



Automatic Service-Integration Framework for Ubiquitous Environments

Noha Ibrahim, Frédéric Le Mouël and Stéphane Frénot

ARES INRIA / CITI, INSA-Lyon, F-69621, France
{noha.ibrahim, frederic.le-mouel, stephane.frenot}@insa-lyon.fr

Abstract—Service-oriented architectures are among the premier middleware approaches to coping with the dynamicity of ubiquitous computing environments. In this article, we propose a new way of integrating services, namely ad-hoc service integration. For a certain class of services, ad-hoc integration is capable of automatically combining services at run time. This allows generating new functionalities from services newly appearing or already being available in the Ubicomp environment, but leaving the services' interfaces unchanged. This way, the extension in service functionality can be kept transparent to applications or users of a service. Nevertheless, our service-integration framework can more precisely distinguish among services. To show the feasibility of ad-hoc integration, we have implemented our service-integration framework based on OSGi/Felix along with a toolkit providing two different techniques to realize the ad-hoc integration: Redirection, i.e. calling interfaces and replication, i.e. copying implementations of services. A first evaluation verifies the viability of our work¹.

I. INTRODUCTION

Service-oriented architectures are among the premier approaches to providing a middleware layer for ubiquitous computing environments. As Ubicomp environments are characterized by a high degree of dynamicity, i.e. devices can join and leave an environment; the services provided by these devices can appear and disappear at any time.

This article contributes to the handling of this dynamicity by discussing automatic service integration. We propose what we call ad-hoc integration. For a certain class of services, ad-hoc integration is able to automatically combine services at run time. Thereby, the functionalities offered in a ubiquitous computing environment can be expanded not only as services appear but also based on the services already available in the environment. Upon intent, the newly created functionalities can be diminished again as services disappear.

As the basis for our ad-hoc service integration, we define a notion of compatibility among services and combine it with an additional condition. Together, this enables ad-hoc integration to automatically extend the functionality of an existing service *S* by integrating it with compatible services in the environment, but leaving the interface of *S* unchanged. This way, the extension in functionality of *S* can be kept transparent to applications or users employing this service. Nevertheless, our service-integration framework presented in this paper has means to distinguish more precisely.

In line with the service paradigm, we assume that every relevant context parameter of a ubiquitous computing environment is provided by some service. Consequently, we generally define context as the collection of services available in such an environment. Employing this definition of context, we can now characterize our automatic ad-hoc service-integration framework as dynamic and contextual. It integrates the available services at run time while taking the whole context into account, and, if intended, it can also dis-integrate services again.

In environments with a large number of services, the automaticity of ad-hoc integration might become an issue. In such environments, service integration can be handled in an on-demand manner, i.e. services are integrated on applications' or users' request.

In the following, an example use case will be depicted, which helps explaining our work throughout this article (section II). In section III, we will start by introducing our service model along with our notion of service equivalence. This is followed by the presentation of our notion of service-compatibility plus the additional condition, which altogether allows us to define ad-hoc integration along with its life cycle. At the end of section III, we discuss the implementation of our concepts, followed by a first evaluation (section IV). In section V, we review relevant related work to position our work. Finally we present conclusions and open issues (section VI).

II. USE CASE EXAMPLE

A use case is described all along the article to motivate and explain our automatic ad-hoc service-integration approach.

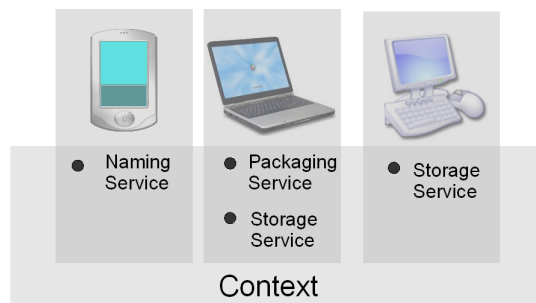


Fig. 1. use case

¹This work is part of the ongoing European project: IST Amigo-Ambient Intelligence for the Networked Home Environment [1].

The use case defines three services:

- the storage service: a service that enables to store an object on a device. Two services, executing on two different devices, offer the same functionality. One implementation is for local storage, the other one for remote storage.
- the naming service: a service that executes a naming strategy defined by a user to name his files and objects.
- the packaging service: a service that packs and unpacks objects.

These services are provided by different devices (cf. fig 1), that can join or leave the environment leading these services to appear and disappear at any time.

III. AUTOMATIC SERVICE-INTEGRATION FRAMEWORK

A. Service model

A service is composed of three parts:

- interfaces: A service can hold two kinds of interfaces: provided functional interfaces defining the functional behaviour of the service and required interfaces specifying required functionalities from other services. A functional interface specifies methods that can be performed on the service.
- implementations: Implementations realize the functionality expected from the service. These are the implementations of the methods defined in the functional interfaces.
- properties: a service will register its interfaces under certain properties. The property is used by the framework to choose services that offer the same interface, but different implementations.

We model a functional interface of a service S , its implementation and property as follow:

$$Ifc_S \begin{cases} m1(params1) \rightarrow r1 \\ \vdots \\ mk(paramsk) \rightarrow rk \end{cases}$$

$$Impl_S \begin{cases} Impl1(m1) \\ \vdots \\ Implk(mk) \end{cases}$$

$$property_S : (Ifc_S)_{atomic}$$

Where Ifc_S is one functional interface of the service S , mk the method name, $paramsk$ the list of parameters, rk the return result, and $impl_S(mk)$ the implementation of method mk . The property describes the interface implementation and specifies whether this implementation is atomic or integrated (resulting from integration). To execute a service, the framework can choose services interfaces considering the property they publish. If no property is specified the framework will randomly choose a service' interface implementation.

Two services are considered by users/applications to be the same if they have the same functional interfaces. They indeed provide, externally, the same functionalities.

Two services are considered by the run-time framework to be the same, if they have not only the same interface but especially the same property. Two services publishing the same interface but under different properties are considered by the framework to be different. The properties describe the implementation of the functional interface and different implementations mean different services.

Use case. the three use case' services (storage, naming and packaging) are modeled as follows:

$$storage \begin{cases} Ifc_{storage} : save(Object\ obj, String\ ID) \rightarrow void \\ Impl_{storage} : impl_{local}(save) \\ prop_{storage} : storagelocal_{atomic} \end{cases}$$

$$storage \begin{cases} Ifc_{storage} : save(Object\ obj, String\ ID) \rightarrow void \\ Impl_{storage} : impl_{ftp}(save) \\ prop_{storage} : storageftp_{atomic} \end{cases}$$

$$naming \begin{cases} Ifc_{naming} : getNextName(String\ ID) \rightarrow String \\ Impl_{naming} : impl(getNextName) \\ prop_{naming} : naming_{atomic} \end{cases}$$

$$packaging \begin{cases} Ifc_{packaging} \begin{cases} pack(Object\ obj) \rightarrow Object \\ unpack(Object\ obj) \rightarrow Object \end{cases} \\ Impl_{packaging} \begin{cases} impl(pack) \\ impl(unpack) \end{cases} \\ prop_{packaging} : packaging_{atomic} \end{cases}$$

For the run-time framework, there are four different services, as the two storage interface are registered under two different properties ($storagelocal_{atomic}$ and $storageftp_{atomic}$) corresponding to two different implementations (local and remote).

B. Definition of compatibility

Two services are compatible if they have two compatible functional interfaces. Two functional interfaces are defined to be compatible if they have at least two compatible methods. Two methods are compatible if the return result of one method is of the same type of one parameter of the other method (cf. fig 2).

Based on the compatibility definition, we define the integration of services as the combination, two by two, of all their compatible functional interfaces, and so of all their compatible methods. The combination of method1 and method2 (cf. fig 2) creates a new method1 with new parameters type corresponding to the parameters of method2 and part of method1' parameters.

The ad-hoc integration must remain transparent to the users and applications. The new method1 must have the same signature as the initial method1 and for that some conditions

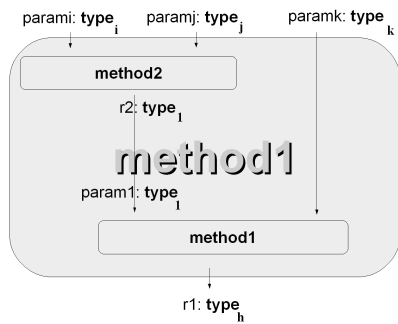


Fig. 2. Combining compatible methods: method1 & method2

must be fulfilled. Method2 must have only one parameter and of the same type as its return result (cf. fig 3).

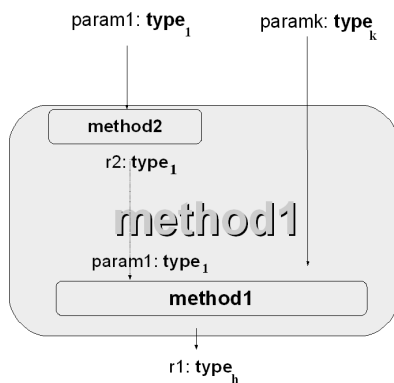


Fig. 3. Keeping the same signature as method1

The condition that needs to be satisfied in order to have an ad-hoc integration of services without generating new functional interfaces in the context is:

condition. One of the two methods to combine must have only one parameter and this parameter must have the same type as the return result of the method.

Two methods are *condition-compatible* if they are compatible and one of the method verifies *condition*. We define the ad-hoc integration of two services as the combination, two by two, of all their *condition-compatible* methods.

C. Automatic ad-hoc integration life-cycle

Automatic ad-hoc service integration is applied upon each appearance of new services in the context. The integration is contextual because it is very dependent on the services in the context, automatic because it is done by the framework upon each appearance of new services.

For the run-time framework a new service is a service with new functional interfaces or new properties.

New services appearing: If these services have new interfaces and so new methods, the framework applies the method matching algorithm. This algorithm returns a list of

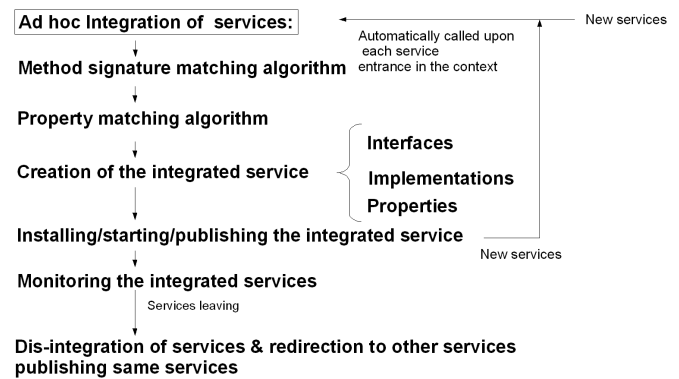


Fig. 4. automatic ad-hoc integration life cycle

all *condition-compatible* methods. The ad-hoc integration can take place and new services are created (same interfaces, new properties). If the services already exist, the framework do the matching on the property to determine if the services are new in the context, which means new atomic property or new integrated property. In case of new atomic property, the framework verifies if the methods of these services belong to the list of *condition-compatible* methods and if it is the case, automatically integrates these methods and creates new services (same interfaces, new properties). In case of new integrated property, the framework needs to insure that no integration must be done if it involves the same services already integrated. This condition insures the stop of our automatic integration. Indeed, the framework never re-integrates services that were previously integrated. All the new services are installed, published and monitored.

Services disappearing: The framework needs to dis-integrate the integrated services. The call to these services will be automatically redirected to other available services offering same interfaces but with different properties. This redirection is kept transparent to the users and applications.

D. Use case example

New services: storage, naming and packaging are now available in the context (fig. 1). The framework automatically executes the steps defined in the life-cycle (fig. 4).

These services have all new interfaces. The framework lists all the interfaces available in the context. Once the interfaces known, a list M of all their methods is created.

use case. fig 5

The framework selects all the methods in this list that has the same parameter and result type. This matching will return a list C of the methods that fulfil the *condition* defined section III-B.

use case. fig 6

The framework verifies the compatibility of all the methods of M to all the methods of C. The result is a list of all *condition-compatible* methods.

use case. fig 7, fig 8

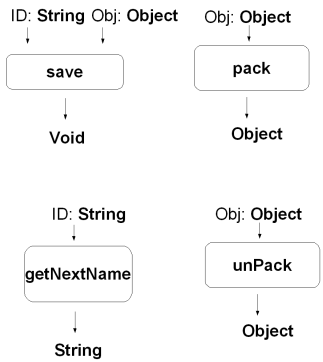


Fig. 5. M: list of all available methods in the context

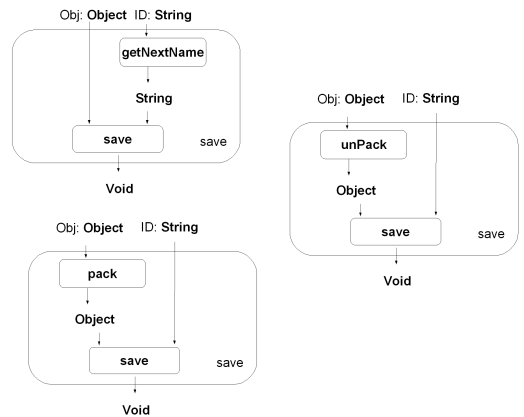


Fig. 9. Same methods signature, different implementations

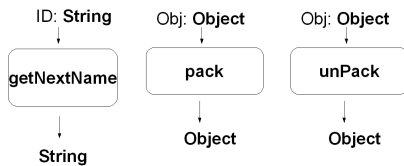


Fig. 6. C: list of methods that has the same parameter and result type

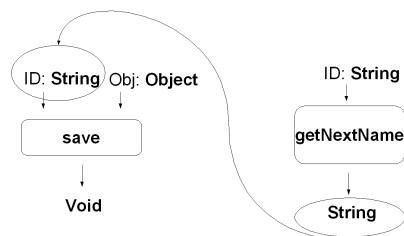


Fig. 7. compatible methods: save and getNextName

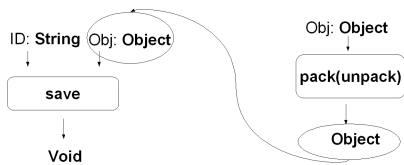


Fig. 8. compatible methods: save and pack(unpack)

The integrated services resulting from ad-hoc integration are services having the same interfaces but different implementations and properties.

use case. fig 9

The new services are now available in the context and registered under these new properties:

- storagelocal_{integrated}(naming_{atomic}),
- storagelocal_{integrated}(packaging_{atomic}),
- storageftp_{integrated}(naming_{atomic}),
- storageftp_{integrated}(packaging_{atomic}).

These new services are reconsidered for a possible re-integration by the framework. As the interfaces are not new, the properties are checked and only non previously integrated

services are allowed to integrate (cf. Table I).

The first integration will create service storage with new property storagelocal_{integrated}(naming_{atomic}, packaging_{atomic}) that will be added to the context. The property matching algorithm is updated. The integration of storagelocal_{integrated}(packaging_{atomic}) with naming_{atomic} that was possible by Table I is no longer possible. The run-time framework considers two interfaces registered under the same property to be the same.

Only two integration are done because of the property matching creating two services: storagelocal_{integrated}(naming_{atomic}, packaging_{atomic}) and storageftp_{integrated}(naming_{atomic}, packaging_{atomic}). The run-time framework reconsiders these new services for integration, but the property matching algorithm indicates that all the integration possibilities have been already done.

use case. fig 10

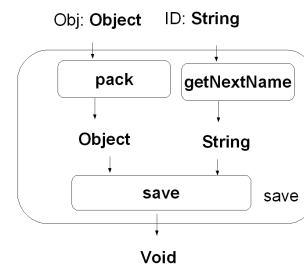


Fig. 10. The new method resulting from the ad-hoc integration (with two different implementations)

E. Ad-hoc service integration toolkit

We developed a toolkit for Felix/OSGi framework. The choice of Java was motivated by its portability and its capability to provide a strong separation between the APIs and their implementations. The OSGi specifications 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

	packaging _{atomic}	naming _{atomic}
storage _{local} _{integrated} (naming _{atomic})	yes	no
storage _{ftp} _{integrated} (naming _{atomic})	yes	no
storage _{local} _{integrated} (packaging _{atomic})	no	yes
storage _{ftp} _{integrated} (packaging _{atomic})	no	yes

TABLE I
PROPERTY MATCHING

capability to manage the life cycle of the software components in the device from anywhere in the network. A unit of deployment called bundle offers the services in the framework. An OSGi bundle is comprised of Java classes and other resources which together can provide functions, services and packages to other bundles. A bundle is distributed as a JAR file. We implement our developing framework on Felix which is open source implementation of OSGi framework specification.

The ad-hoc integration call (fig 4) is done by the framework via a simple method call: `integrate(context)`.

The framework executes this method call upon each appearance of a new service in the context.

In OSGi, creating the service is done by creating the unit of deployment, called bundle. To create a bundle we need to tackle several needs:

- unit of deployment: a bundle to deploy the new integrated service.
- integration glue (cf. Table II): The java code that do the technical integration. We provide two different techniques: the redirection or interface call, done via method call and RMI, and the replication or implementations copy done via method call to the local replicated implementations.
- needed libraries: in case of replication, the implementations of the replicated services are needed and added to the bundle.
- services dependencies: the new service will have to verify the dependencies of the services involved in the integration.

Once the service created, it is installed, started and its interfaces registered in the context (listing 1).

```
Properties props = new Properties();
props.put("StorageIfc", "Storage-integrated(Naming-atomic)");
context.registerService(
    StorageIfc.class.getName(), serv, props);
```

Listing 1. Example of a service registration

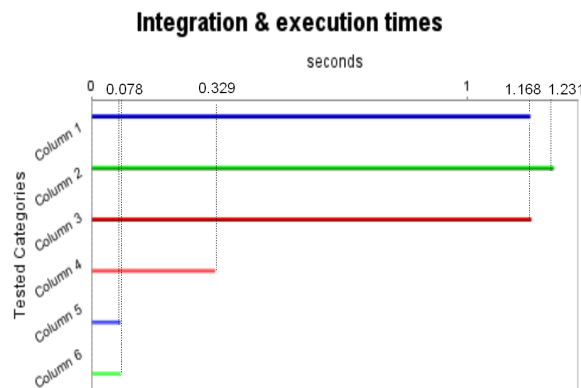
The run-time framework monitors all the integrated services. For each change in the context involving the integrated services, the framework stops the services and dis-integrates them. All the calls are redirected to services publishing the same interfaces but with different properties.

IV. EVALUATION

To test our toolkit we implemented the above described use case employing two Dell Latitude D410 laptops (Intel(R) Pentium(R) M, processor 1.73GHz, 0,99Go RAM) running Microsoft Windows XP Professional (version 2002) and Ubuntu

6.06 LTS.

We measured the time of our matching algorithm, service-integration techniques, and execution of the services.



- Column 1: Redirection: calling interfaces
- Column 2: Replication: copying implementations
- Column 3: Redirection and Replication
- Column 4: Execution time of matching algorithm on 100 services
- Column 5: Execution time of an integrated service
- Column 6: Execution time of a normal service

Fig. 11. Average of a 100 test runs

The time of our integration techniques is about 1 second for integrating two services. One can choose which technique to apply depending on the context. The redirection technique is more appropriate for constraints devices whereas the replication technique is more recommended for integrating services executing on devices that disconnect very often. The contextual choice of the technique will be the subject of another article.

The integrated service has the same execution time as any other atomic service (cf. fig 11).

For n services in the run-time framework, the complexity of our matching algorithm is $O(n)$ upon each entry of a service in the context and $O(n^2)$ if a matching is done between all the services of the context.

The matching algorithm is relatively quick, but the integration time is not scalable for large context. For run-time frameworks with 100 services, if matching only takes 329 ms, the integration time is much slower. Adding to that the time it takes to get distant access between remote run-time frameworks, one can quickly see the limits of the automaticity in large context.

	unit of deployment	integration glue	needed libraries	services dependencies
Redirection	Bundle (jar)	Method Call or RMI		S1, S2
Replication	Bundle (jar)	Method Call	S1 bundle, S2 bundle	dependencies S1, S2

TABLE II
INTEGRATION TECHNIQUES

V. RELATED WORK

Basically, the process of service integration responds to an external and explicit request (e.g. from users or applications) by providing new services in the environment. This integration is known in related work as “business logic of a client” [2], “on-demand basis composition” [3] or “users tasks descriptions” [4]. On the contrary and to the best of our knowledge, ad-hoc service-integration is considered more as an adaptation of the service itself rather than an integration of services. The idea of a framework that extends and shrinks automatically is not very exploited in the literature.

Integrating services by matching their interfaces has already been done in [5], [6]. The matching is especially done on semantic description of the parameters input and output. The matching descriptions is usually language-described at an abstract level. A service takes in charge to find the services corresponding to the semantic descriptions and to execute them at run-time.

[2] and [3] classify composition of web services framework for ubiquitous computing. They emphasize three important characteristics for service composition: dynamicity ([7], [8]), automaticity and context-awareness ([8], [9]) of the composition. While a certain number of works dealt with the dynamicity and context awareness of service-composition, few were interested in providing a real automaticity. We proposed to apply the characteristics defined in the above classifications to the interface matching and provide an automatic, dynamic and contextual service-integration framework for ubiquitous computing, that integrates services not only on a on-demand basis.

VI. CONCLUSION AND FUTURE WORKS

In this article, we proposed an automatic ad-hoc service-integration framework for ubiquitous computing. The proposed integration contributes in extending automatically the context with new functionalities. The more services offer their functionalities in such ubiquitous computing environments, the more it will be beneficial to combine them. The only constraints we need to satisfy is to have the same services’ interfaces so that the same users and applications can employ these services.

The contributions of our approach are in:

- extending ubiquitous computing environment with new functionalities.
- providing an automatic, dynamic and contextual integration for the service-oriented architectures.
- proposing a toolkit with two techniques of integration: Redirection i.e. calling interfaces and replication, i.e. copying implementations of services.

The perspectives of our approach are:

- scalability: The automaticity of our integration is heavy for large context and more appropriate for localized scalability. In large context, automaticity disappears and is replaced by on-demand integration.
- interoperability: The offered toolkit is only for java technology. We plan to use Amigo interoperable services [1] and extend our toolkit to .Net.
- generality of the matching algorithm: If a return type of method2 matches several parameters’ types of method1, only one match is taken into consideration. The property specifies that two services are already integrated and two methods can not be combined more than once. To resolve that issue, we want to describe semantically our services. The matching will be done on semantic description and not on methods’ signature to take all the cases into consideration.
- context-awareness: We want to define contextual strategies for the run-time framework for choosing the integration techniques depending on the context. We also want to add negotiation between services before integrating.

REFERENCES

- [1] N. Georgantas, Ed., *Detailed Design of the Amigo Middleware Core: Service Specification, Interoperable Middleware Core*, ser. Deliverable D3.1b, IST Amigo project, 2005.
- [2] S. Dustdar and W. Schreiner, “A survey on web services composition,” *Int. J. Web and Grid Services*, vol. 1, no. 1, pp. 1–30, 2005.
- [3] A. Alamri, M. Eid, and A. E. Saddik, “Classification of the state-of-the-art dynamic web services composition techniques,” *Int. J. Web and Grid Services*, vol. 2, no. 2, pp. 148–166, 2006.
- [4] S. B. Mokhtar, N. Georgantas, and V. Issarny, “Ad-hoc composition of user tasks in pervasive computing environments,” in *4th international workshop on Software Composition, co-located with ETAPS’05*, Apr. 2005, edinburgh, Scotland.
- [5] J. Zhou Zhang, S. Jian Yu, X. Kun Ge, and G. Wu, “Automatic Web Service Composition Based on Service Interface Description,” in *International Conference on Internet Computing*. CSREA Press, 2006, pp. 120–126.
- [6] S. Kalasapur, M. Kumar, and B. Shirazi, “Seamless service composition (SeSCo) in pervasive environments,” in *International Multimedia Conference, Proceedings of the first ACM international workshop on Multimedia service composition*. ACM Press, 2005, pp. 11–20.
- [7] H. Sun, X. Zhou, and P. Zou, “Research and Implementation of Dynamic Web Services Composition,” in *APPT 2003*, vol. LNCS 2834. Springer-Verlag Berlin Heidelberg, 2003, pp. 457–466.
- [8] S. B. Mokhtar, N. Georgantas, and V. Issarny, “COCO: COnteraction-based Service COmposition in Pervasive Computing Environments,” in *Proceedings of the IEEE International Conference on Pervasive Services (ICPS’06)*. IEEE Computer Society, 2006.
- [9] M. Mrissa and D. Benslimane, “Towards a semantic- and context-based approach for composing web services,” *Int. J. Web and Grid Services*, vol. 1, no. 3/4, pp. 268–286, 2005.