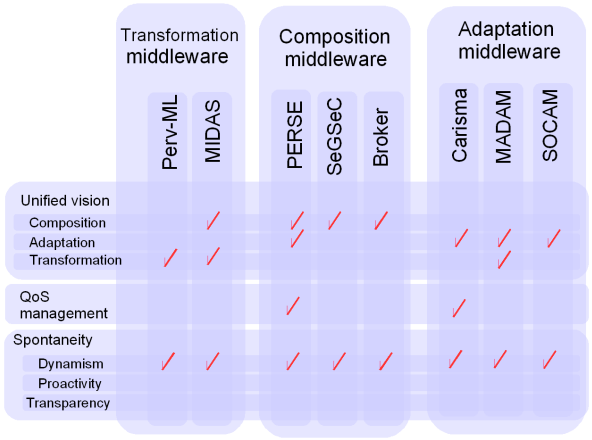


Figure 2: Classification of the major pervasive middleware

pects defined above: unified vision of the integration, the QoS service management, and the spontaneity issue. If some of the current approaches deal with one or two aspects of the integration, no unified vision for the integration as we define it is provided. If the dynamism aspect is tackled by the major middleware, no middleware proposes solution for spontaneous integration of services in a proactive and transparent way, without the user intervention. A need for a spontaneous service integration middleware with respect to the non functional QoS properties of services appears.

3. SPONTANEOUS SERVICE INTEGRATION

We begin by defining and formalizing our service model as a first step before integrating services. Then, we explain the notion of spontaneity for the service integration. We give a brief explanation of the integration techniques. Finally, we list the events that lead to applying the integration actions.

3.1 Service Model

We define a generic service model as composed of several parts: an interface, an implementation and QoS non functional properties. A functional interface specifies operations that can be performed on the service. An operation is described by a concept, a set of inputs and an output. The implementation is the implementation of the operations defined in the functional interface. The QoS non functional properties describe the operation capabilities. These capabilities reflect the quality of the functionality expected from the service, such as dependability (including availability, reliability, security and safety), accuracy of the operation, speed of the operation, and so on. The service is also semantically described. The semantic description is upon the operations and QoS properties and is based upon common ontology concepts.

Consider finite sets of grammatical alphabet Σ, ontologies $\mathtt{O}$, concepts $\mathtt{N}$ belongings to these ontologies $\mathtt{O}$, operations $\mathtt{Op}$, inputs $\mathtt{In}$, outputs $\mathtt{Out}$, concepts $\mathtt{Cpt}$, non functional properties $\mathtt{Np}$, quantitative and qualitative non-functional properties $\mathtt{Np}_{QN}$, $\mathtt{Np}_{QL}$.

Consider the following operators: $*$ (repetition zero or more times), $+$ (repetition one or more times), and 0..1 (repetition zero or one time). We define an operation op belonging to $Op \subset \mathtt{Op}$ as follows:

$(op \in Op \Leftrightarrow \exists\ In \subset \mathtt{In},\ \exists\ Out \subset \mathtt{Out},\ \exists\ cpt \in \mathtt{Cpt},\ \exists\ Np \subset \mathtt{Np},\ \exists\ Np_{QN} \subset Np_{QN},\ \exists\ Np_{QL} \subset Np_{QL})$:

$$op : < In^*,\ Out^{0..1},\ cpt,\ Np^* >$$

$$in : < name, type, semantic >,\ name \in \Sigma^*$$
$$out : < type, semantic >$$
$$cpt : < name, semantic >,\ name \in \Sigma^*$$
$$type : < language, name >,\ \{name,\ language\} \in \Sigma^*$$
$$semantic : < o, n >,\ o \in O \subset \mathtt{O},\ n \in N \subset \mathtt{N}$$
$$np : < Np^*_{QL}, Np^*_{QN} >$$
$$np_{QL} : < name, semantic >,\ name \in \Sigma^*$$
$$np_{QN} : < name, numericValue, operator >,\ name \in \Sigma^*$$
$$numericValue \in \mathbb{R}$$
$$operator : \{<,\ >,\ \leq,\ \geq\}$$

where:

- In is the set of the operation op inputs. in is defined as a tuple where $name$ is the chosen input syntactic name, $type$ is the syntactic input type, and $semantic$ the input semantic description.
- $out \in Out$ is the operation op output. out is defined as a tuple where $type$ is the output syntactic type, and $semantic$ its semantic description.
- cpt is the concept the operation op defines. The operation op concept cpt is defined as a tuple, where $name$ is the syntactic name through which the operation is called and $semantic$ its semantic description.
- Np is the set of non functional properties characterizing op. Np can be qualitative or quantitative. $np_{QN} \in Np_{QL}$ is the qualitative non functional properties defined as a tuple $< name, semantic >$. $np_{QN} \in Np_{QN}$ is the quantitative non functional properties defined as a tuple, where $numericValue \in \mathbb{R}$ and $operator \in \{>, <, \leq, \geq\}$. $operator$ specifies the order applied to $numericValue$. For $\{>, \geq\}$ the greater the $numericValue$ is, the best is the QoS property for the service runtime execution. For $\{<, \leq\}$ the smaller the $numericValue$ is, the best is the QoS property for the service runtime execution.

The $type$ depends strongly on the programming language the op is defined in, whereas the $semantic$ is independent of the technology and more related to the set of defined ontologies O.

USE CASE 2. *In our MyStudio use case, the five services - webcam, storage, printer, printing, and packaging - are described under the generic service model as depicted figure 3. For simplicity, the five services have five interfaces containing each, one operation. Each operation has a set of inputs described by a name, a type (Java language), and a semantic description, an output described by a type (Java) and a semantic description, and a concept described by a name and a semantic concept. Each operation can have one or several non functional properties, qualitative or quantitative.*

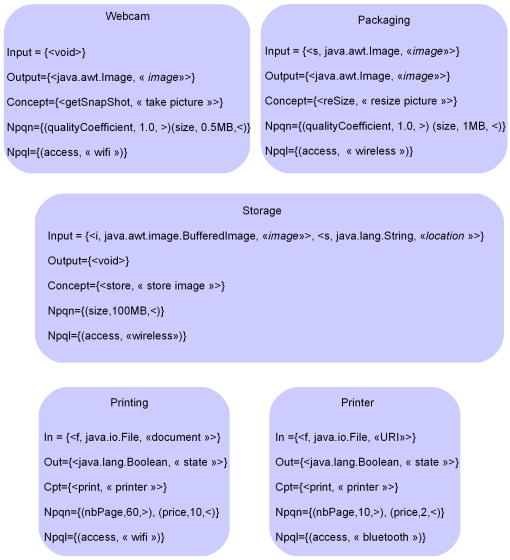


Figure 3: MyStudio syntactic and semantic service descriptions

3.2 A Generic Service Integration

A Transparent Service Integration

In nowadays middleware, a specific demand need to be formulated by applications for executing a service integration. Users specify their demands explicitly - searching for specific functionalities or specific non functional QoS properties - and the environment tries to respond to these demands by integrating the available services, if no atomic services can directly respond to these requests. On the contrary, the spontaneity of a service integration is in providing users and applications with new services (new functionalities or better non functional QoS properties) but in a completely transparent way and without previous user demand or external control over the integration. Indeed, a spontaneous integration is not initially required by users and applications, and the proactivity of the middleware need to be transparent enough to hide the integration results from users and at the same time provide them with the best capabilities their environment can offer. This spontaneous service integration is done technically in three ways (cf. figure 4): spontaneous service transformation, spontaneous service composition, and spontaneous service adaptation.

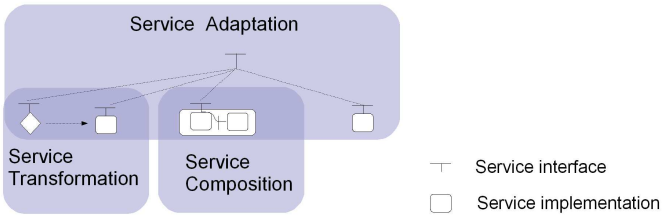


Figure 4: Service integration techniques

The spontaneous service transformation enables a given service expressed and provided in a predefined technology to be transformed into our generic service model and later on into another technology. This service transformation allows applications to re-use the services even if provided in different technologies. The transformation is essential in order to allow applications to use the available services in their vicinity but also to compose services together once they are expressed in the same model or formalism.

The spontaneous service composition enables services to be composed two by two, extending by that the environment with new functionalities. For this composition to be transparent, for users and applications, the new resulting "composed" services need to publish the same interface description (syntactic or semantic) as the already available services in the environment as users and applications access services via these interfaces. Providing new services with the same interfaces but different implementations and hence different functionalities, or different QoS properties remain transparent for users.

The spontaneous service adaptation adapts the service execution at runtime, by transparently insuring to users and applications a viable service for their executions. If a service is unavailable for any reason, the middleware searches for functionally equivalent services in the environment, to replace these services without disturbing user tasks nor application execution. If a new service is available and fits better the application needs, the middleware searches to substitute the already existing services with this new service. Users and applications have access to the service interface, and by accessing the same interface all the time, they are not aware of the implementation changes done behind.

We define our spontaneous service integration as a transparent integration of several services that returns new services with the same well known interfaces for the users and applications, but different functionalities, implementations, and non-functional QoS properties (cf. figure 5). The environments that are enriched with the spontaneous integration middleware, extend and shrink with service implementations, service non functional QoS properties, depending on what is available in the environment at a certain time. For the users and applications, the environment still publishes the same functional interfaces as before the spontaneous integration.

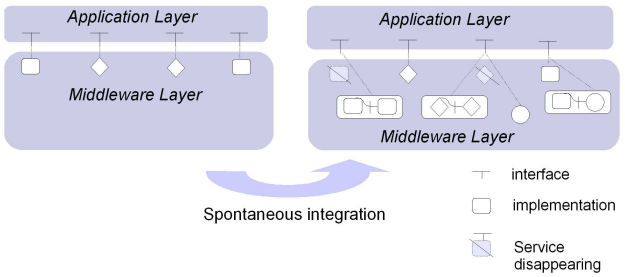


Figure 5: Available services, before and after spontaneous integration

An Event Based Service Integration

We distinguish two major events that can affect an environment in term of services and by that can lead to a spontaneous service integration:

- New services appearing in the environment leading to an automatic execution of the spontaneous integration

(transformation, composition, and adaptation) that extends the environment with their new functionalities and non functional QoS properties. When a new service appears in the environment, it brings with it a new functionality accessed by a new interface. The spontaneous integration is the ability of integrating this service with the already existing services, offering by that extended functionalities and/or different non functional properties, for the user of these environments. The service is first transformed into the service model understood by the middleware. Once this transformation done, the middleware will seek to compose this service with other services in a transparent way, which mean creating a composite service with a well know interface but new implementations, and new non functional QoS properties. The interface corresponds to the user visible part of the service, and by keeping a well-known interface, users can still access the service. The middleware will also seek to substitute already existing services with the new service if this one offers the same functionalities as these pre-existing services but better non-functional QoS properties for the applications.

- Services leaving the environment leading to spontaneous service adaptation for users and applications. Service adaptation replaces the vanishing services with others publishing the same interfaces but not necessarily the same implementations nor the same non-functional QoS properties. The middleware will seek, in the environment, for services that provide the same functionality as the disappearing service and for that publishing the same interfaces. In that case the new service can be transparently accessed by the applications as it publishes the same interfaces, even if the implementations and non functional QoS properties are slightly different.

The spontaneous service integration has its own life cycle, and stops automatically once there is no new services, new implementations, or new non functional QoS properties to add. For that, a stop condition is provided to prevent duplicating services. For more details on the spontaneous service integration life cycle please refer to C-ANIS [8].

4. MYSIM MIDDLEWARE

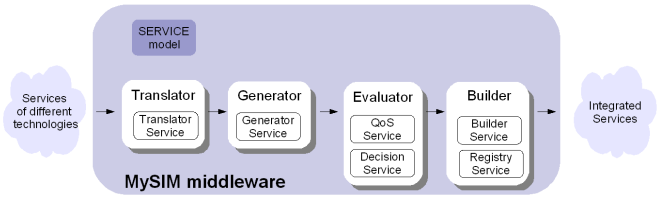


Figure 6: `MySIM` middleware

`MySIM` is composed of several services, gathered under four modules (cf. figure 6): the `Translator` module responsible for the service transformation, the `Generator` module responsible for the functional service relations (equivalence or composition), the `Evaluator` module responsible for evaluating the QoS non functional properties of the service relations, and finally the `Builder` module responsible for executing the real service integration, registering and monitoring the services. We will detail each module, explaining the techniques that are employed and illustrating them with the *MyStudio* use case.

4.1 The Translator Module

The `Translator Service` translates the diverse service technologies offering the functionalities available in the environment into our generic service model. The transformation rules extract and identify the three main parts of a service: the interface by extracting the operation signatures and semantic description, the implementations by extracting the operation implementations and the QoS non-functional properties by extracting the operation non functional properties. Depending on the technology the transformation can range from simple to complex. If the service provider has the same service model as the one we propose (such as OSGi, Web service, Fractal) the translation is relatively easy. If not, more complex mechanisms need to be introduced. Our `Translator Service` provides rules to map from OSGi services, and Web service descriptions to our service model. We only detail the OSGi mapping rules. The OSGi specifications [1] define a standardized, component oriented, computing environment for networked services. Adding an OSGi Service Platform to a networked device (embedded as well as servers), adds the capability to manage the life cycle of the software components in the device from anywhere in the network. OSGi proposes a service layer where an OSGi service is a Java interface, provided by a bundle (unit of deployment) and registered to a repository. An OSGi service can be implemented in different ways, and can have several implementations for a same interface. The OSGi service layer is composed of the following parts: service interfaces, service references and bundles. Service interfaces specify the service public methods. Service references encapsulate properties and meta data information about the service. Bundle is the unit of deployment and provides the service implementations. Mapping the OSGi specification to our generic service model is relatively easy and is done as follows:

- Service interfaces in OSGi correspond to the functional interface of our service model. If in OSGi a service can publish multiple interfaces, it will be mapped to our service model as different services, as a service correspond to one functional interface. The OSGi methods are the operations of the service, the OSGi parameters of the method are the inputs, and the result returned by the methods is the output. The name of the OSGi method through which the method is called is the concept part of the operation.

- Service references in OSGi are the semantic description of the operations and the non-functional QoS properties. These meta-data information about a service provide non-functional property descriptions of a service.

- OSGi bundles are the implementations of the service operations.

4.2 The Generator Module

The `Generator Service` is responsible of the syntactic (operation signatures) and semantic matching of the functional interfaces (set of operations) of services in order to

compose, substitute or adapt services. It returns all possible functional relations between services: the functional equivalence and composition relations. Two services are syntactic (or semantic) equivalent if they have syntactic (or semantic) equivalent interfaces - set of operations. Two operations are syntactic (or semantic) equivalent if respectively their inputs, output and concept are syntactic (or semantic) equivalent. The syntactic equivalence is a simple type equivalence for inputs and outputs and a simple name equivalence for the concept. The semantic equivalence relies on the semantic matching proposed by Paolucci [13]. It defines four level for the semantic matching between two concepts (exact, plug in, subsume and fail). We stipulate that a concept may substitue another one if an exact or plug in matching relations are defined between the two conceps. Two services are composable if syntactically (or semantically) the output of one operation of a service can be used as an input for an operation of the other service as explained figure 7. For the syntactic equivalence, the output type need to be of the same type or a sub-type of the input type. For the semantic equivalence, the output concept need to be exact or plug in matching with the input concept.

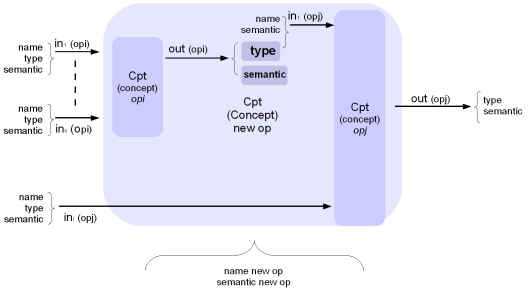


Figure 7: Service composition type or semantic compatibility

In the composition process, the `Generator Service` finds, for each service, all the services that respond to a composition relation, syntactic or semantic, between the functional parts of services. From all these combinations the `Generator Service` will only keep the services that are interface equivalent to an already existing service in the environment (cf. figure 8).

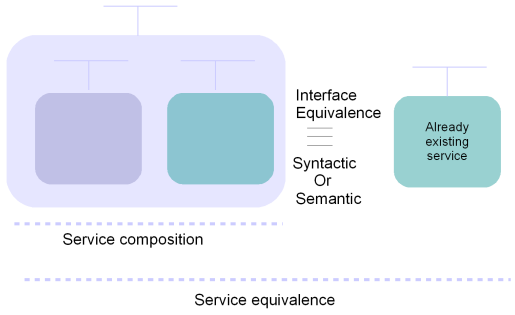


Figure 8: Transparent service composition for users and applications

This combination of the composition and equivalence relations guarantees a transparent composition for users and applications using the services of the environment. Indeed, the new composite services will still publish well known interfaces with new implementations corresponding to the service composition.

USE CASE 3. *In our MyStudio use case, the* `Generator Service` *distinguish one syntactic transparent composition - webcam and packaging - and one semantic transparent composition - storage and packaging (cf. figure 10). These service compositions produce new services, respectively a webcam and a storage service, publishing the same interfaces as the initial webcam and storage services but adding the packaging functionality to these services.*

In the adaptation process, for each new service in the environment, the `Generator Service` finds all the services available in the environment and that are syntactically or semantically equivalent. This notion of equivalence is based on the functional part of a service. By that, the middleware can propose the new service as a substitute to these services if the implementation it proposes or the non functional properties fit better the application needs. On the other hand, when a service is no longer available, the middleware searches, in the environment, for interface equivalent services to propose them as a substitute for the applications.

USE CASE 4. *In our MyStudio use case, the* `Generator Service` *distinguishes one syntactic transparent equivalence - printing and printer. The two services have equivalent operation signatures but different non functional QoS properties (cf. figure 3).*

4.3 The Evaluator Module

The `QoS Service` is responsible of evaluating the previous equivalence or composition relations by analyzing the service non-functional QoS properties. If the `Generator Service` indicates the possible functional relations between services, it does not pay attention to the service non-functional QoS properties. We distinguish the service composition issue and the service adaptation one. In a service composition, combining functionalities without paying attention to the service non-functional QoS can lead to non executable service composition. `QoS Service` needs to check for every service composition, that the non-functional QoS properties of inputs and outputs of service operations are compatible (cf. figure 9).

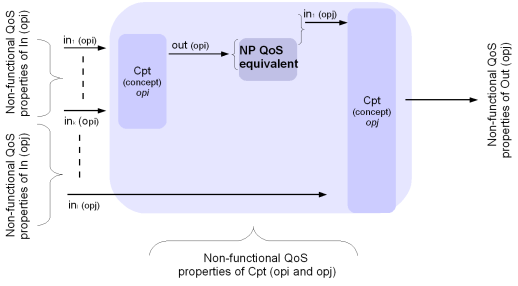


Figure 9: QoS properties compatibility

USE CASE 5. *Combining webcam and packaging, packaging and storing is possible as the non functional QoS properties of the output of the webcam operation (respectively packaging operation) respect the non functional QoS properties of the input of the packaging operation (respectively storing operation) (cf. figure 10).*

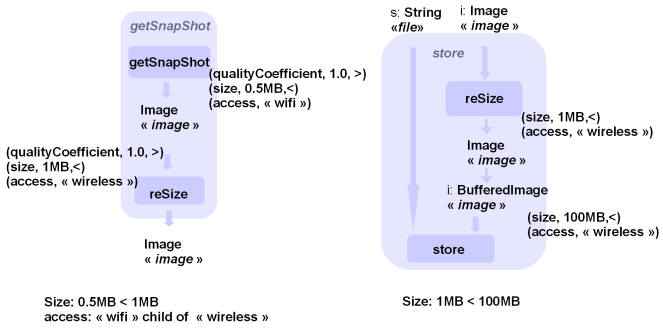


Figure 10: Composing transparently services: webcam and packaging, packaging and storing

In a service adaptation, a service is replaced by another one, providing equivalent functionalities. Two services can be functionally equivalent but provide different non-functional QoS properties. The `QoS Service` assures good substitution between equivalent services by choosing the best QoS properties of services for a given application. We define a metric that measures the non-functional QoS degree of equivalence. This metric allows to assign a normalized degree that measures the degree of non-functional QoS similarities between two equivalent syntactic or semantic services. The non-functional equivalence degree $QoS_{Degree}(opi, opj)$ between two functional equivalent operations is evaluated upon their quantitative and qualitative property similarities. Considering two operations *opi* and *opj*, we define the degree of equivalence between the two operations $QoS_{Degree}(opi, opj)$ as a function that measures how close is *opj* from *opi* in terms of non-functional QoS. We consider the non-functional properties of *opi*, NP_{opi} and calculate as follows the degree of equivalence *opj* has upon these properties:

$$QoS_{Degree}(opi, opj) = \sum_{k=1}^{|Np_{opi}|} w_k * \ deg(npk_{opi}, npk_{opj})$$

where, w_k is the assigned weight for a particular non-functional QoS property with the following conditions $\sum_{k=1}^{|Np_{opi}|}(w_k) = 1$. The more w_k is closer to zero, the more important is the property Npk. This ponderation allows to decide, when searching for equivalent services, if certain non-functional QoS properties are more important than others for the required service replacement. $deg(npk_{opi}, npk_{opj})$ are normalized values between 0 and 1 corresponding to the equivalence degree between npk_{opi} and npk_{opj}. These values are calculated using the z-score or standardization of the *npk* values for quantitative properties and semantic distance between concepts belonging to the same ontology for qualitative properties. The z-score of a quantitative property np_{QN}, indicates how far and in what direction, the property deviates from its distribution's mean, expressed in units of its distribution's standard deviation. We use the z-score standardization in order to provide a way of comparing all the different non-functional QoS by including consideration of their respective distributions. The semantic distance to compare the concepts of the qualitative properties returns a normalized value between 0 and 1. The more the concepts are close (parents/children depth association in the ontology) the more the value is close to zero. The QoS_{Degree} between two services will return a value that indicates how close these two operations are, related to their non functional properties. The less this value is, the closest the operations are in terms of non functional QoS similarities.

USE CASE 6. *A new service the Impression has appeared in the environment (cf. figure 11). Considering the Printing operation, it is syntactically equivalent to Printer and semantically equivalent to Impression. We calculate the non-functional QoS degree of equivalence to determine which of Printer or Impression replaces the better Printing.*

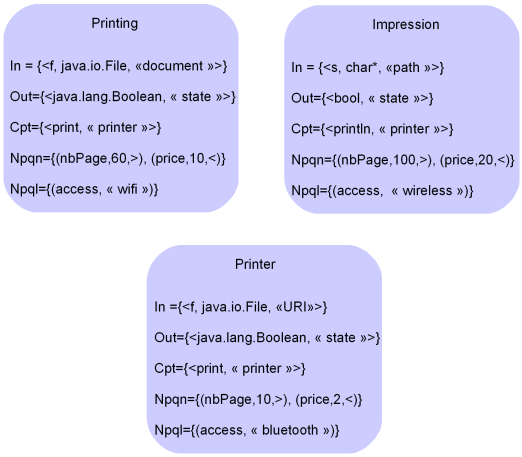


Figure 11: Three syntactic or semantic equivalent operations

The z-score is computed by normalizing the values obtained by the mean and deviation for each quantitative property (nbpage and price). The semantic distance for the qualitative property (access) is based upon the matching value between concepts (0 for the exact matching, 0.2 for the plugin matching, 0.8 for the subsume matching and 1 for the fail matching).
The QoS_{Degree} of the three operations are:

$QoS_{Degree}(Printing, Impression) = w1 * deg(nbpage_{printing}, nbpage_{impression}) + w2 * deg(price_{printing}, price_{impression} + w3 * deg(access_{printing}, access_{impression}$
$QoS_{Degree}(Printing, Impression) = w1 * 0.27 + w2 * 0.35 + w3 * 0.2$

$QoS_{Degree}(Printing, Printer) = w1 * deg(nbpage_{printing}, nbpage_{printer}) + w2 * deg(price_{printing}, price_{printer} + w3 * deg(access_{printing}, access_{printer}$
$QoS_{Degree}(Printing, Printer) = w1 * 0.33 + w2 * 0.33 + w3 * 1$

The QoS degree of the nbpage property shows that the Impression service is closer to Printing service (0.27 < 0.33). The nbpage operator > indicates that higher the value is, the better service it proposes. In that case, Impression service can print 100 pages per minute which satisfies the 60 pages proposed by Printing service. For the price property, the price operator < indicates that lower the value is, the better service it proposes. The Printer service is more suitable in replacing the Printing service and this is reflected in the QoS degree 0.33 < 0.35. For the qualitative property price, the "wifi" and "wireless" are plugin matching ("wireless" is a parent of "wifi" in the ontology), and the value of QoS degree is close to zero (0.2), where as for "wifi" and "bluetooth" that are fail matching, the value of QoS degree is 1. If we suppose the three non-functional QoS properties of the same importance $w1 + w2 + w3 = 1$, we obtain: $QoS_{Degree}(Printing, Impression) = 0.27$, and $QoS_{Degree}(Printing, Printer) = 0.55$. The Impression operation offers non-functional QoS that are closer to Printing if we assign the same weight to the three non-functional properties. Thus, if Printing service disappear the Impression service is more appropriate to replace it than the Printer service.

The `Decision Service` is based on an event-based mechanism of service appearance and disappearance. It uses the `Generator Service` and the `QoS Service` to construct abstract possible equivalence relations or composition compatibilities between services. For a spontaneous service composition, the `Decision Service` needs to ask the `Generator Service` and the `QoS Service` to provide all the possible transparent service composition, even if these compositions were not initially required by users. Nevertheless, interface equivalence relations, defined and verified by the `Decision Service` and executed by the `Generator Service` and the `QoS Service` need to be applied in order to allow a transparent use and access to these new implemented services. The `Decision Service` makes sure not to compose the same services over and over again. For that, a stop condition is needed. The `Decision Service` analyses the available services, by analyzing the interfaces they publish but also the implementations they propose. A new service is a service with new interface or an already existing interface but new implementations or/and non functional QoS properties. By that, it can verifies if the proposed composition plans are already available in the environment or really provide a new service to the environment and decides not to initiate a new composition that will duplicate services in the environment.

A spontaneous service adaptation may occur upon the appearance and/or disappearance of services. In case of a disappearance, the `Decision Service` asks the `Generator Service` and the `QoS Service` to provide all the available services that are equivalent (syntactically or semantically) to the disappeared service. The `Decision Service` chooses from all these available and equivalent services, atomic or composed ones, the most appropriate one depending on the QoS properties they provide. It uses the QoS degree function defined above to compute which service is the best to replace the disappearing service. The best service is the one minimizing the QoS degree function (cf. figure 12). In case of a service appearance, the `Decision Service` checks via the `Generator Service` if some other services, equivalent to the new one, are being used by applications.

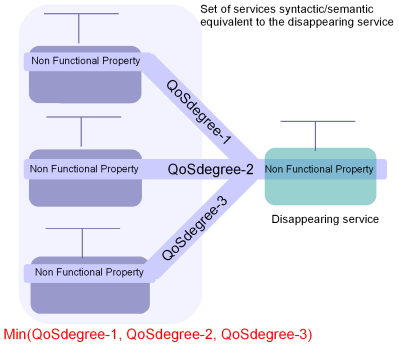


Figure 12: Choosing the best equivalent service according to non functional properties

If so, using the QoS degree proposed by the `QoS Service`, it computes whether the new service is QoS closer to the application needs than the services being used. In that case, a transparent substitution can be made.

4.4 The Builder Module

The `Builder Service` implements the combinations provided by the `Generator Service` and the `QoS Service` and approved by the `Decision Service`. The service integration is technically realized. The `Builder Service` creates new functionalities respecting the service model and directly implements these services in a chosen technology model. These newly implemented and generated services will continue to exist even after a user or application has finished using them. If many integration middleware propose to integrate services on the fly, as a runtime workflow where services are chained at execution time, few implement and generate at runtime new services as independent components. The `Builder Service` proposes several techniques to realize the real integration of services. Whether the integration is a transformation, composition, or an adaptation the techniques are different. For the service transformation, the `Builder Service` implements the desired service in an OSGi service and that by creating a new unit of deployment, a bundle that implements the published interface. For the service composition the `Builder Service` proposes the composition by service replication or the composition by service redirection. Redirection consists in creating OSGi services that redirect every call to the service to a set of chained services. Replication consists in creating OSGi services that replicate within their implementations the implementations of other services in order to be independent of them. For a service adaptation, the `Builder Service` proposes a facade pattern that adapt the service execution to the real service implementations available in the environment.

The `Registry Service` registers the interfaces of the newly transformed or/and composed services in the environment and monitors these services as they are very often dependent on the services they integrate. They are also dependent on the employed integration technique. The `Registry Service` checks periodically if these services execute correctly. When needed, it can suspend, stop, and start the services. Accessing these services can sometimes be impossible as one of the services involved in the integration can be unavailable. In that case, the new service does not execute properly and the `Registry Service` is quickly aware of this change.

The `Registry Service` notifies the `Decision Service` of the events related to service appearance (new service registering) or disappearance (services unregistering or being stopped).

USE CASE 7. *The newly composed services (new webcam service extended with packaging, new impression service replacing printing service, and new storage service extended with packaging) are deployed in the environment (cf. figure 13). They publish their interfaces for a possible use. This deployment is completely transparent for users, as the published interfaces are similar to the already available interfaces - webcam, storage, and printing interfaces. Only the middleware is aware of the implementation or non functional QoS property differences between the services and can propose to user the best implementations or non functional QoS properties he can ask for.*

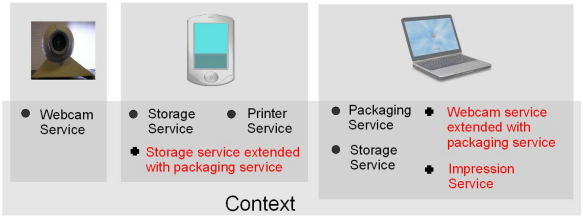


Figure 13: MyStudio after spontaneous service integration

5. MYSIM PROTOTYPE EVALUATIONS

We implemented all the major functionalities of our `MySIM` middleware under an OSGi service platform implementation, the Apache Felix. The service transformation rules are for now statically defined and can transform OSGi services and Web services (via their WSDL descriptions) to our generic service model. The service builder is OSGi based and implements new services in the OSGi technology. The composition is done syntactically using the Java language, the Java API introspection and semantically using online reasoner OWL-S ontologies and the matching relations of Paolucci [13]. The non functional QoS properties are for now defined in the service description and we do not yet consider the dynamic changes affecting these properties while service execution. For the evaluations we developed a use case composed of 10 OSGi services, inspired from the *MyStudio* use case. Then, we extended the platform test to 100 services in a small environment deployed on three laptops (Dell Latitude D410, 1,73 GHz, and 0,99 Go of memory).

The service composition and service generation time is approximatively 1 second for transparently composing two services. It is costly in terms of memory as it uses the **sun.tools.javac.Main** and **sun.tools.jar.Main** classes to build services from scratch. The `MySIM` service composition implements and generates new units of deployment for the new services in the environment. If this feature is interesting for the persistence of services in the environment, it is much less interesting in terms of memory consuming than the approaches that compose services on the fly without generating unit of deployment for these service composition.

Figure 14 gives time and memory values for syntactic service matching done by the `Generator Service` in order to find all the spontaneous composable services in the environment. The syntactic spontaneous composition - based on operation signatures - takes no time at all, and can be executed over relatively constraints devices.

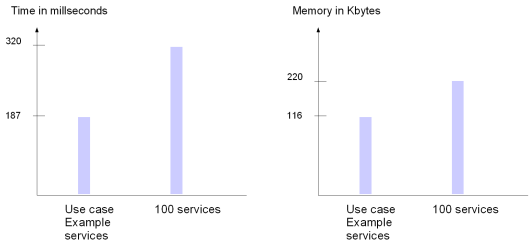


Figure 14: Time and memory consumption for syntactic service matching

The semantic matching - based on interface semantic description - is much longer than the syntactic one (cf. figure 15). The OWL-S API takes about 12 seconds to compare and matches 10 services owl-s descriptions (*MyStudio*) and 55 seconds for about 100 services. The pellet matching engine that reads all the owl-s files by adding them to the reasoner and extracts the inputs, outputs and concepts fields is much slower and much more memory consumer than as simple syntactic matching based on introspection methods provided by the Java language. We can improve the semantic matching time and memory consuming by employing techniques as in PERSE [10] that propose efficient semantic service matching using encoding classified ontologies.

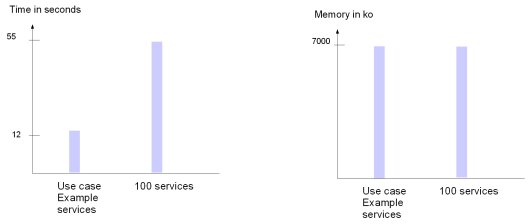


Figure 15: Time execution for semantic service matching

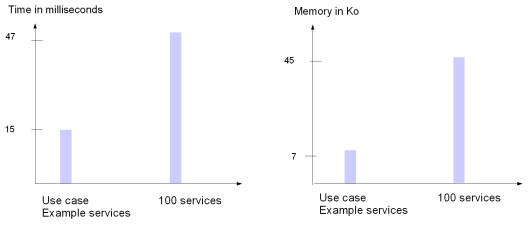


Figure 16: Time and memory consumption for QoS degree computing

Figure 16 gives the time execution and memory consumption for quantitative non-functional properties QoS_{degree} function computing. We suppose that each service has one

quantitative non-functional property. When a service leaves the environment, the time to adapt to this changes is the time required to compute and sort the QoS degree of available services publishing the same interfaces. When a service appears in the environment, the `QoS Service` computes the QoS degree of this services to find if it better suits the applications using equivalent services. If so, the `Registry Service` will propose to applications the new service and the adaptation would be done in no time for the application, as it is showed figure 16.

6. CONCLUSION

In pervasive environments, many middleware dealt with one or more of our service problems - transformation, composition, adaptation - but few proposed a unified vision for the service integration in pervasive environment, a management of the functional and non-functional properties of services during the integration, and especially, a spontaneous service integration based on events affecting the environment rather than explicit demands coming from the application layer. In this article, we proposed `MySIM` middleware a service integration middleware that spontaneously and in a transparent way transforms, composes and adapts services. The users and applications are unaware of these integrations and use all the possibilities provided by the middleware in a transparent way. We proposed a metric to compare services, based on syntactic and semantic interface matching, and a metric for computing the non functional QoS property similarities between services. We implemented a prototype under Java OSGi framework as a proof of concept and evaluated the efficiency of our proposal. One of the aspects that is not yet tackled by our `MySIM` middleware is the state of a service [14] that disappears while executing. If a service disappears while executing an application needs, to replace it in a transparent way, `MySIM` needs not only to find equivalent services in terms of functional and non-functional QoS properties but to know from which state to start the execution of the new service, so that the application does not loose what has been already executed by the previous service. Mechanisms of logging and checkpoints need to be introduced at the service execution time level to save the state of a service at runtime. These mechanisms allow our `MySIM` middleware to keep a trace over the state of services and to know when they disappear at which state of execution they were. Another important issue would be to test our prototype in large pervasive 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 of our spontaneous approach vis à vis to users. Our spontaneous service integration would surely have problem to scale to these service numbers and a more smart selection, based on semantic ontologies and user profiles, would be appropriate to choose a subset of services to compose, adapt, and transform.

7. REFERENCES

[1] OSGI Alliance. OSGi Service Platform, Core Specification Release 4. Draft, 07 2007.

[2] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, 2003.

[3] 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.

[4] Valeria de Castro, Esperanza Marcos, and Marcos López Sanz. A model driven method for service composition modelling: a case study. *Int. J. Web Engineering and Technology*, 2(4):335–353, 2006.

[5] Jacqueline Floch, editor. *Theory of adaptation*, Delivrable D2.2, Mobility and ADaptation enAbling Middleware (MADAM), 2006.

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

[7] Tao Gu, Hung Keng Pung, and Da Qing Zhang. A Middleware for Building Context-Aware Mobile Services. In *Proceedings of IEEE Vehicular Technology Conference*, Los Angeles, USA, September 2004.

[8] Noha Ibrahim, Frédéric Le Mouël, and Stéphane Frénot. C-ANIS: a Contextual, Automatic and Dynamic Service-Oriented Integration Framework. In *International Symposium on Ubiquitous Computing Systems (UCS)*, volume 4836/2008. LNCS, November 2007.

[9] Fuyuki Ishikawa, Nobukazu Yoshioka, and Shinichi Honiden. Mobile agent system for web service integration in pervasive network. *Syst. Comput. Japan*, 36(11):34–48, 2005.

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

[11] Jishnu Mukerji and Joaquin Miller. Technical guide to model driven architecture: The mda guide v1.0.1. Technical report, OMG's Architecture Board, June 2003.

[12] Javier Munoz, Vicente Pelechano, and J.Fons. Model driven development of pervasive systems. In *International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, June 2004.

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

[14] Davy Preuveneers and Yolande Berbers. Pervasive services on the move: Smart service diffusion on the osgi framework. In *UIC '08: Proceedings of the 5th international conference on Ubiquitous Intelligence and Computing*, Berlin, Heidelberg, 2008. Springer-Verlag.