ANIS: A Negotiated Integration of Services in Distributed Environments

Noha Ibrahim, Frédéric Le Mouël

▶ To cite this version:

HAL Id: inria-00394968
https://hal.inria.fr/inria-00394968

ANIS: A Negotiated Integration of Services in Distributed Environments

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

INRIA ARES, CITI Lab., INSA Lyon
21 Avenue Jean Capelle
F-69621 Villeurbanne Cedex, France

Abstract. The development of highly dynamic distributed environments modifies the runtime behavior of applications. Applications tend to use services available everywhere in the environment and would like to, whenever it is possible and/or needed, integrate services offered by the local environment. 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.

In this article, we propose *ANIS*: A Negotiated Integration System. Our system provides a framework including a set of integration management interfaces - `Integrable`, `Negotiable`, `IntegrationLifeCycle` - and the tools implementing these interfaces. These tools offer different techniques of integration (local/remote composition, local/remote weaving, deployment by downloading/uploading), negotiation by contracts and the capability to manage the life cycle of the integration.

A prototype based on Java platform and OSGi technology is implemented as a proof-of-concept to demonstrate the potential of ANIS[1].

Keywords: services, integration, negotiation, life cycle, OSGi, distributed systems.

1 Introduction

For more than one decade, integration of services: the problem of combining different separate services into a unified one to achieve new functionality, is generating considerable interest in several computer science communities. The middleware is a particularly interesting domain for service integration for several reasons. Firstly, increasing numbers of service providers are moving into the middleware, providing a collection of useful services for applications [1]. Another reason is the development of highly dynamic distributed environments that modifies the execution of applications at runtime. Applications tend to use services available everywhere in the environment and would like to, whenever it

---

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

is possible and/or needed, integrate services offered by the local environment. 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. Third, services integration has the potential to reduce the effort of applications that continuously look for services, by combining complementary middleware services provided by independent providers to achieve the end-application's needs [2].

The service integration is particularly challenging because of the diversity of the services available in the middleware. Integrating these services may require different technologies or techniques. Our objective is to propose a system with A customizable Negotiated Integration of Services, *ANIS*. *ANIS* provides a framework including a set of integration management interfaces - `Integrable`, `Negotiable`, `IntegrationLifeCycle` - and the tools implementing these interfaces. These tools offer different techniques of integration (local/remote composition, local/remote weaving, deployment by downloading/uploading, etc..), negotiation by contracts and the capability to manage the life cycle of the integration.

The idea behind *ANIS* is:

  – To propose a generic model of service that is independent of any implementation and technology. Our system can be applied to all types of services, from Entreprise Java Beans [3] to Web Services [4].
  – To propose a developing framework including APIs for managing the integration of services. The framework includes three essential interfaces: `Integrable` for integrating services, optional `Negotiable` for negotiating the integration and `IntegrationLifeCycle` for managing the life cycle of the resulting integration.
  – To propose tools that implement the defined APIs. These tools offer different techniques of integration (deployment of services by downloading a service from a platform or uploading a service into an environment, composition of services and weaving of services), a negotiation protocol based on contracts between the services willing to integrate and a manager of the life cycle of the integration.

Nowadays, number of existing framework do the integration of services but impose, very often, a precise toolkit to do so. In *ANIS*, the framework is independent of the tools implementing it and a service can use our framework with his own toolkit implementation. The framework supplies the general road map for an efficient integration. Applications, depending on their needs, can use the provided toolkit or another one.
We demonstrated the feasibility of *ANIS* by implementing a prototype. We choose to implement it using Java platform and OSGi technology. The tools we propose are specific to the choices we made. The choice of Java was motivated by its portability and its capability to provide a strong separation between the APIs and their implementations. This separation reflects the one we made in *ANIS* between the framework and the toolkit. OSGi [5] was chosen for its

facility to provide the Inversion Of Control (IOC) [6]. OSGi simplifies the development, deployment and management of services by decoupling the service's specification from its implementation.

The strength of *ANIS* is in its framework that can be adapted to all different existing platform (.Net, OSGi, Fractal, J2EE, etc.). The limit of the current version of *ANIS* is in its toolkit; the applications that need to use it has to support the Java Runtime Environment and the OSGi platform.

The paper is organized as follows: section 2 explains the requirements we impose to *ANIS*. Section 3 details the architecture of the *ANIS* system. Section 4 shows implementation of our framework prototype. Section 5 compares our work to existing related integration works. Finally, section 6 concludes and gives future research directions.

2   Requirements for the *ANIS* System

A service integration framework must adhere to certain requirements in order to provide an efficient and viable integration. These requirements depend a lot on the application field we wish to apply *ANIS* to. For the current version of *ANIS*, we choose to limit ourselves to three properties of the integration [7]. We consider these three properties to be essential for integrating services in general distributed environments. Many other properties are interesting to study, such as interoperability or adaptability, but we consider them to be more specific to certain environments such as mobile or pervasive environments. According to future application domain, these properties will be studied later.
The integration has to verify these three core properties:

– Genericity: In distributed environments, all kinds of applications would like to integrate services available around. To make it possible, the framework must be generic. It has to be applied to all types of services, and also to be used by all types of applications.
– Life-time cycle: The integration can be set for a certain time. It can have a life cycle. An integrated service, can therefore last a certain span and disappear once completed its cycle. This property enables applications to integrate services for a purpose, and be sure of the disappearance of the integrated service once the purpose reached.
– Reversibility: As an integrated service has to disappear once its cycle end, integration must be reversible. We mean by reversible the faculty to return to the state previous to the integration process. The integrated service disappears and the services, that were involved in the integration, are still in an operational state.

The integration verifies another property not related to the distributed nature of applications, efficiency. The integration gains in efficiency if previously to integration, services negotiate on terms and conditions of this one. This property is optional and services can integrate without negotiating. Our toolkit propose negotiation by contracts between services but other toolkits can implement the framework without specifying any negotiation protocol.

3   *ANIS* Framework Architecture

This section introduces the architecture of the *ANIS* framework. We consider that all services are gathered in a middleware layer providing a run-time environment. The application layer uses the services of the middleware and the services of the middleware rest on the system layer which provides the management of the hardware and the network. We define a special service *IntegServ*. This service can reside on every host wishing to integrate services. The integration of services, whether it is local or remote, will be managed by an *IntegServ* service. As we will consider integration of services, we will first define our model of service. Then, we detail the *IntegServ* architecture and its sub-parts.

3.1   Service Model

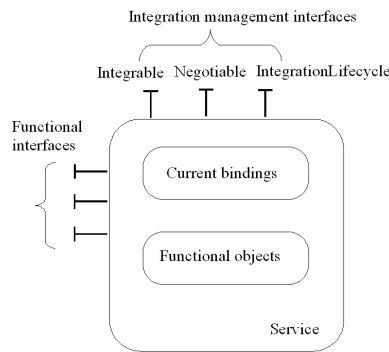As shown in the figure 1, our service is composed of three parts:

Fig. 1. Service model

– Interfaces: an interface specifies methods that can be performed on the service. Service's interfaces are public and so published for an external use. A service can hold two kinds of interfaces: functional interfaces defining the functional behavior of the service (e.g. for a video streaming service, a functional interface can allow to specify frame's size, frame's rate, etc.) and integration management interfaces defining the way to manage the integration of this service (e.g. a Integrable interface can allow to specify that the service can be integrated).
– Bindings: a service can provide and/or require functionalities from other services. Bindings express these run-time dependencies (e.g. if a video streaming service requires a QoS communication interface, at one moment, it can bind to a H.323 service, and at some later moment, to a SIP/RTP service).

– Objects: objects realize the functionality expected from the service (e.g. in our video streaming service, objects multiplex/demultiplex, order/reorder video and audio frames, etc.).

Our service model is independent of any implementations and can be applied to EJBs [3], CORBA Components [8], Fractal components [9], OSGi bundles/services [10] or Web Services [4].

Technically, interfaces can be expressed in language-native way, such as Java interfaces or by using an Interface Description Language (IDL), such as in CORBA [8]. Bindings can be expressed by an Architecture Description Language (ADL) [11] . Objects implementations are language-dependent and result from the instantiation of classes.

3.2 *IntegServ* Architecture

Several services run on different hosts in a distributed environment (cf. figure 2). We consider that these services can be final or integrated. A service needing to be integrated has to implement the `Integrable` interface. As specified in the requirements above, it can optionally negotiate before integrating, and can so implement the `Negotiable` interface. After the integration, the integrated service will hold the `IntegrationLifeCycle` interface.

Two middleware services are involved in the integration process. Once an application use the `Integrable` interface of a service the interface can execute an integration defined by the service provider, redirect the call to a local *IntegServ* (step 1 figure 2) or to a remote one if no local one is found. A *Discovery service* takes in charge the task of searching and reporting for the available *IntegServ* (step 2 figure 2). *ANIS* do not provide the *Discovery service* and relies on existing systems such as Jini Discovery Service [12] or UPnP [13].

ANIS applies an integration reflexivity model on its own services. The *IntegServ* service results itself from the integration of several special services: the *Technical Integration Service*, the *Negotiation Manager Service* and the *Life Cycle Manager*.

3.3 Technical Integration Service

The *Technical Integration Service* applies different techniques of integration involving internal parts of services (i.e. bindings and objects) but also external parts (i.e. interfaces). These different techniques can be applied one before the other and/or combined. They can be applied to a set of services. The *Technical Integration Service* provides three techniques of integration:

– Composition of services is the technique of integrating services by connecting their interfaces. The *Technical Integration Service* defines two methods for
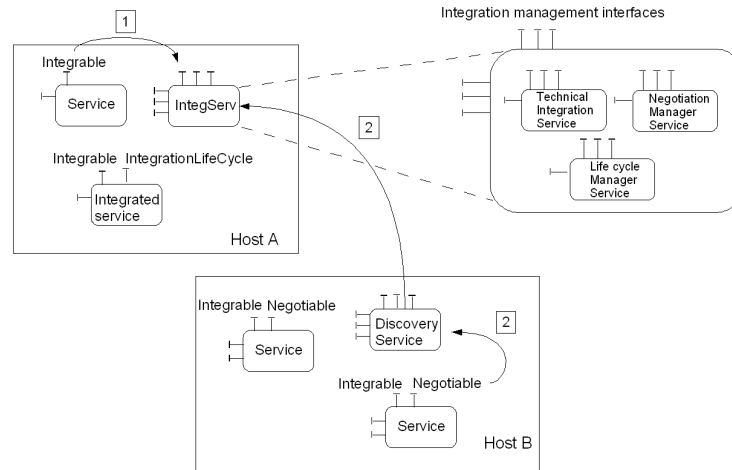
Fig. 2. Architecture of the *IntegServ* service

choosing how to connect the interfaces. The first one is satisfied by republishing the same interfaces. The composed service provides all the interfaces of the services composing it (cf. figure 3). The second one chooses to provide only one interface which represents all the others (cf. figure 4). This method is used if the output of an interface matches the input of another, making a chain of matching interfaces.

– Weaving of services is the technique of interlacing the object code of a service into object code of another one. Only object parts of services are concerned (cf. figure 5). The weaved service publishes the same functional interface it possessed before the weaving process, but the functionality that the service offers has changed after the weaving.

– Deployment of services is the technique of distributing services in a new environment by downloading or uploading them. Only binding parts of services are concerned. Once a service deployed, new bindings are created and old ones deleted (cf. figure 6).

Nothing prevents from adding new techniques to the *Technical Integration Service*. The only requirement, is that these new techniques have to respect the *ANIS* service model.

3.4 Negotiation Manager Service

Negotiation is the process of getting an agreement between two services about terms and conditions of the integration. This step occurs during the integration process, just before applying the techniques of integration. Three main parts compose the negotiation process: contracts, strategies and negotiation phases. Contract is conceptually composed of several distinct parts [14]:
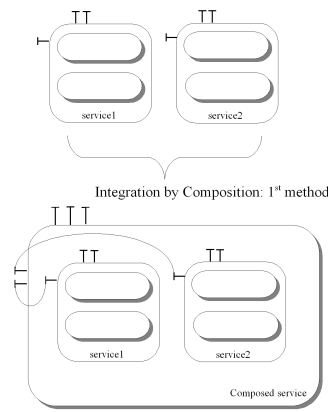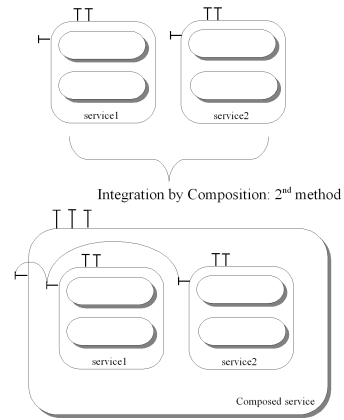
Fig. 3. Composition of services: first method

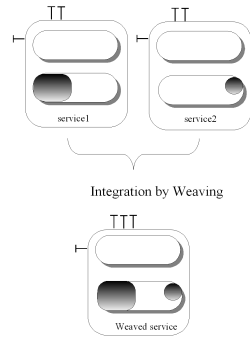**Fig. 4.** Composition of services: second method

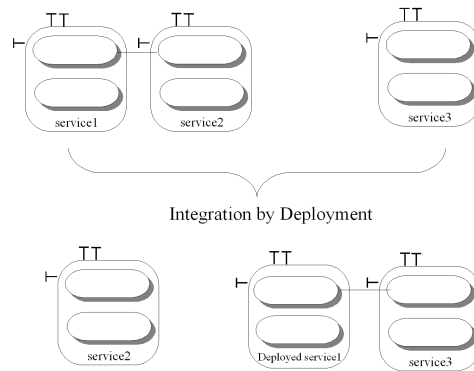**Fig. 5.** Weaving of services

Fig. 6. Deployment of services

- The contract descriptions. Contracts [15] synthesize the terms and conditions of the negotiations in clauses.
- The clauses of the contract. Clauses contain elements that can be measured and quantified, such as QoS or memory resources. We enrich clauses with integration parameter such as duration of the integration, etc.
- The penalties to apply in case clauses are not respected.

Strategies define the methodologies of negotiation. The framework defines two kinds of strategies: service level strategy vs host level strategy. We need these two strategies to make viable integrations on any host. Service level strategy is local to the service. Each service know the integrations it is part of and can establish contracts according to this information. The host level strategy is managed by the *Negotiation Manager Service*. This 'global' information is necessary to prevent services, on the same host, from making contract that can not be

honored. For example, if the negotiation between two services is about memory resources, the service level strategy will take into account only the memory consumed by the service whereas the host level strategy will consider the memory consumed by all services on the host.

The negotiation is split into three phases: Initialization, Active Negotiation and Destruction.

- **Initialization**: This list of stages occurs in this specific order and for a fixed number of time.
    1. propose contract: the service S1 that requires the integration initiates the negotiation with the service S2 by proposing a contract.
    2. analyze contract: S2 analysis the clauses of the contract and verifies if it can satisfy them. Two kinds of analyze may occur, service level analyse and/or host level analyse.
    3. accepte/refuse contract: in case of acceptance, the contract is signed. If not, another proposition can be made (back to step 1).
- **Active Negotiation**: This stage may occur multiple times during the life time of the contract.
    1. isAlive contract: verify the duration of the contract and notify when it ends or when one of the services violate the contract. This verification can be monitored by the services which signed the contract or the negotiation manager service.
    2. re-negotiate contract: services can re-negotiate a contract.
- **Destruction**: This stage occurs only once during the life time of the Contract.
    1. invalidate contract: the contract is invalidated.

3.5   Life Cycle Manager Service

The *Life Cycle Manager Service* is in charge of the integration life time. The integration can have, from its creation, a life time cycle, known and managed by the *Life Cycle Manager Service*. Once this time expires, the integrated service must be disintegrate.

To manage the life cycle of integrated services, each integrated service implements a *IntegrationLifeCycle* interface.

The Life cycle of an integrated service is divided into three parts: Initialization, Active Integrated Service, and Destruction.

- **Initialization**: This list of stages occurs in this specific order, and occurs only once during the life of the integrated service. Beside each stage, we precise the needed interface.
    1. enableLogging: activate and save logs [`IntegrationLifeCycle`].
    2. verify that the service implements the `Negotiable` interface: make sure that the service to integrate accepts negotiation [`Integrable`].
    3. if the service implements `Negotiable` then propose contract: establish a negotiation [`Negotiable`] else proceed to step 4.

4. integrate: integrate services using one of the techniques defined by the *Technical Integration Service* for a specified life-time [`Integrable`].
5. initialize: initialize the integrated service [`IntegrationLifeCycle`].
6. start: start the integrated service [`IntegrationLifeCycle`].

– **Active Integrated Service**: This list of stages occurs in this specific order, but may occur multiple times during the life of the integrated service. Beside each stage, we precise the needed interface.
1. suspend: suspend for a certain time the integrated service [`IntegrationLifeCycle`].
2. isAlive: verify the duration of the life cycle [`IntegrationLifeCycle`].
3. resume: re-start the integrated service [`IntegrationLifeCycle`].

– **Destruction**: This list of stages occurs in the order specified, and occurs only once during the life of the integrated service. Beside each stage, we precise the needed interface.
1. stop: stop the integrated service [`IntegrationLifeCycle`].
2. invalidate contract: to invalidate a contract [`Negotiable`].
3. disintegrate: separate the integrated service into several services [`Integrable`].

The *Life Cycle Manager Service* satisfies the life-time requirement of the *ANIS* system.

4 *ANIS* Prototype

In this section, we present our developing API framework and the toolkit implementation using Java and OSGi technologies. In this article, we detail the three integration techniques proposed by the *Technical Integration Service*. Negotiation and integration life cycle implementations will be the subject of another article.

4.1 ANIS Framework: APIs

`IntegServ` package contains three other packages implementing the integration management interfaces. Package `TechnicalService` (cf. figure 7) is composed of three classes implementing the three integration techniques. Package `NegotiationService` (cf. figure 8) is composed of classes referring to the three main parts of negotiation: Contracts, strategies and negotiation phases and package `LifecycleService` for managing the integrated services. Interface `Integrable` provides several methods allowing to manage the integration of services: method `integrate`, method `disIntegrate`, method `getIntegratedServices` and method `isNegotiable` (to verify if a service accept negotiation). Interface `Negotiable` provides several methods allowing to manage the negotiation between at least two services: method `propose` (to propose a contract), method `accept` (to accept a contract), private method `analyzeContract` (to analyze the contract whether with a service level or host level strategy), method `monitoringContract` (to monitor the contract).
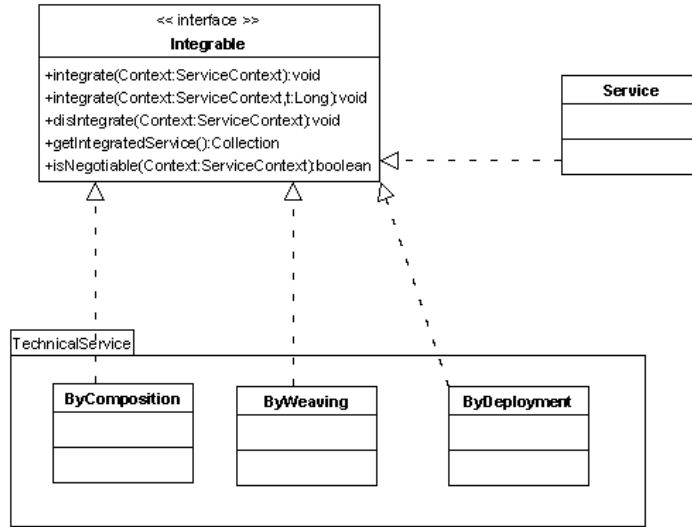
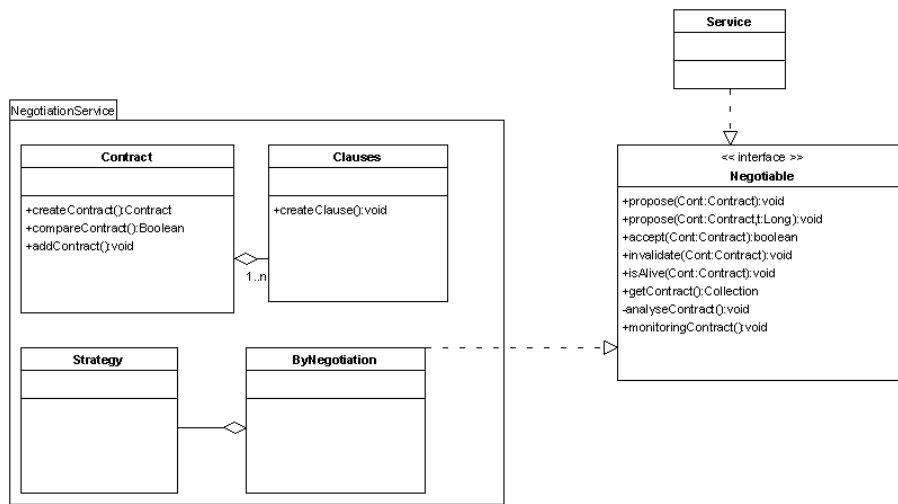**Fig. 7.** UML class diagram of *TechnicalService*

Fig. 8. UML class diagram of *NegotiationService*

4.2 *ANIS* Toolkit: OSGi Implementation

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. We implement our developing framework on Oscar [16] which is open source implementation of OSGi framework specification. We enrich it with an instantiation of our *IntegServ* service.

We apply our service model to OSGi's bundle and service interfaces; objects are Java runtime objects instantiation of classes started by an `Activator`; bindings are modeled by the `manifest.mf` file. A service is provided by a bundle. A bundle can provide diverse services.

Use case scenario In this article we choose to detail the implementation of our deployment (download with RMI), composition by redirection call and weaving techniques. We implemented the following scenario. Max, an architect, wants to accomplish a video of one of his models. Equipped with his PDA, he enters a studio of production of video clip. The studio has a camera and movie maker software. The PDA integrates the software of the video camera automatically as soon as Max enters the studio. Max begins taking the shots he needs of his model. He can command the camera by using a familiar interface installed on his PDA. He doesn't need to know how the camera works! The PDA downloads the driver of the camera (cf. figure 9) and weaves it with the interface software that Max always uses (cf. figure 9). Then, Max uses the movie maker software that the PDA integrated to create his video clip. The PDA composes the movie maker software with the driver of the camera proposing then a new service that enables Max to take shots and create the sequence of the video clip (cf. figure 9). This new service, initially not offered by environment is now available on Max's PDA.

Integration call The service interface of the PDA calls `integrate` method of `Integrable` interface with the remote context that includes the host of the camera (step 1 figure 10):

```
serviceInterface.integrate(context);
```

Listing 1.1. Service B integrates context containing service A

The `context` parameter of `integrate` method is a `ServiceContext` type that inherits the `BundleContext` class of OSGi. This `ServiceContext` contains the list of services available in the context of the service we wish to integrate. It can be `ConcretContext` class when the context is the host of the service. It can also be `ConcretContextSerializable` class when the context is another host. This call to `Integrable` interface will be redirected to the local *IntegServ* on the PDA.
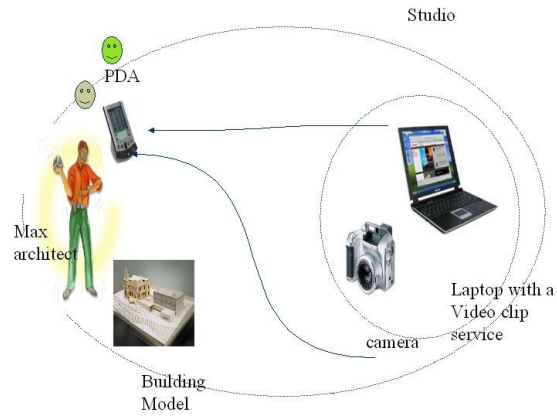
Fig. 9. Use case scenario

Deployment Technique Technically, services are provided by bundles. On each host we define a distributed decentralized Oscar Bundle Repository (OBR) where all bundles are stored. The *IntegServ* service running on the PDA analysis the location of the camera service and decides to pull it from its host and run it locally before integrating it with service interface on the PDA. Service camera is stopped and the bundle is serialized and streamed (step 3 figure 10).

The bundle is then downloaded from the stream and installed in the local OBR of the PDA.
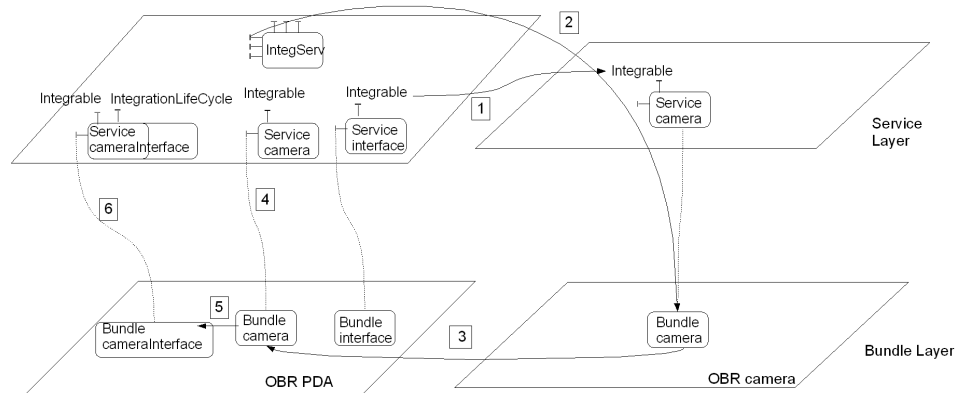

Fig. 10. Integration of the camera service in the PDA

Weaving Technique Once downloaded, bundle camera is installed and started (step 4 figure 10), service camera is now available on the PDA. *IntegServ* can now apply a local weaving to integrate service interface of the PDA and service camera. The weaving technique weaves into the code of service interface the call for the service camera. As shown in figure 11, this technique adds a `Aspect.class` to the bundle interface and creates a new bundle cameraInterface after weaving this class to `serviceInterface.class`.

The service interface is stopped, classes are extracted from the bundle interface. The AspectJ compiler, given an aspect in source form, produces a binary aspect and runs the weaver. Byte-code weaving takes classes and aspects in `.class` form and weaves them together to produce binary-compatible `.class` files that run in any Java VM and implement the AspectJ semantics. Practically, AspectJ takes `Aspect.class` and weaves it with `ServiceB.class`. Finally, the whole are jarred into a new bundle cameraInterface offering service cameraInterface (figure 11).

The `Aspect` aspect defines a `callMethodB` pointcut that captures calls to all public static methods with names `methodB` taking any arguments. Then, the aspect defines one advice after the `callMethodB` pointcut : calling `methodA` of service A, `callMethodA(context)`. We are testing another way to weave the two services. For now weaving is done using AspectJ. The aspect simply do a redirection call to service A after the execution of method B of service B. Another way is to write aspects that weave not a call for `methodA` in `ServiceB.class` but the implementation of `methodA` as it is in `ServiceA.class` (cf. figure 11). To do that, service interface and service camera are stopped, classes are extracted from the two bundles interface and camera. Byte-code weaving takes `ServiceB.class`, `ServiceA.class` and `Aspect.class` and weaves them together to produce a new `ServiceB.class` file. Finally, we jar the whole into a new bundle cameraInterface that offers service cameraInterface.
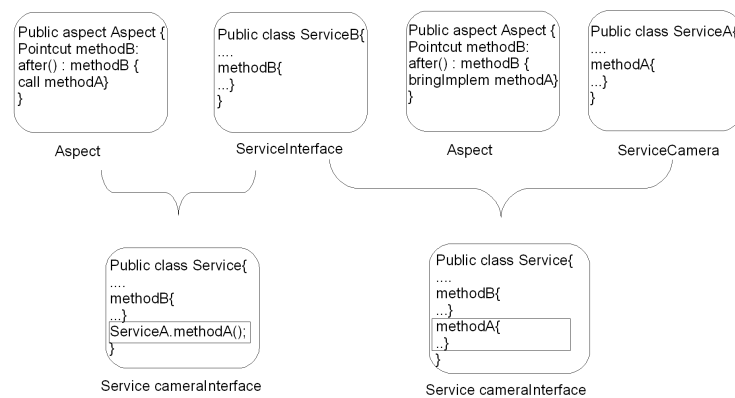

Fig. 11. Two different weaving methods for integrating the camera service and the interface service

Composition by Redirection Call Technique A composition by redirection call (cf. figure 12) is then decided by the *IntegServ* to integrate the cameraInterface with the videoClip. A bundle C is created offering a new service C accessible by the method `methodC()` and redirecting to services cameraInterface and videoClip (if these two services are available). Method C launches method cameraInterface followed by method videoClip. The `ByComposition` class figure 7 implements here the second method of composition (figure 4) because the method cameraInterface and videoClip match.
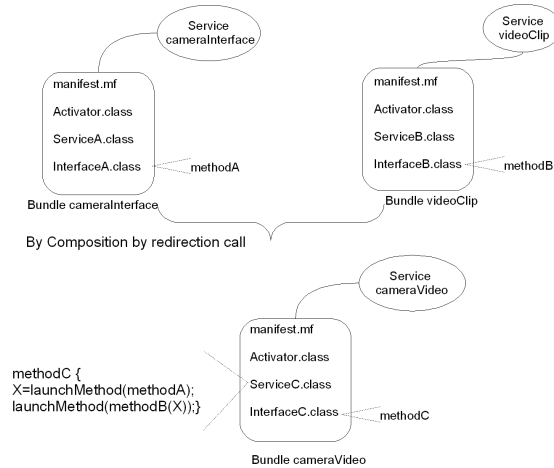

Fig. 12. Composition by redirection call to the cameraInterface service and video clip service

The *IntegServ* creates the bundle C by writing source file, compiling them and jarring them. The `compile()` method uses the `sun.tools.javac.Main()` of the `com.sun.tools.javac.Main` package and the `jar()` method the `sun.tools.jar.Main()` method.

Nothing prevents from adding a new technique and enrich the OSGi toolkit. The procedure is simple. All we need is to add a new implementation of the method `integrate` in the `TechnicalService` package defined figure 7. Nothing also prevents from using another Toolkit based on another technology that respects *ANIS* API Framework and *ANIS* generic service model.

5 Related Work

Three major domains of object-oriented programming lean over the concept of integration: the Component-Based Software Engineering (CBSE) [17], the Aspect-Oriented Programming (AOP) [18] and the Service-Oriented

Programming (SOP) [19]. Each of these domains has several definitions and techniques of integration according to different existing platforms.

Different types of models based on components as EJBs [3], CORBA Component Model [8] and Fractal [9] allow the interaction between distant components. The integration of components in these different models is often reduced to the deployment and/or the parameterization of these components. In these models, the definition of new components is rather difficult during execution, so the integration of components is often predefined beforehand.

Fractal is a modular and extensible component model. Fractal uses the separation of concerns: separation of interface and implementations, component oriented programming and inversion of control. Fractal vision of integration is to provide composite components defined by their sub-components and the bindings between them.

Recursive and Dynamic Software Composition with Sharing [20] presents a Fractal model which allows sharing of sub-components between components. It defines a structural composition based on containment and binding relationships between components. Two types of bindings are defined, but still these bindings generalize the notion of connectors in architecture definition language. A Fractal framework, which is a projection of the Fractal model in the Java programming language, is presented. This framework offers three different forms of configuration, from static to dynamic configuration. The fact remains that composition is seen as an assembly of components in order to form new, higher-level components. It is reduced to dynamically change bindings to reuse and share components.

Aspect-Oriented programming allows to establish independent transverse concerns (aspects) and to combine them (the weaving) later to produce final application. AspectJ [21], Fac [22] and [23] are models based on aspect, applying the weaving of aspect as method of integration. Fac is an extension of the Fractal component model to support Aspect-Oriented programming. FAC realizes a twofold integration of Component-Based Software Engineering (CBSE) and Aspect-Oriented Software Development (AOSD). Integration is defined as introducing a concern (usually a non functional one such as persistence or security) to a set of components.

Validation of Context-Dependent Aspect-Oriented adaptations to Components [24] aim to take advantage of the expressive power of Aspect-Oriented Programming to modularize concerns about non-functional properties within domain-specific frameworks and components. In this case, both frameworks and components can be refined to satisfy specific requirements at deployment time, composition time as well as at runtime. The key to the integration of Aspect-Oriented Software Development to Component-Based Software Engineering lies in the ability to derive and validate the specification of a component that has been subject to aspect weaving. By explicitly specifying the desired properties of components and aspects, this article aims to increase the capability to

reason about aspect weaving, so that the correctness of a refined component can be verified with respect to a specification. The fact remains that weaving is not done between two component but between a concern and a component.

In the terminology of Service-Oriented Programming, the composition of service very often refers to integration of services. Web services [4] provided a new solution for reusing and assembling web software or components under the distributed service-oriented architecture and across different platform environment with a series of XML-based protocol. Nowadays, researches aim at developing an architecture which allows the composition of service by using a logical reasoning given by the languages of description of service as the DARPA Agent Markup Language (DAML) [25], Universal Description, Discovery and Integration (UDDI) and Web Service Description Language (WSDL) [26]. These languages define standard ways for service discovery, description and invocation (message passing).
SWORD [2] is a developer toolkit for building composite web service. It does not deploy the emerging service description standards such as WSDL and DAML-S, instead, it uses rule-based plan generation, it specifies the web services by using Entity-Relation model.
Architecture-based Web Service Composition Framework and Strategy [27] proposes a kind of relationship-oriented service composition description language, RSCDL, for description the composition process. RSCDL language can guarantee that the complex service can be composed of the reused atomic services by different structural types and can provide a value-added service to different users. Relying on this language description, an architecture-based service composition model was proposed providing a mechanism for dynamic services management and deployment. These works focuse more on services matching and searching services for composition than on defining different techniques to compose them.

6   Conclusion and Future Works

In this paper, we have presented *ANIS* A Negotiated Integration System. *ANIS* provides a framework including a set of integration management interfaces - `Integrable`, `Negotiable`, `LifeCycle` - and the tools implementing these interfaces.
The contributions of *ANIS* are three-fold:

- A simple and efficient integration system for a generic model of service.
- An API developing framework for the integration offering different techniques of integration, negotiation by contracts and the capability to manage the life cycle of the integration.
- The implementation of OSGi toolkit that demonstrates the feasibility of *ANIS* and details the integration techniques (deployment, composition by redirection call and weaving).

Our system verifies the defined requirements. The *ANIS* framework is generic. It applies to all types of services, and can be used by all types of applications. The *ANIS* service model is also generic and can be applied to existing implementations. We provide a special API the *IntegrationLifeCycle* to manage the life cycle of the integrated service. We provide within the `Integrable` API a `disintegrate` method that reverse the integration. Our negotiation protocol based on contract permits the negotiation before the integration assuring more efficient integration. Our integration process is simple, services only call `integrate` method of `Integrable` interface without knowing the techniques used behind.

We implement our prototype using Java platform and OSGi technology. We enriched OSGi platform with our *IntegServ* service. In this article, we detail the different techniques of integration: deployment, composition by redirection call and weaving. The limit of the current version of *ANIS* is in its toolkit; the applications that need to use it has to support the Java Runtime Environment and the OSGi platform.

We are improving ANIS by enriching our service model and negotiation process with semantic description. Services but also contracts will be described in OWL-S. For services, this will allow to integrate services having the same semantic description but belonging to different platforms. For contracts, we will define an ontology for the terms of the clauses of the contracts. The strategies of negotiation that are now based on simple matching rules will be enriched using this ontology.

We are also improving our *ANIS* system by adding a new requirement, automation. *ANIS* will become Automatic Negotiated Integration of Services system. This requirement is particularly interesting in pervasive and context-aware environments. We can define context-aware strategies for choosing the services to integrate depending on the execution context. In such environments, we also aim to test the performance of our ANIS prototype for instance, its reactivity to service availability.

References

1. Kien, T.N., Erradi, A., Maheshwari, P.: WSMB: a middleware for enhanced web services interoperability. In: Interop-ESA'05, First International Conference on Interoperability of Entreprise Software and Applications. (2005) Geneva, Switzerland.
2. Ponnekanti, S.R., Fox, A.: SWORD: A Developer Toolkit for Web Service Composition. In: 11th World Wide Web Conference. (2002) Honolulu, USA.
3. Monson-Haefel, R.: Entreprise JavaBeans. O'Reilly & Associates (2000)
4. Iverson, W.: Real Web services. O'Reilly (2004)
5. Alliance, O.: OSGi Service Platform, Core Specification Release 4. Draft (2005)
6. Loritsch, B.: Developing With Apache Avalon. Technical report, Apache Software Foundation (2001)
7. Le Mouël, F., André, F., Segarra, M.T.: AeDEn: An Adaptive Framework for Dynamic Distribution over Mobile Environments. Annales des Télécommunications **57**(11-12) (2002) 1124–1148

18

8. Zahavi, R.: Entreprise Application Integration with Corba Component and Web-Based solutions. Johne Wiley & sons (1999)
9. Bruneton, E.: Developing with Fractal. The ObjectWeb Consortium, France Telecom (R&D). (2004) version 1.0.3.
10. OSGIalliance: About the OSGI service platform. Technical report, OSGI alliance (2004) revision 3.0.
11. Rennie, M.W., Misic, V.B.: Towards a Service-Based Architecture Description Language. Tr 04/08, University of Manituba (2004)
12. Kumaran, S.I.: JINI Technology An Overview. Prentice Hall PTR (2002)
13. Corporation, M.: Understanding UPnP : A white paper. Technical report, UPnP Forum (2000)
14. Jin, H., Wu, H.: Semantic-enabled Specification for Web Services Agreement. International Journal of Web Services Practices **Vol.1**(No.1-2 pp. 13-20) (2005)
15. Clotet, D.P., Pallotta, V., Rajman, M.: Systematic definition and assent to eContracts for Web Services. In: Workshop on Contract Architectures and Languages (CoALa 2005), in conjunction with the 9th IEEE International Enterprise Computing Conference (EDOC 2005). (2005) Enschede, The Netherlands.
16. Hall., R.S.: Oscar an OSGI framework implementation. Technical report, Objectweb organisation (2005)
17. Heineman, G.T., Councill, W.T.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley (2001)
18. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: European Conference on Object-Oriented Programming (ECOOP). (1997) 220–242
19. Singh, M., Huhns, M.N.: Service-Oriented Computing. Wiley (2005)
20. Bruneton, E., Coupaye, T., Stefani, J.: Recursive and Dynamic Software Composition with Sharing. In: Seventh International Workshop on Component-Oriented Programming (WCOP02) at ECOOP 2002. (2002) Malaga, Spain.
21. Laddad, R.: AspectJ in Action: practical Aspect-Oriented Programing. Manning publications (2003)
22. Pessemier, N., Seinturier, L., Duchien, L.: Components, ADL and AOP: Towards a common approach. In: Workshop ECOOP Reflection, AOP and Meta-Data for software Evolution (RAM-SE04). (2004)
23. Douence, R., Fritz, T., Loriant, N., Menaud, J.M., Ségura-Devillechaise, M., Südholt, M.: An expressive aspect language for system applications with Arachne. In: 4th International Conference on Aspect-Oriented Software Development (AOSD'05). (2005)
24. Cottenier, T., Elrad, T.: Validation of Context-Dependent Aspect-Oriented adaptations to Components. In: WCOP, Oslo (2004)
25. Sheshagiri, M., des Jardins, M., Finin, T.: A planner for composing services described in DAML-S. In: International Conference on Automated Planning and Scheduling (ICAPS) 2003 Workshop on planning for web services. (2003)
26. Walsh, A.E.: UDDI, SOAP and WSDL: the Web Services specification Reference book. Pearson Education (2002)
27. Yuan, R., li Zunchao, Boqin, F., Jincang, H.: Architecture-based Web Service Composition Framework and Strategy. In: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems(ECBS'05). (2005)