C-ANIS: a Contextual, Automatic and Dynamic Service-Oriented Integration Framework

Noha Ibrahim, Frédéric Le Mouël and Stéphane Frénot
{noha.ibrahim, frederic.le-mouel, stephane.frenot}@insa-lyon.fr

ARES INRIA / CITI, INSA-Lyon, F-69621, France

Abstract. Ubiquitous computing environments are highly dynamic by nature. Services provided by different devices can appear and disappear as, for example, devices join and leave these environments. This article contributes to the handling of this dynamicity by discussing service integration in the context of service-oriented architectures. We propose C-ANIS: a Contextual, Automatic and dyNamic Integration framework of Services. C-ANIS distinguishes two different approaches to service integration: automatic integration and on-demand integration. Automatic integration automatically extends the capabilities of an existing service S, leaving the interface of S unchanged. On-demand integration builds a new service on request from a list of given services. We have implemented C-ANIS based on the OSGi/Felix framework and thereby demonstrated the feasibility of these two service integration concepts. We have also implemented a toolkit providing two different techniques to realize the automatic and on-demand service integration concepts: Redirection, i.e. calling interfaces and replication, i.e. copying implementations of services[1].

1 Introduction

Ubiquitous computing environments are highly dynamic by nature. Services provided by different devices can appear and disappear as, for example, devices join and leave these environments. This article contributes to the handling of this dynamicity by discussing service integration in the context of service-oriented architectures. We propose to distinguish two different approaches to service integration: automatic integration and on-demand integration. Automatic integration automatically extends 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. On-demand integration builds a new service on request from a list of given services. It integrates an existing service S with a list of services in the environment, creating new interfaces. These new interfaces are employed at least by the users or applications which have requested their creation.

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

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 propose C-ANIS: a Contextual Automatic and dyNamic Integration framework of Services. C-ANIS integrates automatically and on-demand the available services at run time while taking the whole context into account, and if intended, it can also dis-integrate services again.

A use case is described all along the article to motivate, explain and evaluate our two integration approaches.

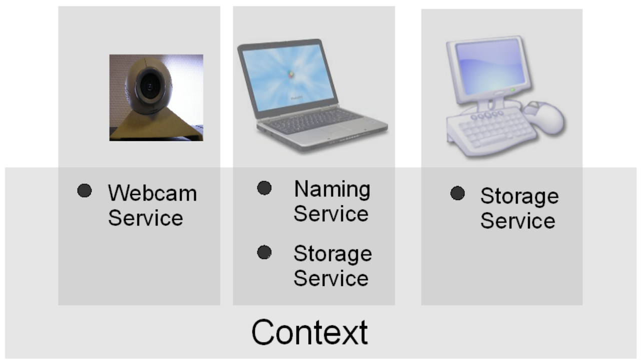


Fig. 1. use case

The use case defines three services:

- The webcam service: a service that enables to take a photo via a webcam.
- The storage service: a service that enables to store an object on a device. Two different services offer the same functionality. One implementation is for local storage, the other one for remote storage.
- The naming service: a service that execute a naming strategy defined by a user to name his files and 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.
We have implemented C-ANIS based on the OSGi/Felix framework and thereby demonstrated the feasibility of the two service integration concepts. We have also implemented a toolkit providing two different techniques to realize the automatic and on-demand service integration concepts: Redirection, i.e. calling interfaces and replication, i.e. copying implementations of services.

In the following, we will start by introducing our service model along with our notion of service integration (section 2). This is followed, section 3 and section 4, by the presentation of our two services integration approaches along with their life cycle. In section 5, we discuss the implementation of our concepts, followed by a first evaluation (section 6). In section 7, we will review relevant related work to position our work. Finally, we present conclusions and open issues (section 8).

2 Basic Definitions of our Service-Integration Approach

2.1 Service Model

A service is composed of three parts:

- interfaces: A service can hold two kinds of interfaces. Provided functional interfaces defining the functional behavior of the service. 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 interfaces, 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.

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

$$webcam \begin{cases} Ifc_{webcam} : getSnapShot() \rightarrow Image \\ \\ Impl_{webcam} : impl(getSnapShot) \\ \\ prop_{webcam} : webcam_{atomic} \end{cases}$$

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

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

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

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. The two storage services are considered to be the same by users. The implementations of these services is kept transparent from the users/applications (cf. fig 2).

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. For the run-time framework, the two storage interface are registered under two different properties (storageftp$_{atomic}$ and storagelocal$_{atomic}$) and considered as two different services (cf. fig 2).

Our service model is independent of any implementations and can be applied to EJBs [2], Fractal components [3], OSGi bundles/services [4] or Web Services [5].

2.2 Service Integration Approaches

In ubiquitous computing environments, services provided by different devices can appear and disappear as devices join and leave these environments. These services are employed by users or applications being in the environment. New services only come from new devices joining the environment. The only other

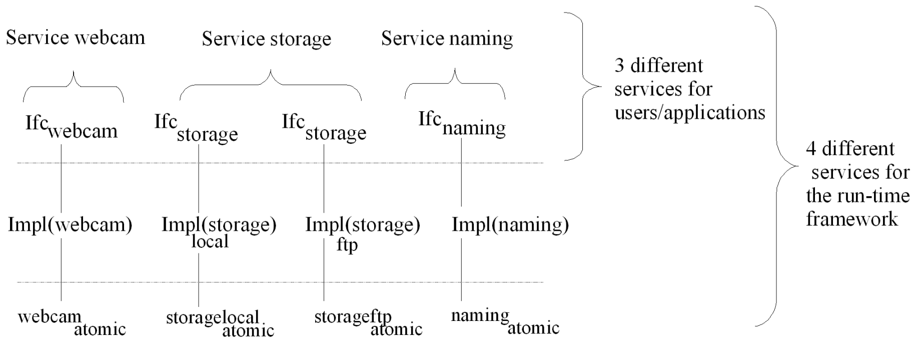


Fig. 2. Different services

way to offer new services in these environments is to respond to an external demand of integration. If new services are offered, without being requested, they are likely not to be used.

For that, we distinguish on-demand integration that builds new services upon users/applications' requests and automatic integration that extends the functionalities of existing services.

- On-demand service integration: The framework responds to an external demand by providing new services in the environment. This demand comes from users or applications being in the environment. Applications or users tend to use services available everywhere in the context and would like to, whenever it is possible and/or needed, integrate services offered by the context. In particular, if no single service can satisfy the functionality required by the application, combining existing services together should be a possibility in order to fulfill the request [6]. The result of this integration is a new service with new interfaces (new methods), new implementations (new functionalities) and new properties.
- Automatic service integration: The framework selects automatically all the compatible services in the environment and integrates them. The result of these integrations is the same services enriched with new functionalities. The service interfaces and methods do not change, only its functionality and properties change. This way, the extension in functionality can be kept transparent to applications or users employing the services. Once new services are in the environment, the framework automatically compares these services to existent services and if compatible services are found, the automatic integration can take place.

3 On-Demand Service Integration

On-demand integration builds a new service on request from a list of given services. We will first define our compatibility notion, followed by the life-cycle of our on-demand integration approach. Finally, we show its application on our use case example.

3.1 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 3).

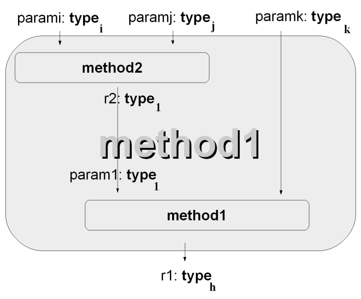


Fig. 3. Combining compatible methods: method1 & method2

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 3) creates a new method1 with new parameters type corresponding to the parameters of method2 and part of method1' parameters.

3.2 Life Cycle of On-Demand Service Integration

When integrating the services (cf. fig 4), all their methods are listed and only compatible methods are selected. The framework selects the most appropriate service' implementations to create the new service. This selection is context aware and must depend on the users/applications preferences. For now no strategies are defined and the selection is done statically. Once the implementations chosen, the new service is created, with its new interfaces, implementations and properties. The new service is installed, started, monitored and its interfaces published. If services involved in the integration leave the environment, the service newly created, is dis-integrated and a new service is created. For that a contextual selection of new service' implementations is done. In the meanwhile, all the calls to the service are buffered.

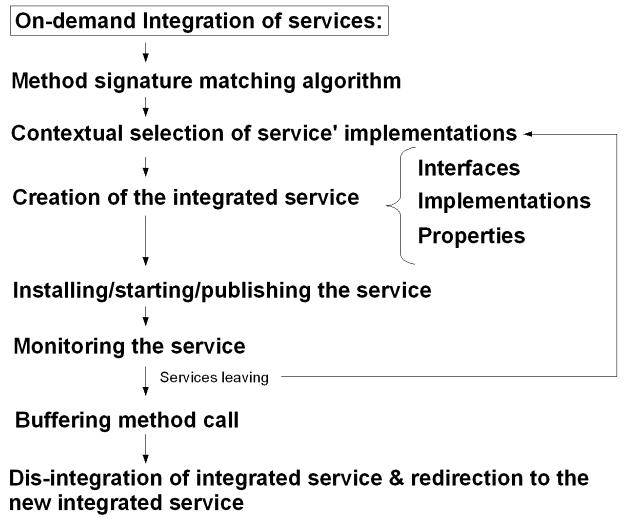

On-demand Integration of services:
Method signature matching algorithm
Contextual selection of service' implementations
Creation of the integrated service
Interfaces
Implementations
Properties
Installing/starting/publishing the service
Monitoring the service
Services leaving
Buffering method call
Dis-integration of integrated service & redirection to the new integrated service

Fig. 4. on-demand integration life cycle

3.3 Example Use Case

An on-demand integration example is the integration of service Webcam and storage (cf. fig 5). The two methods `save` and `getSnapShot` are compatible. Indeed, the return result of `getSnapShot` is of type `Image` which inherits `Object` the type of one parameter of `save`. The two methods can be combined as shown fig 5, and a new method `saveGetSnapShot` can be created. To integrate the two services storage and Webcam, the framework must choose the most appropriate services' implementations. A contextual choice is made upon users/applications preferences. If a user has a constraint device, he will probably prefer to store the image on a remote computer and for that the framework will choose the ftp storage implementation. If the user has a PDA and would like to store the photo on his device, the local storage is selected by the framework. For now, strategies are hard coded and chosen statically.

4 Automatic Service Integration

Automatic integration automatically extends the functionality of an existing service S by integrating it with compatible services in the environment, but leaving the interface of S unchanged. We will first define the modified compatibility definition for automatic integration, followed by its life-cycle. Finally, we show its application on our use case example.

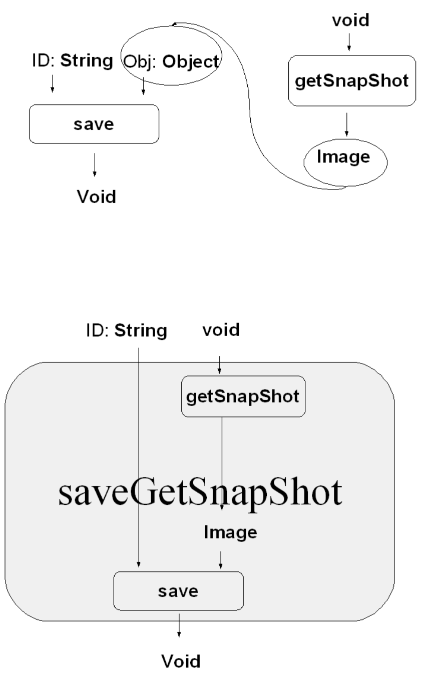


Fig. 5. on-demand service webcam and storage integration

4.1 Definition of Condition-Compatibility

The notion of compatibility is the same as defined for on-demand integration but with additional condition. The automatic integration must remain transparent to the users and applications. The new method1 must have the same signature as the initial method1 so that it can be employed by applications and for that some conditions must be fulfilled. Method2 must have only one parameter and of the same type as its return result (cf. fig 6).

The condition that needs to be satisfied in order to have an automatic 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 automatic integration of two services as the combination, two by two, of all their *condition-compatible* methods.

4.2 Life Cycle of Automatic Service Integration

Automatic service integration is applied upon each appearance of new services in the context. The integration is contextual because it is very dependent on

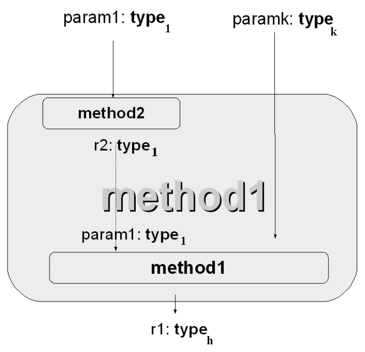


Fig. 6. Keeping the same signature as method1

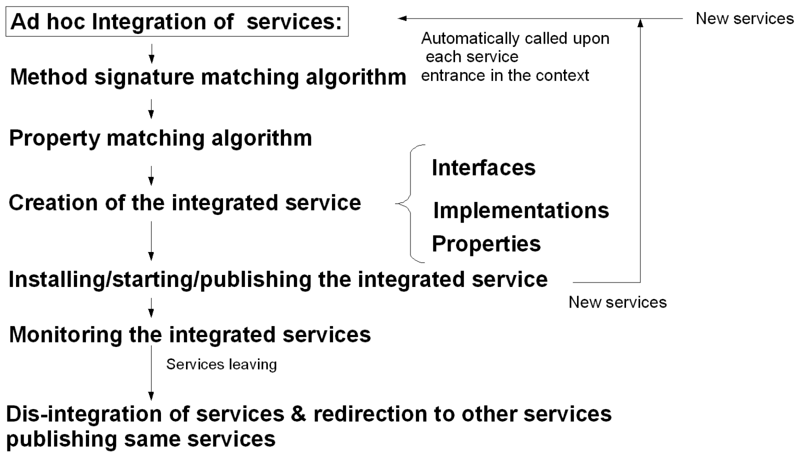

Ad hoc Integration of services:
Automatically called upon each service entrance in the context
New services
Method signature matching algorithm
Property matching algorithm
Creation of the integrated service
Interfaces
Implementations
Properties
Installing/starting/publishing the integrated service
New services
Monitoring the integrated services
Services leaving
Dis-integration of services & redirection to other services publishing same services

Fig. 7. automatic integration life cycle

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 all *condition-compatible* methods. The automatic 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.

4.3 Example Use Case

New services: storage, naming and webcam are now available in the context (fig. 1). The framework automatically executes the steps defined in the life-cycle (fig. 7).

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 (cf. fig 8).

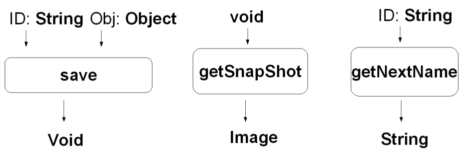


Fig. 8. M: list of all available methods in the context

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 3.1 (cf. fig 9).

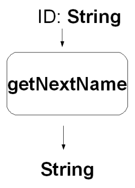


Fig. 9. C: list of methods that has the same parameter and result type

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 (cf. fig 10).

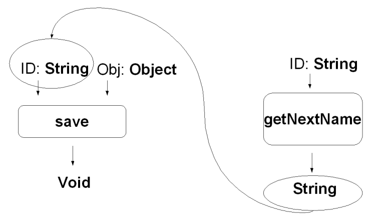


Fig. 10. compatible methods: save and getNextName

The integrated services resulting from automatic integration are services having the same interfaces (cf. fig 11) but different implementations and properties.

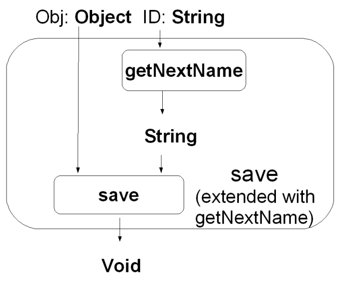


Fig. 11. Same methods signature, different implementations

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

storagelocal$_{integrated}$(naming$_{atomic}$), storageftp$_{integrated}$(naming$_{atomic}$).

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 1).

	webcam$_{atomic}$	naming$_{atomic}$
storagelocal$_{integrated}$(naming$_{atomic}$)	no	no
storageftp$_{integrated}$(naming$_{atomic}$)	no	no

Table 1. Property matching

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 (cf. Table 1). Indeed, the run-time framework considers two interfaces registered under the same property to be the same.

5 Contextual Service-Integration Toolkit

We developed a toolkit for the C-ANIS framework under Felix/OSGi. 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 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. We implement our developing framework on Felix which is open source implementation of OSGi framework specification.

The integration call is done by the framework.

- automatic integration call:

```
integrate(context);
```

Listing 1.1. Integrating services of the context

  The framework executes this integration call upon each entrance of a new service in the context. The new service is compared to all other services available in the environment.
- on-demand integration: Integrating the specified services is done via an integration call:

```
integrate(webcam, storage);
```

Listing 1.2. Integrating services storage and webcam

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

- unit of deployment: a bundle to deploy the new integrated service.
- integration glue (Table 2): 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.

	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 2. Integration techniques

Once the service created, it is installed, started and its interfaces registered in the context (listing 1.3).

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

Listing 1.3. 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. For automatic integration, all the calls are redirected to services publishing the same interfaces but with different properties. For on-demand integration, the calls are buffered, for a certain time, while the service is re-created with new services' implementations.

6 Evaluation

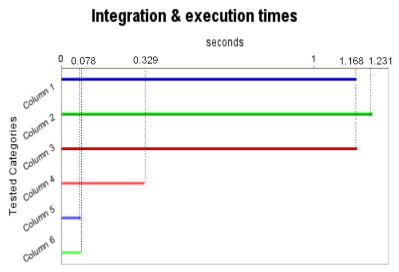
To test our prototype we implemented the above described use case employing a Logitech USB webcam (vfw:Microsoft WDM Image Capture (Win32):0), two Dell Latitude D410 laptops (Intel(R) Pentium(R) M, processor 1.73GHz, 0,99Go RAM) running Microsoft Windows XP Professional (version 2002) and Ubuntu 6.06 LTS.
We measured the time of our matching algorithm, service-integration techniques, execution of the services (cf. fig 12) and bundles' size (cf. fig 13).

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 12).

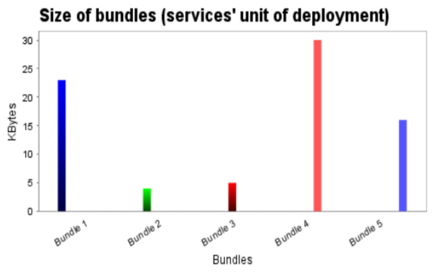
For n services in the run-time framework, the complexity of our automatic matching algorithm is $O(n)$ upon each entry of a service in the context and $O(n^2)$ if a matching is done between all the services of the context.

The matching algorithm is relatively quick, but the automatic 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.



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. 12. Average of a 100 test runs



Bundle1:	Webcam
Bundle2:	Storage
Bundle3:	Storage-Webcam (Redirection)
Bundle4:	Storage-Webcam (Replication)
Bundle5:	Storage-Webcam (Redirection & Replication)

Fig. 13. Bundle size for on-demand integration of webcam and storage services

7 Related Work

There has been a lot of work in developing different kinds of pervasive computing environments such as Gaia [7], Oxygen [8], Project Aura [9] and PCOM [10] to cite only these. All these environments try to make life easier for users by deploying various devices and supporting middlewares.

Project Oxygen [8] enables pervasive, human-centered computing through a combination of specific user and system technologies. Oxygen aims to enable pervasive, unobtrusive computing. The project proposes a user-centric support for ubiquitous applications, emphasizing specially the automatic and personalized access to information, adapting applications to users preferences and necessities as they moves through different spaces.

Aura's [9] goal is to provide each user with an invisible halo of computing and information services that persists regardless of location. Meeting this goal will require effort at every level: from the hardware and network layers, through the operating system and middleware, to the user interface and applications. Aura separates the user's intent from the application she uses to satisfy that intent and tries maintain the user's intent as the computing environment changes.

The Gaia system [7] provides a ubiquitous computing infrastructure for active spaces or smart rooms. Pervasive computing required the use of discovery, authentication, events and notification, repositories, location, and trading. Gaia depends on a distributed object, client/server architecture. Gaia adds a level of indirection between traditional applications and input/output resources to enable resource swapping.

These systems structure pervasive applications in terms of tasks and their subtasks, which is a software composition problem. The aim of these systems is to realize users demands. Our approach proposes not only the integration of services on an on-demand basis but also providing an automatic generation of services that can be or not used by the users. It deals more with the proactive property of pervasive environment. Another distinction is that our framework is distributed on the contrary of the previous systems which usually have a star-shaped architecture with central components.

PCOM [10] is a light-weight component system supporting strategy-based adaptation in spontaneous networked pervasive computing environments. Using PCOM, application programmers rely on a component abstraction where interdependencies are contractually specified. PCOM is all about adaptation of applications in pervasive environments and integration is seen as contract dependencies between components. PCOM is not service-oriented. An application is modeled as a tree of components and their dependencies where the root component identifies the application. Components are unit of composition with contractually specified interfaces and explicit context dependencies. Whereas our application and context abstraction are represented as services.

8 Conclusion and Future Work

In this article, we proposed C-ANIS: a Contextual Automatic and dyNamic Integration framework of Services. C-ANIS framework realizes automatically and at runtime the integration of available services in the environment, generating enriched services and new services as described in our two integration approaches. We have implemented C-ANIS based on the OSGi/Felix framework and thereby demonstrated the feasibility of these two service integration concepts.

The contributions of C-ANIS are in its:

- automaticity: Each time a new service is in the framework, a possible integration is done. We distinguished two major integration approaches: automatic integration and on-demand integration.
- context-awareness: Both automatic and on-demand integrations are context-aware. For on-demand integration, The choice of the services' implementations is depending on the context. For automatic integration, the services available in the context define the integration to do.
- dynamicity: Once the integration decided (method matching signature done), the choice of the implementations is done at run-time and can changes with the context changes.

The perspectives of our approach are:

- service model: Our service model do not take into consideration for now the non-functional properties and the state of the services. These two characteristics are important in pervasive environment especially when updating the integration due to context changes.

– generality of the matching algorithm: If a return type of method2 matches several parameters' types of method1, only one match is taken into consideration. Indeed, the properties specify 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 only on methods' signature to take all the cases into consideration.
– interoperability: The offered toolkit is only for java technology. We plan to use Amigo interoperable services [1] and extend our toolkit to .Net.
– context-awareness: The context is for now restricted to the set of services available in the environment. Commonly, context also reflects the social context of users and their preferences. We want to define contextual strategies for the run-time framework for choosing the integration techniques (replication or redirection) and services' implementations depending on the context. And for that a new notion of context needs to be defined.

References

1. Georgantas, N., ed.: Detailed Design of the Amigo Middleware Core: Service Specification, Interoperable Middleware Core. Delivrable D3.1b, IST Amigo project (2005)
2. Monson-Haefel, R.: Entreprise JavaBeans. O'Reilly & Associates (2000)
3. Bruneton, E.: Developing with Fractal. The ObjectWeb Consortium, France Telecom (R&D). (2004) version 1.0.3.
4. OSGIalliance: About the OSGI service platform. Technical report, OSGI alliance (2004) revision 3.0.
5. Iverson, W.: Real Web services. O'Reilly (2004)
6. Ponnekanti, S.R., Fox, A.: SWORD: A Developer Toolkit for Web Service Composition. In: 11th World Wide Web Conference. (2002) Honolulu, USA.
7. Roman, M., Campbell, R.H.: A Middleware-Based Application Framework for Active Space Applications. In: ACM/IFIP/USENIX International Middleware Conference (Middleware 2003). (2003)
8. MIT. Project Oxygen: Pervasive, Human-Centered Computing. Website: http://oxygen.lcs.mit.edu/ (2007)
9. Garlan, D., Siewiorek, D., Smailagic, A., , Steenkiste, P.: Project aura: Towards distraction-free pervasive computing. IEEE Pervasive Computing, special issue on "Integrated Pervasive Computing Environments" **21**(2) (2002) 22–31
10. Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM - A Component System for Pervasive Computing. In: the 2nd IEEE Annual Conference on Pervasive Computing and Communications (PERCOM'04), Washington, DC, USA, IEEE Computer Society (2004)