No d'ordre 2008 ISAL 0062

Thèse

Spontaneous Integration of Services in Pervasive Environments

À présenter devant

l'Institut National des Sciences Appliquées de Lyon
École Doctorale Informatique et Information pour la Société

pour l'obtention du

Grade de Docteur

spécialité informatique

par

Noha IBRAHIM

Soutenue le 23 septembre 2008

Devant le jury composé de

Rapporteurs :	Guy Bernard,	Professeur,	Telecom & Management SudParis (France)
	Satoh Ichiro,	Professeur,	National Institute of Informatics (Japan)
	Valérie Issarny,	Directeur de recherche,	INRIA Arles (France)
Examinateur :	Michel Riveill,	Professeur,	Polytech'Nice (France)
Directeurs :	Frédéric Le Mouël,	Maître de conférences,	INSA de Lyon (France)
	Stéphane Ubéda,	Professeur,	INSA de Lyon (France)
Invitée :	Christine Collet,	Professeur,	Grenoble INP - ENSIMAG (France)

Thèse effectuée au sein du Centre d'Innovation en Télécommunications et Intégration de Services (CITI) de l'INSA de Lyon, équipe *AMbient Architectures: Services Oriented, Networked, Efficient and Secure* (AMAZONES) de l'INRIA Rhône-Alpes

Résumé

La prolifération croissante des dispositifs numériques sur le réseau ouvre la voie à une nouvelle vision de l'informatique, la vision d'informatique diffuse. Toutefois, composer les services offerts par ces dispositifs pour réaliser des applications, ou services, répondant aux attentes des usagers reste un problème ouvert.

Cette thèse vise à proposer une solution à ≪ l'intégration spontanée de services dans les environnements de l'informatique diffuse ≫. La solution proposée se base sur une architecture orientée service et se decompose en trois fonctionnalités majeures: la transformation de service, la composition de services et l'adaptation de services. Un modèle générique d' ≪ intergiciel d'intégration de services ≫ SIM est proposé, ainsi qu'une instanciation particulière MySIM, est développée. Le modèle d'intergiciel SIM intègre les services présents dans l'environnement à travers quatre fonctionnalités: la transformation de descriptions de services propriétaires en un modèle de référence, la génération de toutes les compositions possibles des services au regard de leur compatibilité, la sélection des compositions de services valide en vue de leur qualité de service et finalement la construction qui instancie les services composites retenues et les publie dans l'environnement. MySIM est une instanciation de SIM qui se distingue par une approche spontanée de l'intégration de services. Les phases de génération, évaluation et construction de services sont réalisées de manière systématique et spontanée par l'intergiciel, sans intervention de l'utilisateur, en fonction des arrivées et départs des services au sein de l'environnement.

Une implémentation de l'intergiciel MySIM basée sur la technologie OSGi a été proposée. Cette implémentation permet de valider pratiquement la solution introduite dans cette thèse pour résoudre les problèmes de spontaneité liés aux environnements pervasifs, et d'en évaluer les performances.

Abstract

If mobile computing brought challenges and constraints to distributed systems, it kept evolving with the evolution of the technology. The mission of mobile computing is to allow users to access any information using any device over any network at any time. When this access becomes to every information using every device and over every network at every time, we can then say that mobile computing has evolved to what we now call pervasive computing.

A computing infrastructure where "everything is a service" offers many new system and application possibilities. Among the main challenges, however, is the issue of standardized way of application development in such heterogeneous environments. The natural way of doing this is by performing service composition, either by creating services and composing them according to requirements, or adapting and reusing existing services in order to achieve a given task. In such open environment the ability of services to adapt and be extended represents the primary driving force. These two actions of composition and adaptation are only possible if services are implemented and described in interoperable languages. For that reason, service transformation is a critical step preceding any composition or adaptation action.

In this thesis, we focus our interest upon the intersection of two major domains, the service-oriented architectures meeting the pervasive computing, and propose a novel solution to integrate services in a pervasive environment. The main contributions of this thesis are threefold. They arise from the lacks noticed in current service integration middleware for the pervasive environments. If many middleware dealt with one or more of our service problems - transformation, composition, adaptation - few proposed a unified vision for the service integration in pervasive environment, a management of the functional and non-functional properties of services during the integration, and especially, a spontaneous service integration that extends environments transparently with functionalities.

We define the SIM model a Service Integration Middleware model adapted to service integration middleware for pervasive environment. We give our middleware instantiation, `MySIM` and provide the services functional and non-functional equivalence and composable relations to define service transformation, service composition, and service adaptation. Based on these relations, spontaneous service integration adapted to pervasiveness is explained. We developed a prototype as a proof of concept that we tested over *MyStudio* environment. We think that our work on spontaneous service integration with others related to intelligence, smartness and pro-activity lead to the development of the Ambient Intelligence, the very likely probable evolution of pervasive computing.

Acknowledgments

I would like to thanks my jury members, Guy Bernard, Valérie Issarny, Ichiro Satoh, and Michel Riveill for the time they spend over my thesis and their interesting feedbacks. I would like also to thanks them for attending my phd defense.

I would like to thank my advisor, Frédéric Le Mouël, for supporting me over the years, and for giving me so much freedom to explore and discover new areas of research. My other team members have also been very supportive. Stéphane Frénot brought a more than useful perspective to the thesis. Stéphane Ubéda as the director of the laboratory gave me opportunities to be involved in the laboratory every day life. I would like also to thank Fabrice Valois for his friendship and support.

I would like to thank my many friends and colleagues at Citi with whom I have had the pleasure of working over the years. In particular, I would like to thank Amira Ben Hamida for her friendship and support all along these years of thesis, for which I am very grateful.

Finally, I would like to thank my parents and family for giving me the motivation to finish this thesis, and especially for their unconditional love.

On ne voit bien qu'avec le coeur, l'essentiel est invisible aux yeux.

Le petit prince

Contents

List of Figures

List of Tables

Listings

Symbol	Association
Σ	alphabet
`O`	set of ontologies
`N`	set of concepts across `O`
`S`	set of services
`Ifc`	set of service interfaces
`Op`	set of operations
`Np`	set of non functional properties
Np_{QN}	set of quantitative non functional properties
Np_{QL}	set of qualitative non functional properties
`In`	set of operation inputs
`Out`	set of operation outputs
`Cpt`	set of operation concepts
`Impl`	set of operation implementations
`Pr`	set of implementation functional properties
`Protocol`	set of services protocols
$\Sigma*$	set of all words over the alphabet Σ
ϵ	empty word
a	word or element over the alphabet, $a \in \Sigma*$
x	word or element over numbers, $x \in \mathbb{R}$

Table 1: List of Symbols

Symbol	Association
$\mu(x)$	mean value of x, $x \in \mathbb{R}$
$\sigma(x)$	standard deviation of x, $x \in \mathbb{R}$
z-score(x)	standardization of x, $x \in \mathbb{R}$
$\eta(x)$	z-score(x) normalized to $[0,1]$, $x \in \mathbb{R}$
$\wedge$	logical and
$\vee$	logical or
$\neg$	logical not

Table 2: List of operators over an element

Symbol	Association
$\wedge$	logical and
$\vee$	logical or
$\neg$	logical not
$<\ldots>$	externally defined token
$\{\ldots\}$	set (one of)
$(\ldots)$	grouping
$\|S\|$	number of elements in the set S
S^*	repetition of the previous element zero or more times, $\|S\| > 0$
S^+	repetition one or more times, $\|S\| > 1$
$S^{0..1}$	repetition zero or one time, $\|S\| = 0 \vee 1$

Table 3: List of operators over a set

Chapter 1

Introduction

The goal of this chapter is to highlight the challenges in computer system research raised by pervasive computing and put the light on a new challenging problem, the integration problem within distributed, mobile, and pervasive middleware. We begin by examining the mobile computing domain, its characteristics, and constraints. Then, we explain the evolution from mobile computing to pervasive computing. This evolution has affected not only networking and application layers, but essentially the middleware layer, the glue of all software systems. We focus on service-oriented architectures (SOA) and explain this middleware emerging approach that uses services as main building blocks. Then, we delve deeper into one of the key research problem raised by pervasive computing meeting the SOA: the service integration problem. Finally, we outline the contributions of this thesis and detail the chapter contents.

1.1 From Mobile to Pervasive Computing

Hardly a day passes without some new evidence of the proliferation of portable computers in the marketplace, or of the growing demand for wireless communication. Support for mobility has been the focus of number of experimental systems, research and commercial products, and that since several decades.

Although many basic principles of distributed system design [Mullender 1993, Coulouris et al. 2001] continued to apply, mobile computing brought more constraints to research problems. Indeed the solution of many previously-encountered problems becomes more complex and new problems, related to mobility, arise.

In his article, Satyanarayanan [Satyanarayanan 1996] listed four main constraints raised by mobile computing:

- Resource-poor: mobile elements are resource-poor relative to static ones. Considerations of weight, power, size, and ergonomics will cause a penalty in computational resources such as processor speed, memory size, and disk capacity. These characteristics are closely related to the embedded technologies of these elements. Even if mobile elements will improve as the technology is improving, they will always be resource-poor relative to static elements.

- Hazardous mobility: mobile elements are more vulnerable to loss or damage. Their mobility makes it difficult to consider their availability. Security problem are also more important for mobile elements as their mobility lead them to many different places and situations, much more complex than in a static position.

- Variable connectivity: mobile connectivity depends strongly on the mobile element's geographical position. Some buildings may offer reliable, high-bandwidth wireless connectivity while others may only offer low-bandwidth connectivity. Outdoors, a mobile element may surely have to rely on a low-bandwidth wireless network with gaps in coverage.

- Finite energy resource: while battery technology will undoubtedly improve over time, the problem of power consumption will not diminish. This concern must be considered at every level, hardware or software, to be fully effective. The finite energy resource is a fact that every developer of applications in mobile environments must deal with.

Due to these constraints, mobile computing rises several research problems listed by [Satyanarayanan 2001]. Many of these problems of computer science deal with the challenges of mobile computing:

- Mobile networking: protocols and techniques that allow mobile device users to move from one network to another by supporting routing to and from mobile hosts. Mobile networking includes Mobile IP [Bhagwat et al. 1996], ad hoc network protocols [Royer et al. 1999], and techniques to improve TCP performance in wireless networks. This area of research tackles the network layer.

- Mobile information access: in mobile environments, data and more generally information should be able to "follow" the mobile host. Mobile information access includes disconnected operation [Kistler et Satyanarayanan 1992], bandwidth-adaptive file access [LBM et Satyanarayanan 1995], data migration and selective control of data consistency [Terry et al. 1995].

- Adaptive protocols and applications: two levels of adaptation are required. Protocols have to adapt to a different set of parameters in mobile wireless networks. They need to be designed for adaptation to multiple parameters such as latency, burst error, disconnection during hand-off, asymmetry of the link, location, and cost. The adaptation we are interested in is more about applications' adaptation [Satyanarayanan et al. 1994]. The support for adaptative applications includes transcoding by proxies [Fox et al. 1996] and adaptative resource management.

- QoS-aware system: with mobility, the flow path of information may change with each move, hence delays will be surely impacted by mobility. This area of research tries to adapt applications, services, and all the functionalities provided to the clients to the ever changing world of mobile computing.
- Location or position sensitivity: use of location, location sensing [Want et al. 1992], and location-aware system behavior are some of many research challenges for the mobile computing domain. These challenges related to location and position are very often located in the network layer.

If mobile computing brought challenges and constraints to distributed systems, it kept evolving with the evolution of the technology. The mission of mobile computing is to allow users to access any information using any device over any network at any time. When this access becomes to every information using every device and over every network at every time, we can then say that mobile computing has evolved to what we now call pervasive computing. This evolution from mobile to pervasive has an impact on the research problems already existing, but also introduces new challenges and problems.

One can not introduce pervasive computing without citing the famous Mark Weiser' 1991 vision. *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it"* - so began Mark Weiser's paper that described his vision of ubiquitous computing, now also called pervasive computing. The essence of that vision was the creation of environments saturated with computing and wireless communications capability, yet gracefully integrated with users [Satyanarayanan 2001]. If many of these features were unavailable in 1991 and more related to science fiction, many key building blocks needed for this vision are now viable commercial technologies: wearable and hand-held computers, high bandwidth wireless communication, location sensing mechanisms, and so on.

Pervasive computing is all about everywhere and anytime computing. It spreads intelligence and connectivity to more or less everything. So conceptually, machines, clothing, tools, appliances, homes and even the human body will be embedded with chips to connect to an infinite network of other devices. This creates an environment where the connectivity of devices is embedded in such a way that it is unobtrusive and always available. Information instantly accessible anywhere and anytime is what pervasive computing is all about. It simplifies life by combining open standards-based applications with everyday activities. It removes the complexity of new technologies, enables us to be more efficient in our work and leaves us more leisure time and thus pervasive computing is fast becoming a part of everyday life. The pervasive computing is a one to several interactions between a user and several personal devices. The use is implicit, discrete, transparent, and usually powerful.

A user is at the center of a pervasive environment. As users move, their computing environment moves with them, changes and extends as users meet, shrinks as users separate. Moreover, users should be able to take full advantage of the local capabilities and resources within a given environment, even with its ever changing topology.

The challenges already brought by mobile computing have more complex solutions in pervasive environments. Satyanarayanan in [Satyanarayanan 2001] cites some of these challenges:

- Effective use of smart spaces: a space may be an enclosed area or a well-defined open area. By embedding computing infrastructure to almost everything and everywhere, smart spaces bring together the world of humans and the world of computing in a new unexplored way. Smartness involves accurate sensing followed by intelligent control or action between the two worlds, namely, machine and human.

- Invisibility: or the complete disappearance of pervasive computing technology from a user's consciousness; if a pervasive computing environment continuously meets user expectations and rarely presents him with surprises, it allows him to interact almost at a subconscious level. Automated techniques to dynamically reconfigure the network when required are also crucial to realizing the pervasive computing vision.

- Localized scalability: future pervasive computing environments will likely face a proliferation of users, applications, networked devices, and their interactions on a scale never experienced before. As environmental smartness grows, so will the number of devices connected to the environment and the intensity of human machine interactions. Scalability is thus a critical problem in pervasive computing. The density of interactions has to fall off as one moves away.

Undoubtedly, research in the pervasive computing field has considerably advanced. At the networking level, for example, wireless communication is already possible through technologies like Bluetooth, Wi-Fi, and Zigbee. Mobility and ad hoc networking protocols have also been developed, like Mobile IP, GPRS, UPnP, and Zeroconf. At an application level, when considering the context-awareness of applications, undoubtedly some efforts have been made in specific application domains. As examples, we can cite several domains such as home automation, health care, learning and multimedia. The task of building pervasive computing applications can be too tedious if performed from scratch. In other words, the developer will need to deal with low level networking protocols to high level applications context awareness. This, of course, deviates the attention of the developers to tasks that are not the purpose of the application. Instead, they should only concentrate on the application logic, that is, the tasks the application must perform. This is why we are interested in middleware for pervasive computing. In the next section we define what is a middleware and introduce the most recent and emergent middleware approach, the service-oriented architecture.

1.2 From RPC Middleware to Service-Oriented Architectures

Middleware are enabling technologies for the development, execution and interaction of applications. These software layers are standing between the operating systems and applications. They have evolved from simple beginnings - hiding network details from applications - into sophisticated systems that handle many important functionalities for distributed applications - providing support for distribution, heterogeneity and mobility. The evolution of middleware has been influenced by numerous developments and standards efforts. Various middleware paradigms were defined. We cite the Remote Procedure Call (RPC) middleware [Myerson 2002], the Message Oriented Middleware (MOM) middleware [Gomolski 1997], the Object Request Broker (ORB) middleware [Puder et al. 2005], and the Service-Oriented Architectures (SOA) middleware.

The term middleware first appeared in the late 1980s to describe network connection management software, but did not come into widespread use until the mid 1990s, when network technology had achieved sufficient penetration and visibility. By that time, middleware had evolved into a much richer set of paradigms and services, offered to build distributed applications more easily and in a more manageable way.

The term middleware was associated with relational databases for many practitioners in the business world through the early 1990s. By the mid-1990s this was no longer the case. Concepts similar to today's middleware previously went under the names of network operating systems, distributed operating systems and distributed computing environments. Systems such as Apollo's Network Computing Architecture (NCA), Sun's RPC standard, and the Open Software Foundation's Distributed Computing Environment (DCE) are all examples of successful RPC-oriented middleware that has been used for significant production applications.

At the same time these systems were being produced in the 1980s and early 1990s, significant research was occurring in the area of distributed objects. Distributed objects represented the confluence of two key areas of information technology: distributed systems and object-oriented design and programming. As a result of research on these and other distributed object systems, Common Object Request Broker Architecture (CORBA) was created.

The CORBA system is built on top of ORB, which encompasses the entire communication infrastructure necessary to identify and locate objects, handle connection management, and deliver data. At the same time, a specific class of middleware, MOM, that operates on the principles of message passing or message queuing appeared.

The MOM middleware unlike RPC and object-orientation is an asynchronous form of communication, i.e. the sender does not block waiting for the recipient to participate in the exchange. While the evolution of distributed objects in the 1990s was marked by significant efforts to establish standards such as CORBA, the evolution of messaging oriented middleware was practically devoid of standards efforts.

In the late 1990s, the growing popularity of Java, the explosive growth of the World Wide Web (WWW), and lessons learned from CORBA and MOM were all combined to form the Java 2 Enterprise Edition (J2EE) specification, a comprehensive component middleware platform. J2EE provides support for numerous application types, including distributed objects, components, web-based applications, and messaging systems. Throughout the development of CORBA, J2EE, and proprietary messaging systems, Microsoft was busy developing its DCOM distributed objects system.

In 1999 and 2000, the Web's influence on middleware started to become readily apparent with the publication of the initial version of Simple Object Access Protocol (SOAP) [Walsh 2002]. Initially dubbed SOAP was the result of trying to create a system-agnostic protocol that could be used over the Web and yet still interface easily to non-SOAP middleware, including CORBA, DCOM, J2EE, and messaging middleware systems. Given the 1990s "middleware contests" between J2EE, CORBA, and DCOM, and between RPC and message passing, the unified support for SOAP was indeed groundbreaking. At the time, it seemed that SOAP, Web Services [Iverson 2004] and Service-Oriented Computing in general might finally provide the basis for broad industry agreement on middleware standards.

The term service-oriented architecture (SOA) emerged to describe the approach of building loosely coupled distributed systems with minimal shared understanding among system components. The main building blocks in SOA are services. Services are self-describing, open components that support rapid, low-cost development and deployment of distributed applications. A Service-Oriented Architecture (SOA) is a form of distributed system architecture that is typically characterized by the following properties [Milanovic 2006]:

- Logical view: the service is an abstracted, logical view of actual programs, databases, business processes, and so on, defined in terms of what it provides.
- Message orientation: the service is formally defined in terms of the messages exchanged between providers and requesters and not the properties of these latter themselves. The internal structure including features such as implementation language, process structure and even database structure, are deliberately abstracted away in the SOA: using the SOA one does not and should not need to know how a provider implements a service.
- Description orientation: a service is described by machine processable meta-data. The description supports the public nature of the SOA: only those details that are exposed to the public and important for the use of the service should be included in the description.
- Granularity: services tend to use a small number of operations with relatively large and complex messages.
- Network orientation: services tend to be oriented toward use over a network, though this is not an absolute requirement.
- Platform neutral: messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML is usually the most obvious and used format that meets this constraint.

A computing infrastructure where "everything is a service" offers many new system and application possibilities. Among the main challenges, however, is the issue of standardized way of application development in such heterogeneous environment. The natural way of doing this is by performing service composition, either by creating services and composing them according to requirements, or adapting and reusing existing services in order to achieve a given task. In such open environment the ability of services to adapt and be extended represents the primary driving force. These two actions of composition and adaptation are only possible if services are implemented and described in interoperable languages. For that reason, service transformation is a critical step preceding any composition or adaptation action.

1.3 Problem Statement: Service Integration

The challenge we are interested in arises from the intersection of the SOA and the pervasive computing. In such open environments the ability of services to integrate into their new environment represents the new challenges of pervasive computing for the SOA. Though pervasive computing elements are already deployed around users, integrating them into a single platform is still a research problem.

One can ask why integrating services? why not rather just use them? The primary driving force of a pervasive environment is to provide a seamless environment of functionalities to the users. If such computing environment does not present means to take advantages of the available services, it will fail to provide what Weiser defined as the disappearing technologies. To let the technology disappears from user and application perspective, the computing environment need to integrate all the services that come and go so that they can operate correctly and be accessed transparently by users. Whenever something new appears in an environment the natural action to do is to try and integrate it into its new environment, so that it can fit perfectly and provide the best of what it can do. Whenever something disappears from the environment the natural action to do is to mask this disappearance by adapting the environment to this loss. This applies to domains such as computing but also to domains far from it such as in biology or social behaviors.

As the number and heterogeneity of devices and applications increases, integration becomes more complex. Indeed the localized scalability makes it possible to have numerous devices available at the same time in a same smart space. For example, servers must handle thousands of concurrent client connections, and the influx of pervasive devices would quickly approach the host's capacities. The invisibility introduced by pervasive environments requires from the integration to preserve this invisibility by providing a seamless integrated environment to users. The new provided functionalities must disappear into the smart spaces and just be available to respond to the users needs. We state our problem and give a definition to the new identified challenge which is the subject of this thesis: service integration in pervasive environment.

The first problem we are faced with when investigating service integration is the lack of a valid definition. We begin by giving dictionary definitions for the employed terms followed by definitions provided by the literature.

Definition 1 — Integration

> *Integration is an act or instance of combining into an integral whole.*

■

In pervasive SOA, the act of combining into an integral whole, concerns the services available in the surroundings as they are the building blocks of such environment. The integration of these services is essential for the existence, expansion, and development of all the applications within the pervasive environments [Ishikawa et al. 2005]. To define the service integration, we provide answer to these three questions: What to integrate in a service? How to integrate? and When to integrate? The answers are given figure 1.1.

Definition 2 — Service Integration

We define a service integration as the combination of three functional aspects: service transformation, service composition, and service adaptation, taking place at run-time and concerning the computational and behavioral parts of services (cf. figure 1.1).

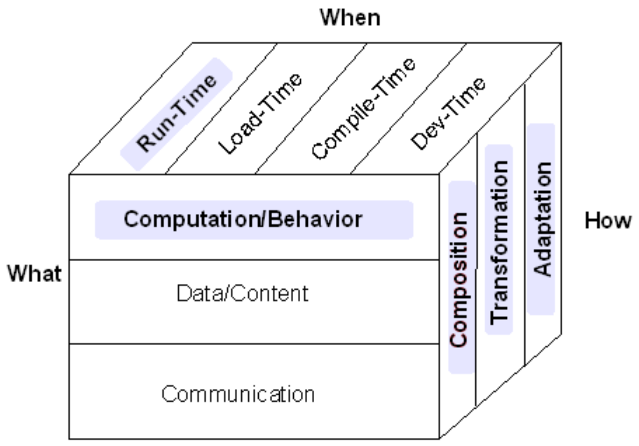


Figure 1.1: Service integration dimensions

■

A service integration is defined as a service transformation, a service composition, or a service adaptation. It can also be a combinations of these three techniques.

Definition 3 — Transformation

Transformation is the process or result of changing from one appearance, state, or phase to another.

■

Nowadays creating enterprise software comes with a lot of compatibility issues. There are simply too many platforms, and too many conflicting implementation requirements, to ever agree on a single choice in any of these fields. Transformation from one service model to another in order to cooperate and integrate is more than essential in pervasive environments. For that an integration to be successful in pervasive environments, it has to be able to transform the diverse services model into models that can interact together, to compose services together [dCastro et al. 2006], and to adapt them or the resulting composition to the ever changing environment. The transformation model-driven development key challenge is in transforming higher-level models into platform-specific models, that can be used to generate implementation level models. Transforming a model into another model means, that a source model is transformed into a target model based on some transformation rules. Different methods can be used for defining the transformation rules. As for the adaptation we limit our "MDA" interest to middleware that propose mechanisms of transformation and translation between service technology model [dCastro et al. 2007].

Definition 4 — Composition

> *Composition is a process of binding two or more entities into a new one. Composition focuses on the global interactions among all participants. Sometimes, term choreography is used instead.*

■

Service composition is defined as the act of linking services together to form or provide something new. This term has been widely used and studied in SOA such as Web services. Composing services in a pervasive environment is all about providing a way to make these services communicate together and create a new functionality or service, unable to exist without the interaction of the whole. Service composition problem has been mainly described as a language description problem. Languages, such as XML, OWL-S [Coalition 2003], and BPEL4WS [Juric et al. 2006] are commonly employed to describe the composition between different services. These languages need to be adapted and extended to the pervasive environment. If some research area are concerned about these extensions and adaptations, others propose new languages for describing composition in pervasive environment, by filling the gap left by the first ones.

Definition 5 — Adaptation

> *Adaptation is the process of making adjustments to suit the environment and to adjust to different conditions.*

■

Unlike the composition, adaptation granularity is defined over one service, that can be itself the result of a composition. Adaptation is very important in dynamic and ever changing environments as it allows to integrate certain services in their new inhabitance. The adaptation problem is one of the biggest challenges for software engineering [Floch 2006, Le Mouël et al. 2002]. This is particularly the case since mobile and pervasive computing have turned adaptation from the slow process of software evolution into a highly dynamic run-time procedure that needs to occur as devices and applications move from network to network. We distinguish several layers where adaptation can be applied: the resource adaptation, the service execution and deployment adaptation, the service composition adaptation, and so on. In this thesis, we are interested in adaptation as a technique of integration. Others such as [Frei 2005] saw also in adaptation a way to allow the architecture and the application components to be extended as required. The premise is to extend the application transparently with new functionality. We limit our studies to the services adaptation defined as a reconfiguration and re-parametrization for a good execution in pervasive environments. The extension in functionality as defined by [Frei 2005] is the result of the service composition defined above.

Many research fields are interested in one of these development trends, but few have tried to combine all these functionalities into one middleware. Current middleware offer mechanisms for the composition and adaptation, or algorithms for the transformation of models, but very few middleware considered the unified view of the service integration as we define it.

1.4 Thesis Outline

This thesis is about service integration in pervasive environments. First, it proposes a survey of available service integration middleware in pervasive environments: service transformation middleware, service composition middleware, and service adaptation middleware. Based on this survey, we highlight three main lacks for service integration in pervasive environments: the unified vision for the integration, the functional and non-functional QoS service integration management, and the spontaneity of the integration in pervasive environments. Then, we present our service integration middleware model and propose our own middleware that responds to the three identified lacks. A prototype implementation is provided as a proof of concept. Finally, we conclude and open perspectives for this work.

- **Chapter 2** provides an overview of the field related to service integration and brings out the drawbacks and limitations of currently existing solutions. First, we depict middleware that do the service transformation, service adaptation, and service composition. Then, we classify these middleware under three requirements that we identify to be essential in pervasive environments.

- **Chapter 3** exposes the contribution of this thesis. A first section presents a unified service integration model, the SIM model with our instantiation middleware, the `MySIM` middleware. The second section introduces and formalizes our generic service model with its relations (equivalence and composition). The third section depicts our spontaneous service integration based on the relations introduced earlier.

- **Chapter 4** presents the implementations of our middleware. First, we depict the overall middleware architecture developed under OSGi. Then, we explain the implementations of the middleware services based on a provided use case (The `MyStudio` use case). Finally the last section details the evaluation of our middleware implementation.

- **Chapter 5** concludes the thesis and presents the major perspectives of this work.

Chapter 2

Overview of the Service Integration Middleware

After, a brief description of the existing integration middleware - component and communication based - we survey in this chapter research efforts investigating service based integration middleware platforms for pervasive computing. First, existing research efforts investigating service transformation middleware, such as Perv-ML [Munoz et al. 2004] and MIDAS [dCastro et al. 2006] are presented in Section 2.2. Then, service composition middleware that focus on issues related with the dynamic composition of pervasive applications such as PERSE [Mokhtar 2007] and SeGSeC [Fujii et Suda 2005] are surveyed in Section 2.3. Finally, we discuss service adaptation middleware that provide solutions for dealing with middleware adaptability in pervasive computing environments, such as CARISMA [Capra et al. 2003] and Madam [Floch 2006] in Section 2.4 and provide an overall discussion and classification in Section 2.5. For conciseness, we choose only two or three systems for each category of the presented middleware. In section 3.1 other middleware are referred to while presenting our own model of Service Integration Middleware.

2.1 Component and Communication Based Middleware

Even if we are interested in this thesis in the SOA, many existing middleware deal with the integration of functionalities modeled in a different way than services. We distinguish the component-based infrastructure and the communication-based infrastructure. If component infrastructure derives from the object concept and is on the way to be replaced by the service oriented infrastructure, many current systems [Pellegrini et Riveill 2003] still use this specific infrastructure. Other middleware are more communication based and do not rely on a predefined infrastructure but propose infrastructures for communication rather than infrastructures for functionalities modeling.

2.1.1 Component-Based Infrastructure

A largely widespread approach is to provide the various functionalities offered by devices through elementary components. These components are very often used to carry out a task or a particular activity. In a component-based infrastructure, components can only interact with their environment through operations at identified access points called interfaces which can be of two main sorts: server (provided) and client (required). Some bindings must be established between components (more precisely between their interfaces) so that they can interact. Bindings are communication paths than can be local, distributed, or secured. Component interfaces are kept separate from implementations, and bindings are exteriorized from components so as to support flexibility.

Systems such as Gaia [Chetan et al. 2005], Aura [Garlan et al. 2002], and Pcom [Becker et al. 2004] built their infrastructures upon the component model. Middlewares, such as Fractal [Bruneton 2004], used the component model to represent their provided functionalities. We briefly introduce these systems and highlight their component-based infrastructure. Mobile Gaia [Chetan et al. 2005] is a component-based middleware that integrates resources of various devices. The notion of ad-hoc pervasive computing in Gaia is a cluster of personal devices that can communicate and share resources among each other. The cluster is referred to as a personal active space. The user can program this cluster through a common interface. Mobile Gaia role is to provide services that discover devices that form the personal space, maintain the composition of the cluster, share resources among devices in the cluster and facilitate communication. It also provides an application framework to develop applications for the device collection. The application framework decomposes the application into smaller components that can run on different devices in this collection. The Gaia system [Roman et Campbell 2003] technology model is implemented as components. The deployment framework forms a container for these components. Each component in Mobile Gaia has its own life cycle and can be dynamically managed regardless of it model and location.

Aura [Garlan et al. 2002] provides user with an invisible halo of computing and information services that persists regardless of location. A personal Aura acts as a proxy for the mobile user it represents. Aura aim is to allow users to execute their tasks regardless their location, and Aura deals more with adaptation, replacement of services, rather than with composition and services' integration. Project Aura provides several pervasive applications adapted to both homes and offices. In Aura, user task is modeled as abstract components. Their implementations consist in wrapping existing services or applications to conform Aura API. The user intent, the abstract services and applications are described using XML-based markup formats. Assumption of common vocabulary tags and their corresponding interpretations is made.

Pcom [Becker et al. 2004], a component system for pervasive computing is a light-weight component system that offers application programmers a high-level programming abstraction which captures the dependencies between components using contracts. 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. Pcom technology model is also component-based. Components are atomic with respect to their distribution but can rely on local or remote components, resulting in a distributed application architecture. Pcom's components enclose contracts that describe their offered functionality and requirements regarding the platform and other components.

Fractal [Bruneton 2004] is a general-purpose software composition framework that supports component-based programming, including both: components (type) definition and configuration (instantiation), and runtime management including dynamic reconfiguration. The Fractal specification defines the component model and the Application Programming Interface (API), i.e. the Fractal programming model. The Fractal component model provides a homogeneous vision of software systems structure with a few but well defined concepts and exhibits distinguishing features such as recursion with sharing and reflection [Bruneton et al. 2002]. These features allow: A uniform management of both so-called business and technical components, a uniform management of resources (data, caches, pools, protocols, connections), activities (threads, transactions, processes) and domains (security, persistence), and finally a possible component-based approach throughout the software life-cycle: development, deployment, and production (runtime management). [Hoareau et Mahéo 2006] presents a method to make Fractal-based application ubiquitous, that is, to allow the services implemented by the application to be invoked on more than one host. This is mainly achieved thanks to a distribution scheme of composite components that can be duplicated on several hosts. The introduction of active interfaces to the model allows the application to perform in a degraded mode when disconnections occur. Some simple mechanisms have also been described to show how ubiquitous Fractal components can be tuned regarding network disconnections and how bindings are automatically reconfigured when network failures occur.

2.1.2 Communication-Based Infrastructure

Another approach consists in using functionalities that are more independent from each other and that carry out their integration and their communication in a generic way. A key objective should be to design the infrastructure for integration and evolution, and not to try and achieve a stable, definitive system. A communication-based infrastructure should be flexible enough to easily integrate heterogeneous devices but also to make it possible for these devices to evolve and for new devices to appear. Some projects such as Oxygen [MIT. Project Oxygen 2007] adopted this kind of infrastructure, other technologies such as Multi-Agent Models, Peer to Peer computing, and Grid computing rely on a communication-based infrastructure as a support to offer an integrated environment for users and applications.

Project Oxygen [MIT. Project Oxygen 2007] enables pervasive, human-centered computing through a combination of specific user and system technologies. Oxygen aims to enable pervasive, unobtrusive computing. To permit this, Oxygenated applications adapt their behavior according to the resources available in their environment. Oxygen technology models three layers, devices, networks and softwares. Project oxygen technology model is typically a communication based infrastructure. Applications developed under the Oxygen project, are written in a special communication-oriented language and middleware. The language manipulates at a high level of abstraction, components such as nodes, edges, messages, and actions. Nodes are just about anything that can be named and communicated via sockets. An edge is a directed connection between nodes. A message is an entity that flows along an edge. Actions make up the final component of the language. An action consists of a trigger and a consequence. There is a middleware system that executes the language.

Multi-Agent Models offer a communication based infrastructure for systems in pervasive environments. Agents are autonomous entities that perceive their environment and decide on their action according to their knowledge and goals. A multi-agent system is a distributed system of agents, which interact and cooperate in order to achieve global tasks. Agents technology is characterized by its language, negotiation and cooperation, personal and social behavior and intelligence. Agents are very often used for managing and integrating other technologies, such as component or services. Agents do not integrate but very often communicate and interact in order to achieve a specific goal, that can be an integration problem. Multi-agent systems enable complex interactions between entities such as flexible compositions [Vallée et al. 2007], using high level semantic languages. This feature seems essential in environments dealing with various, heterogeneous information from physical sensors, services or users preferences. Integration of such data is only possible at a higher level where all kind of information (about services context, etc.) is expressed semantically. In a multi-agent, autonomous entities with limited capabilities coordinate in order to achieve complex tasks. Emergent coordination and flexible organization patterns enable groups of agents to create and reconfigure application dynamically depending on conditions. Such patterns seem well adapted to dynamic integration of elementary functionalities in an open, dynamic environment [Vallée et al. 2005].

Peer-to-peer (P2P) computing covers a wide range of infrastructures, technologies and applications that share a single characteristic: they are designed to create networked applications in which every node (or deployed system) is in some sense equivalent to all others, and application functionality is created by potentially arbitrary interconnection between these peers. The consequent absence of the need for centralized server components to manage P2P systems makes them highly attractive in terms of robustness against failure, ease of deployment, scalability and maintenance. This communication infrastructure allows systems to integrate different technologies in pervasive environments [Frénot et al. 2003]. Indeed, it provides a connectivity to all the entities available on the devices. Grid computing is the high-performance computing infrastructure for supporting large-scale distributed scientific endeavor that has recently gained heightened and sustained interest from several communities. The Grid thus refers to an infrastructure that enables the integrated, collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations. The integration concept is more related to the integration of multiple independent computing clusters, in order to work together.

The communication-based infrastructures offer more a way of communication and interaction rather than a technology model for integration. This infrastructure is not really oriented to model a specific technology but rather to allow a better interaction between entities and devices populating the pervasive environments.

We adopt the SOA and investigate the service integration middleware in pervasive environments. We decompose these middleware into three categories: service transformation, service composition, and service adaptation. After a brief description of the major middleware of these domains, we give a classification that highlights the lacks in service integration middleware for the pervasive. We need to point out that we deliberately chose for conciseness not to consider the service integration problem outside the frame of pervasiveness. Many service transformation, composition, and adaptation middleware were designed and developed for distributed or mobile computing. The surveyed middleware are those respecting the requirements of pervasiveness.

2.2 Service Transformation Middleware

Service transformation middleware tackle the problem of heterogeneity brought by pervasiveness. If Web services technology provide seamless and automatic connections from one software application to another using standard protocols such as SOAP, WSDL, and UDDI, to discover and call a method in a software application – regardless of location or platform, these technologies fail to address problems such as management and composition for services other than Web services from early stage conception to system development process. Service transformation middleware propose to transform the different service technologies available in the vicinity into one common model to enable service interaction and communication. The area that tackles this problem is the Model-Driven Development [Mukerji et Miller 2003]. It proposes a software development methodology in which software are developed not by writing code directly in implementation languages, but by constructing high level models that can be transformed into code by automated transformation engines and code generators. The slogan of MDD is “Model one, generate anywhere”. The two middleware depicted in this section, Perv-ML [Munoz et al. 2004] and MIDAS [dCastro et al. 2006], repose on MDD principles.

2.2.1 Perv-ML: Pervasive Modelling Language

Pervasive Modelling Language (Perv-ML) [Munoz et al. 2004] is a language with the aim of providing the pervasive system with a set of constructs that allow to precisely describe the pervasive system itself. Perv-ML [Munoz et al. 2004] promotes the separation of roles where system developers can be categorized as analysts and architects. System analysts capture system requirements and describe the pervasive system at a high level of abstraction. Analysts use three diagrams (models) that constitute the analyst view. On the other hand, system architects specify what commercial off-the-shell devices and/or software systems realize system services. Architects build other three models that constitute what we call the architect view. This separation facilitates the reuse of the models defined in the analyst view by any other architects to build models for the architect view.

Perv-ML is built on the model driven development model, and thus follows the MDD techniques. Perv-ML proposes a PIM (Platform Independent Model) language to describe the system with high level constructs. It chooses a PSM (Platform Specific Model) reposing on the OSGi technology. It establishes a set of transformation rules defining how a PIM can be converted to a PSM. Finally, it generates the code from the PSM to develop the pervasive services. The proposed technique is described figure 2.1.

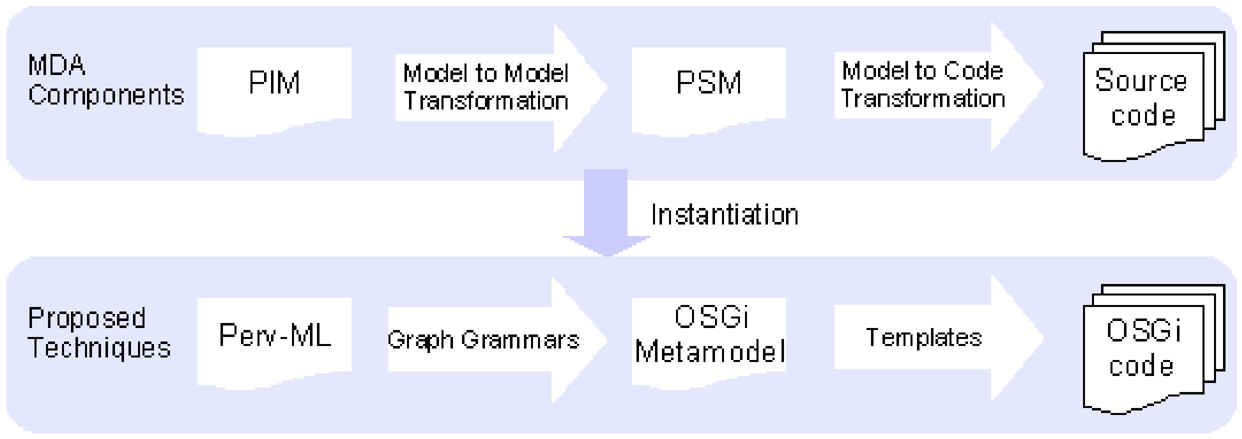


Figure 2.1: Perv-ML MDA

The definition of transformations between PIM and PSM involves jumping a wide gap between abstraction levels. [Munoz et al. 2004] are using graph transformations and graph grammars as the model transformation engine. From a mathematical point of view, a model can be seen as a graph where model elements are labeled nodes and the relationships between model elements are edges. In this way they can apply all the existing knowledge for defining graph transformations in order to achieve model transformations in the MDA context. Graph grammars have many advantages over other proposed techniques: a formal mathematical sound, algorithms for applying them and a graphical representation for defining intuitively transformations. Perv-ML proposes rules for model transformation from Perv-ML models to OSGi-based models. Every rule is composed by a Left Hand Side (LHS), that defines a pattern to be matched in the source graph, and a Right Hand Side that defines the replacement for the matched subgraph. For instance, when a Perv-ML `component` element is found it must be transformed into a `Bundle` element, and references to a `ServiceActivator` and `Manifest` elements have to be created and linked to the `Bundle`. In this simple rule example, the Perv-ML `component` element belongs to the PIM language description. The `Bundle`, `ServiceActivator` and `Manifest` elements are special to the OSGi specifications.

Following these rules, the system transforms an abstract definition of a service into a platform dependent one, and abstract model of services becomes real component pervasive services.

2.2.2 MIDAS: Model drIven methodology for the Development of web InformAtion Systems

MIDAS [dCastro et al. 2006] presents a model driven method for service composition modeling, which could solve the limitations of current service composition languages such as BPEL. MIDAS [dCastro et al. 2006] defines new modeling concepts and also new models in which these concepts are represented. Moreover, the method proposes a process which consists of the description of the tasks for the generation of each new model and which also defines the mapping rules between them. The modeling process starts identifying the services that will be offered to users; it allows MIDAS [dCastro et al. 2006] to obtain a platform-independent model, called the service composition model, which makes the mapping to a specific Web service technology easier. Within the context of this work, a service is defined as a complex functionality, offered by the system, that satisfies a specific need of the user, and the term Web services refers to the technology chosen to implement a service or a part of it.

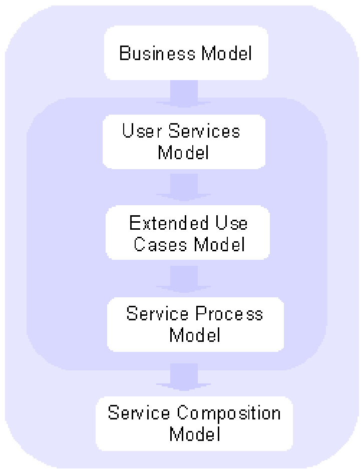


Figure 2.2: MIDAS concepts

The method for the service composition modeling defines a process, new models and mappings between them. the method introduces a new set of concepts necessary for the service composition modeling, which are represented on the different models proposed by the method. The new concepts and the relations between them are:

- Service is a complex functionality, offered by the system, which satisfies a specific need of the user. A service models how the systems should behave in response to a user's needs.
- Use service is a functionality required by the system to carry out a service. MIDAS model as use services all the functionalities needed for the execution of the services.

- Service process represents the execution flow of a service. Each service defines a service process, which is modeled through an activity diagram.
- Service activity is the specification of a behavior that is part of the execution flow of a service. The service activities represent the behavior of the basic use services in the activity diagram that models the service process.
- Activity operation declares an action that is supported by the service activity.
- Business collaborator is an organizational unit that carries out some activity operation, which is involved in the services offered as Web services.

The method for the service composition modeling (see Figure 2.2) includes the new models defined above. The input of the method is the business model, in which the user, business collaborators, and services are identified. The output is the service composition model, from which it is possible to make the transformation to the Web service technology. MIDAS [dCastro et al. 2006] defines three intermediate models to obtain the service composition model: the user services model, the extended use cases model and the service process model. The process comprises several steps, each one related to the generation of a different model (user services model, extended use cases model, service process model and service composition model). In this way, the process is defined as the set of tasks to be carried out in order to generate each model. MIDAS [dCastro et al. 2006] describes the transformation rules in natural language. To illustrate the process, MIDAS have presented a case study. Using the case study, MIDAS have explained the process, the way of identifying the new concepts for building models and also how to make the transformation between models.

2.3 Service Composition Middleware

Service composition allows the combination of multiple services into a single composite service, which may be achieved at design-time (static) or at run-time (dynamic). In current middleware and systems, dynamic service composition is very often associated with the realization of user tasks on the fly. Indeed, service composition can be a major key for the user-centrism paradigm by enabling the user to be at the heart of the realization of his daily tasks through the combination of relevant services available in the vicinity. If some service discovery mechanisms such as DLNA, which is based on UPnP, support service compositions, many dedicated service composition middleware exist. Service composition has been tackled in SOA in general such as in SWORD [Ponnekanti et Fox 2002] and is not specific to the pervasive environments. Software composition in general and service composition in particular were widely developed and used in web services technologies but also in middleware such as CORBA. For conciseness, in this section we will be interested only in service composition middleware adapted for pervasive environments. The three middleware depicted in this section (PERSE [Mokhtar 2007], SeGSeC [Fujii et Suda 2005], and Broker [Chakraborty et al. 2005]) are, as all other major current service composition middleware (SeSCo [Kalasapur et al. 2007], WebDG [Medjahed et al. 2003], eFlow [Casati et al. 2000], , and Contract-Based composition [Milanovic 2006]), goal-oriented as they dynamically compose services in response to a user task. For conciseness, we choose to detail only three service composition middleware.

2.3.1 PERSE: PERvasive SEmantic-aware Middleware

PERSE [Mokhtar 2007] proposes a semantic middleware, that deals with well known functionalities such as service discovery, registration and composition (cf. figure 3). This middleware provides a service model to support interoperability between heterogeneous both semantic and syntactic service description languages. The model further supports the formal specification of service conversations as finite state automata, which enables the automated reasoning about service behavior independently from the underlying conversation specification language. Hence, pervasive service conversations described with different service conversation languages can be integrated toward the realization of a user task. The model also supports the specification of service non-functional properties based on existing QoS models to meet the specific requirements of each pervasive application through the `QoS aware Composition` service.

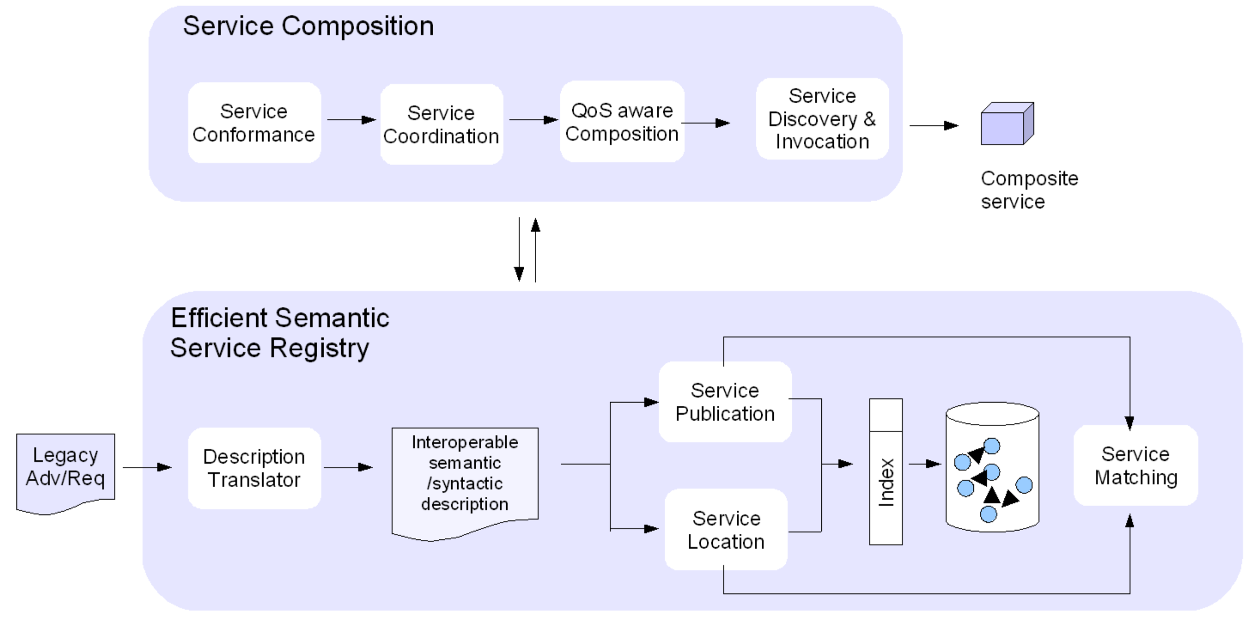


Figure 2.3: Semantic service middleware

PERSE [Mokhtar 2007] introduces the architecture of a semantic service registry for pervasive computing. This registry allows heterogeneous service capabilities to be registered and retrieved by translating their corresponding descriptions to a predefined service model through the `Description Translator`. Service discovery protocol interoperability requires the translation of service advertisements into a common service description language for enabling service matching and composition to be performed independently from the specific underlying languages. Once the translation done, the services can be published, stored, compared or composed depending on what is needed in the environment.

In PERSE [Mokhtar 2007] networked services of the pervasive computing environment and user tasks are modeled as conversations using OWL-S [Coalition 2003]. OWL-S, formerly DAML-S [Pagels 2005], is the acronym for web ontology language for Web services which aims at connecting the Web services world with the semantic web. This language describes Web services using three parts: service profile for advertising and discovering services; the process model, which gives a detailed description of a service's operation; and the grounding, which provides details on how to interoperate with a service, via messages. Within the task description, the capabilities are abstract, i.e., they do not refer to any specific networked service, as these capabilities have to be dynamically provided by the pervasive computing environment and may be realized by either atomic or composite processes of networked services. Indeed, this depends on the service implementation. The same capability can be developed as a single client/service interaction or as a sequence of client/service interactions.

In a composition process, PERSE [Mokhtar 2007] performs a semantic matching of interfaces - through the `Service Matching` - that leads to the selection of the set of services that may be useful during the composition. Then, PERSE [Mokhtar 2007] performs a conversation matching starting from the set of previously selected services, thus obtaining a conversation composition that behaves as the task's conversation. The matching is based on a mapping of OWL-S conversations to finite state automata. This mapping facilitates the conversation composition process, as it transforms this problem to an automaton equivalence issue. Once the list of sub-automata that behaves like the task automaton is produced, a last step consists in checking - through the `Service Conformance` and `Service Coordination` - whether the atomic conversation constraints, have been respected in each sub-automaton. After rejecting those sub-automata that don't verify the atomic conversation constraints, PERSE [Mokhtar 2007] selects arbitrarily one of the remainders, as they all behave as the user task. Using the sub automaton that has been selected, an executable description of the user task that includes references to existing environment's services is generated, and sent to the `Service Discovery & Invocation` that executes this description by invoking the appropriate service operations.

[Mokhtar 2007] have implemented a prototype of PERSE using Java 1.5. Selected legacy plugins have been developed for SLP using jSLP, UPnP [Cooperation 2000] using Cyberlink, and UDDI using jUDDI. The efficiency of PERSE has been tested and proved in the evaluation of the cost of semantic service matching, the time to organize the semantic service registry, the time to publish and locate a semantic service description as well as the comparison of the scalability of the registry compared with a WSDL service registry, and finally the processing time for service composition with and without the support of QoS.

2.3.2 SeGSeC: Semantic Graph-based Service Composition

SeGSeC [Fujii et Suda 2005] proposes an architecture that obtains the semantics of the requested service in an intuitive form (e.g. using a natural language), and dynamically composes the requested service based on its semantics. To compose a service based on its semantics, the proposed architecture supports semantic representation of services - through a component model named *Component Service Model with Semantics (CoSMoS)* - discovers services required for composition - through a middleware named *Component Runtime Environment (CoRE)* - and composes the requested service based on its semantics and the semantics of the discovered services - through a service composition mechanism named *Semantic Graph-Based Service Composition (SeGSeC)*.

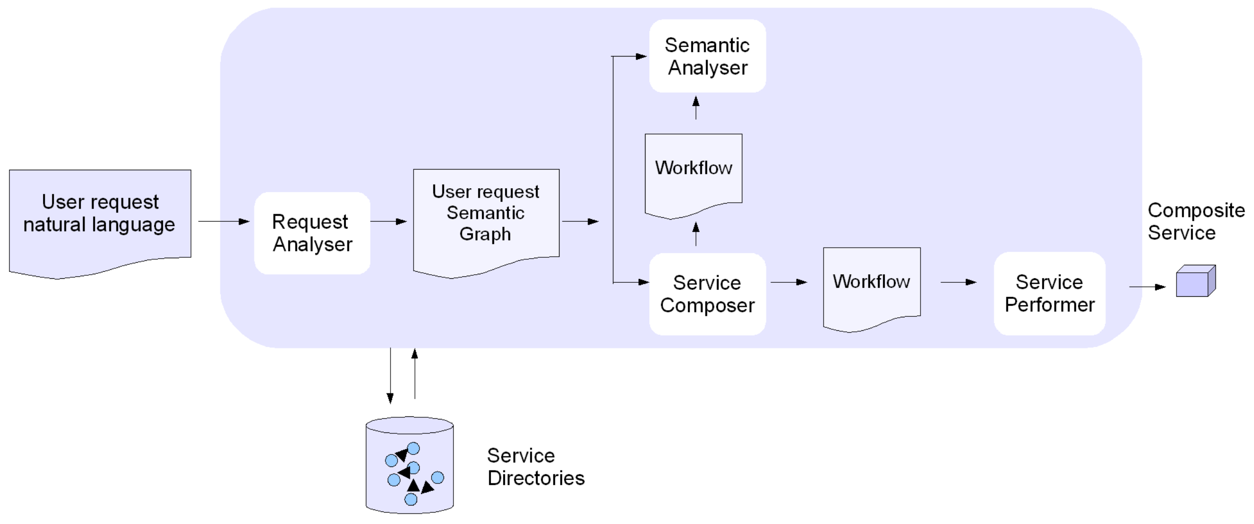


Figure 2.4: Modules in SeGSeC

SeGSeC [Fujii et Suda 2005] consists of four modules: `RequestAnalyzer`, `ServiceComposer`, `SemanticsAnalyzer`, and `ServicePerformer` (cf. figure 2.4). When a user requests a service in a natural language `RequestAnalyzer` parses the request into a CoSMoS semantic graph representation. In CoSMoS, a service is represented as a graph of nodes and links, where nodes represent data types, concepts, logics, or components, and links represent relationships between nodes. Then, `RequestAnalyzer` passes the semantic graph (i.e., the user request) to `ServiceComposer`.

Upon receiving the user request from `RequestAnalyzer`, `ServiceComposer` discovers services based on the user request. After discovering the services, `ServiceComposer` creates an execution path by finding other services that can be supplied as the inputs of the operation. Then, `ServiceComposer` computes all possible combinations of the services. `ServiceComposer` sorts the combinations based on their similarity values (number of common node in the combination and the user request) and compatibility rules and chooses the top one. `ServiceComposer` iterates the process an extends the execution path until all the operations in the execution path become executable. Then, `ServiceComposer` gives the execution path (in the form of semantic graph) and the user request (which is also a semantic graph) to `SemanticsAnalyzer` to check whether the semantics of the execution path is satisfied by the user request.

`SemanticsAnalyzer` applies the semantic matching rules onto the execution path in order to derive the semantics of the path. Those rules are designed such that they derive semantic facts from the given execution path, by adding proper links between concepts in the execution path based on the data flows (i.e. argument links) and the structure of components (i.e. operations and properties) defined in the execution path. In order to derive the semantic facts from the execution path and compare them with the user request, `SemanticsAnalyzer` interprets the links in the user request as goal statements and the links in the execution path as fact statements, and performs reasoning (such as forward chaining) to check if the goals (i.e. user request) can be derived by applying the rules onto the facts (execution path). If `SemanticsAnalyzer` could successfully conclude that the goals are derived by applying the rules, `SemanticsAnalyzer` notifies `ServiceComposer` that the given execution path satisfies the user request. In this case, `ServiceComposer` proceeds to the next step, service execution, to execute the path.

`ServicePerformer` module takes in charge invoking the Web services. After `ServiceComposer` concludes that the semantics of the execution path matches the user request, `ServiceComposer` passes the execution path to `ServicePerformer`. `ServicePerformer` then asks the user whether to execute the path or not. If the user agrees to execute, `ServicePerformer` executes the given execution path by accessing the services, i.e., by invoking operations and retrieving properties of services in the specified order, through specific access interface. If the user disagrees with the path, `ServicePerformer` asks `ServiceComposer` to continue the combination process to search for alternative paths.

SeGSec [Fujii et Suda 2005] was evaluated according to the number of services deployed and the time needed to discover, match and compose services. Another set of evaluations took not only the number of deployed services but especially the number of operations these services implement. Their results show that SeGSeC [Fujii et Suda 2005] performs efficiently when only a small number of services are deployed and that it scales to the number of services deployed if the discovery phase is done efficiently.

2.3.3 Broker Approach for Service Composition

Broker [Chakraborty et al. 2005] presents a distributed architecture and associated protocols for service composition in mobile environments that take into consideration mobility, dynamic changing service topology, and device resources. The composition protocols are based on distributed brokerage mechanisms and utilize a distributed service discovery process over ad-hoc network connectivity. The proposed architecture (cf. figure 2) is based on a composition manager, a device that manages the discovery, integration, and execution of a composite request. Two broker selection-based protocols - dynamic one and distributed one - are proposed in order to distribute the integration requests to the composition managers available in the environment. These protocols depend on device-specific potential value, taking into account services available on the devices, computation and energy resources and the service topology of the surrounding vicinity.

The broker-based approach [Chakraborty et al. 2005] proposes a *Description-level Service Flow (DSF)* which stands for a declarative description of a composite service or a composite request. A service is defined as any software component, data, or hardware resource on a device that is acces-

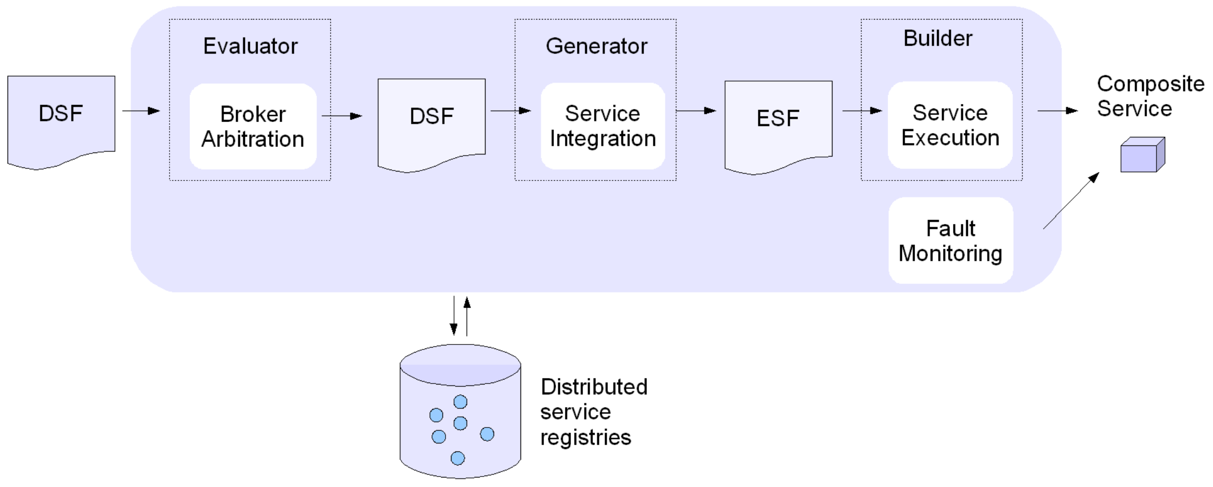


Figure 2.5: Service composition and management layer

sible by others. Services are located on devices connected to each other using ad-hoc networking protocols. [Chakraborty et al. 2005] uses an ontology DAML [Pagels 2005] to effectively describe services. Services are classified into a hierarchical group based on their functionalities. DAML-S [Pagels 2005] supplies Web services providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer interpretable forms. DAML-S markup of Web services will facilitate the automation of Web services tasks including automated Web services look up, execution, interoperation, composition and execution monitoring.

The Broker based composition proposed by [Chakraborty et al. 2005] uses DAML-S for specifying a composite service. The specification consists of a list of service descriptions along with the desired flow of execution that constitutes the composite service. This description is referred to as the *Execution-level Service Flow (ESF)*. *ESF* stands for a complete specification of the composite service with execution-level details required to invoke the services in the service provider.

The Broker based composition [Chakraborty et al. 2005] proposes two protocols for selecting an adequate broker for a given composition request, the dynamic broker selection and the distributed broker selection. This selection takes place in the `Broker Arbitration` (`Evaluator`) (cf. figure 2). The first protocol applies a broker arbitration that consists of controlled broadcasting of solicitation requests to devices in the nearby vicinity of the request source. The broker arbitration module in each device replies with information regarding its potential to be a composition manager for the request. The request source elects the best possible composition manager from the available devices and sends the *Description-level Service Flow (DSF)* to it. The elected composition manager on receiving the request increments its load level (number of executing composite requests) and starts processing it via the `Service Integration` (`Generator`) (cf. figure 2). The protocol evaluates the potential value of each node using formal methods evaluating local resources such as number of matching local atomic services, battery life, current processing load on the node, and the number of remote service advertisements cached by the device of the node. An equation calculates the potential of each nodes. And the best node is selected to be the broker for a given composition request. The second protocol relaxes the single composition manager execution policy imposed by the first protocol. It distributes the task of composing a service to multiple composition managers on an as-needed basis. And by this way the new protocol reduces the burden on the request source since it would not have to handle any more the faults of failure of the central composition manager.

[Chakraborty et al. 2005] broker based composition relies on a star-based execution pattern. The composition manager with its `Service Execution` (`Builder`) (cf. figure 2) coordinates the execution of the services in the order specified by the *Execution-level Service Flow*. The broker uses the underlying routing protocol to transmit results received from one service to another during the execution. The partial results flow from one device to another through the composition manager. A mesh-based execution pattern where the result would directly flow from one service to the next service is under study.

[Chakraborty et al. 2005] implemented a protocol as part of a distributed event-based mobile network simulator, to test the two proposed broker arbitration protocols and the composition efficiency. Simulation results show that their protocols - especially the distributed approach - exceed the usual centralized broker composition in terms of composition efficiency and broker arbitration efficiency.

2.4 Service Adaptation Middleware

Service adaptation middleware adapt the functional and non functional behavior of a service to meet the application needs. This adaptation is done at run-time (dynamic), as pervasive applications need to cope with unpredictable changes. While an adaptive behavior implies the capability of a middleware to run in a number of different configurations, these middleware also need to dynamically perceive the characteristics of the surrounding environments and that by being context aware. Adaptation to changes might happen [Kjaer 2007] in middleware such in Aura [Garlan et al. 2002], Carisma [Capra et al. 2003], Cortex [Sorensen et al. 2004], Carmen [Bellavista et al. 2003], and Madam [Floch 2006] or in the applications such in Cooltown [Debaty et al. 2003], Socam [Gu et al. 2004], Gaia [Chetan et al. 2005], MiddleWhere [Ranganathan et al. 2004], and MobiPADS [Chan et Chuang 2003] . For concisness, we only detail three of these middleware: Madam [Floch 2006], Carisma [Capra et al. 2003], and Socam [Gu et al. 2004].

2.4.1 MADAM: Mobility and ADaptation enAbling Middleware

A main idea in MADAM [Floch 2006] is to model MADAM Middleware as a Component framework, allowing middleware services to be composed in a flexible manner and thus supporting adaptability of the middleware. MADAM [Floch 2006] presents three components: the context model, the resource model, and the property model. The context model and the resource model are used to describe the environment where components are executing. The property model is used to describe the properties of the services offered or required by components. In MADAM, a component is defined as a unit of composition with contractually specified interfaces and explicit dependencies. A component type is associated to a set of port types that components of that type should implement. MADAM component architecture also defines the component plan, which maintains the association between a component type and a component, together with the component's context and resource dependencies expressed by property statements.

The component model, the context model, the resource model and the property model are used to build the architecture model of an application. To enable the MADAM Middleware to adapt an application, a run-time representation of the application component framework must be made available for the middleware. This representation is called the application architecture model, and it includes specification of application structure, variability and distribution aspects, and specifies the properties of each variant which can be derived from it. It is built and maintained dynamically from context and resource information and the set of component plans.

A plan describes a component as one alternative realization of a component type. A plan specifies the behavior of a component through its provided and required ports, the properties of the component, and the implementation resources for that component. A plan may refer to different kinds of implementation resources. Several implementation resources may be used to implement the same component type, but they may exhibit differences in offered and needed properties. In MADAM, properties of interest are QoS characteristics, device and network capabilities and user needs. The developer(s) may publish several plans associated with one component type, where each plan specifies the properties of the implementation resources referred to by that plan. The property specifications are used by the MADAM platform to distinguish and select between alternative implementations of a component type. The association of component type with plans is thus a central mechanism for describing variability in the architecture model.

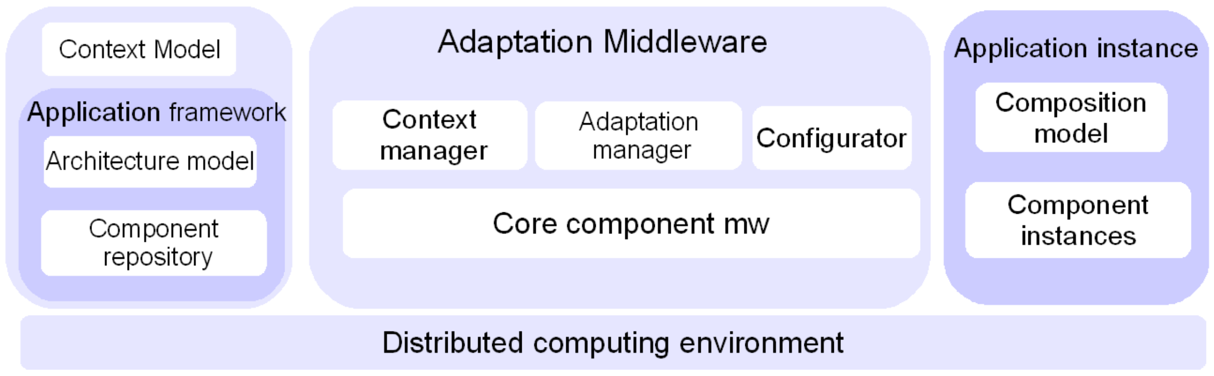


Figure 2.6: Architecture of Madam

The middleware architecture is shown in Figure 2.6. The core provides the fundamental platform independent services for the management of applications, components and component instances. The core relies on the basic mechanisms for instantiation, deployment and communication provided by the distributed computing environment. Beyond the core, every thing else is a component. The adaptation middleware includes three main services:

- The `Context manager` monitors the user and execution context for detection of relevant changes. `Context Manager` is responsible for sensing and capturing context information and changes, providing access to context information (pull) and notifying context changes (push) to the `Adaptation Manager`.
- The `Adaptation Manager` is responsible for reasoning on the impact of context changes on the application(s), and for planning and selecting the application variant or the device configuration that best fits the current context. As part of reasoning, the `Adaptation Manager` needs to assess the utility of these variants in the current context. The `Adaptation Manager` produces dynamically a model of the application variant that best fits the context.
- The `Configurator` is responsible for co-ordinating the initial instantiation of an application and the reconfiguration of an application or a device. When reconfiguring an application, the `Configurator` proceeds according to the configuration template for the variant selected by the `Adaptation Manager`.

In order to allow the development of adaptable applications developers need modeling language extensions and tools that enable them to specify the adaptation capabilities at a platform-independent level and tools that perform the transformation of the abstract adaptation model to platform-specific source code. MADAM defines the MADAM UML Profile which enhances and tailors the semantics of UML modeling elements with regard to the task of specifying the adaptation capabilities of self-adaptive applications. MADAM provides a complete MADAM Tool Chain covering the whole path from the platform-independent application adaptation model to the platform-specific source code. The tool chain incorporates UML modeling tools that support the MADAM UML Profile in order to allow the specification of the application adaptation model and incorporates the transformation tool MOFScript which is able to generate the corresponding source code as expected by the MADAM middleware.

2.4.2 CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications

CARISMA [Capra et al. 2003] is a middleware model that exploits reflection to enable context-aware interactions between mobile applications. It focuses on the adaptation aspect of a running application that uses services in a mobile environment. Applications are allowed to dynamically inspect middleware behavior (introspection), and also to dynamically change it (adaptation), by means of a meta-interface that enables run-time modification of the internal representation previously made explicit. The middleware is in charge of maintaining a valid representation of the execution context, by directly interacting with the underlying network operating system. By context, we mean everything that can influence the behavior of an application, from resources within the device, such as memory, battery power, screen size and processing power, to resources outside the physical device, such as bandwidth, network connection, location and other hosts within reach, to application-defined resources, such as user activity and mood.

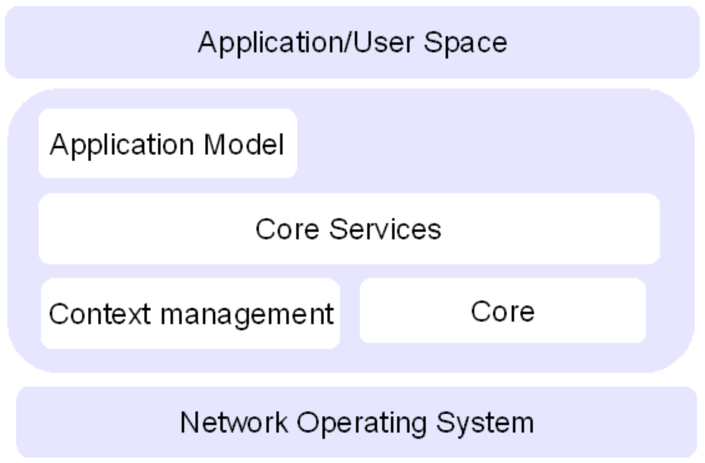


Figure 2.7: Architecture of Carisma

CARISMA [Capra et al. 2003] provides application engineers with an abstraction of the middleware as a customizable service provider. In particular, the behavior of the middleware with respect to a specific application is described as a set of associations between the services that the middleware customizes, the policies that can be applied to deliver the services, and the context configurations that must hold in order for a policy to be applied. Each time a service is invoked, the middleware consults the profile of the application that requests it, queries the status of the resources of interest to the application itself, as declared in the profile, and determines which policy can be applied in the current context, thus relieving the application from performing these steps. Through a reflective API , applications can dynamically inspect the content of their profile (i.e., the current configuration), and alter it by adding, deleting and updating the associations previously encoded.

CARISMA [Capra et al. 2003] also provides a mechanism based on sealed bit auctions to resolve conflicts or lets say to be able to choose between different possible policies for a same context. A formalization of this auction is given and tests are conducted to show that all these techniques - reflection, policies, auctions - respect the simplicity, dynamicity, and other properties imposed by pervasive environments.

The CARISMA architecture is made up of four main components, as shown in Figure 2.7 the `Core` component provides basic functionalities, such as support for asynchronous communication, service discovery, etc. The `Context Management` component is responsible for interacting with physical sensors and monitoring context changes. `Core Services` take care of answering service requests with application-defined QoS levels. The `Application Model` defines a standard framework to create and run context-aware applications on top of our middleware model.

2.4.3 SOCAM: Service-oriented Context-Aware Middleware

SOCAM [Gu et al. 2004] architecture aims to enable rapid prototyping of context-aware services in pervasive computing environments. The middleware converts various physical spaces where contexts are acquired from a semantic space where contexts can be easily shared and accessed by context-aware services. SOCAM consists of `Context Providers`, `Context Interpreter`, `Context Database`, `Service Location Service` and `Context-aware Mobile Services` as shown figure 2.8. The SOCAM middleware components are designed as independent service components which may be distributed over heterogeneous networks and can interact with each other.

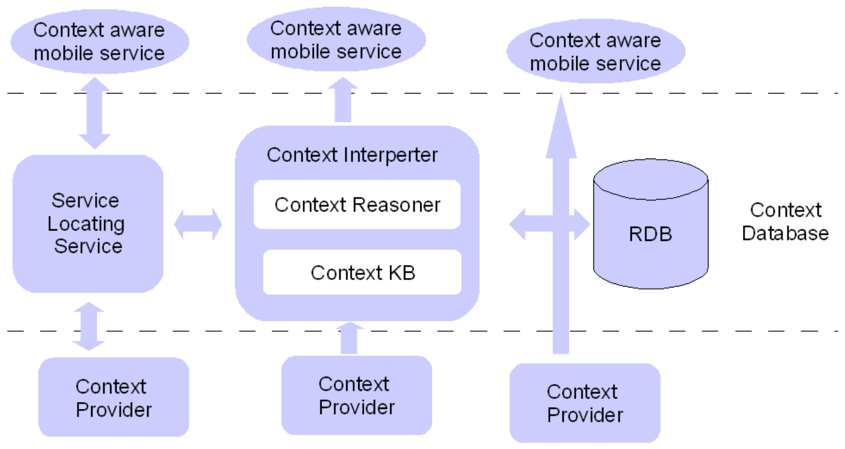


Figure 2.8: Architecture of Socam

`Context Providers` provide context abstraction to separate the low-level context sensing from the high-level context manipulation. Each context provider needs to be registered into a service registry by using the Service Locating Service mechanism and can be discovered by others.

The `Context Interpreter` also acts as a context provider as it provides high-level contexts by interpreting low-level contexts. It consists of a context reasoner and a context KB. The context reasoner has the functionality of providing deduced contexts based on direct contexts, resolving context conflicts and maintaining the consistency of the context Knowledge Base (KB). The Context KB provides a set of API's for other service components to query, add, delete or modify context knowledge. The Context KB contains: context ontologies in a sub-domain and their instances. These instances may be specified by users in case of defined contexts or acquired from various context providers in case of sensed contexts.

The `Service Locating Service` allows user, agents and applications to locate different context providers. The main features include scalability, dynamics and multiple matching.

`Context-aware mobile services` are applications and services that make use of different level of contexts and adapt the way they behave according to the current contexts. By querying the service registry provided by the Service Locating Service, we are able to locate all the context providers which provide a set of interested contexts.

To construct context-aware mobile services, a common way is to specify actions that are triggered by a set of rules whenever the current context changes. In the SOCAM, service developers can easily write pre-defined rules and specify what methods to be invoked when a condition becomes true. All the rules will be saved in a file and pre-loaded into the Context Reasoner.

2.5 Classification and Discussion

We classify the service middleware - PERSE [Mokhtar 2007], SeGSeC [Fujii et Suda 2005], Broker [Chakraborty et al. 2005], Carisma [Capra et al. 2003], MADAM [Floch 2006], SO-CAM [Gu et al. 2004], Perv-ML [Munoz et al. 2004], and MIDAS [dCastro et al. 2006] under three requirements.

The chosen requirements are the unified vision, QoS management and spontaneity of the integration.

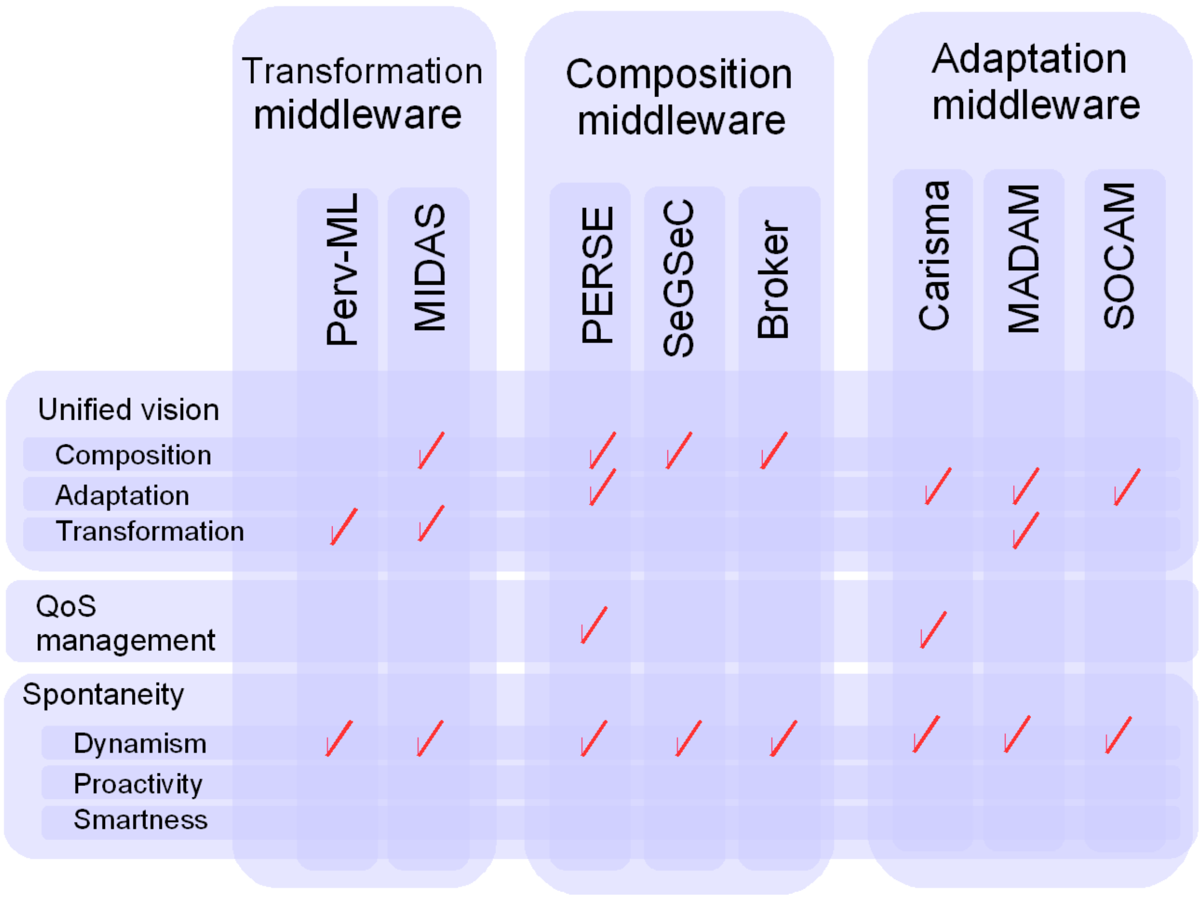
Unified vision *The service integration as we define it, is composed of three essential actions: transformation, composition, and adaptation. These three actions are all needed in order to offer a service integration middleware. If the listed middleware are all specialized in one specific task, some of them can do more than one action. We classify the six above middleware to distinguish the ones that do a complete or partial service integration as we define it.*

QoS management *QoS represents the set of those qualitative and quantitative characteristics (QoS characteristics) of an application necessary to achieve the required functionality from the application. QoS management is essential in dynamic environments, where connectivity is very variable. A pervasive middleware for service integration need to take the non functional parameters of applications, and devices into consideration in order to provide viable and flexible integration plans and composite services. The integration execution need to rely on these parameters in order to take place in the best conditions.*

Spontaneity *concerns the ability of a pervasive middleware to integrate services independently of user and application requests. The middleware spontaneously transforms all the available services in the vicinity into a common model, composes services that are compatible together producing a new composite service into the environment, and adapts the services to the all possible changes. Spontaneous service integration is an interesting feature in pervasive environments, as services meet when the user encounter, and interesting composite service can be generated from these meetings, even though not required at that moment by users. To evaluate the spontaneity of a service integration middleware, we evaluate the three following properties: proactivity, dynamism and smartness.*

Our classification (cf. figure 2.9) points out the limitations of the current integration service middleware concerning the three requirements we defined.

PERSE [Mokhtar 2007] middleware proposes to resolve user tasks by seeking for the desired functionality in the environment. For that it first translates the diverse service descriptions into one common model and then composes them to provide the functionality. It adapts this service composition result to the changing nature of the environment, and if many service compositions for a same request are possible, it chooses the best one depending on their QoS properties. If composition mechanisms are well elaborated in PERSE. The transformation and adaptation model as we define them are not really addressed. Interoperability is achieved in PERSE through the translation of



	Transformation middleware		Composition middleware			Adaptation middleware		
	Perv-ML	MIDAS	PERSE	SeGSeC	Broker	Carisma	MADAM	SOCAM
Unified vision								
Composition		✓	✓	✓	✓			
Adaptation			✓			✓	✓	✓
Transformation	✓	✓					✓	
QoS management			✓			✓		
Spontaneity								
Dynamism	✓	✓	✓	✓	✓	✓	✓	✓
Proactivity								
Smartness								

Figure 2.9: Service middleware classification

heterogeneous service advertisements to a common language. PERSE does not define a PIM model and transformation rules to pass from the PIM to some PSMs. The service technology in PERSE is the Web services technology and the translation mechanisms are especially used when discovering services. The adaptation concerns the composition itself and not any service in general evolving in the environment. Service composition can be adapted to changes in the environment. For that it is sufficient to replace a certain sequence of services by another. PERSE describes the QoS of services and reposes upon these when composing services together. Services that return compatible QoS are composed together. PERSE is clearly goal-oriented as it aims at fulfilling the users need. Spontaneous service composition, without the user intervention is not considered.

SeGSeC [Fujii et Suda 2005] proposes a mechanism to compose Web services based on a semantic description of these latter. SeGSeC does not propose service integration mechanisms as it does not deal with the service transformation or service adaptation. The service technology used is the Web services. By using Web services, SeGSeC argues that it fulfills the interoperability requirement. The composition is based on a semantic matching of the functional parts of the services.

A module is responsible for checking the validity of the composition. The non functional part of services, such as QoS are not taken into account. SeGSeC is also reactive as it fulfills user requirements and comes back to users once a valid composition path is found for validation. Spontaneous service integration is not considered at all.

Broker [Chakraborty et al. 2005] proposes dynamic and distributed protocols for service composition in mobile environments. It does not deal with service transformation aspect as it uses an ontology DAML [Pagels 2005] to effectively describe services. Adaptation aspects are only taken into account when deciding which manager is responsible of a service composition depending on the device it executes on. The QoS management of services is not taken into account. Although service composition is distributed over devices that are hierarchized according to their resource capacities, no real QoS management mechanisms are taken into account when composing services together. The service composition is initiated upon user requests and is goal-oriented and not pro-actively initiated by the environment.

MADAM [Floch 2006] is a service adaptation middleware for pervasive environments. It considers application as being the composition and instantiation of component services, which are possibly distributed on different hosts and try to adapt these applications to the environment changes. MADAM does not provide mechanisms for composing components. It rather adapts the already existing service composition seen as applications. Adaptation is needed when there is a mismatch between the quality of the services provided by the application and the user needs, or between the application needs and the context. The QoS provided by a component service is described and quantified using the `property` model: each QoS characteristic is represented by a property. MADAM relies over the MDA concept to be independent of any technology. It therefore proposes a service transformation middleware beside its service adaptation middleware. Adaptation is defined as the activity of adjusting to varying contexts. It is a response to an action and the spontaneity is not considered.

Carisma [Capra et al. 2003] offers application engineers an abstraction of middleware as a dynamically customizable service provider, where each service can be delivered using different policies when requested in different contexts. Carisma is an adaptable middleware and does not offer mechanism to compose services. It does not deal with the service transformation issues as it considers a common component model. The QoS management is verified in Carisma as in any adaptation middleware. Carisma proposes an auction based mechanism in order to choose between the different possible QoS offered by the services of the application. Spontaneity is not addressed as Carisma allows applications to adapt to the context, by letting them define their policies and strategies.

SOCAM [Gu et al. 2004] provides an architecture for application adaptation based on specific rules that are triggered upon certain events related to context changes. The service composition and transformation aspects are not tackled by this architecture. There is no unified vision for the service integration. The proposed adaptation responds to context changes and is not proactive and spontaneous. SOCAM supports rules for specifying which methods should be invoked on events. The rules are predefined and loaded into the context reasoner. There are no mechanisms for QoS management.

perv-ML [Munoz et al. 2004] and MIDAS [dCastro et al. 2006] are both service transformation middleware that rely over the MDA principles. They define a PIM model for the pervasive services and a set of rules to transform their PIM model to a PSM. In perv-ML the chosen PSM is OSGi based platform, and in MIDAS the PSM is the Web services technology model. Both middleware deal only with the functional parts of services and do not deal with the QoS. In MIDAS the service transformation is used as a tool for service composition. Service composition is defined as a PIM model before being mapped to a PSM model in a BPEL format. In both middleware, spontaneity is not considered.

After analyzing these middleware, a need for a service integration middleware model that does the service transformation, composition, and adaptation appears. If some of the current approaches deal with one or two aspects, no unified vision that respects the QoS of services is provided. Even more, no middleware proposes solution for spontaneous integration of services in a proactive and smart way, without the user intervention. In the next chapter, we introduce a generic model of a service integration middleware and propose our own service integration middleware that provides a spontaneous service integration with non-functional QoS management.

Chapter 3

Spontaneous Service Integration Middleware

In this chapter, we detail our contributions in service integration for the pervasive environment. Our goal is to provide a service integration middleware that responds to the challenging requirements defined in the previous chapter. First, We introduce a unified vision for the service integration by defining a novel and intuitive way to model service integration middleware: the SIM model. To motivate and proof the genericity of this model, we show that all current service integration middleware - and not only those presented in the state of the art - can be described using this model. Then, we introduce our own `MySIM` integration middleware, explain its general architecture and show how it respects the three requirements defined in the previous chapter: the unified vision, the non-functional QoS management, and the spontaneity of service integration. Finally, we delve deeper into each of `MySIM` modules to explain our spontaneous service integration that takes into account non-functional QoS management in pervasive environments.

3.1 A Unified Vision for Service Integration Middleware

We define a service integration middleware as a framework providing tools and techniques for integrating services. If the integration problem is naturally decomposable into three subproblems that are the transformation, the adaptation, and the composition, we try to adopt another decomposition of the integration problem that unify the vision of the three subproblems defined above. Several studies [Yang et al. 2005, Rao et Su 2004] already split the service composition process problem into several fundamental problems. They adopt a runtime vision for the service composition and split this latter into several phases from the composition specification till the composition execution. Based on these approaches, we apply this decomposition principle not only to service composition as they do but to service integration in general as we define it.

We define a service integration middleware model, the SIM model, as an abstract layer, general enough to describe all existing service integration middleware - transformation, adaptation and composition middleware. The SIM model is at a high-level of abstraction, without considering a particular service technology, language, platform or algorithm used in the integration process. We show that this model is general enough to describe all the existing service integration middleware and platforms. Then, we introduce and define our integration middleware, `MySIM`, by depicting its main functionalities through the various services it contains.

3.1.1 A Service Integration Middleware Model: the SIM Model

The SIM is modeled by four modules (cf. figure 3.1): the `Translator`, the `Generator`, the `Evaluator`, and the `Builder`. As an entry point of the SIM are the services of different technologies available in the environment. These services are potential targets for integration. The services are modeled differently as they can be provided from different technologies platforms. At the end point of the SIM are the resulting integrated services (via transformation, composition, or adaptation). In between, the process of service integration passes through four phases of integration, that corresponds to the three functional problems: transformation, adaptation, and composition. In the following we explain the four modules of the SIM model. We depict how the current service transformation, composition, and adaptation middleware implement these modules.

1. The services available in the environment are implemented and described with diverse languages or techniques as they are provided from different sources. These services technologies and services descriptions are transformed to a generic middleware comprehensible

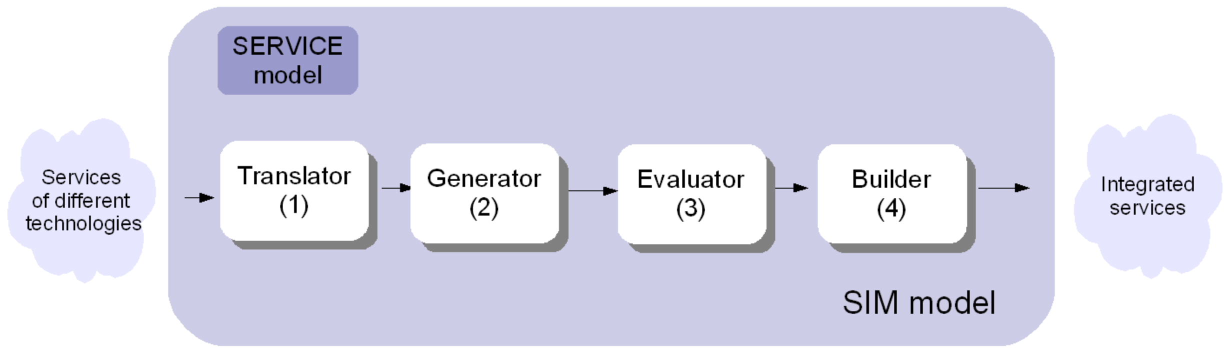


Figure 3.1: Service Integration Middleware model

language in order to be used by the middleware framework. The `Translator` module is responsible of describing the services available in the environment, the functionalities requested by the applications and users in a system comprehensible language, and for that it transforms the diverse existing technologies that are platform-dependent into one service model independent from the platforms, the SERVICE model (figure 3.1). In most of the service composition middleware, we find this `Translator` module even if provided in various forms. Some research, such as PERSE [Mokhtar 2007], provide a translation mechanism of the available services technologies and services descriptions into one model. Others, such as SELF-SERV [Constantinescu et al. 2004], propose a wrapper to provide a uniform access interface to services. [Fujii et Suda 2005] allows users to express their demands in a natural language transformed into a semantic machine-understandable language. These middleware usually realize transformation from one model to another or from one technology to another. The technologies are predefined in advance and usually consist of the legacy ones. If new technologies model appear in the environment, the `Translator` module will need to be expanded to take these technologies into consideration. Another family of research do not provide `Translator` module as they use common model to describe all the services of the environment. [Sirin et al. 2003, Chakraborty et al. 2005] that deal with web services composition use common description languages such as DAML-S [Pagels 2005] - recently called OWL-S [Coalition 2003] - for describing atomic services, composed services and users queries, [Constantinescu et al. 2004, Medjahed et al. 2003] define their own abstract description model for a service and suppose that all services in the environment are described using this model. Other approaches [Hashemian et Mavaddat 2006, Casati et al. 2000, Kalasapur et al. 2007] use graphs as a common infrastructure representing services. All the available services, provided by different technologies, are modeled in graphs. By this way, services can be compared and matched even if originally they are provided in different technologies.

2. Once transformed into a common model, the SERVICE model, the services are sent to the

Generator in order to be integrated (adapted and/or composed). The Generator will try to provide new functionalities by composing the available services, or to adapt the execution of services and applications to the environment changes. It tries to generate one or several composition or adaptation plans with the services available in the vicinity. It is quite common to have several integration plans, as the number of available functionalities in pervasive environment is always in expansion. The Generator module is the one responsible of elaborating all possible plans for adapting and composing applications and services. A plan specifies both the services to integrate, the way to combine them, and to adapt them in order to accomplish a required functionality. In the literature, two main approaches were widely adopted to model this planning: the graph-based description approach and the language composition one. These two approaches model the integration process in a graph or describe the integration process using a specific language. The following composition oriented middleware: Pcom [Becker et al. 2004], eFlow [Casati et al. 2000], SeSCo [Kalasapur et al. 2005], and SeGSeC [Fujii et Suda 2005] use graphs to construct their composition plans in order to fulfill a user task. Others such as PERSE [Mokhtar 2007] model their networked services of the pervasive computing environment and user tasks as conversations using extended OWL-S [Coalition 2003]. Number of languages have been proposed in the literature to describe data structure in general and functionalities offered by devices in particular. If some languages are widely used, such as XML, and generic for multiple uses others are more specific to certain tasks as service composition, orchestration or choreography such as Business Process Execution Language (BPEL4WS or BPEL) [Juric et al. 2006], and OWL-S [Coalition 2003].

3. The Evaluator relies on QoS criteria or applications and users profiles [Capra et al. 2003] to choose the most suitable and realizable integration plan for a given situation. This selection is done from all the plans provided by the Generator. In pervasive environments, this evaluation depends strongly on many criteria like the context, the technology model, the quality of the network, the services QoS, and so on. The evaluation needs to be dynamic, proactive, and adaptable as changes may occur unpredictably and at any time. Two main approaches are commonly used: the rule-based planning and the formal methods approach. Many systems such as eFlow [Casati et al. 2000], SOCAM [Gu et al. 2004] and SWORD [Ponnekanti et Fox 2002] employ rules to evaluate whether a given integration plan is appropriate or not in the actual context. If rules were commonly used as an evaluation approach, their use lacks of dynamism proper to pervasive environments. A major problem of the evaluation approach is namely the lack of dynamic tools to verify the correctness - functional and non functional aspects - of the service integration plan. This aspect is at the main advantage of what most formal methods offer. The nowadays most popular and advanced technique to evaluate a given integration plan is the evaluation by formal methods (like Petri nets and process algebras like the π-calculus). Many middleware such as Broker [Chakraborty et al. 2005], SeSCo [Kalasapur et al. 2007], PERSE [Mokhtar 2007], and WebDG [Medjahed et al. 2003] employed formal methods with mathematical proofs to evaluate a given plan by giving it a value. Automata or labeled transition systems are a well-known model underlying formal system specifications. On the other hand, Petri nets

are a framework to model concurrent systems. Their main attraction is the natural way of identifying basic aspects of concurrent systems, both mathematically and conceptually. Petri nets are very commonly merged with composition languages such as BPEL [Juric et al. 2006] and OWL-S [Coalition 2003].

4. The `Builder` executes the selected integration plans and produces several implementations corresponding to the required integrated services. It can apply a range of techniques to realize the effective service integration. These techniques depend strongly on the service technology model we are composing or adapting and on the context we are evolving in. Several transformational rules that maps a PIM (platform independent model) to PSMs (platform specific model) are provided by the `Builder` in order to implement the resulting service integration in a given technology model. Service transformation middleware such as Perv-ML [Munoz et al. 2004] and MIDAS [dCastro et al. 2006] provide the building blocks of this functional aspect. Once the integrated services available, they are deployed in the environment for a possible use. In the literature, we distinguish different kinds of builders provided by the service integration middleware. Some builders are very basic and use only simple invocation in sequence to a list of services. Others provide complex discovery protocols adapted to the heterogeneous nature of the pervasive environments. Finally some systems propose solutions not only located in the middleware layer but especially in the networking one. Many systems use the well known and well defined services discovery protocols such as UPnP [Cooperation 2000] and Jini [Kumaran 2002]. Others provide their own services discovery mechanisms. PERSE [Mokhtar 2007] reposes over MUSDAC [Raverdy et al. 2006] (MUlti-protocol Service Discovery and ACcess) middleware. MUSDAC enables explicit translation of service discovery protocol messages. In this middleware, the interoperability layer is located on top of the existing service discovery protocols, and provides an explicit discovery API to service requesters. Once the services discovered, as most systems use web services, the service invocation model is a simple call using the SOAP protocol over the web through an URL to the chosen web services.

As shown above, the SIM model is generic enough to provide generic functional modules that describe the existing service integration middleware. In the following, we define our integration middleware: `MySIM` middleware. `MySIM` is an instanciation of the `SIM` model that also respects and fulfills the two other pervasive requirements - QoS management and spontaneity - left behind by most of the current service integration middleware.

3.1.2 An Instantiation of the SIM Model: the `MySIM` Middleware

In this section, we describe our instantiation of the SIM model: the `MySIM` middleware. We decompose the `MySIM` into several services, each of them corresponding to a specific functionality and belonging to certain modules of the `SIM` modules. We distinguish the following services: the `Translator Service`, the `Generator Service`, the `Decision Service`, the `QoS Service`, the `Builder Service`, and the `Registry Service`.

3.1.2.1 `MySIM` Services Descriptions

Mapping `MySIM` to the SIM model, we gather our services under the four SIM modules (figure 3.2). The SIM `Translator` module is our `Translator Service`. The SIM `Generator` module is our `Generator Service`. The SIM `Evaluator` module is composed of several services: `Decision Service`, `QoS Service`. The SIM `Builder` module is the `Builder Service` and `Registry Service`. As shown figure 3.2, our `MySIM` provides all the modules defined by the SIM model.

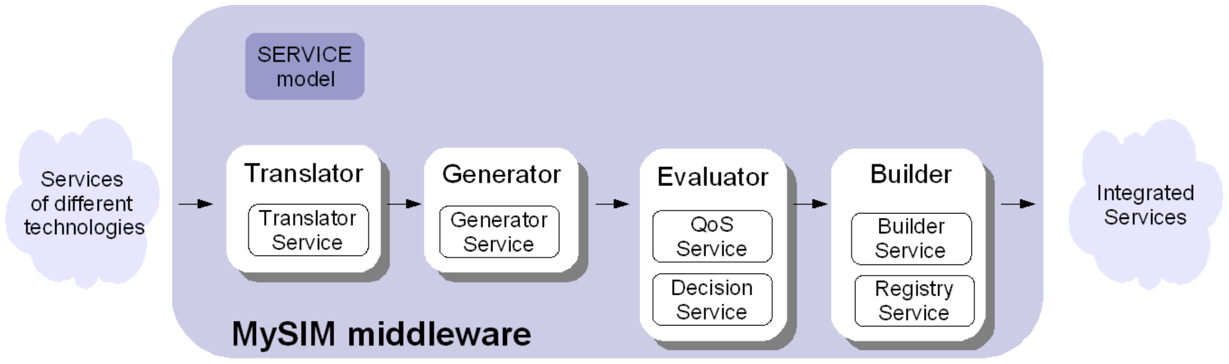


Figure 3.2: `MySIM` middleware following the SIM model

Each of these services has a specific task. The two services `Translator Service` and `Builder Service` are responsible of the technical issues of the integration. They provide, each at their levels, techniques allowing the service transformation, and the service integration implementation and generation. The `Generator Service` deals with the functional aspect of services essential for matching, comparing, and composing services. The `QoS Service` is responsible of the non-functional QoS properties of services taking part in an integration. The `Decision Service` is responsible of the spontaneity of the service integration and the adaptation issues. `MySIM` reposes over a generic SERVICE model defined in section 3.2.1. We briefly explain each of `MySIM` services:

- The `Translator Service` translates the diverse technologies offering the functionalities available in the environment into one generic SERVICE model. As a common representation is more than necessary in order to compare, match, adapt, and combine the functionalities. It is unrealistic to suppose that all functionalities of a pervasive environment are described in a same language or programmed under the same platform. Translators are needed for every technology model available in the environment. In the section 3.2.1, we define our platform independent SERVICE model. In chapter 4, we detail the chosen platform dependent service technology and detail the transformation rules that allow to transform from this model to our generic SERVICE model and vice versa.

- The `Generator Service` generates all possible combinations between services based on functional service relations such as functional equivalence and composition relations. Several abstract combinations are elaborated depending on the environmental available services, at a given time. In our middleware, service combinations and relations are done upon syntactic - signature - and semantic service matching. For each service in the environment, the `Generator Service` can find all the services available in the environment and that are syntactically or semantically equivalent. This notion of equivalence is based on the functional part of a service. The `Generator Service` can also find, for each service, all the services that respond to a composition relation, syntactic or semantic, between the functional parts of services. The services relations are described in sections 3.2.2.1 and 3.2.3.1

- The `QoS Service` is responsible of evaluating the previous equivalence or composition combinations by analyzing the services non-functional QoS. If the `Generator Service` indicates the possible relations and combinations between services, it does not pay attention to the services non-functional QoS properties. The `QoS Service` adds to every abstract services equivalence or composition compatibility the best non-functional QoS properties it can have, depending on the non-functional properties of the services involved in the integration. We distinguish two important cases for dealing with the non-functional QoS properties of services. The service composition issue and the service adaptation one. In service composition, combining functionalities without paying attention to the services non-functional QoS can lead to non executable service composition. `QoS Service` need to check for every service composition combinations, that the non-functional properties of services are respected. In a service adaptation, a service is replaced by another one, providing equivalent functionalities. Two services can be functionally equivalent but provide different non-functional QoS properties. The non-functional QoS services relations are described in sections 3.2.2.2 and 3.2.3.2.

- The `Decision Service` is responsible of the initialization of the spontaneous service integration. It is based on an event-based mechanism of services appearance and disappearance. It uses the `Generator Service` and the `QoS Service` to construct abstract possible equivalence relations or composition compatibilities between services. It decides which abstract combinations to implement and execute. Our `MySIM` in pervasive environment provides a spontaneous service integration: The environment need to be extended spontaneously and automatically with services, corresponding to the integration between already available services and others arriving in the environment. For a spontaneous service composition, the `Decision Service` needs to ask the `Generator Service` and the `QoS Service` to provide all the possible service composition, even if these compositions were not initially required by users. Nevertheless, some conditions, defined and verified by the `Decision Service` and executed by the `Generator Service` and the `QoS Service` modules need to be applied in order to allow a transparent use and access to these new implemented services. For a spontaneous service adaptation, the `Decision Service` needs to ask the `Generator Service` and the `QoS Service` to provide all the possible service equivalences. The `Decision Service` chooses from all these possible available services, atomic or composed ones the most appropriate ones depending on the context, the applications needs, and services QoS. In a changing environment, such as pervasive environments, decisions also need to be taken to cope with changes that may affect services. The `Decision Service` is thus responsible of all the adaptation issue such as redirection to another services, and so on.

- The `Builder Service` implements the combinations provided by the `Generator Service` and the `QoS Service` and approved by the `Decision Service`. The service integration is technically realized. The `Builder Service` creates new functionalities respecting our SERVICE model (defined in section 3.2.1) and directly implements these services in the chosen technology model. These newly implemented and generated services will continue to exist even after that a user or application has finished using them. If many integration middleware propose to integrate services on the fly, as a runtime workflow where services are chained at execution time, few implement and generate at runtime new services as independent components. The `Builder Service` proposes several techniques to realize the real integration of services. Whether the integration is a composition, or an adaptation the techniques are different. For the service composition the `Builder Service` proposes the composition by services replication or the composition by services redirection. Redirection consists in creating services that redirect every call to the service to a set of chained services. Replication consists in creating services that replicate within their implementations the implementations of other services in order to be independent of them. For a service adaptation, the `Builder Service` proposes a facade service that adapt the services execution to the real services implementations available in the environment. All these techniques are thoroughly explained in sections 3.3.2.2 and 3.3.3.2.

- The `Registry Service` registers the interfaces of the newly integrated services in the environment and monitors these services as they are very often dependent on the services they integrate. They are also dependent on the employed integration technique. The `Registry Service` checks periodically if these services execute correctly. It can if needed suspend, stop, and start the services. Accessing these services can sometimes be impossible as one of the services involved in the integration can be unavailable. In that case, the new service does not execute properly and the `Registry Service` is quickly aware of this change. It notifies the `Decision Service` of this failure. This latter is responsible of the adaptation issues.

3.1.2.2 `MySIM` Services Interactions

We highlight in the following the role that each `MySIM` services plays depending on the nature of the service integration. For a service transformation, the services technologies are transformed into our SERVICE model by the `Translator Service`. Then, The `Builder Service` implements these services into the chosen technology. Finally the `Register Service` registers these new services in the environment (cf. figure 3.3). In our implementation chapter, we explain our transformation rules that map from the OSGi service model to our SERVICE model defined in section 3.2. We also explain the implementation and generation of the newly integrated services in OSGi.

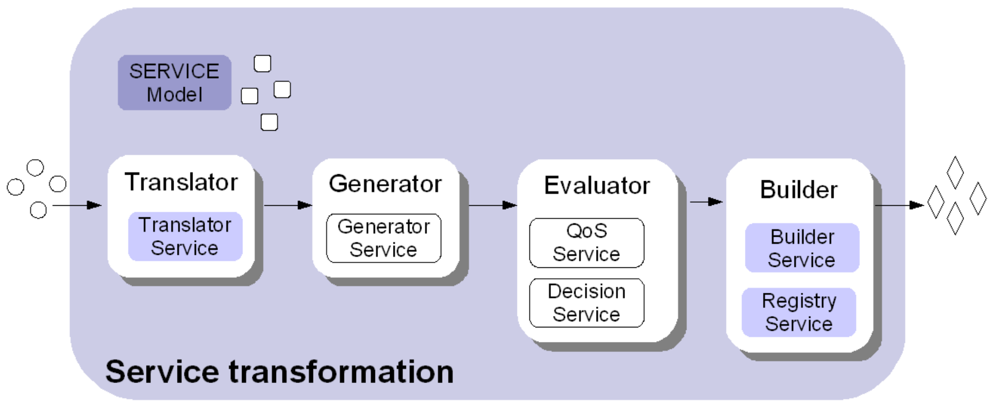


Figure 3.3: Service transformation by `MySIM` middleware

In service composition, the `Generator Service` generates all the possible abstract composition services from the set of available services. We suppose that services are represented in our SERVICE model and if not a previous transformation is done. Once the composition plans generated the `QoS service` selects from all these plans the possible composition plans and gives each potential composition service the adequate QoS in order to produce operable and executable composition services. Finally the `Builder Service` implements the new composed services which are registered via the `Registry Service` into the environment. The Decision Service has a control role over the functional quality of the composition and is responsible of starting and stopping the composition process (cf. figure 3.4).

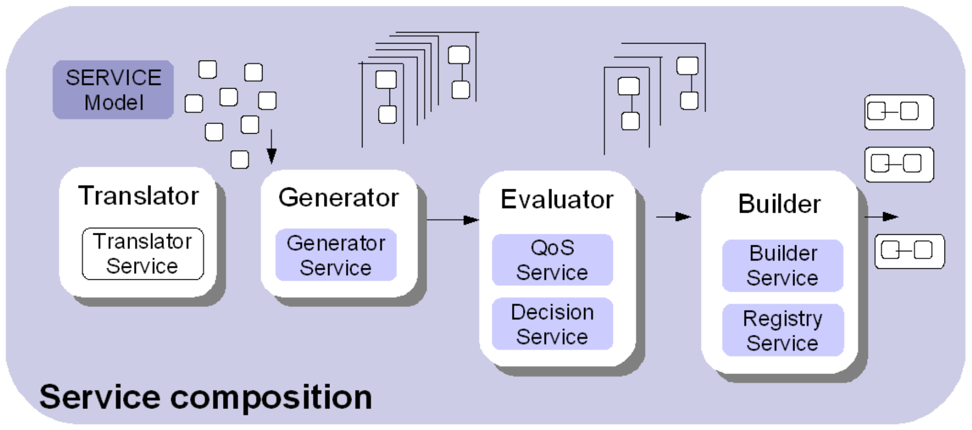


Figure 3.4: Service composition by `MySIM` middleware

In service adaptation, a new service may appear in the environment and be more appropriate for an application, or a service may be unavailable for multiple reasons and need to be replaced. The `Registry Service` notifies the `Decision Service` of such appearance or unavailability. The `Decision Service` will have the task to analyze if the new service fits better the applications requirements or to replace the disappearing service by a functional equivalent one that can be atomic or composed (cf. figure 3.5). The `Decision Service` uses both the `Generator Service` and the `QoS Service` to compute the functional and non-functional equivalences between services.

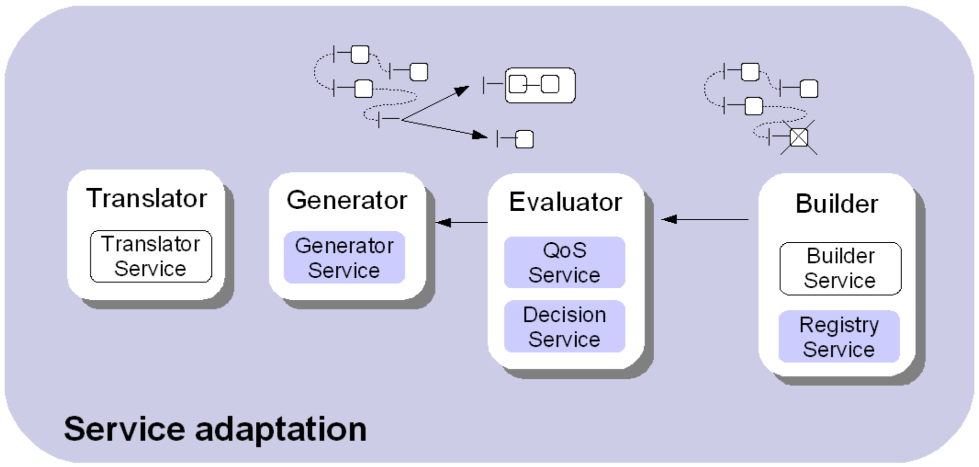


Figure 3.5: Service adaptation by MySIM

Section Conclusion

In this section, we presented a general SIM model - Service Integration Middleware. The SIM presented a unified vision for the three problems of service transformation, composition and adaptation. We validated the genericity of the SIM by showing that the current service transformation, adaptation, and composition middleware can be depicted using the SIM modules. We then introduced our `MySIM` middleware and explained its diverse services and their functionalities. This section explained how we fulfill the unified vision requirement. In the next section, we will depict our general SERVICE model and the service relations related to composition - the composition relation - and to adaptation - the equivalence relation. We will present and will discuss our service functional and non-functional QoS equivalence and composition relations provided by the `Generator Service` and `QoS Service` and respecting our SERVICE model.

3.2 Service Functional and Non-Functional QoS Integration Relations

In this section, we present our generic SERVICE model and the functional and non-functional QoS relations between services that we use for our service integration. This platform independent model relies upon the service-oriented programming paradigm. We formalize our SERVICE model and depict several service relations the service equivalence and almost equivalence relations (concerning functional aspects), the QoS equivalence degree (concerning non-functional QoS properties) and the service composition relation (composition of functional interfaces respecting non-functional properties). The service equivalence and almost equivalence relations are used in services adaptation as a service can be replaced by another one providing the same functionalities but with possible different non-functional properties. The service composition relation is used in services composition.

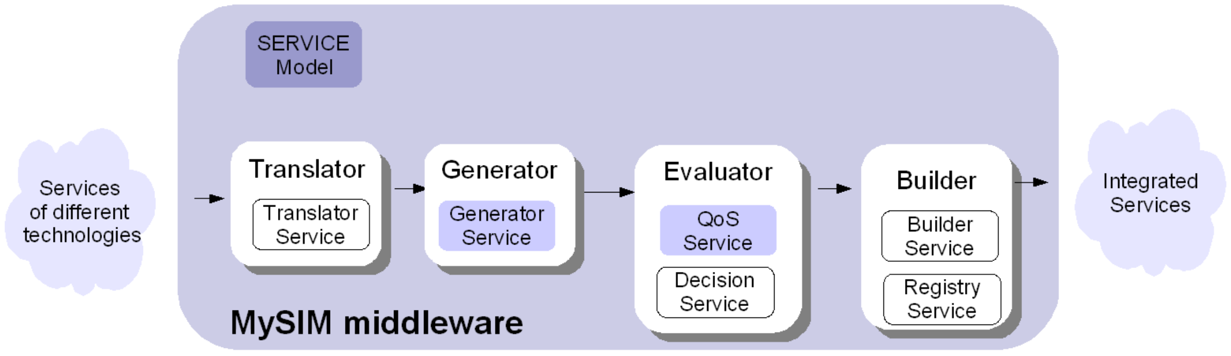


Figure 3.6: Modules using the functional and non-functional aspects of a service

Figure 3.6 highlights the different services of `MySIM` middleware that are detailed in this section and that play a role in providing our platform independent SERVICE model and in verifying the different services relations. The SERVICE model provides a generic way to describe the functionalities of the environment whether they are atomic or integrated and their services QoS. The `Generator Service` is responsible of services syntactic and semantic matching to determine services equivalence and services composition relations using their functional interfaces. The `QoS Service` is responsible of checking, comparing, and adjusting the non-functional QoS properties of the services involved in the integration.

In a service adaptation, `Generator Service` returns all the services functionally equivalent to the service to adapt and the `QoS Service` orders these equivalent services from the closest equivalent service to the farthest one in terms of non-functional QoS properties. Indeed, services can be functionally equivalent but providing different levels of QoS.

In a service composition, the `Generator Service` returns all possible service composition plans and the `QoS Service` takes in charge verifying and assigning the correct QoS values to the resulting service composition.

The rest of this section is as follows. First, our generic SERVICE model is defined and formalized using symbols and operators introduced table 1, 2, and 3. Then, we thoroughly explain our service equivalence, almost equivalence, (syntactic and semantic) and our service composition (syntactic and semantic) relations.

3.2.1 Formal Definitions

We define a service as composed of several parts (cf. figure 3.7):

- Interfaces: a service can hold two kinds of interfaces. Provided functional interface defining the functional behavior of the service. Required interfaces or operations specifying required

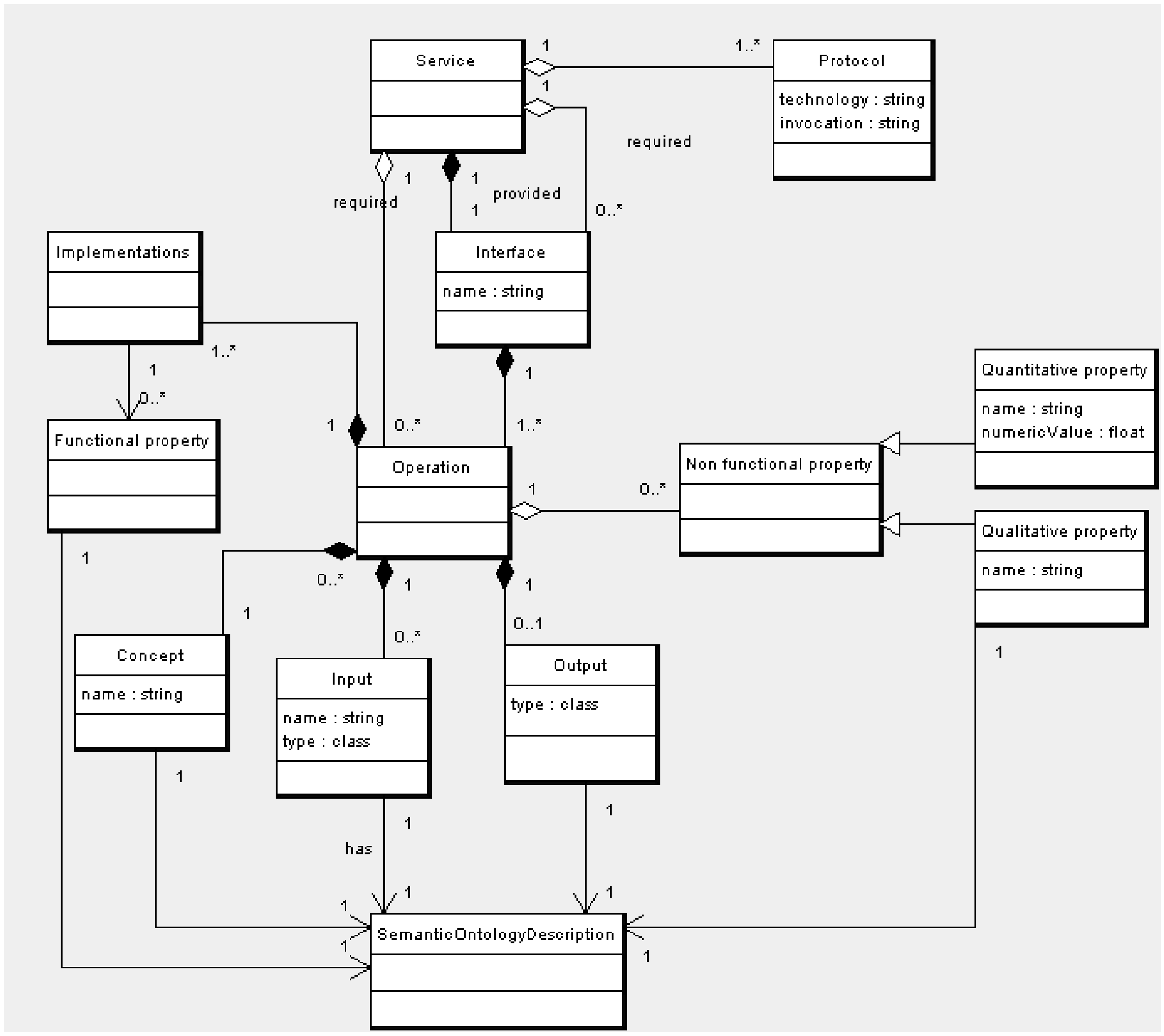


Figure 3.7: Service Model

functionalities from other services. A functional interface specifies operations that can be performed on the service. An operation is described by a concept, a set of inputs and an output. A required interface specifies the set of operations a service requires in order to execute appropriately. A service can also specifies one or several required operations not interfaces, as sometimes interfaces contain many other operations the service does not need.

- Implementations: implementations realize the functionalities expected from the service. These are the implementations of the operations defined in the functional interfaces.

- Functional properties: a service will register its interface implementations under certain properties. The functional property is used by the `Service Integration` to choose services that offer the same interfaces, but different implementations. These functional properties specify whether the service is atomic or integrated (resulting from a service integration).

- Non functional properties: they can be quantitative or qualitative. The non functional properties describe operations capabilities. These capabilities reflect the quality of the functionality expected from the service, such as dependability (including availability, reliability, security and safety), accuracy of the operation, speed of the operation, and so on.
- Semantic description based on ontologies: these descriptions use common ontologies to describe concepts (of interfaces, operations, properties), outputs, inputs and the non functional properties in a semantic way. Service description is important in order to locate, publish, compose, or execute services coming from everywhere, in a pervasive environment.
- Interaction protocols: describe the communication protocol used by the service. These protocols can be standards such as SOAP for web services, or remote call through a specific language such as in RPC. Translators must be provided to resolve interoperability problems if the service technology models used are different.

For all the potential users of a service (applications, clients, etc.), a service is seen as a functional interface with a number of operations providing specific functionalities, a semantic description, and interaction protocols specifying how to communicate with the service. The operations' implementations are more related to the internal state of the service and are usually not accessible to the outside world.

In the rest of the section, we use the symbols and operators introduced in tables 1, 2 and 3. The chosen formalism has not been inspired from a well known formalism but has been rather chosen for its simplicity in describing our concepts. The notation are simple and we use rules introduced by first order logic and formal language grammar to describe these concepts. Another formalism could be used instead if it similarly defines the introduced concepts.

Definition 6 — Service

Consider finite sets of grammatical alphabet Σ, services $\mathtt{S}$, interfaces $\mathtt{Ifc}$, operations $\mathtt{Op}$, non functional properties $\mathtt{Np}$, and protocols $\mathtt{Protocol}$.
We define a service s belonging to $S \subset \mathtt{S}$ as follows:

$$(s \in S \Leftrightarrow \exists\, Ifc \subset \mathtt{Ifc},\ \exists\, Op \subset \mathtt{Op}, \exists\, Protocol \subset \mathtt{Protocol}):$$

$$s : < ifc, Ifc^*, Op^*, Protocol^+ >$$

$$ifc : < name_{ifc},\ Op^+ >,\ name_{ifc} \in \Sigma^*$$
$$protocol \in Protocol,\ protocol : < technology, invocation >,\ \{technology, invocation\} \in \Sigma^*$$

■

A service $s \in S$ is defined as a tuple of a provided interface ifc, eventual required interfaces Ifc or operations Op, and specified protocols of communication $Protocol$. An interface ifc is specified by a syntactic $name_{ifc}$ and define a set of operations Op. A *protocol* $\in$ *Protocol* indicates the service technology model and the employed service method invocation.

DEFINITION 7 — Operation

Consider finite sets of grammatical alphabet Σ, ontologies $\mathtt{O}$, concepts $\mathtt{N}$ belongings to these ontologies $\mathtt{O}$, operations $\mathtt{Op}$, inputs $\mathtt{In}$, outputs $\mathtt{Out}$, concepts $\mathtt{Cpt}$, non functional properties $\mathtt{Np}$, quantitative and qualitative non-functional properties $\mathtt{Np}_{QN}$, $\mathtt{Np}_{QL}$, implementations $\mathtt{Impl}$, and functional properties $\mathtt{Pr}$. We define an operation *op* belonging to $Op \subset \mathtt{Op}$ as follows:

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

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

$in \in In,\ in : < name_{in}, type_{in}, semantic_{in} >,\ name_{in} \in \Sigma^{*}$
$out \in Out,\ out : < type_{out}, semantic_{out} >$
$cpt : < name_{cpt}, semantic_{cpt} >,\ name_{cpt} \in \Sigma^{*}$
$type : < language_{type}, name_{type} >,\ \{name_{type},\ language_{type}\} \in \Sigma^{*}$
$semantic_{\Sigma^{*}} : < o, n >,\ o \in O \subset \mathtt{O},\ n \in N \subset \mathtt{N}$
$np \in Np,\ np : < Np^{*}_{QL}, Np^{*}_{QN} >$
$np_{QL} \in Np_{QL},\ np_{QL} : < name_{np_{QL}}, semantic_{np_{QL}} >,\ name_{np_{QL}} \in \Sigma^{*}$
$np_{QN} \in Np_{QN},\ np_{QN} : < name_{np_{QN}}, numericValue_{np_{QN}}, operator_{np_{QN}} >,\ name_{np_{QN}} \in \Sigma^{*}$
$numericValue_{np_{QN}} \in \mathbb{R}$
$operator_{np_{QN}} :\ \{<,\ >,\ \leq,\ \geq\}$
$impl : < Pr^{0..1} >$
$pr \in Pr,\ pr :\ \{< semantic_{pr}, value_{pr} >\}^{+}$
$value_{pr} \in \{''atomic'',\ ''integrated''\}$

■

where:

- In is the set of the operation *op* inputs. $In = \{in_1 \ \dots \ in_{|In|}\}$. $\forall\, j \in \{1..|In|\}$, in_j is defined as a tuple: $< name_{in_j}, type_{in_j}, semantic_{in_j} >$, where $name_{in_j}$ is the chosen input syntactic name, $type_{in_j}$ is the syntactic input type, and $semantic_{in_j}$ the input semantic description.
- $out \in Out$ is the operation *op* output. *out* is defined as a tuple $< type_{out}, semantic_{out} >$ where $type_{out}$ is the output syntactic type, and $semantic_{out}$ its the semantic description.
- *cpt* is the concept the operation *op* defines. The operation *op* concept *cpt* is defined as a tuple $< name_{cpt}, semantic_{cpt} >$, where $name_{cpt}$ is the syntactic name through which the operation is called and $semantic_{cpt} =< o, n >$ where $o \in O \subset \mathtt{O}$ and $n \in N \subset \mathtt{N}$ is the semantic ontologies-based description of the operation concept.

- Np is the set of non functional properties characterizing *op*. Np can be qualitative or quantitative, $Np =< Np^*_{QL}, Np^*_{QN} >$. $np_{QN} \in Np_{QL}$ is the qualitative non functional properties defined as a tuple $< name_{np_{QL}}, semantic_{np_{QL}} >$. $np_{QN} \in Np_{QN}$ is the quantitative non functional properties defined as a tuple $< name_{np_{QN}}, numericValue_{np_{QN}}, operator_{np_{QN}} >$, where $numericValue_{np_{QN}} \in \mathbb{R}$ and $operator_{np_{QN}} \in \{>, <, \leq, \geq\}$. $operator_{np_{QN}}$ specifies the order applied to $numericValue_{np_{QN}}$. For $\{>, \geq\}$ the greater the $numericValue_{np_{QN}}$ is, the best is the QoS property for the service runtime execution. For $\{<, \leq\}$ the smaller the $numericValue_{np_{QN}}$ is, the best is the QoS property for the service runtime execution.

- *impl* is the operation *op* internal implementation. The operation *op* implementation *impl* is described by a functional property (Pr). Property $pr \in Pr$ is a tuple $\{< semantic_{pr}, value_{pr} >\}^+$ where $semantic_{pr} =< o, n >$, $o \in O, n \in N$ and $value_{pr} \in \{''atomic'' \vee ''integrated''\}$. The property describes the operations implementation and specifies whether this implementation is atomic or integrated (resulting from service integration).

The *type* depends strongly on the programming language the *op* is defined in, whereas the *semantic* is independent of the technology and more related to the set of defined ontologies O. We define *semantic* as a tuple $< o, n >$, where $o \in O$ and $n \in N$.

Our service model is general enough to respect the SOA specifications, and to offer a common model to the heterogeneous technologies usually available in pervasive environments. The model proposes semantic descriptions relying on common ontologies, and by that it allows to abstract from the programming languages. It also models the *type* of parameter inputs In, and outputs Out. Later on we motivate our choice of modeling the syntactic part of a service. For now, we would like to focus on what this model brings to the SOA in pervasive environments: the semantic description of the integration process. Components and frameworks using services in the middleware layer, are aware of the integrated functionalities. Services can be chosen not only relying on a description of the concept, inputs and outputs of their operations as usual, but also using this functional property that indicates exactly whether a service is atomic or not, and in case of an integrated service it indicates all the functionalities a service combines. For a same service interface specification multiple implementations can exist. These implementations are described by different properties. Adaptation techniques rely on these properties to choose the "best" implementation of a service interface at a given time and for a given purpose.

Now that we defined our service model, we detail the different possible relations between services. We define two kinds of relations between two services s_1 and s_2: equivalence relation and composition relation (cf. figure 3.8). Service equivalence is used for service adaptation in pervasive environments. Unavailable services are replaced when needed by other services that can offer the same functionalities. Service composition relation is used for service composition. Service composition and service adaptation are explained section 3.3.

3.2.2 Services Equivalence Relations

Service equivalence relations determine whether two services offer the same functionality or not. A service is considered equivalent to another one if it can offer the same functionality (same interface,

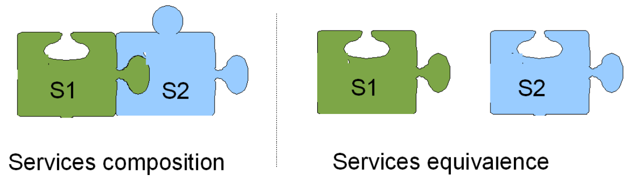


Figure 3.8: Abstract service relations descriptions

same implementations) even with different non-functional QoS properties. The aim of this section is to provide definitions of possible relations between the diverse parts of a service in order to identify and decide when a service can be replaced by another one. Two relations are introduced. The equivalence ($\equiv$) and the almost equivalence ($\triangleright$) relations. These relations are defined over the two main functional parts of a service, its interface and its implementations. In an equivalence relation, the two equivalent entities can interchange and be replaced one by the other. The equivalence relation is reflexive, symmetric, and transitive. In an almost equivalence relation only one entity can replace the other one. This relation is non reflexive, asymmetric, and transitive. It is based on sub-typing for syntactic almost equivalence and on sub-concept for semantic almost equivalence. This almost equivalence relation does not define an order relation between entities, but allow to relax the equivalence relation and find entities that are functionally similar even if not equivalent.

The relations tackle two main parts of a service: its functional interface and the implementations of the interface operations, described by the property. Indeed, two services can be equivalent or almost equivalent to users and applications if they provide the same interfaces but not necessarily the same interface implementations. And, two services can be equivalent or almost equivalent for the middleware layer if they provide not only equivalent or almost equivalent interfaces but also equivalent or almost equivalent implementations. In the rest of this section, we define our interface equivalence relations, our implementation equivalence relations, and our non-functional QoS degree of equivalence. All these relations are thoroughly used and explained in our spontaneous service composition and adaptation described in section 3.3.

3.2.2.1 Functional Interfaces Equivalence Relations

We distinguish between two forms of interfaces equivalence - syntactic and semantic interfaces equivalences. In the following, we detail the syntactic interface equivalence followed by the semantic one. We mean by syntactic equivalence, equivalence based on operations signature and typing. This kind of equivalence is possible between interfaces provided in the same technology languages. Semantic equivalence concerns the semantic description of interfaces. It can be applied to interfaces independently of the technology language they are implemented in.

(1) Syntactic equivalence

We define the syntactic interface equivalence using the following operator $\equiv_{syntactic}$. Two interfaces are considered by users and/or applications to be syntactically the same if they provide, externally, the same functionalities. We mean by same functional interfaces, interfaces providing exactly the same operations - the same operations' number and signatures. Indeed, for an interface to be invoked as a replacement of another one, its operations need to have the same signatures as the other interface for a transparent interface call. This condition do not apply upon the non-functional QoS properties of operations. Two interfaces can be syntactically equivalent even if their operations do not offer and require the same non-functional QoS properties.

The interface syntactic equivalence $\equiv_{syntactic}$ is based upon an operation syntactic equivalence which is based on a type equivalence $\equiv_{type}$. We first define what we mean by a type equivalence, then we explain our operation syntactic equivalence, followed by our interface syntactic equivalence.

Type equivalence

Two anonymous types will be considered equivalent $\equiv_{type}$ if all of the following properties are true:

- They have the same number of fields
- They have fields of the same syntactic name declared in the same order
- The types of each of the fields are identical

A type system has:

- A set of basic (primitive) types
- A mechanism for defining new types
- Rules for type equivalence - when are the types of two values the same?
- Rules of type compatibility - when can a value of type a be used in a context that expects type b?

Two principal ways of defining type equivalence exist: structural and name equivalence. In a name equivalence, every type declaration defines a new type. In a structural equivalence, two types are equivalent if they consist of the same components. Algol-68 uses structural equivalence, Standard Pascal and Java use name equivalence, whereas C uses a hybrid form of equivalence.

The equivalence $\equiv_{type}$ defined above is a type compatibility. So the question one can ask is: when can a value of type a be used in a context that expects type b? a type of value a can replace a type of value b if a is type equivalent to b, or almost equivalent to b.

We define these two notions of type equivalence and almost equivalence:

Definition 8 — Type equivalence $\equiv_{type}$

Two types a and b are equivalent ($\equiv_{type}(a,b) = true$) if:

- a and b are name or structural equivalence depending on the programming language.

■

Definition 9 — Type almost equivalence $\triangleright_{type}$

Two types a and b are almost equivalent ($\triangleright_{type}(a, b) = true$) if at least one of these conditions is true:

- (a is a subtype of b)
- $\vee$ (a value of type a can be coerced to a value of type b)

■

Using this definition of types relations, we define operation and service syntactic equivalence ($\equiv_{syntactic}$) and almost equivalence ($\triangleright_{syntactic}$).

Operations syntactic equivalence

Considering two operation opi and opj, we define when these two operations can be compared.

Definition 10 — Comparable Operations $\propto$

We define two operations opi and opj to be comparable ($\propto(opi, opj) = true$) if they have the same number of inputs and the same number of outputs and if it exists a bijection f over their inputs allowing to compare the inputs parameters two by two. $\forall\ k, l \in \{1..|In_{opi}|\}$

$$\begin{aligned}
&|In_{opi}| = |In_{opj}| \\
&\wedge (|Out_{opi}| = |Out_{opj}|) \\
&\wedge (\exists\ f : In_{opi} \rightarrow In_{opj},\ \forall\ in_l \in In_{opj},\ \exists!\ in_k \in In_{opi},\ f(in_k) = in_l)
\end{aligned}$$

■

Definition 11 — Operation syntactic equivalence

$\forall\ \{i, j\} \in \mathbb{N}$, opi and opj are syntactically equivalent ($\equiv_{syntactic}(opi, opj) = true$) if:

$$\begin{aligned}
&\propto(opi, opj) = true \\
&\wedge (name_{cpt_{opi}} = name_{cpt_{opj}}),\ name_* \in \Sigma^* \\
&\wedge (\forall\ in_k \in In_{opi},\ (\equiv_{type}(type_{in_k}, f(type_{in_k})) = true)),\ k \in \{i..|In_{opi}|\} \\
&\wedge (\equiv_{type}(type_{out_{opi}}, type_{out_{opj}}) = true)
\end{aligned}$$

■

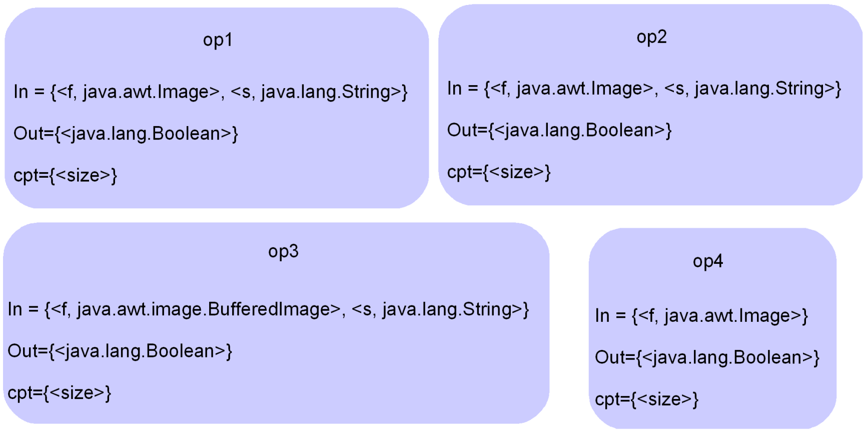
DEFINITION 12 — Operation syntactic almost equivalence

$\forall\ \{i,j\} \in \mathbb{N}$, opi and opj are syntactically almost equivalent ($\triangleright_{syntactic}(opi, opj) = true$) if opi and opj are not syntactic equivalent and at least one input or output parameters have an almost equivalence of types:

$$\begin{aligned}
&\propto(opi, opj) = true\\
&\wedge\ (name_{cpt_{opi}} = name_{cpt_{opj}}),\ name_{*} \in \Sigma^{*}\\
&\wedge\ (\forall\ in_k \in In_{opi},\ (\triangleright_{type}/\equiv_{type}(type_{in_k}, f(type_{in_l})) = true)),\ k \in \{i..|In_{opi}|\}\\
&\wedge\ (\triangleright_{type}/\equiv_{type}(type_{out_{opi}}, type_{out_{opj}}) = true)
\end{aligned}$$

■

EXAMPLE 1 *Considering the following four operations signatures in figure 3.9:*


op1

In = {<f, java.awt.Image>, <s, java.lang.String>}

Out={<java.lang.Boolean>}

cpt={<size>}

op2

In = {<f, java.awt.Image>, <s, java.lang.String>}

Out={<java.lang.Boolean>}

cpt={<size>}

op3

In = {<f, java.awt.image.BufferedImage>, <s, java.lang.String>}

Out={<java.lang.Boolean>}

cpt={<size>}

op4

In = {<f, java.awt.Image>}

Out={<java.lang.Boolean>}

cpt={<size>}

Figure 3.9: operations specifications

If we consider the two operations op1 and op2, the following relations are verified:

$$\begin{aligned}
&(cpt_{op1} = cpt_{op2} = size)\\
&\wedge\ (\equiv_{type}(java.awt.Image, java.awt.Image) = true)\\
&\wedge\ (\equiv_{type}(java.lang.String, java.lang.String) = true)\\
&\wedge\ (\equiv_{type}(java.lang.Boolean, java.lang.Boolean) = true)
\end{aligned}$$

These relations implies $\Rightarrow\ (\equiv_{syntactic}(op1, op2) = true)$

Similarly we can conclude to the following relations:

$$\triangleright_{syntactic}(op3, op1) = true$$

$$\triangleright_{syntactic}(op3, op2) = true$$

$$\propto(op4, opi) = false,\ \forall\ i \in \{1,2,3\}$$

Syntactic interfaces equivalence

We define two interfaces to be comparable ($\propto (ifc_i, ifc_j) = true$) if they have the same number of operations and if it exists a bijection f over their operations allowing to compare them two by two:

Definition 13 — Comparable interfaces $\propto$

We define two interfaces ifc_i and ifc_j to be comparable ($\propto (ifc_i, ifc_j) = true$) if:

$$|Op_{ifc_i}| = |Op_{ifc_j}|$$
$$\wedge\ (\exists\ f : Op_{ifc_i} \rightarrow Op_{ifc_j},\ \forall\ op_l \in Op_{ifc_j},\ \exists!\ op_k \in Op_{ifc_i},\ f(op_k) = op_l)$$

■

Definition 14 — Interface syntactic equivalence

Considering two interfaces ifc_i and ifc_j. ifc_i and ifc_j are syntactic equivalent ($\equiv_{syntactic} (ifc_i, ifc_j) = true$) if:

$$\propto (ifc_i, ifc_j) = true$$
$$\wedge\ (name_{ifc_i} = name_{ifc_j})$$
$$\wedge\ (\forall\ op_k \in Op_{ifc_i},\ (\equiv_{syntactic} (op_k, f(op_k)) = true)),\ k \in \{1..|Op_{ifc_i}|\}$$

■

Definition 15 — Interface syntactic almost equivalence

Considering two services ifc_i and ifc_j. ifc_i and ifc_j are syntactic almost equivalent ($\triangleright_{syntactic}(ifc_i, ifc_j) = true$) if:

$$\propto (ifc_i, ifc_i) = true$$
$$\wedge\ (name_{ifc_i} = name_{ifc_j})$$
$$\wedge\ (\forall\ op_k \in Op_{ifc_i},\ (\triangleright_{syntactic}(op_k, f(op_k)) = true)),\ k \in \{1..|Op_{ifc_i}|\}$$

■

Interfaces can be compared and matched on a subset of operations. This is useful when we search to replace a service interface by another one for a specific operation and not only for the service as a whole.

We thus introduce the interface equivalence and almost equivalence on a set of operations:

Definition 16 — Interface syntactic equivalence over a set of operations

Considering two interfaces ifc_i and ifc_j, $\exists\ Op_{ifc_i} \subset ifc_i$, $\exists\ Op_{ifc_j} \subset ifc_j$, $|Op_{ifc_i}| = |Op_{ifc_j}|$ and ($\propto (ifc_i, ifc_j) = true$) over these operations sets. ifc_i and ifc_j are syntactic equivalent over these operations ($\equiv^{Op}_{syntactic} (ifc_i, ifc_j) = true$) if:

$$\forall\ op_k \in Op_{ifc_i},\ (\triangleright_{syntactic}(op_k, f(op_k)) = true),\ k \in \{1..|Op_{ifc_i}|\}$$

■

Definition 17 — Interface syntactic almost equivalence over a set of operations

Considering two services ifc_i and ifc_j, $\exists\, Op_{ifc_i} \subset ifc_i$, $\exists\, Op_{ifc_j} \subset ifc_j$, $|Op_{ifc_i}| = |Op_{ifc_j}|$ and $(\propto (ifc_i, ifc_j) = true)$ over these operations sets. ifc_i and ifc_j are syntactic almost equivalent over these operations $(\triangleright^{Op}_{syntactic}(ifc_i, ifc_j) = true)$ if:

$$\forall\, op_k \in Op_{ifc_i},\ (\triangleright_{syntactic}(op_k, f(op_k)) = true),\ k \in \{1..|Op_{ifc_i}|\}$$

■

Example 2 *Considering the following three interfaces in figure 3.10:*

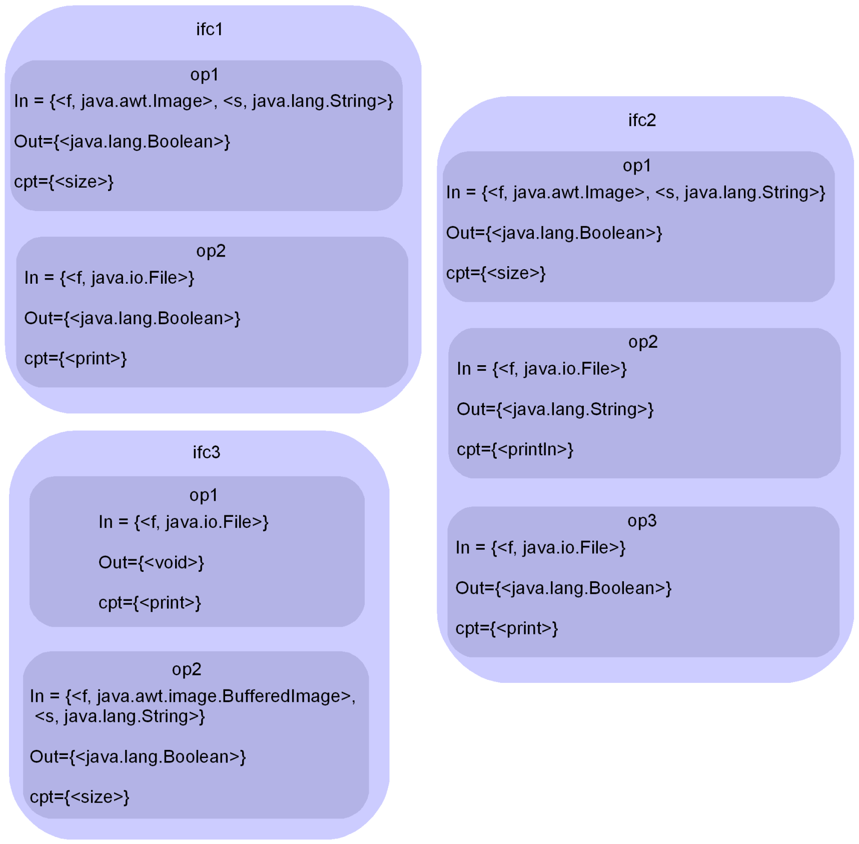

ifc1
op1
In = {<f, java.awt.Image>, <s, java.lang.String>}
Out={<java.lang.Boolean>}
cpt={<size>}
op2
In = {<f, java.io.File>}
Out={<java.lang.Boolean>}
cpt={<print>}
ifc2
op1
In = {<f, java.awt.Image>, <s, java.lang.String>}
Out={<java.lang.Boolean>}
cpt={<size>}
op2
In = {<f, java.io.File>}
Out={<java.lang.String>}
cpt={<println>}
op3
In = {<f, java.io.File>}
Out={<java.lang.Boolean>}
cpt={<print>}
ifc3
op1
In = {<f, java.io.File>}
Out={<void>}
cpt={<print>}
op2
In = {<f, java.awt.image.BufferedImage>, <s, java.lang.String>}
Out={<java.lang.Boolean>}
cpt={<size>}

Figure 3.10: interfaces specifications

The two interfaces $ifc1$ and $ifc2$ verifies the following relations:

$$(\equiv_{syntactic}(op1_{ifc1}, op1_{ifc2}) = true) \wedge (\equiv_{syntactic}(op2_{ifc1}, op3_{ifc2}) = true)$$

This implies an equivalence relations between $ifc1$ and $ifc2$ over a set of their operations

$\Rightarrow\ (\equiv_{syntactic}^{\{op1_{ifc1},\ op2_{ifc1}\}}(ifc1, fc2)\ =\ true)$

The two interfaces $ifc2$ and $ifc3$ verifies:

$$\rhd_{syntactic}(op1_{ifc3}, op1_{ifc2})\ =\ true$$

This implies an almost equivalence relation over a set of their operations:

$\Rightarrow\ (\rhd_{syntactic}^{\{op2_{ifc3}\}}(ifc3, fc2)\ =\ true)$

The syntactic matching is applied upon interface services defined in the same technology model, as typing is strongly dependent on the programming language. If services are defined in two different programming languages, such as Java and C++, a type translation should be done in order to apply our definition. Indeed, translators are needed in order to verify if the types are similar but expressed in respective programming languages, such as *java.lang.Boolean* expressed in Java and *bool* expressed in C++. With these translators we can declare that two operations are syntactically the same but are expressed in two different technology languages. This category can also be resolved by semantics, as each operation has a semantic description. Providing translators to all existing programming languages is not really a solution. We adopt the semantic approach due to the interoperability of semantics based on common ontologies, which are independent of the employed technologies, and we define the service semantic equivalence for services expressed in different technology models.

Example 3 *The two operations defined figure 3.13 Printing and Printer are syntactically equivalent, as both of them are defined using Java language and have type equivalence over the inputs and outputs and syntactic equivalence over the concept. The $Impression$ operation is defined using C++ language and at that stage it is difficult to compare it with the others without translation mechanisms. We can notice that the two operations Printing and Printer even if functionally equivalent have different non-functional QoS properties. This aspect is dealt with and explained in the section 3.2.2.2.*

(2) Semantic equivalence

If the syntactic interface equivalence considered type equivalence, we define the semantic interface equivalence $\equiv_{semantic}$ upon semantic operation equivalence which itself is defined upon a concept matching $M_{concept}$ with concepts belonging to a defined ontology.

Concept matching

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

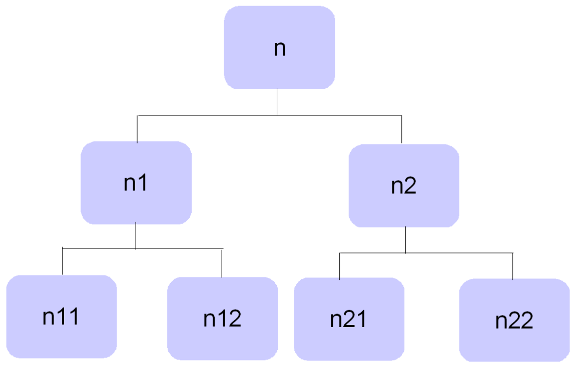


Figure 3.11: An ontology example

A concept n belonging to an ontology o (figure 3.11), can provide all its immediate sub-concepts n_1 and n_2 or one of its sub-concepts n_1 or n_2. This distinction depends strongly on the ontology definitions and providers. If some research such as Paolucci [Paolucci et al. 2002] made the assumption that by selecting a concept n, we implicitly suppose that it provides all its immediate sub-concepts, others made the other assumption that by selecting a concept n, it provides at least one of its immediate sub-concepts, but not necessarily all of them. Consider the set $\{n_1, n_2, .., n_n\}$ of all the sub-concepts of a concept n in an ontology o, the assumption of Paolucci [Paolucci et al. 2002] is formalized as follows: $n \ \equiv_{provide} \ (n_1 \wedge n_2 \wedge ... \wedge n_n)$ which means that n can replace $n1$, $n2$, etc. Others, do not make strong assumptions as this and suppose that a concept n provides one or more of its sub-concepts but not necessarily all of them, $n \ \equiv_{provide} \ (n_1 \vee n_2 \vee ... \vee n_n)$. We fall into the first category, stipulating that a super-concept offers what its sub-concepts offer, and hence can replace them.

Definition 18 — Concept matching $M_{concept}$

Defining n and m, two concepts belonging to the same ontology o. We define the four values of concept matching $M_{concept}$ inspired from Paolucci [Paolucci et al. 2002] as follows:

$$M_{concept}(n,m) \quad \begin{array}{l} Exact : If \ n \ and \ m \ are \ equivalent \ concept \\ PlugIn : If \ n \ is \ a \ super\text{-}concept \ of \ m \\ Subsume : If \ n \ is \ a \ sub\text{-}concept \ of \ m \\ Fail : If \ n \ and \ m \ do \ not \ verify \ the \ above \ conditions \end{array}$$

■

EXAMPLE 4 *Using our ontology example figure 3.12, we give an example of $M_{concept}$.*

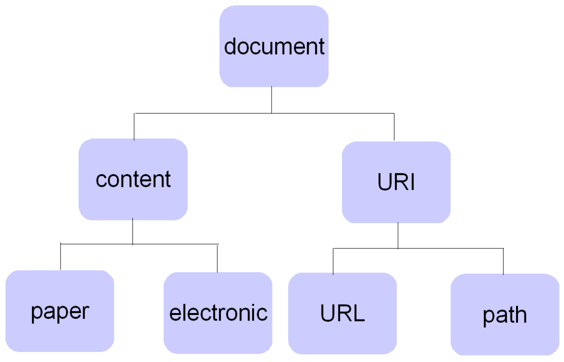


Figure 3.12: A document ontology example

$$\textit{Example}\ \left|\ \begin{array}{l} M_{concept}(''content'',\ ''electronic'') = PlugIn \\ M_{concept}(''document'',\ ''URL'') = PlugIn \\ M_{concept}(''paper'',\ ''document'') = Subsume \\ M_{concept}(''content'',\ ''path'') = Fail \end{array}\right.$$

Semantic operations equivalence

Using this concept matching we define our operation followed by the interface semantic matching values.

First we explain how we match two operations op_i and op_j.

$\forall\ \{i, j, l, k\} \in \mathbb{N}$, we define the semantic matching of two comparable operations op_i and op_j ($\propto (op_i, op_j) = true$), $M_{semantic}(op_i, op_j)$, considering the semantic matching of their concepts, inputs and outputs.

We can quickly realize that the semantic matching of these three items - inputs, outputs, and concepts - can be different, as the concept matching can take multiple values. In syntactic equivalence, the condition was to have a strict syntactic equivalence for the three items. In semantic matching, the three items can range from *Exact* matching to *Fail* passing by the *PlugIn* and *Subsume* values.

We define the different values a semantic matching between two operations op_i and op_j can take as follows:

Definition 19 — $M_{semantic}(op_i, op_j)$

Two operations op_i and op_j verifying $(\propto (op_i, op_j) = true)$ are *Exact* semantic matching if all the matching values between concept, inputs and output are *Exact*. $\forall\ k \in \mathbb{N}$:

$(M_{concept}(n_{semantic_{cpt_{op_i}}}, n_{semantic_{cpt_{op_j}}}) = Exact)$
$\wedge\ (\forall\ in_k \in In_{op_i},\ M_{concept}(n_{semantic_{in_k}}, f(n_{semantic_{in_k}})) = Exact)$
$\wedge\ (M_{concept}(n_{semantic_{out_{op_i}}}, n_{semantic_{out_{op_j}}}) = Exact)$

They are *PlugIn* semantic matching if they are not *Exact* matching and all the matching between concept, inputs or output values are *Exact* or *PlugIn*. $\forall\ k \in \mathbb{N}$:

$M_{semantic}(op_i, op_j) \neq Exact$
$\wedge\ (M_{concept}(n_{semantic_{cpt_{op_i}}}, n_{semantic_{cpt_{op_j}}}) \in \{Exact \vee PlugIn\})$
$\wedge\ (\forall\ in_k \in In_{op_i},\ M_{concept}(n_{semantic_{in_k}}, f(n_{semantic_{in_k}}))\ in\ \{Exact \vee PlugIn\})$
$\wedge\ (M_{concept}(n_{semantic_{out_{op_i}}}, n_{semantic_{out_{op_j}}}) \in \{Exact \vee PlugIn\})$

They are *Subsume* semantic matching if they are no *Exact* or *PlugIn* matching and at least one matching value between concept, inputs or output is *Subsume* and no *Fail* matching value is found between outputs, concepts, and the corresponding comparable inputs. $\forall\ k \in \mathbb{N}$:

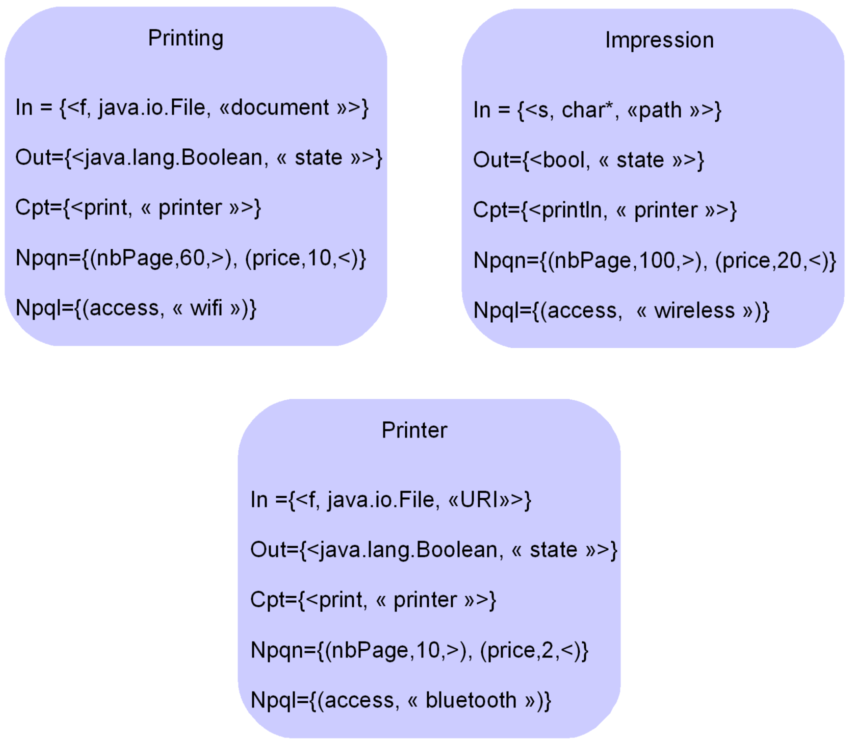
$M_{semantic}(op_i, op_j) \neq Exact$
$\wedge\ (M_{semantic}(op_i, op_j) \neq PlugIn)$
$\wedge\ (M_{concept}(n_{semantic_{cpt_{op_i}}}, n_{semantic_{cpt_{op_j}}}) = \neg(Fail))$
$\wedge\ (\forall\ in_k \in In_{op_i},\ M_{concept}(n_{semantic_{in_k}}, f(n_{semantic_{in_k}})) = \neg(Fail))$
$\wedge\ (M_{concept}(n_{semantic_{out_{op_i}}}, n_{semantic_{out_{op_j}}}) = \neg(Fail))$

They are *Fail* semantic matching if they have different inputs or outputs numbers or at least one semantic matching value between concepts, inputs or outputs is *Fail*. $\forall\ \{k, l\} \in \mathbb{N}$:

$(|In_{op_i}| \neq |In_{op_j}|)$
$\vee\ (|Out_{op_i}| \neq |Out_{op_j}|)$
$\vee\ (M_{concept}(n_{semantic_{cpt_{op_i}}}, n_{semantic_{cpt_{op_j}}}) = Fail)$
$\vee\ (\exists\ in_k \in In_{op_i},\ \forall\ in_l \in In_{op_j},\ M_{concept}(n_{semantic_{in_k}}, n_{semantic_{in_l}}) = Fail)$
$\vee (M_{concept}(n_{semantic_{out_{op_i}}}, n_{semantic_{out_{op_j}}}) = Fail)$

■

Example 5 *Considering the three operations defined in figure 3.13*



Printing

In = {<f, java.io.File, «document »>}

Out={<java.lang.Boolean, « state »>}

Cpt={<print, « printer »>}

Npqn={(nbPage,60,>), (price,10,<)}

Npql={(access, « wifi »)}

Impression

In = {<s, char*, «path »>}

Out={<bool, « state »>}

Cpt={<println, « printer »>}

Npqn={(nbPage,100,>), (price,20,<)}

Npql={(access, « wireless »)}

Printer

In ={<f, java.io.File, «URI»>}

Out={<java.lang.Boolean, « state »>}

Cpt={<print, « printer »>}

Npqn={(nbPage,10,>), (price,2,<)}

Npql={(access, « bluetooth »)}

Figure 3.13: Three services operations specifications

The semantic matching between these operations give the following values:

$$
\begin{aligned}
M_{concept}(Printing, Impression) &= PlugIn\\
M_{concept}(Printing, Printer) &= PlugIn\\
M_{concept}(Impression, Printer) &= Subsume\\
M_{concept}(Impression, Printing) &= Subsume\\
M_{concept}(Printer, Printing) &= Subsume\\
M_{concept}(Printer, Impression) &= PlugIn
\end{aligned}
$$

Definition 20 — Operation semantic equivalence

We define two operations op_i and op_j to be semantically equivalent $\equiv_{semantic}$ if:

$$(\equiv_{semantic}(op_i, op_j) = true) \Leftrightarrow (M_{semantic}(opi, opj) = Exact)$$

The semantic operation equivalence $\equiv_{semantic}$ is reflexive, symmetric, and transitive. As for the syntactic operation equivalence, the semantic equivalence satisfies the conditions an equivalence relation $\Re$ needs to fulfill.

■

Definition 21 — Operation semantic almost equivalence

We define two operations op_i and op_j to be semantically almost equivalent $\triangleright_{semantic}$ if:

$$(\triangleright_{semantic}(opi, opj) = true) \Leftrightarrow (M_{semantic}(opi, opj) = PlugIn)$$

The semantic almost equivalence is non reflexive, asymmetric, and transitive. This relation of almost equivalence specifies that *opi* is equivalent to *opj* and can replace it but that the contrary is not true. *opj* can not always replace *opi*.

■

Example 6 *Coming back to our example in figure 3.13, where we had these matching values between the three operations Printing, Impression, and Printer:*

$$\begin{aligned}
&M_{concept}(Printing, Impression) = PlugIn\\
&M_{concept}(Printing, Printer) = PlugIn\\
&M_{concept}(Impression, Printer) = Subsume\\
&M_{concept}(Impression, Printing) = Subsume\\
&M_{concept}(Printer, Printing) = Subsume\\
&M_{concept}(Printer, Impression) = PlugIn
\end{aligned}$$

we can conclude the following almost equivalent relations:

$$\begin{aligned}
&\triangleright(Printing, Impression) = true\\
&\triangleright(Printing, Printer) = true\\
&\triangleright(Printer, Impression) = true
\end{aligned}$$

Semantic interfaces equivalence

As for operations we define the semantic matching between two interfaces ifc_i and ifc_j:

Definition 22 — $M_{semantic}(ifc_i, ifc_j)$

Two interfaces ifc_i and ifc_j are *Exact* semantic match if $\propto (ifc_i, ifc_j) = true$ and:

$$\forall op_i \in Op_{ifc_i},\ M_{semantic}(op_i, f(op_i)) = Exact$$

They are *PlugIn* semantic match if $\propto (ifc_i, ifc_j) = true$, and:

$$M_{semantic}(ifc_i, ifc_j) \neq Exact$$
$$\wedge\ (\forall op_i \in Op_{ifc_i},\ M_{semantic}(op_i, f(op_i)) \in \{Exact \vee\ PlugIn\})$$

They are *Subsume* semantic match if $\propto (ifc_i, ifc_j) = true$, ifc_i and ifc_j are not *Exact* nor *PlugIn* semantic match and:

$$M_{semantic}(ifc_i, ifc_j) \neq Exact$$
$$\wedge\ (M_{semantic}(ifc_i, ifc_j) \neq PlugIn)$$
$$\wedge\ (\forall op_i \in Op_{ifc_i},\ M_{semantic}(op_i, f(op_i)) \in \{Exact \vee\ PlugIn\ \vee\ Subsume\})$$

The equality over the number of operations is not required. As a *Subsume* relation between services is not used to define equivalence or almost equivalence relations.

They are *Fail* semantic match if:

$$\propto (ifc_i, ifc_j) = false$$
$$\vee\ (\exists\ op_i \in Op_{ifc_i},\ \forall\ op_j \in Op_{ifc_j},\ M_{semantic}(op_i, op_j) = Fail)$$

It is sufficient to have only one operation op_i of ifc_i that do *Fail* match with any operation op_j of ifc_j to declare that the two services matching fails.

■

Based on these interface semantic matching definitions, we define the interface semantic equivalence and almost equivalence.

Definition 23 — Interface semantic equivalence

We define two interfaces ifc_i and ifc_j to be semantically equivalent $\equiv_{semantic}$ if:

$$(\equiv_{semantic}(ifc_i, ifc_j) = true) \Leftrightarrow (M_{semantic}(ifc_i, ifc_j) = Exact)$$

The semantic equivalence $\equiv_{semantic}$ is reflexive, symmetric, and transitive.

■

Definition 24 — Interface semantic almost equivalence

We define two services ifc_i and ifc_j to be semantically almost equivalent $\triangleright_{semantic}$ if:

$$(\triangleright_{semantic}(ifc_i, ifc_j) = true) \Leftrightarrow (M_{semantic}(ifc_i, ifc_j) = PlugIn)$$

As for operations, the semantic almost equivalence is non reflexive, non symmetric, and transitive. This relation of almost equivalence specifies that ifc_i is equivalent to ifc_j and can replace it but that the contrary is not true. ifc_j cannot always replace ifc_i.

■

Example 7 *Considering the three interfaces and their semantic descriptions in figure 3.33:*

The semantic matching between their different operations gives the following values:

$$\begin{aligned}&\propto (ifc1, ifc3) = true\\ &\wedge (M_{concept}(op1_{ifc1}, op1_{ifc3}) = PlugIn)\\ &\wedge (M_{concept}(op2_{ifc1}, op2_{ifc3}) = PlugIn)\end{aligned}$$

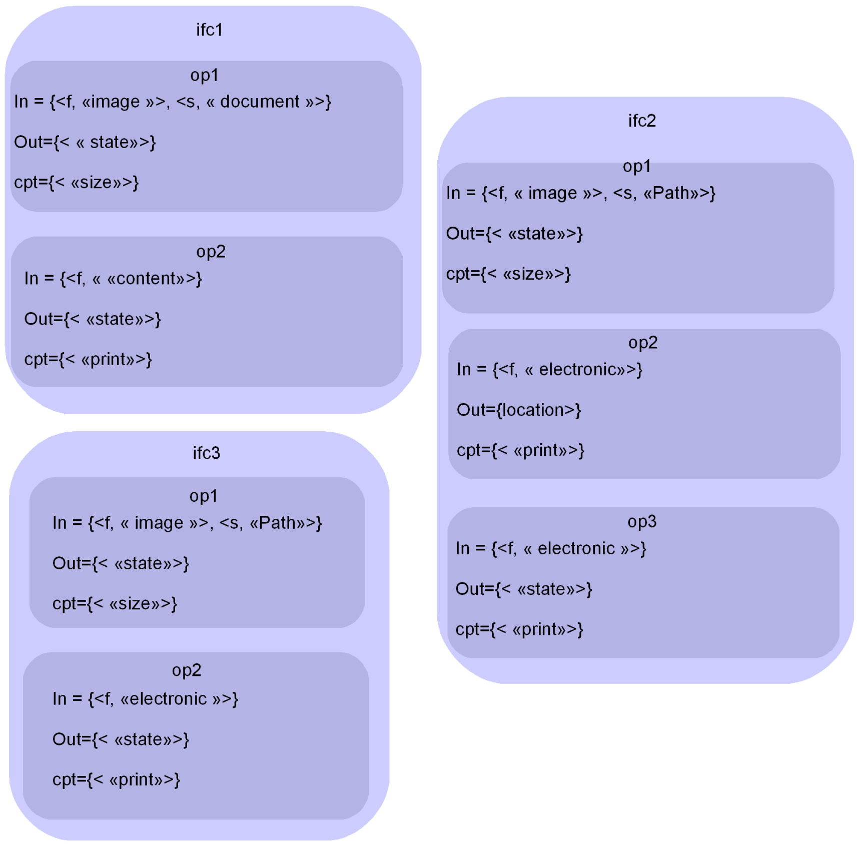


Figure 3.14: Three services operations specifications

We can implies $\Rightarrow$ $(\triangleright_{semantic}(ifc1, ifc3) = true)$

The two interfaces $ifc1$ and $ifc2$ are not comparable as they do not have the same number of operations. Nevertheless, some of their operations are $PlugIn$ semantic.

To resolve the issue brought in example 7. We define the matching over a set of operations for two interfaces ifc_i and ifc_j.

Definition 25 — $M^{Op}_{semantic}(ifc_i, ifc_j)$

Two interfaces ifc_i and ifc_j are *Exact* semantic matching over a subset of operations Op, if:

$$\begin{aligned}&\propto(ifc_i, ifc_j) \;=\; true\\ &\wedge\,(Op \;\subset\; Op_{ifc_i},\; \forall opi \;\in\; Op,\; M_{semantic}(op_i, f(op_i)) \;=\; Exact)\end{aligned}$$

Two services ifc_i and ifc_j are *PlugIn* semantic matching over a subset of operations Op, if:

$$\begin{aligned}&\propto(ifc_i, ifc_j) \;=\; true\\ &M^{Op}_{semantic}(ifc_i, ifc_j) \;\neq\; Exact\\ &\wedge\,(Op \;\subset\; Op_{ifc_i},\; \forall opi \;\in\; Op,\; M_{semantic}(op_i, f(op_i)) \;\in\; \{Exact \vee\; PlugIn\})\end{aligned}$$

■

We thus define interface semantic equivalence and almost equivalence between interfaces over a subset of operations:

Definition 26 — Interface semantic equivalence over a subset of operations

We define two interfaces ifc_i and ifc_j to be semantically equivalent over a subset of operations Op, $\equiv^{Op}_{semantic}$:

$$(\equiv^{Op}_{semantic}(ifc_i, ifc_j) \;=\; true) \;\Leftrightarrow\; (M^{Op}_{semantic}(ifc_i, ifc_j) \;=\; Exact)$$

■

Definition 27 — Interface semantic almost equivalence over a subset of operations

We define two services ifc_i and ifc_j to be semantically almost equivalent over a subset of equivalence Op, $\triangleright^{Op}_{semantic}$:

$$(\triangleright^{Op}_{semantic}(ifc_i, ifc_j) \;=\; true) \;\Leftrightarrow\; (M^{Op}_{semantic}(ifc_i, ifc_j) \;=\; PlugIn)$$

■

Example 8 *Coming back to our example in figure 3.33. The semantic matching between the different operations of $ifc1$ and $ifc2$ gives the following values:*

$$\begin{aligned}&\propto(ifc1, ifc3) \;=\; true\\ &\wedge\,(M_{concept}(op1_{ifc1}, op1_{ifc2}) \;=\; PlugIn)\\ &\wedge\,(M_{concept}(op2_{ifc1}, op3_{ifc2}) \;=\; PlugIn)\end{aligned}$$

From these matching values, we can implies $\Rightarrow$ $(\triangleright_{semantic}^{\{op1_{ifc1},\ op2_{ifc1}\}}(ifc1, ifc3) \ = \ true)$

The two interfaces $ifc1$ *and* $ifc3$ *are almost equivalent upon the two operations of* $ifc1$.

This equivalence and almost equivalence over subsets of operations is useful for service adaptation issues, as a service can be replaced by another one if certain operations are specified to be required at a given time. This aspect is detailed in section 3.3.3.

A ranking of the semantic matching values need to be introduced. This ranking will help ordering services that have semantic almost equivalence with different concept values for the respective operations' inputs, outputs and concepts. It is also used to rank interfaces and operations that have *Subsume* semantic matching. This operations' ordering allows users and applications to choose services that best suit their requirements at a given time, and re-adapt their choice if other services that have a closer semantic equivalence appear. We introduce a semantic distance $D_{semantic}$ between two interfaces. It calculates the distance between two interfaces semantic descriptions. The more this value is closer to zero the more these two services are equivalent.

Semantic distance

Definition 28 — Concept semantic distance

We first define a normalized concept distance $D_{concept}$ between two concepts n and m:

$$
\begin{aligned}
D_{concept}(n,m): \quad & 0 \ if \ M_{concept}(n,m) &= Exact \\
& 0.2 \ if \ M_{concept}(n,m) &= PlugIn \\
& 0.8 \ if \ M_{concept}(n,m) &= Subsume \\
& 1 \ if \ M_{concept}(n,m) &= Fail
\end{aligned}
$$

The more the distance is close to zero, the best is the value of the semantic matching between two concepts. An *Exact* value is preferred to a *PlugIn* one, which is preferred to a *Subsume* one.

■

Definition 29 — Operation semantic distance

We define the semantic distance between two comparable operations opi and opj ($\propto(opi, opj) = true$): ($D_{semantic}(opi, opj),\ i, j \in \mathbb{N}$). This semantic distance is the sum of the ponderated concept distance of the operation concept, inputs and output semantic description:

$$w_1 * D_{Concept}(n_{semantic_{cptopi}}, n_{semantic_{cptopj}}) + w_2 * D_{Concept}(n_{semantic_{out_{opi}}}, n_{semantic_{out_{opj}}}) + \sum_{k=1}^{|In_{opi}|}(w_k * D_{Concept}(n_{semantic_{ink_{opi}}}, n_{semantic_{f(ink_{opi})}}))$$

where $\sum_{i\in\mathbb{N}}(wi) = 1$

wi corresponds to the weight we wish to give to the concept, inputs and output. When matching two operations, the focus may be put on inputs, outputs parameters or on the concept. wi allows to ponderate the ranking of operations.

■

Definition 30 — Interface Semantic Distance

The semantic distance between two comparable interfaces ($D_{semantic}(ifc_i, ifc_i)\ i, j \in \mathbb{N}$) is the sum of all the semantic distance between their comparable operations, ponderated by a weight allowing to focus on some operations rather than others.

$$\sum_{k=1}^{|Op_{ifc_i}|}(w_k * D_{semantic}(opk_{ifc_i}, f(opk_{ifc_i})))$$

■

Example 9 *We come back to our example and calculate the semantic distance $D_{semantic}$ of our two interfaces:*

$$\begin{array}{rcl|l}
D_{semantic}(impression, printing) & = & 0.8 & D_{concept}("printer", "printer") = 0 \\
 & & & D_{concept}("location", "document") = 0.8 \\
 & & & D_{concept}("state", "state") = 0 \\
D_{semantic}(printing, impression) & = & 0.2 & D_{concept}("printer", "printer") = 0 \\
 & & & D_{concept}("document", "location") = 0.2 \\
 & & & D_{concept}("state", "state") = 0
\end{array}$$

The overall value of $D_{concept}(printing, impression) < D_{concept}(impression, printing)$.

In our semantic distance calculation, we gave the three items of an operation - inputs, output, and concept - the same importance. We can ponderate the semantic distance by introducing weights to each of the operation items.

As we can see the interface semantic equivalence is richer than the syntactic one. Interfaces are considered equivalent to a certain degree. Our semantic distance, allow to find, focusing on the priority an application want, the most suitable service as an equivalent to another one. This aspect is crucial in pervasive environment adaptation.

The equivalences introduced so far concern the interfaces of services. If two services can publish the same interface, they can provide different implementations and different non-functional properties. Next section, we define the equivalence upon the property describing the implementations followed by a metric to calculate the non-functional QoS equivalence degree for syntactic and semantic interface equivalence.

3.2.2.2 Property Implementation Equivalence Relations

We defined our syntactic and semantic interface equivalence and almost equivalence relations. Two services are composed not only of a functional interface but also of the implementations of these interfaces. In this section, we define the implementation equivalence and almost equivalence relations. We later motivate our choice of studying these relations. An implementation is described by the functional property of a service Pr. The properties describe the implementations of the functional interfaces and different properties mean different implementations of the services interface.

So, beside the syntactic and semantic interface equivalence relations defined above, a property equivalence and almost equivalence relations are introduced. A property pr_{opi} of the operation is a composed tuple $\{< semantic, value >\}^+$ where *semantic* is a tuple $< o, n >$ of an ontology $o \in O$, $n \in N$ a set of concepts belonging to the ontology set o, and *value* an expression denoting whether the service is *atomic* or *integrated* which means composed of several *atomic* services.

Definition 31 — Comparable properties $\propto$

We define two properties Pr_i and Pr_j to be comparable ($\propto (Pr_i, Pr_j) \; = \; true$) if they have the same number of tuples and if it exists a bijection f over their tuples allowing to compare the tuples two by two. $\forall \; k, l \; \in \; \{1..|Pr_i|\}$

$$|Pr_i| \; = \; |Pr_j|$$
$$\wedge \; (\exists \; f : Pr_i \; \rightarrow \; Pr_j, \; \forall \; pr_l \; \in \; Pr_j, \; \exists! \; pr_k \; \in \; Pr_i, \; f(pr_k) = pr_l)$$

■

Definition 32 — Property equivalence

We define two properties $Pr1$ and $Pr2$ to be equivalent ($\equiv_{Pr} \; (Pr1, Pr2) \; = \; true$) if they are comparable and for each tuple pr_k in $Pr1$

$$M_{concept}(n_{semantic_{pr_k}}, n_{semantic_{f(pr_k)}}) = Exact$$
$$\wedge \; (\equiv_{syntactic} \; (value_{pr_k}, value_{f(pr_k)}) \; = \; true)$$

■

Definition 33 — Property almost equivalence

We define two properties Pr_i and Pr_j to be almost equivalent ($\triangleright_{Pr}(Pr_i, Pr_j) \; = \; true$) if they if they are comparable and for each tuple pr_k in Pr_i:

$$M_{concept}(n_{semantic_{pr_k}}, n_{semantic_{f(pr_k)}}) \in \; \{Exact, PlugIn\}$$
$$\wedge \; (\equiv_{syntactic} \; (value_{pr_k}, value_{f(pr_k)}) \; = \; true)$$

■

Example 10 *Considering the following properties for two interfaces:*

$$\begin{aligned}
Pr_i &= < n2,\ integrated > (< n22,\ atomic >)\\
Pr_j &= < n21,\ integrated > (< n22,\ atomic >)\\
Pr_k &= < n1,\ integrated > (< n22,\ atomic >)\\
Pr_l &= < n2,\ integrated > (< n22,\ atomic >)(< n21,\ atomic >)
\end{aligned}$$

Using the definition relations introduced above we can compute the following values:

$$\begin{aligned}
\propto (Pr_l, Pr_{j/i/k}) &= false\\
\triangleright_{Pr}(Pr_i, Pr_j) &= true\\
M_{concept}(Pr_i, Pr_k) &= fail\\
M_{concept}(Pr_j, Pr_k) &= fail
\end{aligned}$$

As done earlier, we introduce the semantic distance to evaluate the equivalence distance between properties Pr_i and Pr_j. This metric is applied if previously the two properties are computed to be equivalent or almost equivalent.

Definition 34 — Property Semantic Distance

The semantic distance $D_{semantic}(Pr_i, Pr_j)$ between two properties Pr_i and Pr_j that are almost equivalent is: $\sum_{k=1}^{|Pr_i|}(D_{concept}(n_{semantic_{pr_k}}, n_{semantic_{f(pr_k)}}))$

■

Example 11 *If we calculate the property semantic distance for the two properties Pr_i and Pr_j verifying ($\triangleright_{Pr}(Pr_i, Pr_j) = true$) we obtain:*
$D_{semantic}(Pr_i, Pr_j) = D_{concept}(n2, n21) + D_{concept}(n22, n22) = 0.2 + 0 = 0.2$
If Pr_i is almost equivalent to another property Pr_z, the semantic distance will express how close is the similarity and allows to choose between the two properties Pr_j and Pr_z which one is closest to Pr_i.
Let $Pr_z = < n21,\ integrated > (< n221,\ atomic >)$
We have ($\triangleright_{Pr}(Pr_i, Pr_z) = true$) and:
$D_{semantic}(Pr_i, Pr_z) = D_{concept}(n2, n21) + D_{concept}(n22, n221) = 0.2 + 0.2 = 0.4$
Pr_j fits better Pr_i than does Pr_z as ($D_{semantic}(Pr_i, Pr_j) < D_{semantic}(Pr_i, Pr_z)$)

The more the semantic distance is closer to zero the best is the equivalence between the properties.

When comparing two properties, we first compare if they are equivalent or almost equivalent. For almost equivalent properties, we can calculate the semantic distance to order between the different almost equivalent properties.

3.2.2.3 Non-Functional QoS Equivalence Degree

Services can be syntactically and/or semantically equivalent, almost equivalent, or having *Subsume* matching relations. These equivalence are based on the functional aspect of services. Services can offer the same functionalities but with different non-functional QoS properties. We will define a metric that measures the non-functional QoS degree of equivalence. This metric allows to assign a normalized degree that measures the degree of non-functional QoS similarities between two equivalent, almost equivalent, or *Subsume* matching services. These degrees are used to choose between diverse services providing different non-functional QoS properties, but offering similar functionalities.

The non-functional QoS of an operation is defined as follows:

Definition 35 — non-functional QoS properties

Consider a finite set of grammatical alphabet Σ, ontologies O, concepts N belongings to these ontologies O, non-functional QoS properties Np, quantitative non-functional properties Np_{QN}, and qualitative non-functional properties Np_{QL}. Considering an operation *op* we define its non-functional QoS as follows:

$$Np =< Np^*_{QL}, Np^*_{QN} >$$
$$Np^*_{QL} = \{np1_{QL},\ np2_{QL},\ ..\ npk_{QL}\},\ k\ =\ |NP_{QL}|$$
$$Np^*_{QN} = \{np1_{QN},\ np2_{QN},\ ..\ npt_{QN}\},\ k\ =\ |NP_{QN}|$$
$$np_{QL} =< name, semantic >,\ name\ \in\ \Sigma^*$$
$$np_{QN} =< name, numericValue, operator >,\ name\ \in\ \Sigma^*\ \&\ numericValue\ \in\ \mathbb{R}$$
$$operator = \{<,\ >,\ \leq,\ \geq\}$$
$$semantic =< o, n >,\ o\ \in\ O,\ n\ \in\ N$$

operator specifies the order applied to *numericValue*. For $\{>,\ \geq\}$ the greater the *numericValue* is, the best is the QoS property for the service runtime execution. For $\{<,\ \leq\}$ the smaller the *numericValue* is, the best is the QoS property for the service runtime execution.

■

The non-functional equivalence degree $QoS_{Degree}(opi, opj)$ between two functional equivalent operations is evaluated upon their quantitative and qualitative properties similarities. Two functional equivalent operations offer the same functionality but not necessarily the same non-functional QoS properties. The $QoS_{Degree}(opi, opj)$ evaluates the degree of similarities of two operations *opi* and *opj* concerning their non-functional QoS properties. We suppose that $\exists\ f\ :\ Np_{opi}\ \rightarrow\ Np_{opj}\ where\ \forall\ npk_{opj}\ \in\ Np_{opj},\ \exists!\ npk_{opi}\ \in\ Np_{opi},\ f(npk_{opi})\ =\ npk_{opj}$. We suppose $\forall\ k\ \in\ \mathbb{N}$, npk_{opi} and npk_{opj} deals with the same non-functional QoS property. If npk_{opi} is a quantitative non-functional QoS we have npk_{opj} also a quantitative non-functional QoS and $name_{npk_{opi}}\ =\ name_{npk_{opj}}$. If npk_{opi} is a qualitative non-functional QoS we have npk_{opj} also a qualitative non-functional QoS and $name_{npk_{opi}}\ =\ name_{npk_{opj}}$.

Definition 36 — $QoS_{Degree}(opi, opj)$

Considering two operations *opi* and *opj*, we define the degree of equivalence between the two operations $QoS_{Degree}(opi, opj)$ as a function that measures how close is *opj* from *opi* in terms of non-functional QoS. We consider the non-functional properties of *opi*, NP_{opi} and calculate as follows the degree of equivalence *opj* has upon these properties:

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

where, w_k is the assigned weight for a particular non-functional QoS property with the following conditions $\sum_{k=1}^{|Np_{opi}|}(w_k) = 1$. The more w_k is closer to zero, the more important is the property Npk. This ponderation allows to decide when searching for equivalent services if certain non-functional QoS properties are more important than other for the required service replacement. $deg(npk_{opi}, npk_{opj})$ are normalized values between 0 and 1 corresponding to the equivalence degree between npk_{opi} and npk_{opj}. These values are calculated using the z-score or standardization of the *npk* values for quantitative properties and semantic distance for qualitative properties.

■

We define $deg(npk_{opi}, npk_{opj})$ as follows:

- $deg(npk_{opi}, npk_{opj}) = deg(npk_{QN_{opi}}, npk_{QN_{opj}})$ for the quantitative properties.
- $deg(npk_{opi}, npk_{opj}) = deg(npk_{QL_{opi}}, npk_{QL_{opj}})$ for the qualitative ones.

We define next how we calculate these two degrees.

Definition 37 — $deg(npk_{QN_{opi}}, npk_{QN_{opj}})$

$deg(npk_{QN_{opi}}, npk_{QN_{opj}}) = |\eta(npk_{QN_{opi}}) - \eta(npk_{QN_{opj}})|$
We define $\eta(npk_{QN})$ as the normalization of z-score value of npk_{QN} for quantitative non-functional QoS.

■

Definition 38 — $\eta(np_{QN})$

Considering $np_{QN} =< name, numericValue, operator >$ we define $\eta(np_{QN})$ as follows

$$\begin{array}{ll} if\ operator_{np_{QN}}\ is\ '<' : & 0\ if\ z\text{-}score(np_{QN}) < -2 \\ & 1\ if\ z\text{-}score(np_{QN}) > 2 \\ & (z - score(np_{QN}))/4 + 0.5\ if\ 2 > z\text{-}score(np_{QN}) > -2 \\ if\ operator_{np_{QN}}\ is\ '>' : & 1\ if\ z\text{-}score(np_{QN}) < -2 \\ & 0\ if\ z\text{-}score(np_{QN}) > 2 \\ & 0.5 - (z\text{-}score(np_{QN}))/4\ if\ 2 > z\text{-}score(np_{QN}) > -2 \end{array}$$

■

For $<$ the $numericValue$ is the best when it is the smallest. $\eta(np_{QN})$ is closer to zero for the smallest value of $numericValue$ and closer to one for the bigger value of $numericValue$, and vice versa for $>$.

The z-score of a quantitative property np_{QN}, indicates how far and in what direction, the property deviates from its distribution's mean, expressed in units of its distribution's standard deviation. We use the z-score standardization in order to provide a way of comparing all the different non-functional QoS by including consideration of their respective distributions.

Definition 39 — *z-score*(np_{QN})

Considering the quantitative np_{QN}, its corresponding z-score is:

$$z\text{-}score(np_{QN}) = (numericValue_{np_{QN}} - \mu(numericValue_{np_{QN}}))/\sigma(numericValue_{np_{QN}})$$

where, $\mu(numericValue_{np_{QN}})$ is the mean of the values of np_{QN}, and $\sigma(numericValue_{np_{QN}})$ is the standard deviation of np_{QN}.

■

In normal distribution we can distinguish that the 95% of z-score(np_{QN}) values are comprises between -2 and 2. Based on this, $\eta(np_{QN})$ calculates a value between 0 and 1 taking into account the nature of quantitative non-functional QoS properties. Indeed the $operator_{np_{QN}}$ indicates whether the properties are stronger with greater values, or with smaller values.

If for the quantitative non-functional QoS properties, we used z-score and normalization to calculate the degree of similarities between two properties, for qualitative non-functional QoS we use the semantic distance to compare the concepts of the qualitative properties np_{QL}. The semantic distance returns a normalized value between 0 and 1.

Definition 40 — $deg(npk_{QL_{opi}}, npk_{QL_{opj}})$

Considering $npk_{QL_{opi}}$ the qualitative non-functional QoS of the operation. We seek to find the best equivalence for it from a set of equivalent operations. Considering $npk_{QL_{opj}} =< name, semantic >$ the qualitative non-functional QoS of the other operations. we define $deg(npk_{QL_{opi}}, npk_{QL_{opj}})$ as follows:

$$deg(npk_{QL_{opi}}, npk_{QL_{opj}}) = D_{semantic}(n_{semantic_{npk_{QL_{opi}}}}, n_{semantic_{npk_{QL_{opi}}}})$$

■

Example 12 *Considering the three operations defined in figure 3.13. Considering the Printing operation, it is syntactically equivalent to Printer and semantically equivalent to Impression. We calculate the non-functional QoS degree of equivalence to determine which of Printer or Impression replace the best Printing.*

First we calculate the values that we need for our degree calculations. We detail the calculations for nbpage.

$$\mu(nbpage) \ = \ 56.66$$
$$\sigma(nbpage) \ = \ \sqrt{((60-56.66)^2 \ + \ (100-56.66)^2 \ + \ (10-56.66)^2) \ \div \ 3} \ = \ 36.84$$
$$z-score(nbpage_{printing}) \ = \ (60-56.66) \ \div \ 36.84 \ = \ 0.09$$
$$z-score(nbpage_{impression}) \ = \ (100-56.66) \ \div \ 36.84 \ = \ 1.176$$
$$z-score(nbpage_{printer}) \ = \ (10-56.66) \ \div \ 36.84 \ = \ -1.26$$
$$\eta(nbpage_{printing}) \ = \ 0.477$$
$$\eta(nbpage_{impression}) \ = \ 0.206$$
$$\eta(nbpage_{printer}) \ = \ 0.816$$

$$\eta(price_{printing}) \ = \ 0.515$$
$$\eta(price_{impression}) \ = \ 0.867$$
$$\eta(price_{printer}) \ = \ 0.186$$

$$D_{semantic}(access_{printing}, access_{printing}) \ = \ 0, \ M_{concept}('wifi','wifi') \ = \ Exact$$
$$D_{semantic}(access_{printing}, access_{impression}) \ = \ 0.2, \ M_{concept}('wireless','wifi') \ = \ PlugIn$$
$$D_{semantic}(access_{printing}, access_{printer}) \ = \ 1, \ M_{concept}('bluetooth','wifi') \ = \ Fail$$

The QoS_{Degree} of the three operations are:

$QoS_{Degree}(Printing, Impression) \ = \ w1 * (|\eta(nbpage_{printing}) \ - \ \eta(nbpage_{impression})|) \ + \ w2 * (|\eta(price_{printing}) \ - \ \eta(price_{impression})|) \ + w3 * (D_{semantic}(access_{printing}, access_{impression}))$

$QoS_{Degree}(Printing, Impression) = w1 * 0.27 \ + w2 * 0.35 \ + w3 * 0.2$

$QoS_{Degree}(Printing, Printer) \ = \ w1 * (|\eta(nbpage_{printing}) \ - \ \eta(nbpage_{printer})|) \ + \ w2 * (|\eta(price_{printing}) \ - \ \eta(price_{printer})|) \ + w3 * (D_{semantic}(access_{printing}, access_{printer}))$

$QoS_{Degree}(Printing, Printer) = w1 * 0.33 \ + w2 * 0.33 \ + w3 * 1$

If we suppose the three non-functional QoS properties of the same importance $w1 + w2 + w3 = 1$, we obtain: $QoS_{Degree}(Printing, Impression) = 0.27$, and $QoS_{Degree}(Printing, Printer) = 0.55$. The Impression operation offers non-functional QoS that are closer to Printing than Printer if we assign the same weight to the three non-functional properties.

3.2.3 Services Composition Relations

After the service equivalence relations definitions, we define the service composition relations. Composition relation determines the degree of complementarity for a composition between two services. A service is considered to be composable with another service, if the functionality it produces can be consumed by the other service, and if the non-functional properties are compatible. Combining two services together means combining their functional interfaces, and by that at least one of their operations. The output produced by the operation of one service is used as an input for the operation of another service. To be able to compose services, non-functional QoS need to be taken into account. We distinguish two kinds of operations' composition compatibility: syntactic and semantic composition.

We illustrate our composability relations definitions upon the following example of two operations that take a picture via a webcam and that change the dimension of a given image file:

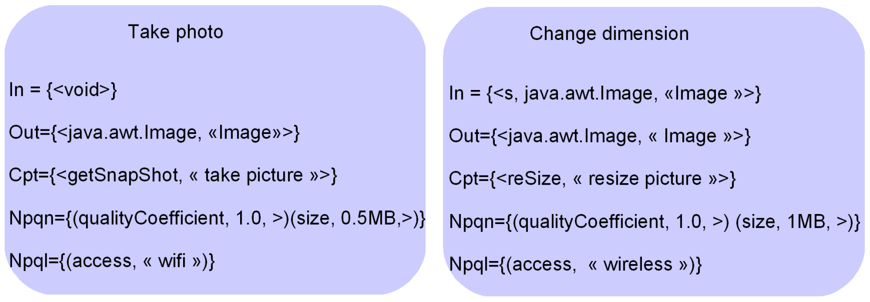

Take photo
In = {<void>}
Out={<java.awt.Image, «Image»>}
Cpt={<getSnapShot, « take picture »>}
Npqn={(qualityCoefficient, 1.0, >)(size, 0.5MB,>)}
Npql={(access, « wifi »)}
Change dimension
In = {<s, java.awt.Image, «Image »>}
Out={<java.awt.Image, « Image »>}
Cpt={<reSize, « resize picture »>}
Npqn={(qualityCoefficient, 1.0, >) (size, 1MB, >)}
Npql={(access, « wireless »)}

Figure 3.15: Two services operation specification

3.2.3.1 Functional Services Composition Relation

We first begin by defining the functional service composability relation. This relation is upon the functional interfaces of services. We distinguish two kinds of composability, the syntactic one and the semantic one.

Syntactic composition relation

Two services are syntactically composable ($O_{syntactic}$) if they have two syntactic composable functional interfaces. Two functional interfaces are defined to be syntactic composable if they have at least two syntactic composable operations. Two operations are syntactic composable if the output of one operation is of the same type as one input of the other operation (cf. fig 3.16). For a syntactic composition relation, we suppose that the services use the same protocol of interaction as this compatibility is based on type matching.

Definition 41 — Syntactic service composability

si and sj are syntactic service composable $(O_{syntactic}(si, sj) = true)$ if:
$\exists\, opk \in Op_{si},\ \exists\, opl \in Op_{sj},\ (O_{syntactic}(opk, opl) = true)$
$\wedge\ (\equiv_{syntactic}(protocol_{si}, protocol_{sj}) = true)$

■

DEFINITION 42 — Syntactic operation composability

opi and *opj* are syntactically composable ($O_{syntactic}(opi, opj) = true$) if:

$$(\exists l \in \{1 \ldots |In_{opj}|\}, (\equiv_{type} (type_{inl_{opj}}, type_{out_{opi}}) = true) \vee (\triangleright_{type}(type_{out_{opi}}, type_{inl_{opj}}) = true)$$

■

Figure 3.16 shows two syntactic composition compatible operations *opi* and *opj*. For that *opi* to be composable with *opj*, the type output of *opi* need to be equivalent or almost equivalent with the type of one input of *opj*. By this way the two operations can be combined.

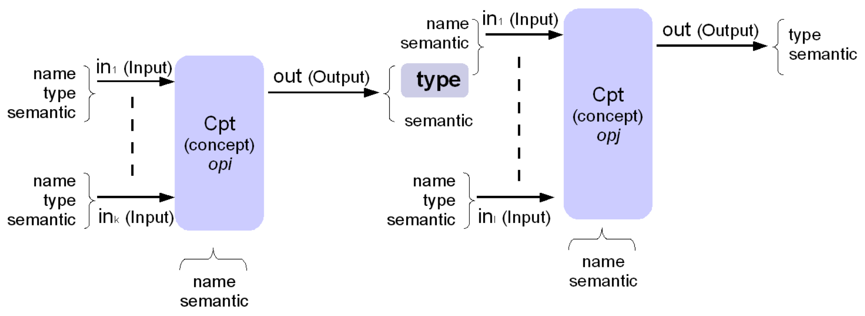


Figure 3.16: Combining compatible operations: *opi* & *opj*

Composing the two operations *opi* and *opj* creates a new operation with a new signature as showed figure 3.17. The new operation inputs are a combination of the *opi* and *opj* inputs, and its output the output of *opj*.

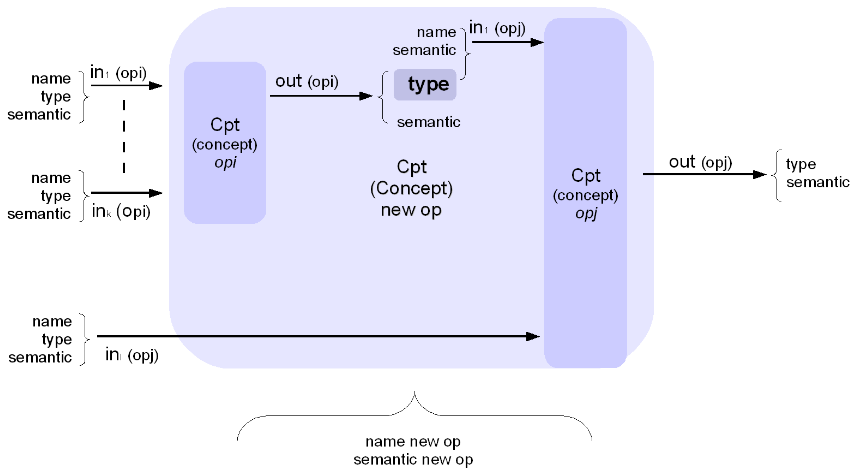


Figure 3.17: new resulting operation

The new resulting signature operation in figure 3.17:

$$\begin{aligned} & type_{In} = type_{In_{opi}} \cup (type_{In_{opj}} - \{type_{ink_{opj}}\}),\ (\equiv_{type} / \triangleright_{type} (type_{Out_{opi}}, type_{ink_{opj}}) = true) \\ & type_{Out} = type_{Out_{opj}} \\ & Cpt =< name_{opi} \bullet name_{opj},\ semantic_{opi} \bullet semantic_{opj} > \end{aligned}$$

The non-functional aspects are explained in section 3.2.3.2.

EXAMPLE 13 *The operation* $op1$: *Takephoto takes a picture using a webcam device. The operation* $op2$: *Changedimension resize a given image and returns an image. Both operations are written using the Java language. Takephoto and Changedimension are composition compatible as the* $Out_{Takephoto}$ *is of the same type of* $In_{Changedimension}$. *This new operation takes a picture and resize it. The new operation specification is:*

$$\begin{aligned} & type_{In} = type_{In_{op1}} \cup (type_{In_{op2}} - \{type_{in1_{op2}}\}) = void \\ & type_{Out} = type_{Out_{op2}} = java.awt.Image \\ & Cpt =< GetSnapShotreSize,\ "takepictureresizepicture" > \end{aligned}$$

The new resulting operation signature:

$$signatureOp_{new} : \ GetSnapShotreSize() \longrightarrow java.awt.Image$$

Semantic composition

Two services are semantically composition compatible ($O_{semantic}$) if they have two composable semantic description of their functional interfaces. Two functional interfaces are defined to be semantic composable if they have at least two semantic composable operations. Two operations are semantic composable if the semantic description of the output of one operation is semantically equivalent to the semantic description of one input of the other operation. At the difference with the syntactic composability, the semantic one does not impose services to have the same protocol of interaction as the semantics aim to resolve the interoperability problem.

DEFINITION 43 — Semantic Service Composability

si and sj are semantically composable ($O_{semantic}(si, sj) = true$) if:

$$\exists\ opi \in Op_{si},\ \exists\ opj \in Op_{sj},\ O_{semantic}(opi, opj) = true$$

■

DEFINITION 44 — Semantic operation composability

opi and opj are semantically composable ($O_{semantic}(opi, opj) = true$) if:

$$(\exists l \in \{1 \ldots |In_{opj}|\}, (M_{semantic}(n_{semantic_{inl_{opj}}}, n_{semantic_{out_{opi}}}) \in \{Exact, PlugIn)\}$$

■

The output produced by *opi* can be transparently consumed as an input for the operation *opj* if the semantic equivalence between their two concepts is *Exact* or *PlugIn*. In case of a *Subsume* equivalence, the output can fail to provide the requirements for the input.

Figure 3.18 shows two composition compatible operations *opi* and *opj*.

Combining the two operations *opi* and *opj* creates a new operation with a new semantic description as showed figure 3.18

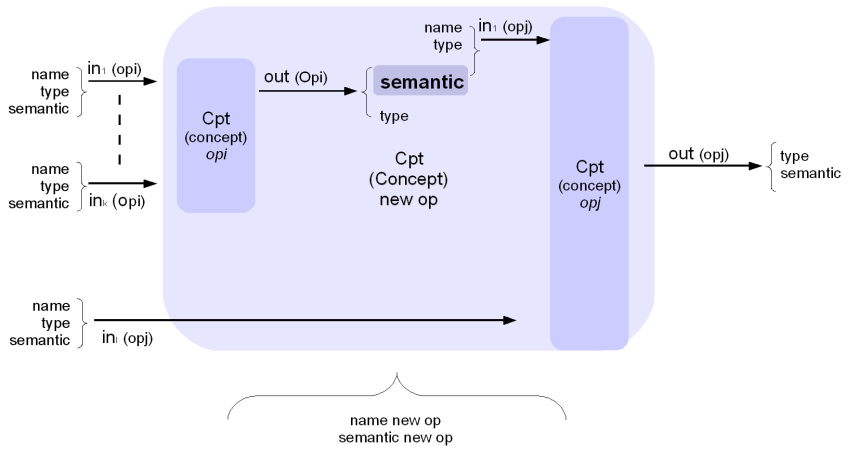


Figure 3.18: new resulting operation

The new resulting semantic operation description in figure 3.18:

$$\begin{aligned}
&n_{semantic_{In}} = n_{semantic_{In_{opi}}} \cup (n_{semantic_{In_{opj}}} - \left\{n_{semantic_{ink_{opj}}}\right\}) \\
&n_{semantic_{Out}} = n_{semantic_{Out_{opj}}} \\
&Cpt =< name_{opi} \bullet name_{opj},\ semantic_{opi} \bullet semantic_{opj} >
\end{aligned}$$

where, $(M_{concept}(n_{semantic_{ink_{opj}}}, n_{semantic_{out_{opi}}}) \in \{Exact, PlugIn\})$.

EXAMPLE 14 *We illustrate the composition compatibility using our example in figure 3.15. The new operation specification is:*

$$\begin{aligned}
&n_{semanticIn} = n_{semanticIn_{Op1}} \cup (n_{semanticIn_{Op2}} - \{n_{semanticIn1_{Op2}}\}) = ''none'' \\
&n_{semanticOut} = n_{semanticOut_{Op2}} = ''Figure'' \\
&Cpt = \{< GetSnapShotreSize,''takepictureresizepicture'' >\}
\end{aligned}$$

The new resulting operation semantic description:

$$''takepictureresizepicture''\ (''none'') \longrightarrow\ ''Figure''$$

we can note that the syntactic operation composability and the semantic one tackle different levels of services compatibility. The syntactic one allows sub-types to replace their types, whereas semantic allows super-concept to replace their concepts (based on Paolucci [Paolucci et al. 2002] definition). We can stipulate that combining these two kinds of composability offers a large spectrum to compose services together.

3.2.3.2 Non-Functional QoS Services Composition Relation

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

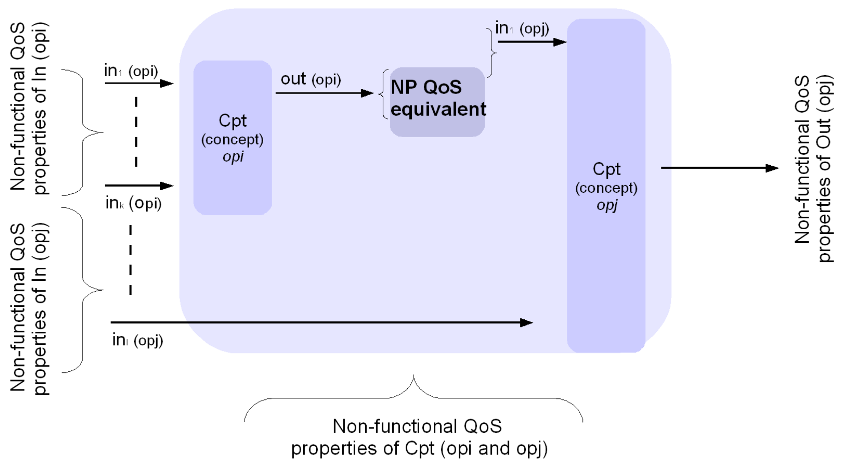

Non-functional QoS properties of In (opi)
Non-functional QoS properties of In (opj)
in₁ (opi)
inₖ (opi)
inₗ (opj)
Cpt (concept) opi
out (opi)
NP QoS equivalent
in₁ (opj)
Cpt (concept) opj
Non-functional QoS properties of Out (opj)
Non-functional QoS properties of Cpt (opi and opj)

Figure 3.19: new resulting operation non-functional QoS properties

We define the QoS compatibility between two non-functional QoS properties that express the same property.

DEFINITION 45 — $Compatible_{QoS}$ for np_{QN}

npi_{QN} and npj_{QN} are compatible ($Compatible_{QoS}(npi_{QN}, npj_{QN}) = true$) if:

$$name_{npi_{QN}} = name_{npj_{QN}}$$
$$\wedge\ (numericValue_{npi_{QN}} \subset numericValue_{npj_{QN}})$$

$\subset$ is $\{< \vee \leq\}$ for $numericValue \in \mathbb{R}$ or $\subseteq$ for $numericValue$ intervals in $\mathbb{R}$.

■

Definition 46 — $Compatible_{QoS}$ for Np_{QN}

Npi_{QN} and Npj_{QN} are compatible ($Compatible_{QoS}(Npi_{QN}, Npj_{QN}) = true$) if:

$$|Npi_{QN}| = |Npj_{QN}|$$
$$\forall npi_{QN} \in Npi_{QN},\ \exists!\ npj_{QN} \in Npj_{QN},\ Compatible_{QoS}(npi_{QN}, npj_{QN}) = true$$

npi_{QL} and npj_{QL} are compatible ($Compatible_{QoS}(npi_{QL}, npj_{QL}) = true$) if:

$$name_{npi_{QL}} = name_{npj_{QL}}$$
$$\wedge\ (M_{concept}(n_{semantic_{npi_{QL}}}, n_{semantic_{npj_{QL}}}) \in (Exact, PlugIn))$$

■

Definition 47 — $Compatible_{QoS}$ for Np_{QL}

Npi_{QL} and Npj_{QL} are compatible ($Compatible_{QoS}(Npi_{QL}, Npj_{QL}) = true$) if:

$$|Npi_{QL}| = |Npj_{QL}|$$
$$\forall npi_{QL} \in Npi_{QL},\ \exists!\ npj_{QL} \in Npj_{QL},\ Compatible_{QoS}(npi_{QL}, npj_{QL}) = true$$

■

Two operations *opi* and *opj* need to provide compatible non-functional QoS properties over the input and output parameters it combines. Besides the functional composability the following conditions needs to be verifed
($Compatible_{QoS}(Np_{out_{opi}}, Np_{ink_{opj}}) = true$) where out_{opi} and ink_{opj} are syntactic or semantic equivalent or almost equivalent.
The new resulting operation will republish the same non-functional QoS properties of the two composed operations *opi* and *opj* but hiding those properties that deal with the composed input and output.

$$Np = (Np^*_{opi} \cup Np^*_{opj}) - \{(Np_{out(opi)}, Np_{ink(opj)})\}$$

Example 15 *In our composition example, the Np_{QN} non-functional QoS properties of the composition concern the qualityCoefficient and size of the picture output returned by getSnapShot and of the image input consumed by reSize. In the example, the non-functional QoS are composable as getSnapShot returns an image of $qualityCoefficient = 1.0$ in conformance of what reSize need, and $size = 5MB < 10MB$. The new operation non functional-properties is thus:*

$$NP_{QL} = (access, wifi)$$

The NP_{QN} related to the $Out_{Takepicture}$ and $In_{Changedimension}$ are not published by the new service as they deal with internal aspect of the new composed operation.

Section Conclusion

In this section, we presented our general SERVICE model and formalism. We detailed several services relations that represent the basis for service adaptation and composition. Our services equivalence and almost equivalence relations, syntactic or semantic, at different levels (functional and non-functional QoS) is used for our service adaptation. Our service composition relations, syntactic or semantic, taking into account all the particularities of our SERVICE model (functional and non-functional QoS properties) is used for our service composition. This section showed how we manage the QoS of services when defining the services relations. In the next section we explain the spontaneity concept for service integration. Based on the services relations defined in this section, we explain the spontaneous service composition, and the spontaneous service adaptations provided by both the `Decision Service` and the `Registry Service`.

3.3 Spontaneous Functional and Non-Functional QoS Service Integration

In this section, we present our spontaneous service integration that takes the functional and non-functional QoS properties of services into account. We begin by explaining the spontaneity approach versus the goal-oriented one. We then explain our spontaneous service composition with its runtime life cycle and techniques, followed by the spontaneous service adaptation with its life cycle and techniques. As already mentioned, the service transformation is described in the implementation chapter as it is more related to the chosen service technology.

3.3.1 Spontaneity Versus Goal-Oriented Service Integration

`MySIM` middleware integrates the services of the environment in a dynamic, proactive, and smart way. The spontaneity of `MySIM` integration is in providing users and applications with new services (new functionalities) but in a completely transparent way and without previous demand or external control over the integration. Indeed, a spontaneous integration is not initially required by users and applications, and the proactivity of our framework need to be smart enough to hide the integration results from users and at the same time provide them with the best capabilities their environment can offer. This spontaneous service integration is done technically in three ways (cf. figure 3.20): spontaneous service transformation, spontaneous service composition, and spontaneous service adaptation.

The spontaneous service transformation enables a given service expressed and provided in a predefined technology to be transformed into another technology model. This service transformation allows applications to re-use the services even if provided in different technology model. This transformation is done in two steps. First, the services are mapped to the generic SERVICE model (introduced section 3.2.1), then these services are implemented by the `Builder Service` in the other chosen technology. In our implementation chapter, we explain transformation rules to transform services implemented in OSGi service specification to the SERVICE model, and how our `MySIM` middleware generates services specified with our SERVICE model into OSGi services.

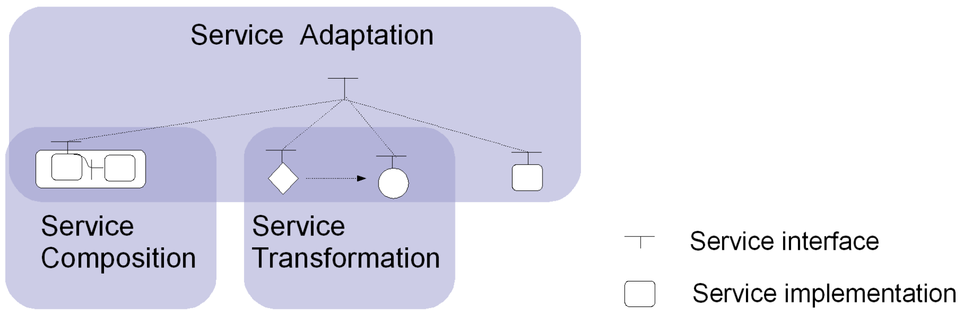


Figure 3.20: Spontaneous service integration

The spontaneous service composition enables services to be composed two by two, creating by that new services that combine all the possible functionalities of the environment. For this composition to be transparent for users and applications, the new resulting "composed" services need to publish the same interfaces as the available services in the environment including the services involved in the composition. Indeed, users and applications access services via their functional interfaces and their semantic descriptions. Providing new services with the same interfaces but different implementations and hence different functionalities remain transparent for users. Our spontaneous service composition seeks for all possible compositions between services leaving the interfaces unchanged and extending by that the environment with new functionalities.

The spontaneous service adaptation adapts the service execution at runtime, by transparently insuring to users and applications a viable service for their executions. If a service is unavailable for any reason, our `MySIM` searches for functionally equivalent services in the environment, to replace these services without disturbing users and applications execution. Users and applications have access to the services interface, and by accessing the same interface all the time, they are not aware of the implementations changes done behind.

We define our spontaneous service integration fig. 3.21 as a spontaneous integration of several services that returns new services with the same well known interfaces for the users and applications, but different functionalities, implementations, and non-functional QoS. This service integration is not a response to a user or application request as in a goal-oriented service composition, but nevertheless it returns functionalities, initially not available in the environment, still accessible via already existing interfaces. The environments that are enriched with our `MySIM`, extend and shrink with services implementations depending on what is available in the environment at a certain time. For the users and applications, the environment still publishes the same functional interfaces as before the spontaneous integration (cf. figure 3.21).

This new way of spontaneously integrating services vis-à-vis to users and applications of the application layer is triggered, managed, and orchestrated by the `Decision Service` of `MySIM` middleware. The `Decision Service` needs to respond to the following questions: *When* to apply the spontaneous integration? and *how* to transparently integrate services?

In a goal-oriented integration a specific demand need to be formulated by applications for executing an integration. Users specify their demands explicitly and the environment tries to respond

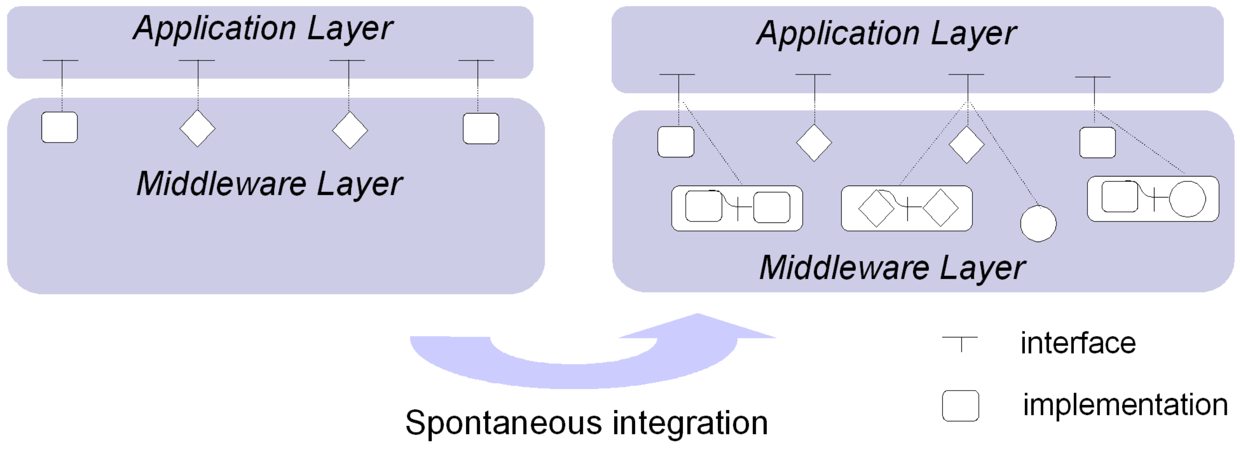


Figure 3.21: Services environment before and after spontaneous integration

to these demands by integrating the available services, if no atomic services can directly respond to these demands. On the contrary, the spontaneity of the integration is more related to specific events occurring in the environment and not controlled and required by users or applications as in goal-oriented service integration. We distinguish two major events that can affect an environment in term of functionalities:

- New services appearing in the environment leading to an automatic execution of the spontaneous integration (transformation, composition, and adaptation) that extends the environment with their new functionalities.

- Services leaving the environment that lead to spontaneous service adaptation for users and applications. Service adaptation replaces the vanishing services with others publishing the same interfaces but not necessarily the same implementations and non-functional QoS properties.

The notion of new services is different whether it is viewed from the application layer or the middleware layer. For the middleware, a new service is a service offering new functionalities whether by providing new functional interfaces or new implementations to pre-existing functional interfaces. A service offering functional interfaces that are already available in the environment, is considered new by the framework if the functional properties associated to the service are new in the context. Whereas for users and applications this service is not new as it publishes well known interfaces even if providing different implementations. Based on these observations we define new services from our `MySIM` point of view as follows:

Definition 48 — `MySIM` new service s ($N_{MySIM}(s) = true$)

Considering S the set of services available in the environment. A service $s \in S$ is considered new to `MySIM` middleware ($N_{MySIM}(s) = true$) if:

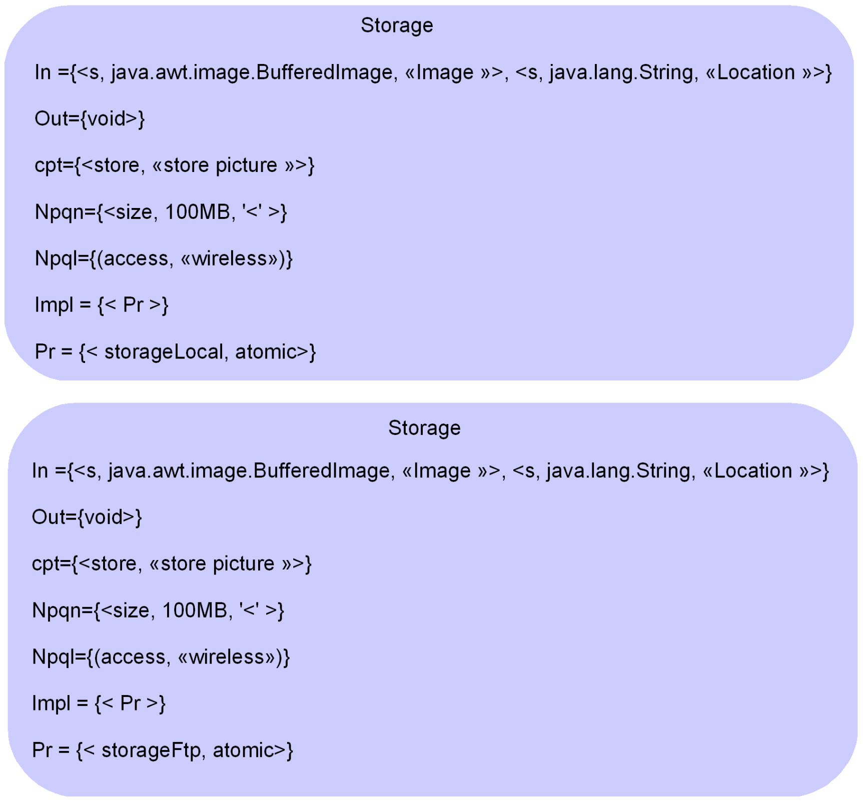
$(\forall si \in S, \equiv_{syntactic} (si, s) = false) \vee (if\ (\equiv_{syntactic} (s, si) = true) \Rightarrow \equiv_{Pr} (si, s) = false)$

$\vee$

$(\forall si \in S, \equiv_{semantic} (si, s) = false) \vee (if\ (\equiv_{semantic} (si, s) = true \Rightarrow \equiv_{Pr} (si, s) = false)$

■

Example 16 *Considering the two services storage, having each one an operation that stores images on hardwares (cf. figure 3.22):*



Storage

In ={<s, java.awt.image.BufferedImage, «Image »>, <s, java.lang.String, «Location »>}

Out={void>}

cpt={<store, «store picture »>}

Npqn={<size, 100MB, '<' >}

Npql={(access, «wireless»)}

Impl = {< Pr >}

Pr = {< storageLocal, atomic>}

Storage

In ={<s, java.awt.image.BufferedImage, «Image »>, <s, java.lang.String, «Location »>}

Out={void>}

cpt={<store, «store picture »>}

Npqn={<size, 100MB, '<' >}

Npql={(access, «wireless»)}

Impl = {< Pr >}

Pr = {< storageFtp, atomic>}

Figure 3.22: Two storage services

The two services have the same operation signature, and semantic descriptions, but two different implementations. The first storage service proposes to store locally images, the second one proposes a remote storage via ftp procedures. These two services can be distinguished by the value of their properties. The property describes the implementation of a specific interface and different implementations correspond to different properties. For `MySIM` middleware, these two storage services are different but for users and applications these two services are the same.

The *how* to transparently integrate services for users and applications is thoroughly explained for the service composition in section 3.3.2 and for the service adaptation in section 3.3.3.

Upon their appearance, new services (whether provided by developers or resulting from services integration), register their interfaces and their respective properties in a registry maintained by the `Registry Service`. This `MySIM` service is aware of all the available services, of the associations between their interfaces and properties, and of the runtime topology of the environment. For every appearance, disappearance of services, it reports these events information to the `Decision Service` which is responsible of triggering the adequate integration technique. In the case of a service appearance, if the service is new for `MySIM` middleware, spontaneous transformations, service compositions, and service adaptation are triggered. If the service already exists no composition is proceeded. If the service does not fit better applications than the already existing services, no adaptation is proceeded. In case of service disappearance, the `Decision Service` spontaneously tries to adapt the execution of applications to this disappearance by replacing the service with equivalent or almost equivalent ones. Service resulting from a spontaneous composition between this services and others need to stopped. In the rest of this chapter, we detail our spontaneous service composition, followed by the spontaneous service adaptation. The service transformation is explained in the next chapter, chapter 4.

3.3.2 Spontaneous Service Composition

The services taking part in realizing the spontaneous composition are the `Generator Service`, the `QoS Service` and the `Builder Service` (cf. figure 3.23). The `Generator Service` generates all possible spontaneous services composition between the services available in the environment. We mean by spontaneous service composition, service composition that returns equivalent or almost equivalent services comparing to already existing services in the environment. These abstract possibilities are passed to the `QoS Service` for inspection and non-functional QoS services decoration. The `QoS Service` verifies that the composition is also valid for non-functional QoS properties of services, and assign the new abstract resulting services the adequate non-functional QoS properties. The `Decision Service` verifies via the property of services that no duplication is made. That means, no services are composed twice by the `MySIM` middleware. The final valid combinations that return transparent functional and non-functional compositions for users are passed to the `Builder Service` for implementations.

In section 3.3.2.1 we detail the runtime life-cycle of a spontaneous service composition cycle, followed in section 3.3.2.2 of a detailed explanation of the employed techniques to realize the effective spontaneous service composition.

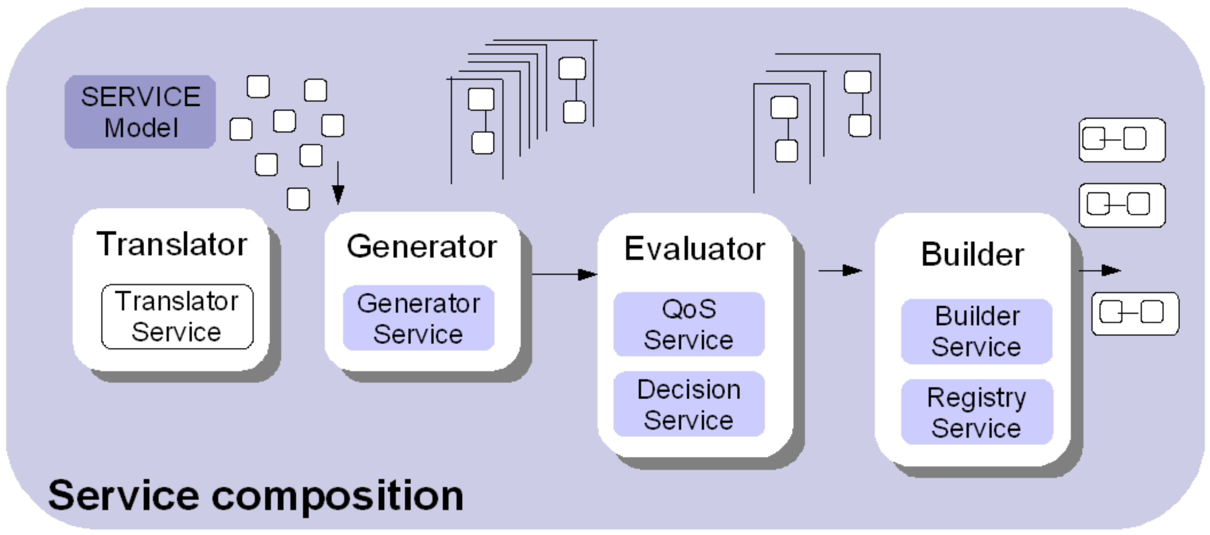


Figure 3.23: Services achieving the technical spontaneous service composition

3.3.2.1 Spontaneous Service Composition Life Cycle

To achieve the spontaneous service composition, every service of the `MySIM` services will have to accomplish a specific task.

In figure 3.24, the different steps of a cycle of a spontaneous service composition are described. We distinguish several important steps:

1. Services technologies transformation: The various service technologies available in the environment are transformed into our generic SERVICE model. The mapping transformation rules are explained in the implementation chapter, chapter 4.

2. Syntactic and semantic abstract service composition: The `Generator Service` matches all the services syntactically and/or semantically and returns all the possible transparent combinations that returns equivalent or almost equivalent services to already existing services in the environment. This is thoroughly explained in section 3.3.2.2

3. These spontaneous service composition possibilities are passed to the `QoS Service` to verify the non-functional QoS between the composed services and to assign the adequate non-functional QoS to the chosen composition results. This is thoroughly explained in section 3.3.2.2

4. The `Decision service` controls the service composition proposition. It uses the semantic to control the syntactic composition but also the property to control that no service duplication is made.

5. The final abstract service compositions are passed to the `Builder Service` for service implementation and generation. This aspect is detailed in the implementation chapter.

6. The newly available services are deployed in the environment and register their interfaces under their new properties through the `Registry Service`.

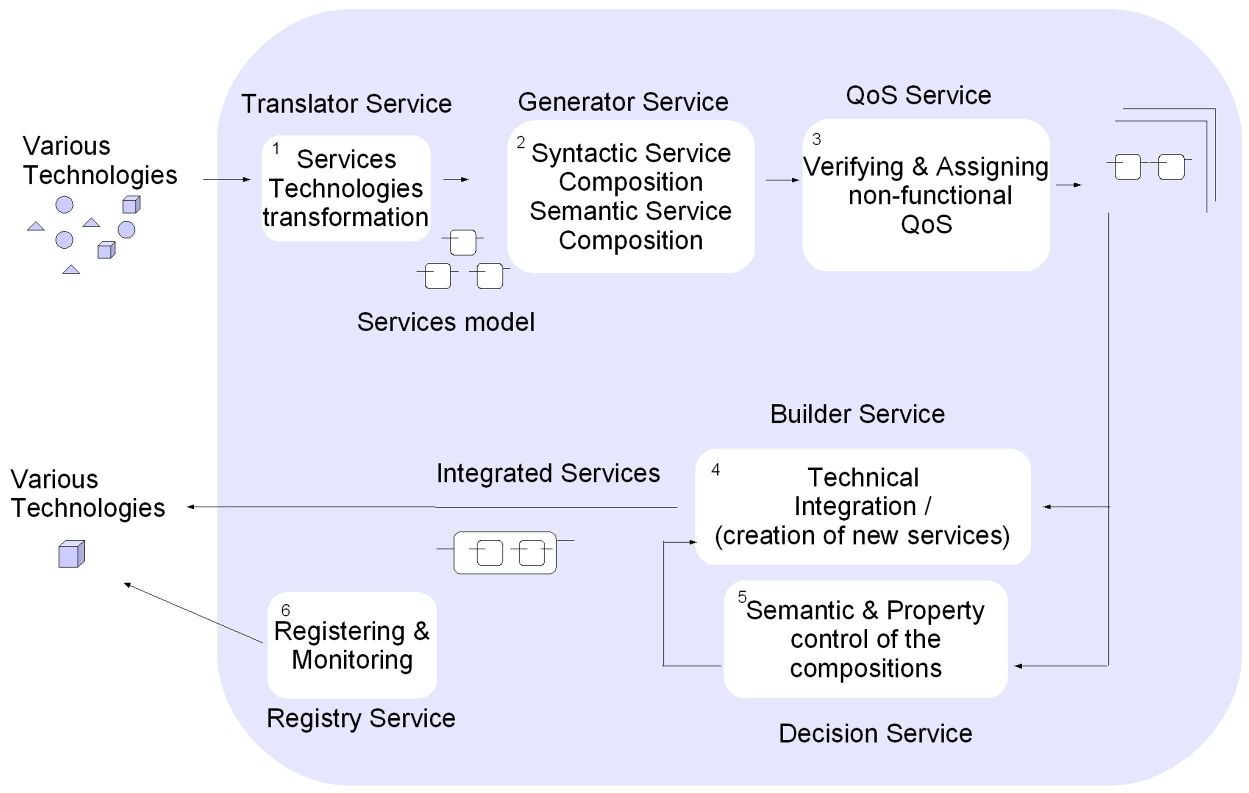

Translator Service
Generator Service
QoS Service
Various Technologies
1 Services Technologies transformation
2 Syntactic Service Composition Semantic Service Composition
3 Verifying & Assigning non-functional QoS
Services model
Builder Service
Various Technologies
Integrated Services
4 Technical Integration / (creation of new services)
6 Registering & Monitoring
5 Semantic & Property control of the compositions
Registry Service
Decision Service

Figure 3.24: Spontaneous Composition runtime phases

The spontaneous service composition is launched upon each appearance of a new service in the environment. The new service can be itself the result of a spontaneous composition and its deployment in the environment triggers automatically the spontaneous service composition. As the matching is done on functional interfaces, the `Generator Service` will always return the same abstract possible combinations and if no stop condition is defined, the same services will be composed over and over again, extending the environment with the same implementations over and over.

Our `Decision Service` for each cycle of a spontaneous service composition verifies just before the `Builder Service` implements the chosen composition that no composition is done twice and that by analyzing the properties of the services. Once a new service is in the environment, whether it is provided by applications or resulting from a spontaneous composition, the `Decision Service` verifies whether it is a new service. A new service is defined as proposing new interfaces or new implementations for existing services interfaces. If the newly arriving service does not verify one of these conditions, it is considered as already existing in the environment and no spontaneous composition is launched. If the service is new, the `Generator Service` and `QoS Service` produce all the possible transparent service composition. Before passing these combinations to the `Builder Service` the `Decision Service` analyzes the property of these possible service compositions, if the properties already exist in the environment, no composition is implemented. This stop conditions ensures that no services composition duplication is made by our `MySIM` middleware. On the other hand, it has severe limitations on the diverse ways that can exist to compose services together. Indeed, two operations may be composable over not only one parameter input but several, and the actual property description do not take this into consideration. One issue is to supply the property with an additional parameter that specifies which elements of the operations (and by generalization services) is actually being composed. By that the options of composing two services are much more important and especially allowed by the `Decision Service`.

Example 17 *We consider the following properties in the environment, describing the functional interfaces published to the user and applications:*

$Pr_1 = \{< takePicture, atomic >\}$
$Pr_2 = \{< resizePicture, atomic >\}$
$Pr_3 = \{< storagePicture, atomic >\}$

We suppose that the functional interfaces described by these properties include composable operations and that the resulting composition produce equivalent or almost equivalent services to the set of already available services. The `Generator Service` and `QoS Service` generates all the possible combinations and the `Decision Service` verifies the generated properties:

$Pr_4 = \{< takePicture, integrated > (< resizePicture, atomic >)\}$
$Pr_5 = \{< storagePicture, integrated > (< resizePicture, atomic >)\}$
$Pr_6 = \{< takePicure, integrated > (< storagePicure, atomic >)\}$

All these properties are different of the already existing properties in the environment. The deployment of these interfaces under these properties, trigger the spontaneous composition and another cycle of matching is produced. The Decision service will analyze the properties and discover that some of the proposed service compositions have already been composed. We take the $TakePicture$ *service and have a look on the possible service compositions:*

$Pr_7 = \{< takePicture, integrated > (< resizePicture, atomic >)(< resizePicture, atomic >)\}$
$Pr_8 = \{< takePicure, integrated > (< storagePicure, atomic >)(< resizePicture, atomic >)\}$

The property Pr_7 *is redundant whereas the property* Pr_8 *corresponds to a real new composition. The `Decision Service` asks the `Builder Service` to implement the new service corresponding to property* Pr_8 *and not the one corresponding to property* Pr_7*. Another cycle of spontaneous composition is triggered due to the deployment of the new service publishing* $Takepicture$ *interface under the new property* Pr_8*. This time all the possible services compositions have already existing properties and the `MySIM` does not compose any new service.*

The choice between realizing a syntactic or semantic matching depends strongly on the nature of the available devices. The strategy usually applied is to perform a syntactic spontaneous composition at first, which allows a quick creation of new services. This syntactic matching is controlled by a semantic evaluation of the concepts compatibilities of the composed services. Indeed, a syntactic matching can be done between two services that are syntactically compatible but do offer completely different functionalities. The semantic service concept matching highlights the non useful combination that a syntactic matching may propose. By that, a decision can be taken to give up certain combinations. If these combinations have already been implemented, the `Decision Service` asks the `Registry Service` to stop these services and to unregister the corresponding properties for the functional interfaces. If these combinations where not yet implemented, the `Decision Service` does not pass the combinations to the `Builder Service`. This is useful in resource constraint environment as syntactic matching is less consuming than a semantic one. In environments where no resource constraint applies a semantic spontaneous service composition is preferred to the syntactic one. If it generates less composed services, it ensures that the generated services are semantically useful for users and applications.

Example 18 *The spontaneous syntactic service composition relies on a syntactic matching of inputs and outputs that do not take into consideration the operations semantic descriptions. The syntactic composition can produce services that are not useful to applications as they have no reasons to be composed together. Consider the following operations signatures:*

$In_{op1} = \{< s, java.lang.String, "Message" >\}$
$Out_{op1} = \{< java.lang.String,\ "Advertisement" >\}$
$Cpt_{op1} = \{< publish,\ "advertise\ message" >\}$

$In_{op2} = \{< s, java.lang.String, "Path" >\}$
$Out_{op2} = \{< java.lang.Boolean,\ "state" >\}$
$Cpt_{op2} = \{< print,\ "print\ document" >\}$

If these two operations are composable as the output of the publish operation is of the same type as the input of the print operation, the semantic meaning of each operation is different as specified in the concept description. The first operation advertises a message whereas the second one prints a document from its path specifications. The semantic control will alert upon this inconsistency and the syntactic composition will be stopped.

3.3.2.2 Spontaneous Service Composition Techniques

To realize the spontaneous service composition, we rely on the service relations defined section 3.2.2. The spontaneous service composition is a composition relation between two services (which at least one of them is new in the environment) that needs to be transparent for the application layer. This transparency is reflected in the services equivalence and almost equivalence relations. Based on these conditions We define this spontaneous composition as follows:

Definition 49 — Environment spontaneous services composition

We define two services si and sj from a set of service S to be environment spontaneous composable for S, $O_{sp}(si, sj) = true$ if si or sj are new in the environment $((N_{MySIM}(si) = true) \vee (N_{MySIM}(sj) = true))$ and the resulting composition service is equivalent or almost equivalent to any $sk \in S$, $\forall\, k \in \{1..|S|\}$

If $((O_{syntactic}(si, sj) = true) \vee (O_{semantic}(si, sj) = true)) \wedge (Compatible_{QoS}(Np_{Out_{si}}, Np_{In_{sj}}) = true)$, we note sl the resulting service of the composition of si and sj.
The spontaneous service composition is:

$O_{sp}(si, sj) = (\equiv_{syntactic}\ (sl, sk)) \vee (\triangleright_{syntactic}(sl, sk)) \vee (\equiv_{semantic}\ (sl, sk)) \vee (\triangleright_{semantic}(sl, sk))$

■

When implementing the service sl, the `Builder Service` needs not only to define the new operations signatures and non-functional QoS properties as explained section 3.2.3, it also needs to specify the property related to this composition and to implement the implementations corresponding to the new operations. The implementations part is thoroughly explained in chapter 4. The new service sl resulting from the composition of the two services si and sj, will publish its interfaces, under a property that specifies that the service sl is a composition of si and sj.

Definition 50 — Services composition property definition

We suppose two services si and sj from a set of service S to be environment spontaneous composable for S, $O_{sp}(si, sj) = true$, we note sl the resulting service of the composition of si and sj. sl publishes the same interfaces as any $sk \in S$, $\forall\, k \in \{1..|S|\}$
The spontaneous service composition sl new property is:

$Pr_{sl} = Pr'_{sk} \bullet (Pr_{si}) \bullet (Pr_{sj}),\ \forall\, k \in \{1..|S|\}$

Pr'_{sk} is the property of sk where the *value* is *integrated* rather than *atomic*. The new property Pr_{sl} describes a new service that is integrated, equivalent to service sk and composed of the two services si and sj.

■

Based on this definition, we introduce a localized spontaneous service composition, that applies the transparent composition on a localized level. This ùay be very useful in pervasive environments, as devices can connect and disconnect at any time, and no general view may be available, for a certain time, on all the services of the environment.

DEFINITION 51 — Localized spontaneous services composition

We define two services si and sj from a set of service S to be localized spontaneous composable, $O_{sp}(si, sj) = true$ if si or sj are new in the environment $((N_{MySIM}(si) = true) \vee (N_{MySIM}(sj) = true))$ and the resulting composition service is equivalent or almost equivalent to sk, $sk = \{si \vee sj\}$

If $((O_{syntactic}(si, sj) = true) \vee (O_{semantic}(si, sj) = true)) \wedge (Compatible_{QoS}(Np_{Out_{si}}, Np_{In_{sj}}) = true)$, we note sl the resulting service of the composition of si and sj.
The spontaneous service composition is:

$O_{sp}(si, sj) = (\equiv_{syntactic} (sl, sk)) \vee (\triangleright_{syntactic}(sl, sk)) \vee (\equiv_{semantic} (sl, sk)) \vee (\triangleright_{semantic}(sl, sk))$

■

And the property definition of the composition is defined:

DEFINITION 52 — Services composition property definition

We suppose two services si and sj from a set of service S to be localized spontaneous composable for S, $O_{sp}(si, sj) = true$, we note sl the resulting service of the composition of si and sj. sl publishes the same interfaces as si or sj.
The spontaneous service composition sl new property is:

$Pr_{sl} = Pr'_{si} \bullet (Pr_{sj}) \vee Pr'_{sj} \bullet (Pr_{si})$

$Pr'_{s(i/j)}$ is the property of $s(i/j)$ where the *value* is *integrated* rather than *atomic*. The new property Pr_{sl} describes a new service that is integrated and composed of the two services si and sj.

■

EXAMPLE 19 *We consider the following two properties in the environment, describing the functional interfaces published to the user and applications:*

$Pr_1 = \{< takePicture, atomic >\}$
$Pr_2 = \{< resizePicture, atomic >\}$

A localized spontaneous service composition of the two services will publish the following property:

$Pr_3 = \{< takePicture, integrated > (< resizePicture, atomic >)\}$

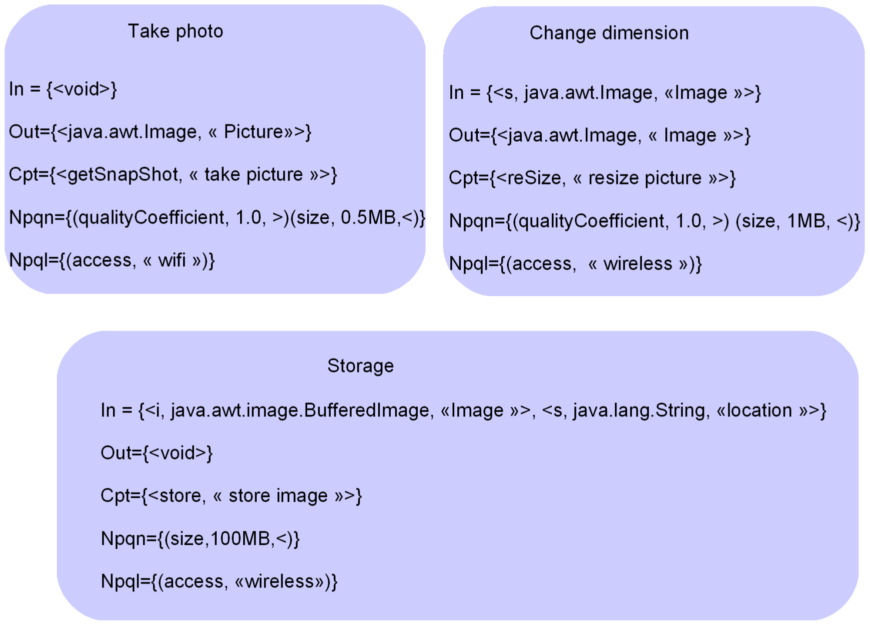
Example 20 *We consider the following two properties in the environment, describing the functional interfaces published to the user and applications:*

$Pr_1 = \{< storePicture, atomic >\}$
$Pr_2 = \{< resizePicture, atomic >\}$
$Pr_4 = \{< storeZip, atomic >\}$

An environment spontaneous service composition of the two services publishing Pr_1 and Pr_2 will publish an interface equivalent to the ones registered under Pr_4 property. The new service composition property is:

$$Pr_5 = \{< storeZip, integrated > (< storePicture, atomic >)(< resizePicture, atomic >)\}$$

Example 21 *Coming back to our example section 3.2.3 we define the three services having each the three operations: takepicture, changedimension and storage (cf. figure 3.25). We explain our spontaneous service composition using this example.*



Take photo

In = {<void>}

Out={<java.awt.Image, « Picture»>}

Cpt={<getSnapShot, « take picture »>}

Npqn={(qualityCoefficient, 1.0, >)(size, 0.5MB,<)}

Npql={(access, « wifi »)}

Change dimension

In = {<s, java.awt.Image, «Image »>}

Out={<java.awt.Image, « Image »>}

Cpt={<reSize, « resize picture »>}

Npqn={(qualityCoefficient, 1.0, >) (size, 1MB, <)}

Npql={(access, « wireless »)}

Storage

In = {<i, java.awt.image.BufferedImage, «Image »>, <s, java.lang.String, «location »>}

Out={<void>}

Cpt={<store, « store image »>}

Npqn={(size,100MB,<)}

Npql={(access, «wireless»)}

Figure 3.25: Three services operations specifications

The syntactic spontaneous composition gives the new operation getSnapShot as shown figure 3.26

$$O_{syntactic}(op_{takepicture}, op_{changedimension}) \quad = \quad true$$

$Compatible_{QoS}(NP_{takepicture}, NP_{changedimansion}) =$
$Compatible_{QoS}(qualityCoefficient_{Takepicture}, qualityCoefficient_{Takepicture}) \wedge$
$Compatible_{QoS}(size_{Takepicture}, size_{Takepicture}) =$
$(1.0 \subset 1.0) \wedge (0.5 \subset 1) = true$

The new operation getSnapShot resulting from the composition of the two operations getSnapShot and reSize has the same signature as the getSnapShot involved in the composition.
$\equiv_{syntactic} (op_{new}, op_{takepicture}) = true$
On the other hand, the two operations - old and new getSnapShot - do not offer exactly the same functionality. The new operation is enriched with the resizing functionality. Once this abstract composition generated by the `Generator Service`*, it is passed to the* `QoS Service` *for non-functional QoS properties verification. The two services have compatible non-functional properties as their respective* NP_{QN} *are compatible. The* `QoS Service` *can hence give the new operation getSnapShot the appropriate non-functional QoS properties based on the non-functional QoS of the two composed services. The new resulting property attached to the new operation will have an "integrated" value specifying that this operation implementation is new in the environment and combines two other "atomic" functionalities "takePicture" and "resizePicture". The* `Decision Service` *after analyzing the property and verifying that no equivalent is found in the environment can ask the* `Builder Service` *to implement the new integrated service. The* `Builder Service` *can hence implement the new service offering the same interface as Takephoto but with different functionality.*

$In = \{< void >\}$
$Out = \{< java.awt.Image,\ “Picture'' >\}$
$Cpt = \{< getSnapShot,\ “take\ picture'' >\}$
$Np_{QL} = \{< access,\ “wireless'' >\}$
$Np_{QN} = \{< qualityCoefficient,\ 1.0, > >,\ < size,\ 0.5MB,\ <>\}$
$operator = \{< >,\ <,\ =>\}$
$Impl = \{< Pr >\}$
$Pr = \{< takePicure, integrated > (< resizePicture, atomic >)\}$

The new Takephoto service is functionally equivalent to the previous Takephoto. But it presents differences in the non-functional QoS it needs, as it has to satisfy those of ChangeDimension service. The property reflects the service integration as it indicates that this Takepicture is an integrated service with the atomic Changedimension service.

The semantic spontaneous composition gives the new operation storage as shown figure 3.27

$(O_{semantic}(op_{storage}, op_{changedimension}) = true)$

The new operation store resulting from the composition of store and reSize has the same semantic description as the store operation, but providing an extended functionality of storing images that resizes them before.

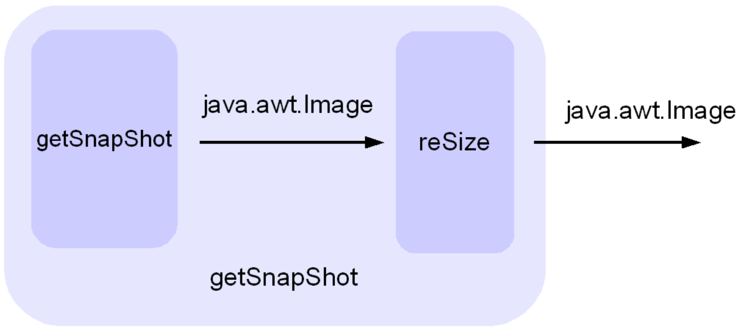


Figure 3.26: New getSnapShot operation

$(\equiv_{semantic}(op_{new}, op_{storage}) \; = \; true)$

Once the ***Generator Service*** *passes this abstract combination to the* ***QoS Service****, this last one verifies the compatibilities of the non-functional QoS and passes the adequate composition with the adequate values of the non-functional QoS to the* ***Builder Service****. Before, the* ***Decision Service*** *controls the new property to verify that no duplication will be made if the service composition is implemented. The new storage service will have NP_{QN} that respect both Changedimension and Storage services. For that reason the qualitative non-functional QoS size will be of $1MB$ respecting by that the qualitative non-functional QoS of Changedimension. The spontaneous composition will generate two different services of the new storage corresponding to the services initially available in the environment.*

$In = \{< i, java.awt.Image, \text{“}Image'' >, < s, java.lang.String, \text{“}location'' >\}$
$Out = \{< void >\}$
$Cpt = \{< store, \text{“}storeimage'' >\}$
$Np_{QL} = \{< access, \text{“}wireless'' >\}$
$Np_{QN} = \{< size, 1MB, <>\}$
$Impl = \{< Pr >\}$
$Pr = \{< storageftp, integrated > (< resizePicture, atomic >)\}$

$In = \{< i, java.awt.Image, \text{“}Image'' >, < s, java.lang.String, \text{“}location'' >\}$
$Out = \{< void >\}$
$Cpt = \{< store, \text{“}storeimage'' >\}$
$Np_{QL} = \{< access, \text{“}wireless'' >\}$
$Np_{QN} = \{< size, 1MB, <>\}$
$Impl = \{< Pr >\}$
$Pr = \{< storageLocal, integrated > (< resizePicture, atomic >)\}$

The two services are different as their functional interfaces are registered under different *properties.*

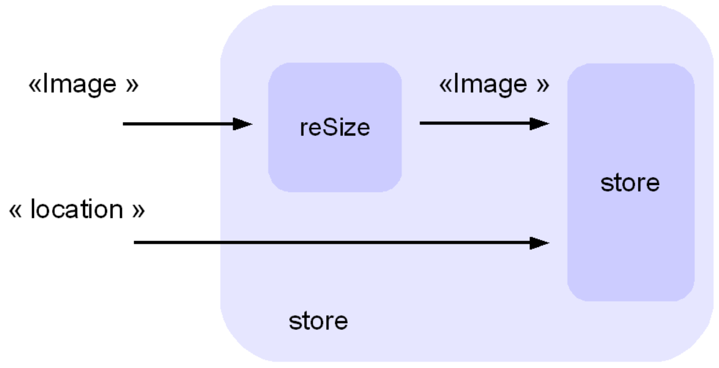


Figure 3.27: New storage operation

To really implement the new storage operations a translator is needed to translate between types as shown figure 3.28. This aspect is more explained in the implementation chapter.

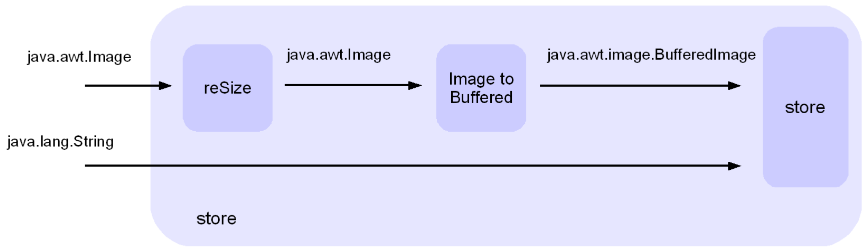


Figure 3.28: Types translation

The new resulting services, the Takephoto and Storage are implemented and deployed by the ***`Builder Service`*** *into the environment. These deployments are seen by* ***`MySIM`*** *middleware as events triggering the spontaneous composition. As their interfaces are still compatible with ChangeDimension a possible integration can be initiated. The* ***`Generator Service`*** *returns the same possibilities as above that combine Takephoto and Storage. After passing the* ***`QoS Service`****, this combination arrives to the* ***`Decision Service`****. The* ***`Decision Service`*** *analyzes the interfaces properties and is aware that a previous integration has already been done. Indeed, the property indicates that the new proposed service combines functionalities already combined. No integration is hence launched.*

A semantic composition relation exists between the two operations *getSnapShot* and *store* but the resulting operation is not semantically nor syntactically equivalent to any of the three operations available in the environment which means that if the composition is proceeded it will produce a new operation with a new signature and hence a new functional interface in the environment. This kind of composition do not provide transparent result to the application layer and can occur only on an a user on-demand composition.

Our `MySIM` middleware has all the techniques to execute an on-demand service composition for users and applications. The `Translator Service` translates a given user request into the SERVICE model. The `Generator Service` searches for all the possible composition combinations that return equivalent or almost equivalent services to the requested service. The `QoS Service` verifies the non-functional properties for the composition and assign the adequate properties for this composition. Finally the `Builder Service` implements the newly composed service in the same technology model as required by the user or applications.

3.3.3 Spontaneous Service Adaptation

In a pervasive environment, devices can come and go, connect and disconnect at any time and without previous prevention. Services, deployed on these devices, can appear and disappear even if used by the applications and users of the environment. `MySIM` middeware proposes service adaptation techniques to respond to these changes and to allow applications to continue their execution regardless of services appearance and disappearance. In this section, we begin by explaining a service life-cycle in a pervasive environment. Then, we explain how our `MySIM` middleware reacts to the events related to this life-cycle. Finally we emphasizes the employed techniques that allow a spontaneous transparent service adaptation for the application layer.

3.3.3.1 Spontaneous Service Adaptation Life Cycle

When a service is deployed in the pervasive environment, it registers its functional interface under the property it wishes to publish. Indeed, a service may wish to publish a specific aspect of the functionalities it provides and implements, and this is typically expressed in the functional property that naturally links between the interface operations and their implementations. These registrations are done through the `Registry Service` that plays the role of a dynamic service registry and service listener. A service may publish its interface without attaching a functional property. If no property is attached to a functional interface, that means that the service implementing the interface does not wish to focus on a particular aspect of this implementation. Once started, the services can be executed by the available application clients. The `Registry Service` may stop, suspend and restart services when needed and upon the `Decision Service` demand.

When a client needs a specific service, it sends its request to the `Registry Service` specifying the functional interfaces it seeks. It can also specify the desired functional properties attached to the functional interface. If no property is specified, the `Registry Service` can choose the services it wants. Services appearing in the environment register their functional interfaces under their properties related to their implementations through the `Registry Service`.

When a service appears in the environment, the `Registry Service` notifies the `Decision Service` of this appearance. The `Decision Service` will ask the `Generator Service` to find out if this service is functionally equivalent or almost equivalent to other services already available in the environment and being used by applications. If this is the case, the `Decision Service` requires from the `QoS Service` to compute the QoS_{degree} of this new service in order to verify whether its non-functional QoS properties are more adapted and fit better some applications requirement than the already used services. This spontaneous and proactive service adaptation occurs even in not required by the applications. If the new service provides better non-functional QoS for applications, it will be proposed spontaneously and transparently to these applications. Applications will continue to have the same required functionalities as the services are interface equivalent or almost equivalent but with better non-functional QoS properties.

When a service disappears, the `Registry Service` that also plays the role of a service listener is aware of this disappearance, it unregisters the services and reports this information to the `Decision Service`. Some actions need to be undertaken to cope with these changes. The call to these services will be automatically redirected to other available services offering same interfaces but with different implementations and hence different properties. This redirection is kept transparent to the users and applications in the application layer (cf. figure 3.29).

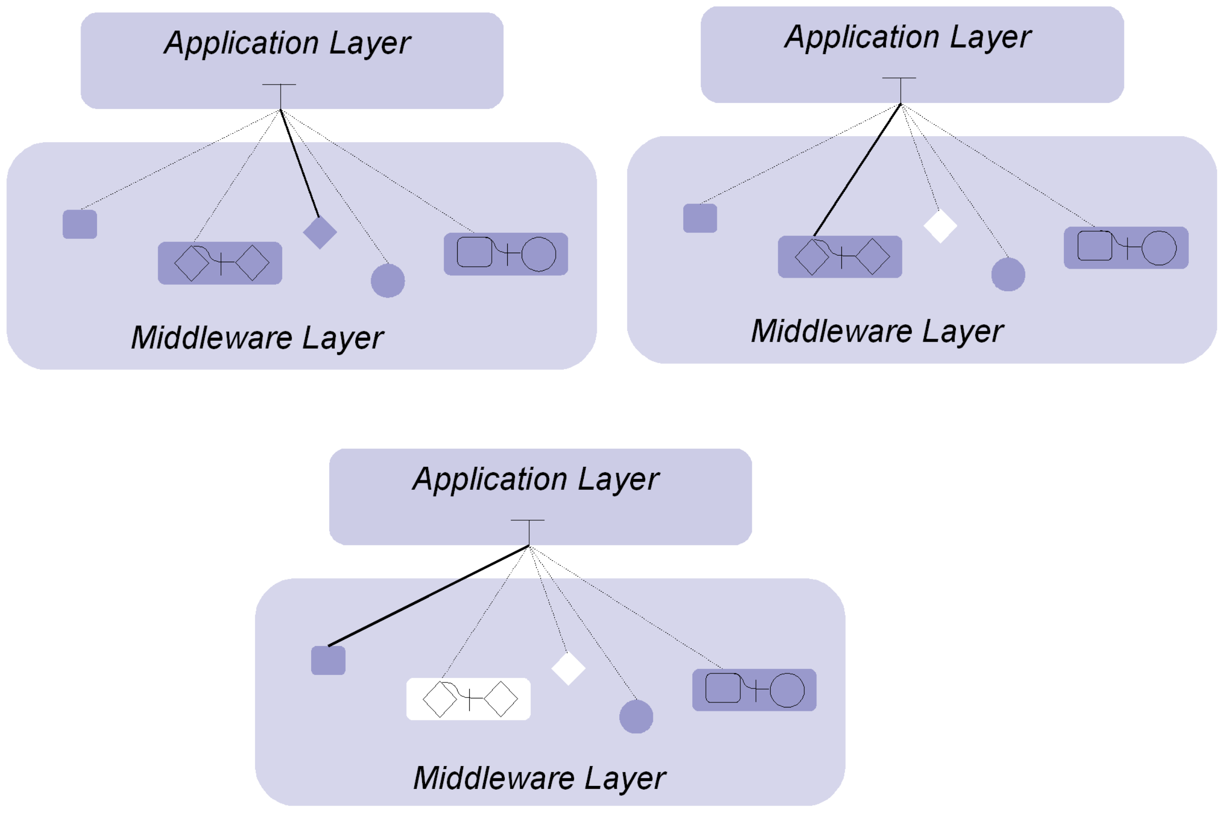


Figure 3.29: Services spontaneous adaptation for the application layer

The `Decision Service` is the one responsible of these service adaptations. it reposes over the `Registry Service` to be notified of possible appearance and disappearance of services and upon the `Generator Service` and the `QoS Service` to find the best services that are interface equivalent to the service that appeared or disappeared and that offer better or similar non-functional QoS properties.

The need for service adaptation occurs when a new service arrives or one a client requires a service that is no longer available. `MySIM` provides transparent adaptation mechanisms that take in charge providing the functionalities requested by the service by switching between the different available equivalent services whether atomic or integrated. This switch is done in a transparent way and the clients are unaware of it.

The `MySIM` services taking part in the spontaneous service adaptation are the `Decision Service`, the `Generator Service`, the `QoS Service`, and the `Registry Service`.

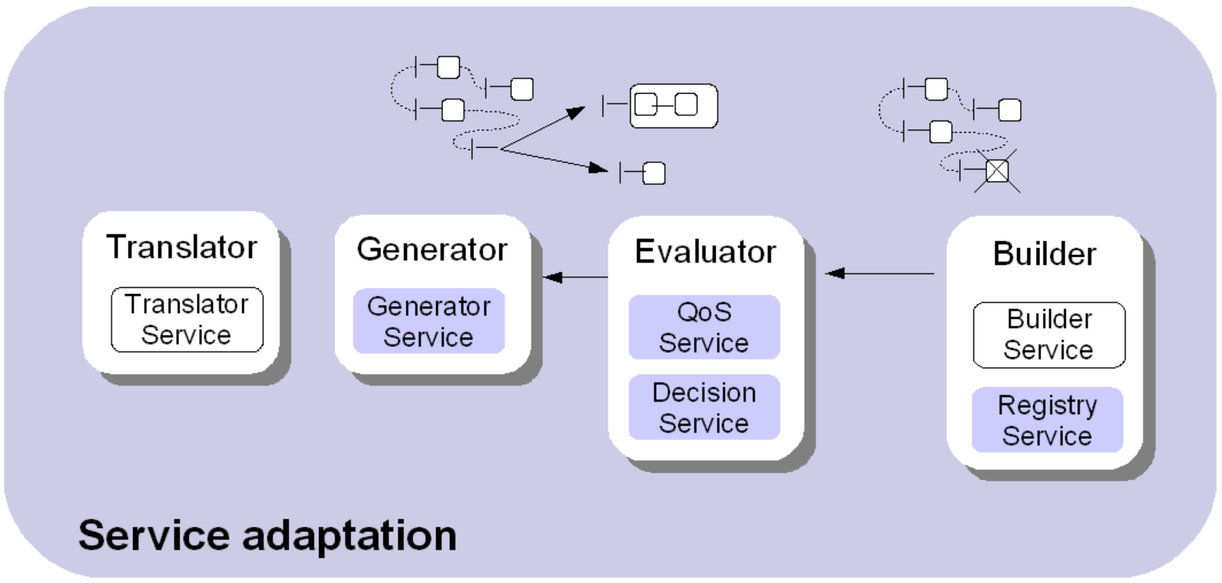


Figure 3.30: Services taking part in the spontaneous service adaptation

3.3.3.2 Spontaneous Service Adaptation Techniques

The service adaptation techniques are applied by the `Decision Service` once a service appears or disappears from the environment and its functional property is registered or unregistered by the `Registry Service`.

Service appearance

Considering a set S of finite services in the environment, we denote si the service that appears. As a first step, the `Decision Service` searches for functionally equivalent or almost equivalent services interfaces in the environment. Indeed, these services are services that provide the same functionality - the same functional interfaces - as the service si, and can be replaced in the application clients execution by the service si .

Definition 53 — Spontaneous service adaptation upon a service appearance

We consider the new service si. We suppose that the service si is equivalent or almost equivalent to other services in the environment:

$\exists\ sj \in S,\ (\equiv_{syntactic}(si, sj) = true) \lor (\equiv_{semantic}(si, sj) = true) \lor (\triangleright_{syntactic}(si, sj) = true) \lor (\triangleright_{semantic}(si, sj) = true)$

The spontaneous service si adaptation succeeds ($A(si) = true$) if si can replace sj for the application execution and that by providing better non-functional QoS properties than sj for the applications. By checking the profile of applications, the `Decision Service` knows the values and the priorities (wi) that the applications would like to assign to the non-functional QoS properties. The `QoS Service` can simulate a service sk, with these values, and calculates the QoS_{degree} using the wi specified by the applications. If no wi are assigned, the `QoS Service` applies the following values: $\sum_{i \in \mathbb{N}} wi = 1$. The service adaptation succeeds ($A(si) = true$) if:

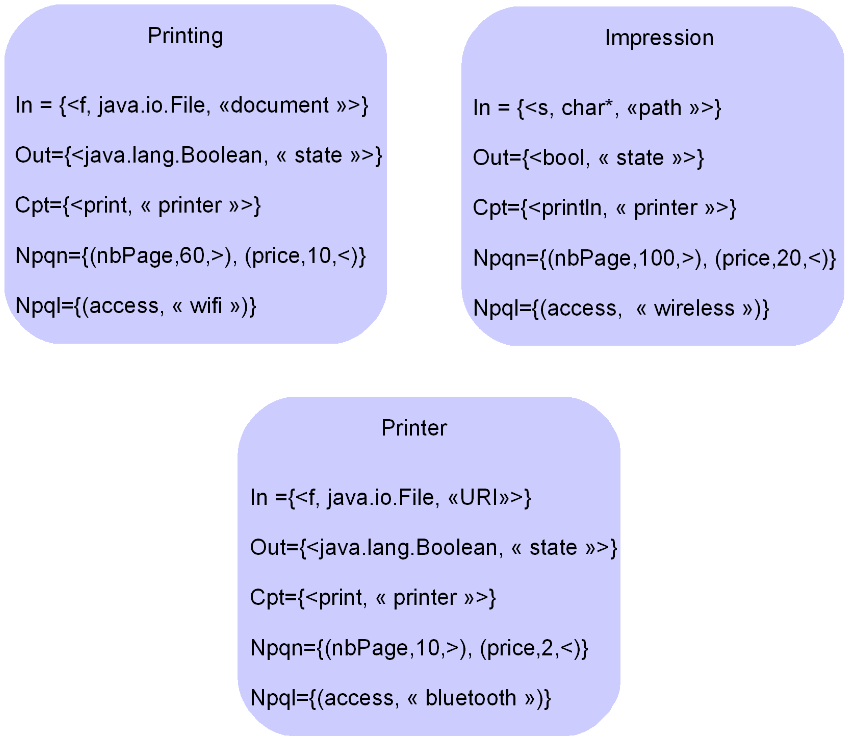
$QoS_{degree}(si, sk) < QoS_{degree}(sj, sk)$

which means that the new service si is closer to sk than sj is to sk in terms of non-functional QoS properties, sk reflecting the applications needs and preferences for the non-functional QoS properties of the service they executes.

■

Example 22 *Considering the three operations defined in figure 3.31.*

The Printing service is a new service appearing in the environment and is semantic almost equivalent to the Impression service. The `Decision Service` *considers applications using the Impression service, and verifies which non-functional QoS properties are the required by the applications. For example, if the price is important, the w_{price} would be much more important than the w_{access} and w_{nbPage}, and the new Printing service fits better for the application. The* `QoS Service` *simulates a new service by assigning it the adequate values of the non-functional QoS properties required by applications. As an example we can give the following application required non-functional QoS properties depicted under service sk:*



Printing

In = {<f, java.io.File, «document »>}

Out={<java.lang.Boolean, « state »>}

Cpt={<print, « printer »>}

Npqn={(nbPage,60,>), (price,10,<)}

Npql={(access, « wifi »)}

Impression

In = {<s, char*, «path »>}

Out={<bool, « state »>}

Cpt={<println, « printer »>}

Npqn={(nbPage,100,>), (price,20,<)}

Npql={(access, « wireless »)}

Printer

In ={<f, java.io.File, «URI»>}

Out={<java.lang.Boolean, « state »>}

Cpt={<print, « printer »>}

Npqn={(nbPage,10,>), (price,2,<)}

Npql={(access, « bluetooth »)}

Figure 3.31: Three services operations specifications

$$Np_{QL} = \{< access,\ “wireless” >\}$$
$$Np_{QN} = \{< nbPage,\ 50,\ '>' >,\ < price,\ 12,\ '<' >\}$$

And $w_{price} = 0.6,\ w_{access} = 0.2,\ w_{nbPage} = 0.2$

First we calculate the values that we need for our degree calculations:

The mean for nbpage property: $\mu(nbpage) = 55$
The standard deviation for nbpage property: $\sigma(nbpage) = 32$
The normalized z-score values are: $\eta(nbpage_{printing}) = 0.46$
$\eta(nbpage_{impression}) = 0.149$
$\eta(nbpage_{printer}) = 0.85$
$\eta(nbpage_{sk}) = 0.54$

The mean for price property: $\mu(price) = 11$
The standard deviation for price property: $\sigma(price) = 6,4$
The normalized z-score values are: $\eta(price_{printing}) = 0.46$
$\eta(price_{impression}) = 0.85$
$\eta(price_{printer}) = 0.15$
$\eta(price_{sk}) = 0.539$

The semantic distance for the non-functional properties are: $D_{semantic}(access_{printing}, access_{sk}) = 0.8,\ M_{concept}('wifi', 'wireless') = Subsume$
$D_{semantic}(access_{impression}, access_{sk}) = 0,\ M_{concept}('wireless', 'wireless') = PlugIn$
$D_{semantic}(access_{printer}, access_{printer}) = 1,\ M_{concept}('bluetooth', 'wireless') = Fail$

Using these values we calculate:

$$QoS_{degree}(Printing, sk) = 0.6 * 0.08 + 0.2 * 0.8 + 0.2 * 0.078 = 0.22$$
$$QoS_{degree}(Impression, sk) = 0.6 * 0.391 + 0.2 * 0 + 0.2 * 0.311 = 0.29$$

We have $QoS_{degree}(Printing, sk) < QoS_{degree}(Impression, sk)$*, which means that the new printing service fits better the application requirement.*

Service disappearance

Another major issue requiring adaptation is the disappearance of services form the environment. If a service disappear, the `Registry Service` is notified and it notifies the `Decision Service`. This one asks the `Generator Service` to come back with all the services that are equivalent or almost equivalent to this service. If many services are found, the `Decision Service` creates sets of services. A set for the services equivalent, almost equivalence, syntactically or semantically. The set are ordered from the syntactically equivalent to the semantically almost equivalent. The equivalence is considered better than the almost equivalence, as services can be interchanged in an equivalence relation (symmetric relation).

We consider a service *si*. We suppose that the service *si* is equivalent or almost equivalent (syntactically and/or semantically) to other services in the environment:

$\exists\ sj \in S,\ (\equiv_{syntactic} (sj, si) = true) \vee (\equiv_{semantic} (sj, si) = true) \vee (\triangleright_{syntactic}(sj, si) = true) \vee (\triangleright_{semantic}(sj, si) = true)$

Definition 54 — Functionally ordered set of equivalence

We define the following:

$$\begin{aligned} S_{\equiv}^{syntactic} &: set\ of\ sj,\ (\equiv_{syntactic} (sj, si) = true) \\ S_{\triangleright}^{syntactic} &: set\ of\ sj,\ (\triangleright_{syntactic}(sj, si) = true) \\ S_{\equiv}^{semantic} &: set\ of\ sj,\ (\equiv_{semantic} (sj, si) = true) \\ S_{\triangleright}^{semantic} &: set\ of\ sj,\ (\triangleright_{semantic}(sj, si) = true) \end{aligned}$$

These sets are ordered as follows $S_{\equiv}^{syntactic} < S_{\triangleright}^{syntactic} \leq S_{\equiv}^{semantic} < S_{\triangleright}^{semantic}$

■

The syntactic equivalence is preferred to the semantic equivalence as it means that services are provided in the same technology model and there is no need for any transformation to replace a service by another one. On the other hand the semantic equivalence offers more possibilities as it searches in all the environment for services offering the same functionalities as the service that disappeared. Thats why a semantic equivalence is sometimes preferred to a syntactic almost equivalence. This is reflected in the $\leq$ operator used instead of $<$.

In every set, services are ordered following the QoS_{degree} function that returns for every equivalent services with the service that disappeared their degree of equivalence concerning the non-functional QoS properties related to the service that the `Decision Service` would like to replace.

We consider the disappearing service si. We suppose the sets of services that are syntactically, semantically, equivalent or almost equivalent to si:

$$\begin{array}{ll} S^{syntactic}_{\equiv} : & set\ of\ sj,\ (\equiv_{syntactic}\ (sj, si)\ =\ true) \\ S^{syntactic}_{\triangleright} : & set\ of\ sj,\ (\triangleright_{syntactic}(sj, si)\ =\ true) \\ S^{semantic}_{\equiv} : & set\ of\ sj,\ (\equiv_{semantic}\ (sj, si)\ =\ true) \\ S^{semantic}_{\triangleright} : & set\ of\ sj,\ (\triangleright_{semantic}(sj, si)\ =\ true) \end{array}$$

For every set, we order the services of the set following their non-functional QoS properties and how they are similar to the ones of the disappearing service. By checking the values on the non-functional QoS properties for each service of every set, the `QoS Service` calculates the $QoS_{degree}(sj, si),\ \forall\ sj\ \in\ S^{*}_{*}$ of each service of a set with the service si. If no ponderation is given by the applications upon the priority of the properties the `QoS Service` employs the same value for wi : $\sum_{i\ \in\ \mathbb{N}} wi\ =\ 1$. The services within each set are ordered from the best one (service sj that minimizes $QoS_{degree}(sj, si)$) to the worst one (service sk that maximize $QoS_{degree}(sj, si)$):

Definition 55 — QoS ordered set of equivalence

$$\begin{array}{ll} T^{syntactic}_{\equiv} : & set\ of\ ordered\ s_j, (QoS_{degree}(s_j, si) < QoS_{degree}(s_{j+1}, si), j \in\ [1..|S^{syntactic}_{\equiv}| - 1]) \\ T^{syntactic}_{\triangleright} : & set\ of\ ordered\ sj, (QoS_{degree}(s_j, si) < QoS_{degree}(s_{j+1}, si), j \in\ [1..|S^{syntactic}_{\triangleright}| - 1]) \\ T^{semantic}_{\equiv} : & set\ of\ ordered\ sj, (QoS_{degree}(s_j, si) < QoS_{degree}(s_{j+1}, si), j \in\ [1..|S^{semantic}_{\equiv}| - 1]) \\ T^{semantic}_{\triangleright} : & set\ of\ ordered\ sj, (QoS_{degree}(s_j, si) < QoS_{degree}(s_{j+1}, si), j \in\ [1..|S^{semantic}_{\triangleright}| - 1]) \end{array}$$

When a service si disappears, the Decision Service asks the Generator Service and QoS Service to compute all possible T^{*}_{*} and chooses the best replacement for the service si by beginning from the most suitable set with the most suitable non-functional QoS properties.

■

Example 23 *Returning to our example section 3.2.2 of the Printing, Impression, and Printer services (cf. figure 3.32).*

If we search to replace the Printing service because of a sudden disappearance and need to choose between the Impression or the Printer services, the calculated QoS_{degree} between these services are different depending on the values assigned to wi.

$$QoS_{Degree}(Printing, Impression)\ =\ w1 * (|\eta(nbpage_{printing})\ -\ \eta(nbpage_{impression})|)\ + w2 * (|\eta(price_{printing})\ -\ \eta(price_{impression})|)\ + w3 * (D_{semantic}(access_{printing}, access_{impression}))$$

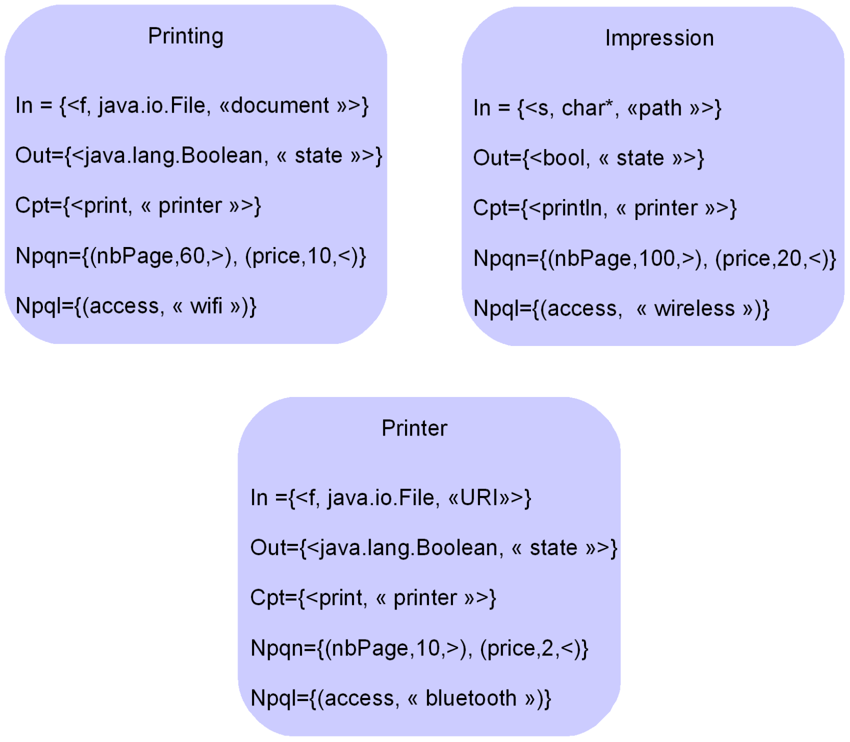

Printing
In = {<f, java.io.File, «document »>}
Out={<java.lang.Boolean, « state »>}
Cpt={<print, « printer »>}
Npqn={(nbPage,60,>), (price,10,<)}
Npql={(access, « wifi »)}
Impression
In = {<s, char*, «path »>}
Out={<bool, « state »>}
Cpt={<println, « printer »>}
Npqn={(nbPage,100,>), (price,20,<)}
Npql={(access, « wireless »)}
Printer
In ={<f, java.io.File, «URI»>}
Out={<java.lang.Boolean, « state »>}
Cpt={<print, « printer »>}
Npqn={(nbPage,10,>), (price,2,<)}
Npql={(access, « bluetooth »)}

Figure 3.32: Three services operations specifications

$QoS_{Degree}(Printing, Impression) = w1 * 0.27 \ + w2 * 0.35 \ + w3 * 0.2$

$QoS_{Degree}(Printing, Printer) = w1 * (|\eta(nbpage_{printing}) - \eta(nbpage_{printer})|) + w2 * (|\eta(price_{printing}) - \eta(price_{printer})|) + w3 * (D_{semantic}(access_{printing}, access_{printer}))$

$QoS_{Degree}(Printing, Printer) = w1 * 0.33 \ + w2 * 0.33 \ + w3 * 1$

If the service Printing is no longer available, the `Decision Service` *finds the service Impression as syntactic equivalent to Printing and Printer as semantic equivalent to Printing, as it is provided in another technology model. For their non-functional properties, it is clear that if the* `Decision Service` *assigns the same value to the three wi, the Impression service would have a closer degree to Printing. Nevertheless, if the application using Printing gives more importance to the price of the printing service, the* `Decision Service` *will assign to w2 a greater importance, and we can notice the Printer service has a closer degree to Printing than the Impression service. Nevertheless the Impression service is preferred as it is a syntactic equivalence and belong to the preferred set of equivalence services.*

It can occurs that no equivalent or almost equivalent services are found, in that case the search may be refined over a set of operations. If the users and applications of the services that disappeared used a particular operation or set of operations, the search may be specified over these operations using the equivalence and almost equivalence service relations defined upon particular operations $(\equiv^{Op}_{syntactic/semantic}, \triangleright^{Op}_{syntactic/semantic})$.

DEFINITION 56 — Spontaneous service si adaptation over Op

The spontaneous service si adaptation over a predefined set of operations Op succeeds if:

$$\exists\ sj \in S,\ (\equiv^{Op}_{syntactic}(sj, si) = true) \vee (\equiv^{Op}_{semantic}(sj, si) = true) \vee (\triangleright^{Op}_{syntactic}(sj, si) = true) \vee (\triangleright^{Op}_{semantic}(sj, si) = true)$$

■

EXAMPLE 24 *Considering the three services interfaces and their semantic descriptions in figure 3.33:*

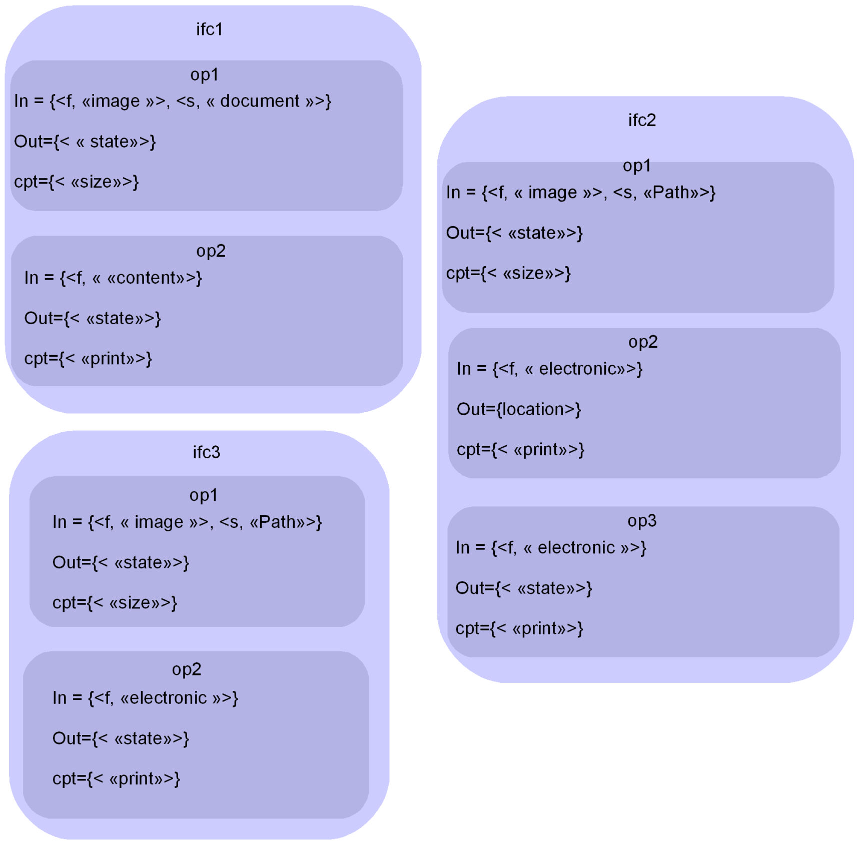


Figure 3.33: Three services operations specifications

We have $(\triangleright^{\{op1_{ifc1},\ op2_{ifc1}\}}_{semantic}(ifc1, ifc3) = true)$, *which means that the services proposing the interface* $ifc1$ *can replace the service* $ifc3$ *over the operations* $op1_{ifc1}$ *and* $op2_{ifc1}$.

As for the service as a whole, the `Decision Service` requires from the `Generator Service` and `QoS Service` to create the sets of equivalent and almost equivalent services over the predefined set of operations and that syntactically and semantically. It also orders the services within these sets depending on the non-functional QoS properties of the concerned operations and not the non-functional QoS properties of all the service.

If no services are found, the `Decision Service` may consider the services that are *Subsume* matching with the service that disappeared. If this replacement can fail to provide the required functionality as a *Subsume* matching between services does not guarantee that the new service can provide all what the other service provided, it can allows the environment to provide something to the applications even if not exactly what is required, while awaiting the appearance of the desired services. The `Decision Service` proposes these services to the applications, specifying that the services they seek are no longer available.

In case of complete failure of finding an appropriate service, the `Registry Service` redirects all the calls to the functional interface of the disappearing service to a proxy. Once a service registers a functional interface responding to the applications needs, the calls of the proxy can be redirected to this new service.

Section Conclusion

In this section, we presented our spontaneous service integration proposed by `MySIM` middleware. The section focused on the spontaneous service composition and the spontaneous service adaptation. Based on the equivalence and composition relations defined in the previous section, we explained the spontaneous service composition and adaptation life-cycle and techniques. The spontaneous service composition extends the already available functional services interfaces with new implementations, extending by that the functionalities an environment can offer to its users and applications. The spontaneous service adaptation hides the changes that affects the environment in terms of functionalities to the users and applications by providing pro-active mechanisms that adapt the call to services interfaces to the available implementations of these latter. In the next chapter, we explain our spontaneous service transformation and OSGi service implementation and generation. We also highlight the utility of our `MySIM` middleware on a given use case and evaluate the efficiency of our prototype.

Chapter 4

`MySIM` Middleware Implementation

We present in this chapter the implementation of the `MySIM` middleware. The middleware prototype implements the service integration functionalities depicted in chapter 3 of this thesis, i.e. spontaneous service transformation, spontaneous service composition and spontaneous service adaptation under an OSGi service platform implementation, the Apache Felix. The reminder of this chapter is structured as follows. First, we present `MySIM` middleware architecture and the different services `MySIM` interacts with. Then, we present *MyStudio* use case, upon which we explain the spontaneous service integration. We first explain the service transformation from OSGi services to our SERVICE model done by the `Translator Service`. We explain the spontaneous service composition and adaptation provided by `Generator Service` and `QoS Service`, and the service implementation and generation of OSGi integrated services provided by the `Builder Service`. Finally, the middleware prototype is evaluated and first performances are given.

4.1 The `MySIM` Middleware Architecture

The `MySIM` middleware is a combination of one or more of its specific services introduced section 3.1.2. These services can be deployed over one device or distributed over several devices in the environment. We adopt the distributed architecture for our middleware as the centralized one does not fit for pervasive environments. `MySIM` middleware can be located on one or several devices, and can be locally used or remotely accessed by the network (cf. figure 4.1).

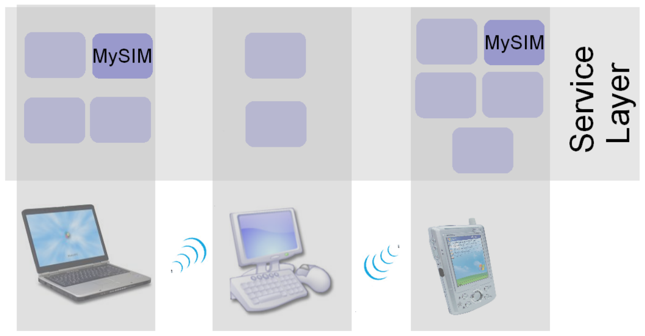


Figure 4.1: Distributed and pervasive environments

Our `MySIM` is based in the middleware layer (cf. figure 4.2). It can interact with other services located in the middleware such as the `Repository Service`, `Discovery Service`, and `Context Service`. Each device of the environment may host one instance of these services or non at all. We briefly explain each of these services and our OSGi based implementation for each of them.

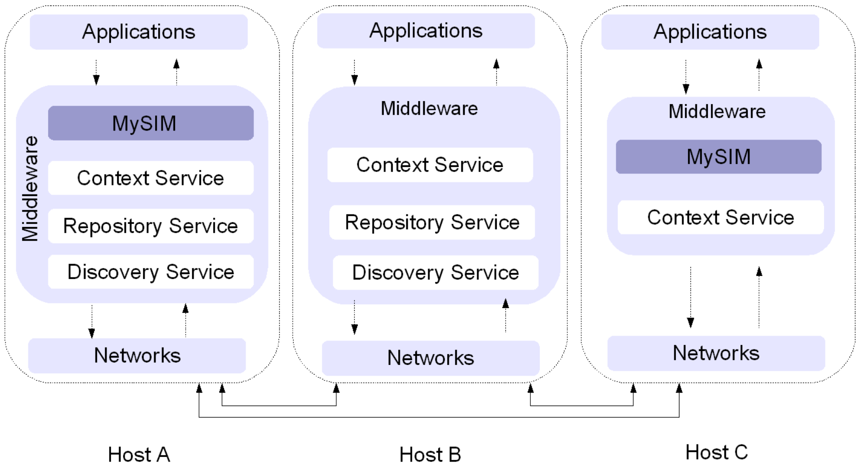


Figure 4.2: Middlewate layer

The `Repository Service` contains a list of all the services deployed on the device. New services deployed on the device and thus appearing in the environment need to register to the repository and to publish their interfaces with their descriptions for a possible use. OSGi specifications propose the OSGi service registry. The OSGi service registry holds the service registrations. A service is registered with the service registry under one or more Java interfaces together with properties. A receipt is provided when a service is registered. The service registry also allows the update of the service properties and the un-registration of the service. When a `MySIM` is deployed on a device, there is no need for a `Repository Service` as the `Registry Service` of `MySIM` plays also the role of a repository. The `Registry Service` offers more functionalities than the `Repository Service` as it can also monitor integrated service and advice the `Decision Service` of any changes occurring in the environment. `MySIM` is designed to be independent and fully operational on any device even if no `Repository Service` is available.

`Context Service` interacts with the physical environment and communicates all valuable information about the context and its changes, to the modules that require these informations, such as `Discovery Service` and `MySIM`. It reposes over an event-based mechanism. OSGi supports several types of events. The bundle event reports changes in the life cycle of bundles which is the unit of deployment of a service in OSGi. Framework event reports that the framework is started, start level has changed, packages have been refreshed, or that an error has been encountered. Service event reports on events holding information about the registration, modification, or un-registration of a service object. We use the service event to define when a service is new in the environment (following our definition 56 in section 3.3.1) and when the service leaves the environment. The `MySIM Registry Service` actively interacts with the `Context Service` and plays the role of a service listener that reacts to these two types of events.

`Discovery Service` is responsible of finding services depending on their descriptions. This service interacts actively with the `MySIM` as services need to be discovered before being integrated. The `Discovery Service` we use is based upon OSGi service reference, that provides a reference to each registered service. It provides access to the service's properties but not the actual service object. The service object must be acquired through a bundle's interface the bundle context. A service reference object encapsulates the properties and other meta-information (such as semantic description) about the service object it represents. This meta-information can be queried by a bundle to assist in the selection of a service that best suits its needs. When a bundle queries the framework service registry for services, the framework must provide the requesting bundle with the service reference objects of the requested services, rather than with the services themselves.

In the rest of the chapter, we first begin by describing our *MyStudio* use case. Based on this use case we depict the `MySIM` services: the `Translator Service` (cf. section 4.3), the `Generator Service` (cf. section 4.4), the `QoS Service` (cf. section 4.5), the `Builder Service` (cf. section 4.6), and the `Registry Service` (cf. section 4.7). Finally, we give performance evaluations of our `MySIM` prototype.

4.2 *MyStudio* Use Case

In the rest of this chapter, we illustrate the diverse key points we would like to explain by a simple use case. The use case is a small environment, a personal studio, composed of several services. The services can appear and disappear as devices come and go at any time. Using this use case, we depict the different aspects of our spontaneous service integration.

The services of *MyStudio* are showed figure 4.3: the $Webcam$ service, the $Naming$ service, the *Packaging* service, the *Storage* service, the *Printing* service, and the *Printer* service. All these services are deployed on devices that can appear and disappear at any time.

These services provide the following functionalities:

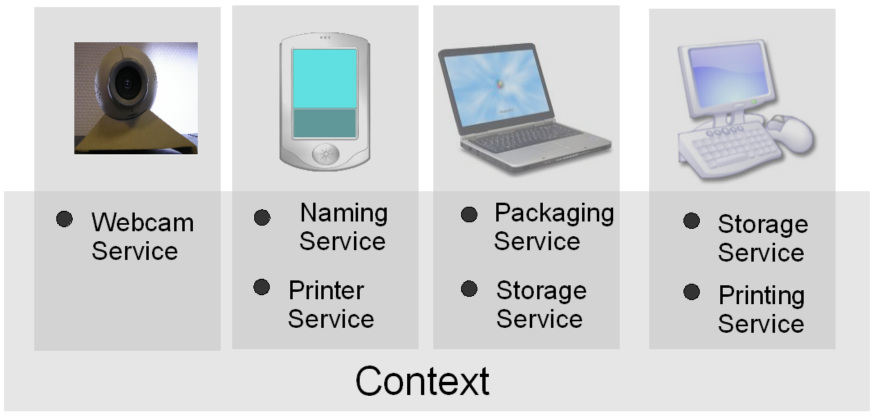


Figure 4.3: *MyStudio* environment

- The *Webcam* service enables to take a photo via a webcam.
- The *Storage* service enables to save an image on a device and to copy a file from one place to another. Two different services offer the same functionality. One implementation is for local storage, the other one for remote storage via ftp.
- The *Naming* service executes a naming strategy defined by a user to name his files and objects.
- the *Packaging* service jars files and resizes images.
- the *Printing* service enables the printing of a document.
- the *Printer* service enables the printing of a document but with different non-functional QoS properties as the *Printing* service.

The aim of our prototype implementation is to provide a proof of concept for our `MySIM` middleware showing how it is capable of spontaneously integrating services and generating new services and new functionalities in the environment. The new services will publish the same interfaces as those already available, and propose extended functionalities.

`MySIM` middleware composes services that are composable and with a resulting service equivalent or almost equivalent to an existing service in the environment. It also adapts the applications execution to possible services appearance and disappearance by providing to applications equivalent or almost equivalent services. `MySIM` triggers the spontaneous integration and controls it via the `Decision Service`. It combines and matches services via the `Generator Service` and `QoS Service`. It generates and deploys these services via the `Builder Service`. Finally, it registers and monitors these new services via the `Registry Service`.

OSGi service interfaces

For each service, we list in listings 4.1 its functional interface with its operations signatures.

```
// Webcam service
public interface cam.interfaces.ifc.WebcamIfc{
public Image getSnapShot();
}
// Storage service
public interface cam.interfaces.ifc.StorageIfc {
public void store(BufferedImage bimg, String filename);
public void copy(File src, File dst) throws IOException;
public void copy(String src, String dst) throws IOException;
}
// Naming Service
public interface cam.interfaces.ifc.NamingIfc {
public String getNextName(String ID);
}
// Packaging Service
public interface PackagingIfc {
public Boolean jar(String repertoire);
public Image reSize(Image img);
}
// Printing Service
public interface PrintingIfc {
public Boolean print(File source);
}
```

Listing 4.1: *MyStudio* Service interfaces

For conciseness, we only give the semantic description of the `Webcam` service. All the other services are described upon this model. We used and adapted the OWL-S ontology to describe our OSGi services. The profile defined in OWL-S corresponds the the interface description and the process to our property describing the implementations of the operations.

```
<!-- Service description -->
<service:Service rdf:ID="WebcamService">
<service:presents rdf:resource="#WebcamProfile"/>
<service:describedBy rdf:resource="#WebcamProcess"/>
</service:Service>

<!-- Profile description -->
<mind:WebcamService rdf:ID="WebcamProfile">
<service:presentedBy rdf:resource="#WebcamService"/>
<profile:serviceName xml:lang="en">Webcam</profile:serviceName>
<profile:textDescription xml:lang="en">This service takes a photo </profile:textDescription>
<profile:hasOutput rdf:resource="#ImageOutput"/>
</mind:WebcamService>

<!-- Process description -->

<process:AtomicProcess rdf:ID="WebcamProcess">
<service:describes rdf:resource="#WebcamService"/>
```

```
<process:hasOutput rdf:resource="#ImageOutput"/>
</process:AtomicProcess>
<process:Output rdf:ID="ImageOutput">
<process:parameterType rdf:datatype="file:///C:/ANIS/webcamDemo/semantic/Diversconcepts.owl">
file:///C:/ANIS/webcamDemo/semantic/Diversconcepts.owl#Image</process:parameterType>
<rdfs:label>image</rdfs:label>
</process:Output>
```

OSGi services implementation properties

The services interfaces properties as registered in the `Service Registry` under the following properties defined listing 4.2. A simple occurrence of “Service” means that the service is atomic, the number of occurrences of Service indicates the services composed within the registered implementations. All our service are atomic.

```java
// Webcam service
props.put("service", "WebcamService");
context.registerService(
cam.interfaces.ifc.WebcamIfc.class.getName(), jmstudio, props);
}
// Storage service via Ftp
props.put("service", "StorageFtpService");
context.registerService(
cam.interfaces.ifc.StorageIfc.class.getName(), serv, props);
}
// Storage Service
props.put("service", "StorageLocalService");
context.registerService(
cam.interfaces.ifc.StorageIfc.class.getName(), serv, props);
}
// Naming Service
props.put("service", "NamingService");
context.registerService(
cam.interfaces.ifc.NamingIfc.class.getName(), serv, props);
}
// Packaging Service
props.put("service", "PackagingService");
context.registerService(
cam.interfaces.ifc.PackagingIfc.class.getName(), serv, props);
}
// Printing Service
props.put("service", "PrintingService");
context.registerService(
cam.interfaces.ifc.PrintingIfc.class.getName(), serv, props);
}
```

Listing 4.2: *MyStudio* Service properties

The property describes the interface implementation and specifies whether this implementation is atomic or integrated (resulting from a spontaneous composition). To execute a service, the framework can choose services' interfaces considering the property they publish. If no property is specified `MySIM` 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. 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 *StorageFtpService* and *StorageLocalService* and considered as two different services.

OSGi services non-functional QoS properties

The non-functional QoS are modeled in our Java code as assertions for the quantitative and qualitative non-functional QoS properties. We use asserts for defining preconditions or what must be true when an operation is invoked, postconditions or what must remain true after an operation completes successfully and invariants what must be true about each instance from the beginning till the end of the execution. As defined in our model, some non-functional properties are more related to parameters inputs, others to the outputs and some to the operations in general. Considering the type of these properties and to what they are related, they are coded as preconditions, postconditions, and invariants assertions. Later, we would like to use OWL-S preconditions and postconditions to represent the non-functional properties. At this stage of our prototype implementation, QoS management is only done in the Java code as assertions over predefined variables defining the non-functional properties to study.

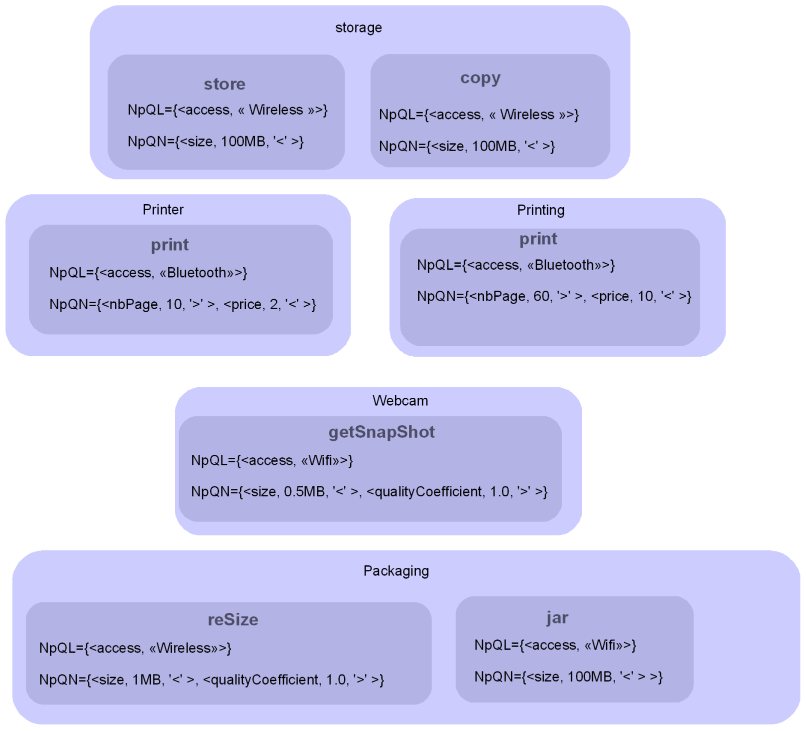

storage
store
NpQL={<access, « Wireless »>}
NpQN={<size, 100MB, '<' >}
copy
NpQL={<access, « Wireless »>}
NpQN={<size, 100MB, '<' >}
Printer
print
NpQL={<access, «Bluetooth»>}
NpQN={<nbPage, 10, '>' >, <price, 2, '<' >}
Printing
print
NpQL={<access, «Bluetooth»>}
NpQN={<nbPage, 60, '>' >, <price, 10, '<' >}
Webcam
getSnapShot
NpQL={<access, «Wifi»>}
NpQN={<size, 0.5MB, '<' >, <qualityCoefficient, 1.0, '>' >}
Packaging
reSize
NpQL={<access, «Wireless»>}
NpQN={<size, 1MB, '<' >, <qualityCoefficient, 1.0, '>' >}
jar
NpQL={<access, «Wifi»>}
NpQN={<size, 100MB, '<' > >}

Figure 4.4: Services non-functional QoS properties

4.3 The MySIM Translator Service

In this section, we detail the Translator Service (cf. figure 4.5) of our MySIM middleware. The chosen service technology is the OSGi service model developed under Java using the Felix platform. This 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 our SERVICE model between the interfaces and their implementations. OSGi [Alliance 2005] was chosen for its facility to provide the Inversion Of Control (IOC) [Loritsch 2001]. OSGi simplifies the development, deployment and management of services by decoupling the service's specification from its implementation.

We begin by introducing the OSGi technology with its service layer. Then, we explain our mapping rule, that map from the OSGi service model to our SERVICE model.

OSGi service layer

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. In OSGi, applications are in the form of bundles, i.e. Jar archives

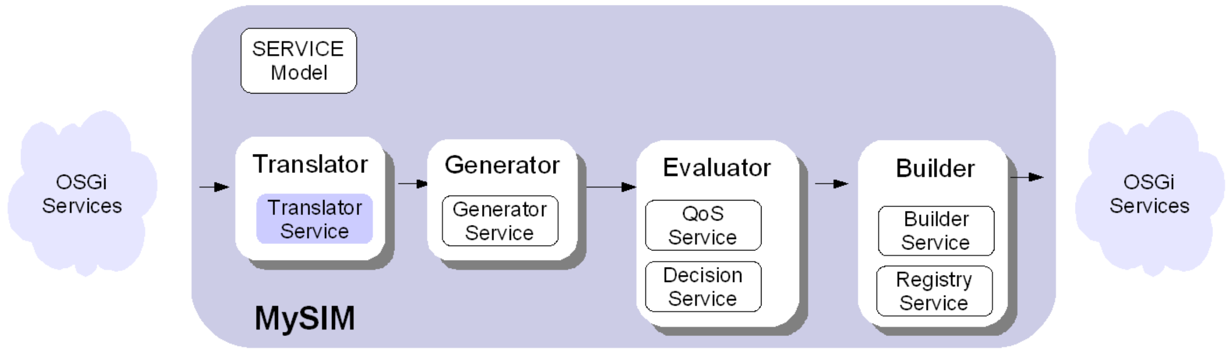


Figure 4.5: Translator Service

containing Java classes, native code, various resources files and so-called meta-data. OSGi proposes a service layer. An OSGi service is a Java interface, provided by a bundle (unit of deployment detailed in the next section) and registered to a repository. An OSGi service can be implemented in different ways, and can have several implementations for a same interface. When a particular service is required, the OSGi repository returns the list of all the available implementations. The bundle that required the service can then choose which implementation to use.

The OSGi service model is a publish, find and bind model. A service is a normal Java object that is registered under one or more Java interfaces with the service registry. Bundles can register services, search for them, or receive notifications when their registration state changes.

In the OSGi Service Platform, bundles are built around a set of cooperating services available from a shared service registry. Such an OSGi service is defined semantically by its service interface and implemented as a service object. The service interface should be specified with as few implementation details as possible. The service object is owned by, and runs within, a bundle. This bundle must register the service object with the Framework service registry so that the service's functionality is available to other bundles under control of the Framework.

The OSGi service layer is composed of the following parts:

- Service interfaces specify the service public methods.
- Service references encapsulate properties and meta data information about the service.
- Service properties associate a key/value argument to each registered interface.
- Bundle is the unit of deployment and provides the service implementations.

Mapping OSGi services into the SERVICE model

Mapping the OSGi specification to our generic SERVICE model is relatively easy (cf. figure 4.6) and is done as follows:

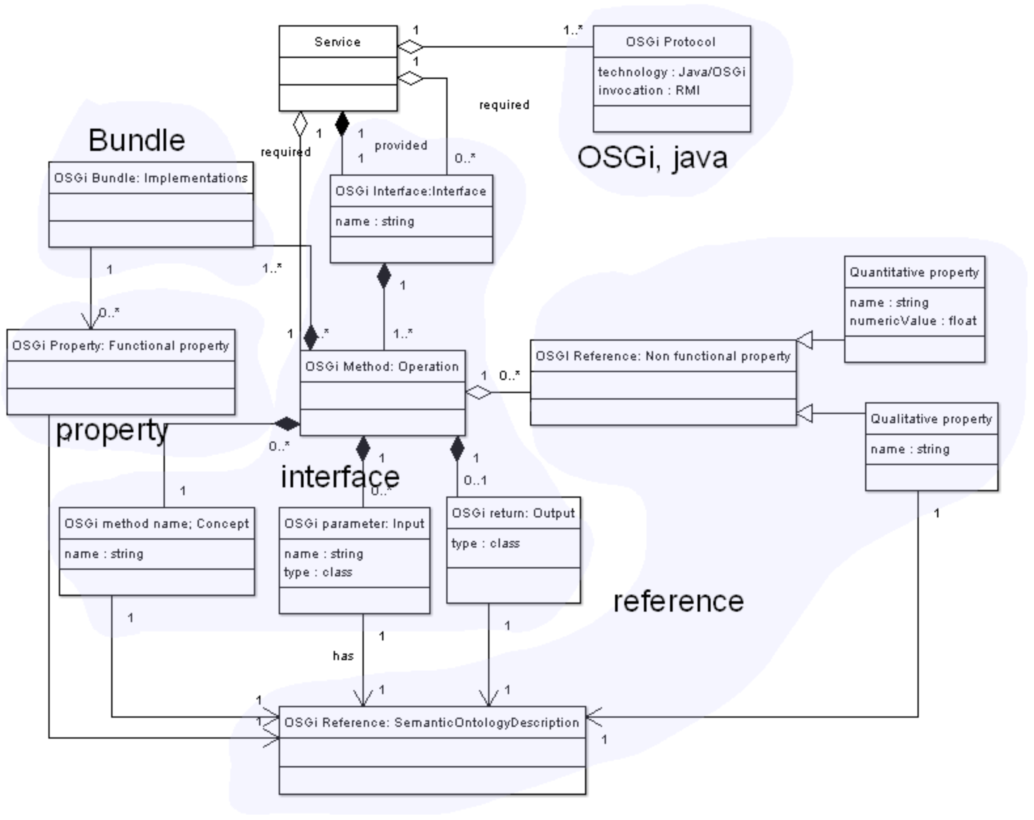


Figure 4.6: Mapping OSGi service to SERVICE model

- Service interfaces in OSGi corresponds to the functional interface Ifc of our SERVICE model. If in OSGi a service can publish multiple interfaces, it will be mapped to our SERVICE model as different services, as a service correspond to one functional interface. The OSGi methods are the operations Op of the service, the OSGi parameters of the method the inputs In, and the result returned by the methods the set of outputs Out. The name of the OSGi method through which the method is called is the concept *cpt* part of the operations Op.

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

- Service properties in OSGi are the functional properties associated to the operations implementations Pr. Interface implementations in OSGi are thus bound to a property that describes the implementations. When registering, a service can associate to its interface the property it wants describing the implementations of this interface.

- OSGi Bundles are the implementations of the service operations $Impl$.

- When mapping to the SERVICE model, the protocol of interaction of a service is specified as being OSGi based, Java language and using RMI procedure calls for remote access.

All the services of *MyStudio* are implemented as OSGi services. Each service is deployed by a bundle containing an activator, manifest, interface classes and the corresponding implementations. The semantic description of the services operations are described using OWL-S, and are considered as the meta-data provided in the property of the service.

Our `Translator Service` maps the OSGi services into the SERVICE model and provides these services to the `Generator Service` and `QoS Service` for spontaneous service composition and adaptation. For now, this mapping is done manually and will be automated in later versions of our prototype.

4.4 The `MySIM Generator Service`

The `MySIM Generator Service` (cf. figure 4.7) is responsible of the syntactic and semantic matching of the functional interfaces of services in order to compose or adapt services. In the following, we explain our spontaneous service composition and adaptation, and how the `Generator Service` matches services and proposes functional composition and functional equivalences that are transparent for users and applications.

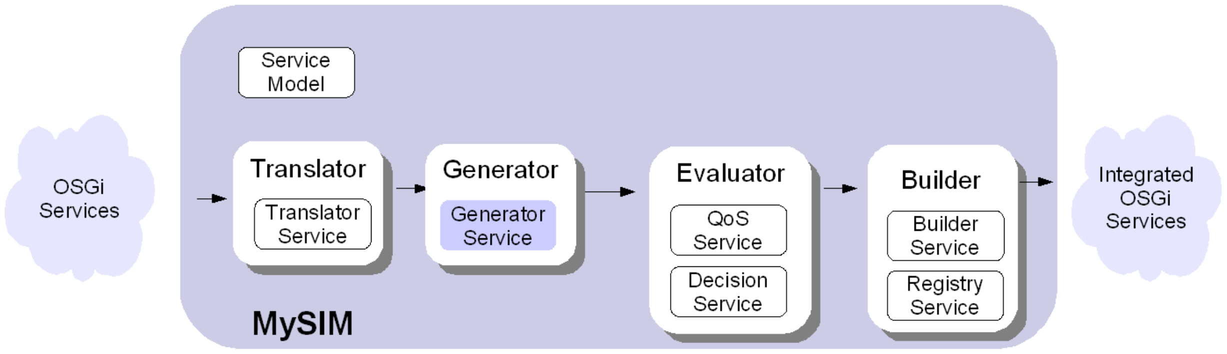


Figure 4.7: Generator Service

The `MySIM` Spontaneous Service Composition

We have defined our composition as a service sequence process which mean that the output of an operation is used an an input of another one. In our composition model, services are combined with each other based on the conformance of their signatures operation syntactic and/or semantic matching. For a composition to be spontaneous, the resulting services need to be equivalent or almost equivalent to an available service in the environment in terms of its functional interface.

The `Generator Service` lists all the operations of the environment and first proceed to a syntactic matching over their types in order to return all the syntactic possible compositions, that are equivalent to a service of the environment (listing 4.3). The matching is done based on a type marching.

```
// All operations of the environment
Method[] Op = getAllOperations(context);
// Search for composable services
for (int i=0; i<Op.length; i++){
        for (int j=0; j<Op.length; j++){
                bool= isSyntacticComposable(Op[i], Op[j]);
                // Composable services
                        if (bool=true){
                        composeOp = Compose(Op[i], Op[j]);
                                for (int l=0; l<Op.length; i++){
                                boolea = isSyntacticEquivalent(Op[k], Op[l]);
                                // Equivalent services
                                        if (boolean=true){
                                        //pass the composition to QoS Service}}}}}
```

Listing 4.3: Spontaneous syntactic service composition

The syntactic matching is done using the introspection provided by the Java language. Java has a well-developed introspection library. This allows a Java program to take a class and find all of the methods (including constructors) and to find the parameters and return types of these methods. By this way, our syntactic matching done by the `Generator Service` is capable of retrieving all the operations of services, accessing their inputs and outputs types and checking all possible compatibilities.

Example 25 *We illustrate the spontaneous service composition using MyStudio use case. First all the operations of the environment are listed figure 4.8.*

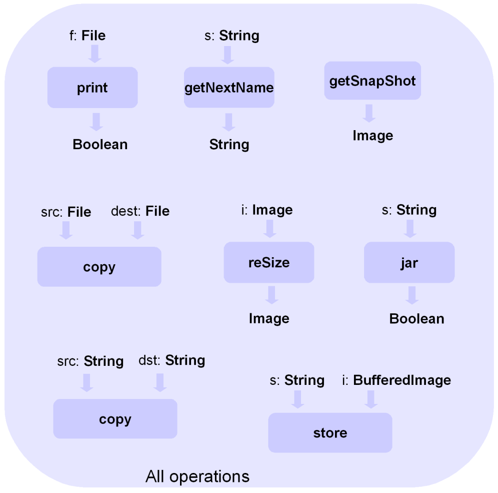


Figure 4.8: All operations available in the environment

The spontaneous syntactic service composition returns the following possibilities (cf. figure 4.9): service storage composed with service naming and equivalent to storage, service storage composed with service packaging equivalent to storage, service webcam composed with service packaging equivalent to webcam, and service packaging composed with service naming and equivalent to packaging.

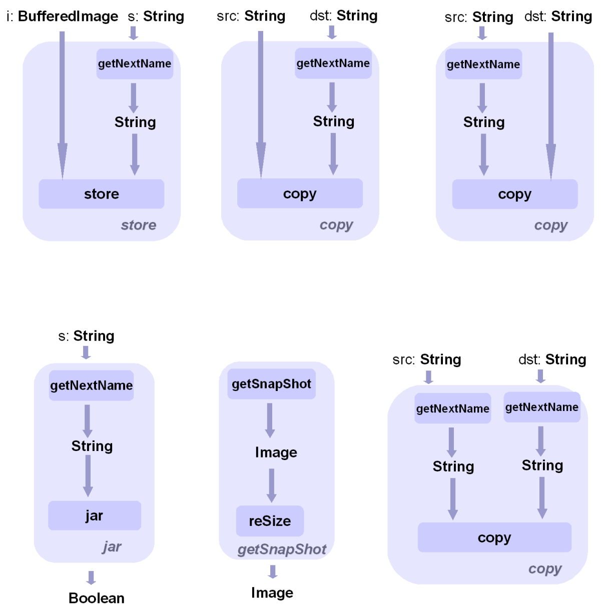


Figure 4.9: Spontaneous Syntactic service composition

The spontaneous syntactic composition may generate inconsistencies by composing services that are not really composable. For that a certain control is needed. Once these compositions have passed through the `Generator Service` the `Decision Service` needs by a semantic matching to control the utility of these syntactic compositions (listing 4.4). The semantic control uses the semantic descriptions of services to verify if the proposed combinations have a meaning or a utility to applications. The verification is done upon three criteria, the semantic description of inputs, the semantic descriptions of outputs and the semantic description of the operations concept.

```
// After the syntactic matching
bool= isSemanticComposable(Op[i], Op[j]);
        // Composable services
        if (bool=true){
        // proceed to the composition}
                else {
                // inform Decision Service}
```

Listing 4.4: Semantic control

For a semantic matching the tools we used are based on the Mindswap OWL-S API. OWL-S API provides a Java API for programmatic access to read, execute and write OWL-S service descriptions. The API supports to read different versions of OWL-S (OWL-S 1.0, OWL-S 0.9, DAML-S 0.7) descriptions. The API provides an Execution Engine based on the Pellet reasoner (listing 4.5) which is an open-source Java based OWL DL reasoner. It can be used in conjunction with both Jena and OWL API libraries. It can invoke atomic processes in our case atomic services, and composite processes, integrated services, that uses control constructs sequence, unordered, and split. Using this API, the `Generator Service` extracts the inputs, outputs and concept semantic value for each operation and compare these values in order to verify a possible compatibility.

```
public Matchmaker() {
kb = OWLFactory.createKB();
kb.setReasoner("Pellet");}
```

Listing 4.5: Pellet reasoner

EXAMPLE 26 *After this matching, `MySIM` decides not to implement the following operations (figure 4.10) as they make no sense for applications and users. Indeed, when an application would like to jar a certain file it does not need to change the naming strategy of the file before jarring it. Same thing applies when copying files, if the application may want to specify a strategy naming to the file destination, it surely does not want to do the same to the source file.*

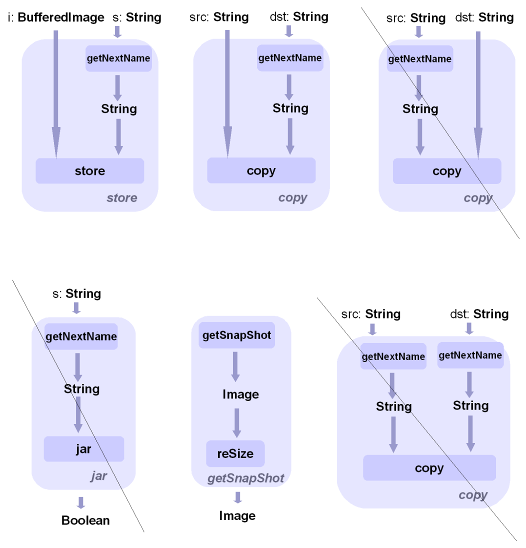


Figure 4.10: Semantic control

If the resources of the environment allow it, the `Decision Service` ask from the `Generator Service` to proceed to a semantic spontaneous composition (listing 4.6).

```
// All operations of the environment
Method[] Op = getAllOperations(context);
        // Search for composable services
        for (int i=0;  i<Op.length;  i++){
                for (int j=0;  j<Op.length;  i++){
                bool= isSemanticComposable(Op[i], Op[j]);
                        // Composable services
                        if (bool=true){
                        composeOp = Compose(Op[i], Op[j]);
                                for (int l=0;  l<Op.length;  i++){
                                boolea = isSemanticEquivalent(Op[k], Op[l]);
                                        // Equivalent services
                                        if (boolean=true){
                                        //pass the composition to QoS Service}}}}}
```

Listing 4.6: Spontaneous semantic service composition

EXAMPLE 27 *Besides the syntactic composition, `MySIM` proposes a semantic spontaneous composition between the two services storage and its save operation and packaging service with its resize operation as described figure 4.11:*

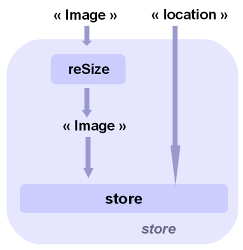


Figure 4.11: Spontaneous semantic service composition

This composition was not detected by the syntactic matching.

The semantic matching is done using the previous introduced execution engine Mindswap OWL-S API. The combinations of the syntactic and semantic compositions detect all possible transparent composition services for users.

EXAMPLE 28 *Once these new services in the environment (after a successful passage through the `QoS Service`), our `MySIM` middleware will try to find other possible combinations but by verifying that no combination is done twice. This verification is done by the `Decision Service` upon the property of each service. After verification one possible combination is found (cf. figure 4.12).*

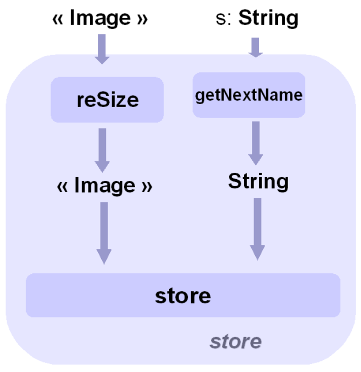


Figure 4.12: Spontaneous semantic service composition

If the service passes the `QoS Service` successfully, it is implemented and deployed in the environment by the `Builder Service`. This new appearance will not trigger a spontaneous service composition as no `MySIM` new services are introduced in the environment. All the interfaces are bound to already visited properties.

The new MyStudio environment resulting from the spontaneous service integration is depicted figure 4.13.

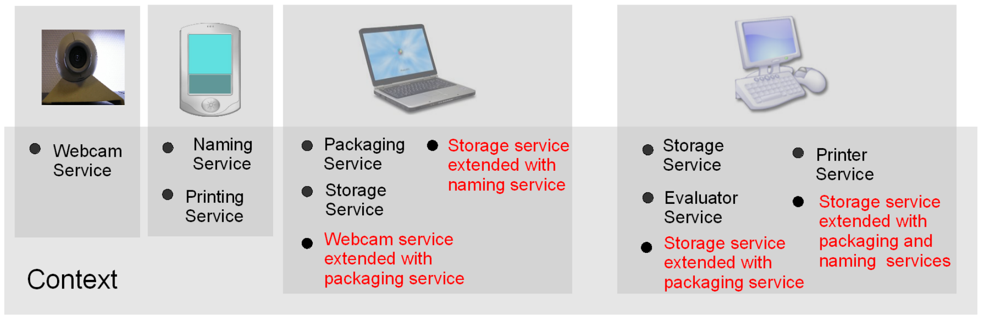

Webcam Service
Naming Service
Printing Service
Packaging Service
Storage Service
Storage service extended with naming service
Webcam service extended with packaging service
Storage Service
Evaluator Service
Storage service extended with packaging service
Printer Service
Storage service extended with packaging and naming services
Context

Figure 4.13: Pervasive environment after the spontaneous integration

The MySIM Spontaneous Service Adaptation

The Generator Service is also responsible of providing equivalent and almost equivalent services or operations to a specific service upon demand (listing 4.7). It first begins by listing all the available operations from the environment. Then, it applies syntactic and semantic matching between operations to construct the sets of syntactic/semantic equivalent operations and syntactic/semantic almost equivalent operations. The listing 4.7 provides the implementation for the operation level. The implementation is similar to the service layer. We just verify that the services have the same number of operations and that the operations provide equivalence relations.

```
// All operations of the environment
Method[] Op = getAllOperations(context);
        // Defining the set of operations
        Method[] SyntacticEquivalent;
        Method[] SyntacticAlmostEquivalent;
        Method[] SemanticEquivalent;
        Method[] SematincAlmostEquivalent;
                // Search for equivalent operations of opi
                for (int j=0;  j<Op.length; i++){
                bool= isSyntacticEquivalent(Opi, Op[j]);
                bool1 = isSyntacticAlmostEquivalent(Opi, Op[j]);
                bool2= isSemanticEquivalent(Opi, Op[j]);
                bool3 = isSemanticAlmostEquivalent(Opi, Op[j]);
                        if (bool=true){
                        SyntacticEquivalent.add(Op[j]);}
                        if (bool1=true){
                        SyntacticAlmostEquivalent.add(Op[j]);}
                        if (bool2=true){
                        SemanticEquivalent.add(Op[j]);}
                        if (bool3=true){
                        SemanticAlmostEquivalent.add(Op[j]);}}
```

Listing 4.7: Service equivalence relations

EXAMPLE 29 *In the MyStudio use case the two services printing and printer are syntactic equivalent. All the storage different services are syntactic equivalent as they publish the same interface. After the spontaneous service composition, the MyStudio is extended with syntactic equivalent services as new implementations of storage and webcam interfaces are provided.*

If the first step to adaptation is by searching for functionally equivalent services (syntactically or semantically), a second important step is done upon a selection based on non-functional QoS properties and is explained in the next section.

4.5 The `MySIM QoS Service`

The `MySIM QoS Service` (cf. figure 4.23) is responsible of the non-functional QoS properties related to service composition and adaptation. In a composition it verifies, whether the non-functional properties are compatible for a possible combinations. In an adaptation it calculates the QoS_{degree} upon the provided sets of equivalent services from the `Generator Service`. Based on the value of this function, the adaptation is optimized.

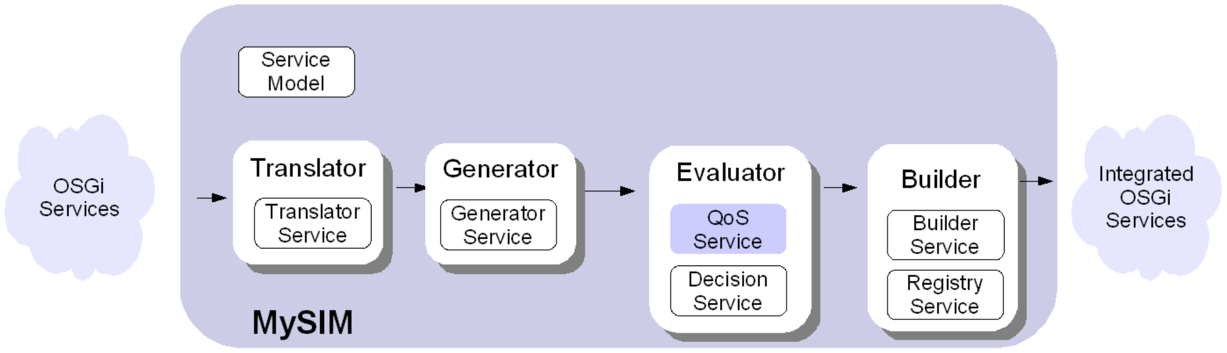


Figure 4.14: QoS Service

The `MySIM` Spontaneous Service Composition

Concerning the non-functional QoS properties, as for now they are considered as variables instantiated by the operations, a simple matching between their values determines whether the resulting service composition holds or not. Later on we would like to integrate the non-functional QoS properties into the OWL-S descriptions and provides a more advanced matching between these values.

When composing two services, the `QoS Service` verifies that the non-functional properties related to the input and output parameters being combined are compatible. For now, as our properties are variables defined by the operations, a simple matching is sufficient to detect inconsistencies.

`QoS Service` uses the `isQoSCompatible()` between two tuples of non-functional properties to verify if they can be combined together (cf. listing 4.9). First it verifies that the two non functional properties correspond to the same property by analyzing the name. Then, depending on the operator, the *npi* input non-functional property needs to verify the *npj* value of the output.

```
// two tuples npi, npj
namei = getName(npi);
namej = getName(npj);
valuei = getValue(npi);
valuej = getValue(npj);
operator = getOperator(npi);
        // testing if the same name
        if (namei=namej){
                if (operator='<'){
                        bool = (valuej < valuei);}
                        else {bool = (valuej > valuei);}
                                return bool;}
```

Listing 4.8: Function `isQoSCompatible` for quantitative properties

For qualitative non-functional properties, the concept matching between the values of the tuples is used based on the OWL-S API introduced earlier.

```
// two tuples npi, npj
namei = getName(npi);
namej = getName(npj);
valuei = getValue(npi);
valuej = getValue(npj);
        // testing if the same name
        if (namei=namej){
                        bool = (isConceptMatch(valuei, valuej));}
                        return bool;
```

Listing 4.9: Function `isQoSCompatible` for qualitative properties

The `isConceptMatch()` function returns true if the values of the concept matching between the two entities is *Exact* or *PlugIn*.

When the `Generator Service` passes the possible combinations of services to the `QoS Service`, the `Decision Service` specifies to this latter on which non-functional QoS properties to focus. Indeed, not all the non-functional properties need to be verified, but especially those of the parameters to combine (input and output). If the role of the QoS degree is rather limited to a matching in a composition, it is much more important in an adaptation as explained later.

The `MySIM` Spontaneous Service Adaptation

For service adaptation, the `QoS Service` receives the set of equivalent or almost equivalent services (syntactically, semantically) from the `Generator Service`, and will have the task to sort these sets using the QoS degree function introduced in chapter 3.

We consider an operation *opi* upon which we would like to find the closest equivalent operations in terms of non-functional QoS properties. *opi* can be an operation belonging to a service that has disappeared, or have non-functional properties representing what an application would like to have.

Listing 4.10 provides a general view of the implementation that allows to sort the sets provided by the `Generator Service` according to their non-functional properties.

```
// All operations of the environment
Method[] Op = getAllOperations(context);
        // set of operations  provided by the Generator Service
        Method[] SyntacticEquivalent;
        Method[] SyntacticAlmostEquivalent;
        Method[] SemanticEquivalent;
        Method[] SematincAlmostEquivalent;
                // Sorting the set following the QoS degree function
                // using  Arrays.sort() function
                for (int j=0;  j<SyntacticEquivalent.length; i++){
                QoS(Opi, Op[j]);
                QoSort(Op[j], SyntacticEquivalent);}

                for (int j=0;  j<SyntacticAlmostEquivalent.length; i++){
                QoS(Opi, Op[j]);
                QoSort(Op[j], SyntacticAlmostEquivalent);}

                for (int j=0;  j<SemanticEquivalent.length; i++){
                QoS(Opi, Op[j]);
                QoSort(Op[j], SemanticEquivalent);}

                for (int j=0;  j<SemanticAlmostEquivalent.length; i++){
                QoS(Opi, Op[j]);
                QoSort(Op[j], SemanticAlmostEquivalent);}
```

Listing 4.10: QoS degree sorting

The `QoS()` operation (cf. listing 4.10) implements the QoS_{degree} defined chapter 3. It calculates the degree of equivalence between the non-functional properties of the operations based on the z-score values for quantitative properties and on concept matching for qualitative one. For each quantitative non-functional property, after a calculation of the mean and standard deviation, `QoS()` provides the z-score followed by the η values for each operations. Once these values known, the `QoS()` operation affect values to wi by querying the `Decision Service`. The `Decision Service` can obtain this information from the applications profiles. For now, `QoS()` function gives all the wi the same value.

The `QoSort()` function sorts the operations of a set using the result of the `QoS()` function. The first element is the one that it QoS function provides the closest value to zero.

Once these sets sorted, the `Registry Service` can automatically choose a service to replace another one. It will be insured that the service it chooses is the best one for the replacement. At each appearance or disappearance of services, and before launching the spontaneous service adaptation, the QoS degree will be re-computed following the new available services and these sets will need to be re-sorted for the adaptation to be context-aware.

4.6 The `MySIM` `Decision Service`

The `Decision Service` (cf. figure 4.15) is the central service of the `MySIM` middleware, as it controls the events that can trigger a spontaneous composition and especially stop it. Indeed, the `Generator Service` matches services upon their functional interfaces, and as these interfaces are invariant in the environment, it will over and over finds the same combinations and requires from the `Builder Service` to generate and implement these compositions.

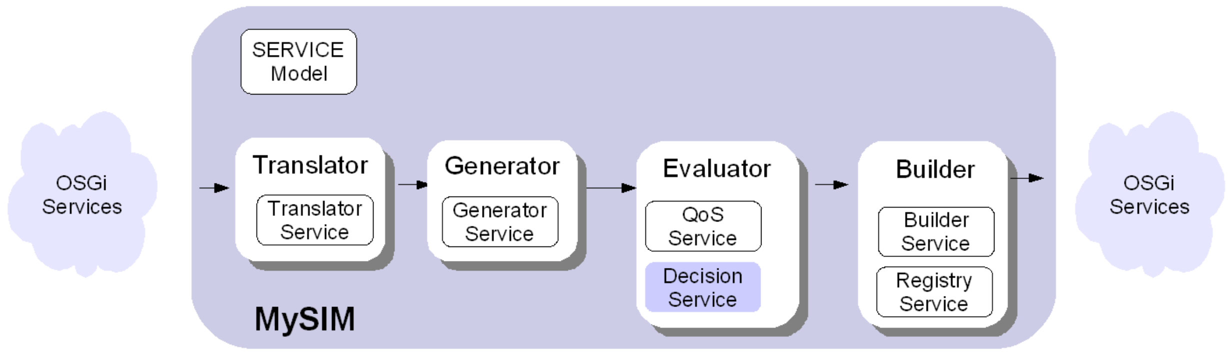


Figure 4.15: Decision Service

We first begin by defining when the spontaneous service composition is launched followed by an explanation on when it is stopped.

New services in the environment

When a service appears in the environment, the `Decision Service` needs to ensure that it is new before launching the spontaneous composition. The provided algorithm in the `Decision Service` providing this guarantee is the one implementing the definition 48 of `MySIM` new service.

```
// name of interface: cam.interfaces.ifc.class.getName()
// property of interface: cam.interfaces.ifc.property
bool = false;
// newly registered interface
if isNewInterface(cam.interfaces.ifc.class.getName()) {
return true;}
// already registered interface
else {
// find all service reference available
ServiceReference[] ref = context.getServiceReferences(
cam.interfaces.ifc.class.getName(), "(service=null)");
for (i=0, i<ref.length, i++){
// compare their properties
ref[i].getProperty(``service'');
bool = cam.interfaces.ifc.property.equals(ref[i]);}
return bool;
```

Listing 4.11: MySIM new service

When a service registers in the `Registry Service`, if it corresponds to a new interface or if it has new property of already existing service interfaces, the `Decision Service` asks the `Generator Service` to provide all possible spontaneous service compositions.

If the service is not new, and provides already existing interfaces with already existing properties, no spontaneous service composition is launched.

Spontaneous composition control

If the trigger of a spontaneous composition is when a service appear in the environment, the stop condition is just before the `Builder Service` implements a composed service. Once a valid combination passes through the `Generator Service` and the `QoS Service`, and before arriving to the `Builder Service`, the `Decision Service` verifies that the generated property that would be bound to the new service is new in the environment and that no occurrences of the same service is noticed. Indeed, two services may be combined more than once, and this would be definitely seen in the property describing the implementation of the newly composed service.

```
// name of interface: cam.interfaces.ifc.class.getName()
// generated property for the interface:
// interface cam.interfaces.ifc.property
bool = false;
// find all service reference available for the same interface
ServiceReference[] ref = context.getServiceReferences(
cam.interfaces.ifc.class.getName(), "(service=null)");
for (i=0, i<ref.length, i++){
// compare their properties
ref[i].getProperty(``service'');
bool = cam.interfaces.ifc.property.equals(ref[i]);}
if (bool = true) {return true;}
else {
// verify the no services are combined twice
bool = occurrenceCheck(cam.interfaces.ifc.property)
return bool;}
```

Listing 4.12: Property control

If this algorithm guarantees the stop of the spontaneous service composition, it has a severe limitation as it can not allow combining services together in multiple ways. Indeed, two operations may combine over several parameter inputs and for that the property generation would be the same even if the implementations are not the same. We are studying how to resolve this limitation by introducing an index to the property specifying which elements is being composed and by that extending the possibilities of compositions.

In the future, we would like to provide the `Decision Service` with means to provide contextual and pervasive strategies to choose the composition technique depending on the environment characteristics and to efficiently deploy integrated services in the environment.

4.7 The MySIM Builder Service

In this section, we detail the `Builder Service` (cf. figure 4.16) of our `MySIM` middleware. The `Builder Service` implements and generates OSGi services for all the composed services provided by the `Generator Service` and `QoS Service`.

We begin by explaining the OSGi bundle which is the unit of deployment of a service in OSGi. Then, we depict how `MySIM Builder Service` implements and generates OSGi services based on this model.

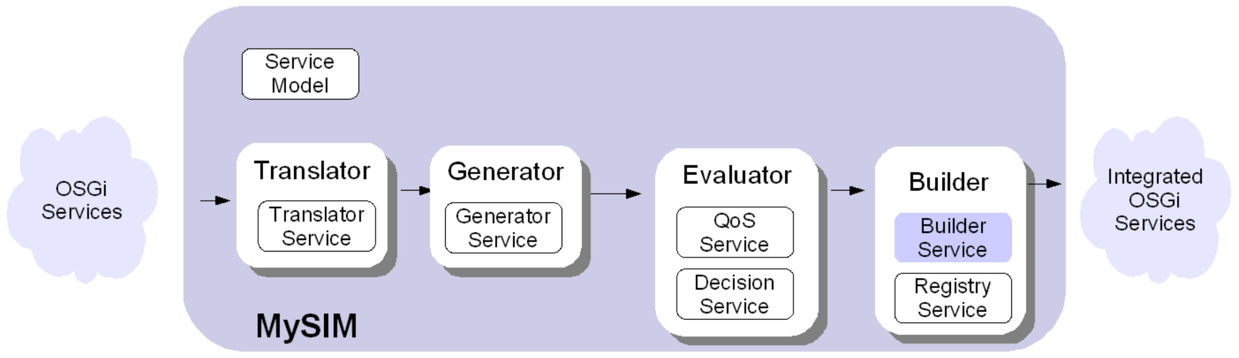


Figure 4.16: Builder Service

OSGi bunlde layer

A bundle can be installed and uninstalled and an OSGi platform verifies that the necessary dependencies are all present. A bundle has a particular class, the activator which contains the code to execute in order to start and stop the application. The bundle is thus a unit of deployment and of execution. A bundle can depend on other software resources to execute. These resources can be particular Java packages (containing Java classes), or particular bundles.

`MySIM` service implementation of the resulting service composition consists in creating new composed services in the environment. In OSGi, creating the service is done by creating a unit of deployment, the 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. Bundles can share Java packages among an exporter bundle and an importer bundle in a well-defined way.

To create a new service that provides the result of a service composition, the `Builder Service` need to tackle several points:

- unit of deployment: a bundle to deploy the new composed service.
- interface implementation: The Java code that provides the implementation of the service interfaces.
- needed libraries: all the possible required libraries depending on the employed composition technique.
- services dependencies: the new service needs to verify the dependencies of the services involved in the integration.

Implementing and generating OSGi bundles

The `Builder Service` tackles each point as follows:

- unit of deployment: it creates the bundle by creating Java classes of an activator, a manifest file, and the service interfaces declarations. The activator is Java class in the bundle implementing an interface that is used to start and stop that bundle, and by that the service. The manifest file is a text file describing the service.
- interface implementations: we provide two different techniques for implementing the operations of an interface, the redirection call to the services that are composed together, done via method call and RMI, and the replication of a copy of the composed services or their implementations done via method call to local replicate available in the newly generated jar bundle.
- needed libraries: in case of replication, the implementations of the replicated services are needed and added to the bundle. The class path to where to find these libraries is specified in the manifest file.
- services dependencies: the new service will have to verify the dependencies of the services involved in the integration.

Finally all these classes are jarred into one bundle. The bundle is then deployed and started in the framework. The corresponding Java code is given listing 4.13

```java
// creating .java
mtc.createDirectory(strDirectory);
mtc.createManifest(ManifestName);
mtc.createActivator(ActivatorName);
mtc.createInterface(ServiceName);
mtc.createService(ServiceName);
// Compilation and generating .class
if (mtc.compileIt("cam/" + functionality1 + functionality2 + "/Activator.java") &&
 mtc.compileIt("cam/" + functionality1 + functionality2 + "/" + functionality1 +
  functionality2 + ".java") ) {
System.out.println("Running␣classes␣:\n\n");
// création of the bundle .jar
if (mtc.jar()) {
System.out.println("jaring␣:\n\n");}
try {
// starting locally the bundle
Comp = context.installBundle(location);
Comp.update();
Comp.start(); }
catch (Exception ex){}}
else {System.out.println("errors␣in␣compilation");}
```

Listing 4.13: Service implementation and generation

Unit of deployment

The `createManifest()` method creates a manifest file that specifies the interface the service exports and any dependencies the service may need to import. In case of composition by replication, the manifest file also indicates the class path where to find the replicates within the bundle.

The `createActivator()` method creates the activator class that is used to start and stop the bundle, and by that the service. It also binds the functional interface implementation of the service to a functional property and that by registering the service interface at the `Registry Service` with this property. The property is automatically generated as described section 3.3.2.2 after a service composition and reflects the properties of the services taking part in the composition process.

The `createInterface()` method specifies all the operations of the newly created service. In our spontaneous service composition model, the new service republish the same interfaces as an available services in the environment. This method re-creates all these operations signatures in an interface class. By that the new implemented service publishes the same operations as an already existing service in the environment.

Implementation code

The `createService()` method provide the implementations code of the new composed service operations. This method specifies the implementation of the interface declaration. This implementation is the real composition of the services involved in the composition. When implementing the operations, the `Builder Service` needs to distinguish between the operations of the interface that have been composed and the other ones that where not composed. Indeed, when two services are composed, it does not mean that all their operations have been composed but only at least one. The new implemented service need to have the same implementations for the operations that where not composed as the service it is equivalent to, and new implementations for the composed ones. The `createService()` is comprised of several methods `createOperation()` that create operations by redirecting the calls to one operation or redirecting the calls to chained operations in case of service composition.

Example 30 *In figure 4.17 an example is given. The new service s_{new} results from the composition of the two services s_i and s_j. The composition result is equivalent to s_i and the operations signatures of s_{new} are the same as those of s_i. The implementations of these operations redirect to those of s_i if no composition is done, and for those where a composition is possible it redirects to the combination of the two composed operations as shown figure 4.17.*

The `Builder Service` implements operations composition via two techniques (cf. table 4.1): redirection call and replication. The redirection call is an RMI call to the services operations taking part in the composition. If a new service is implemented using this technique it will be always dependent on the services it composes. If these services disappear, the new service can no longer execute and need to be stopped by the `Registry Service`. In the replication service, the

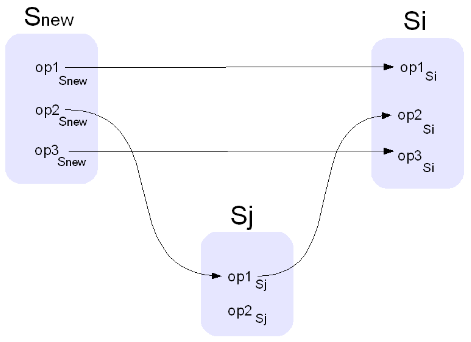


Figure 4.17: Creating the operation implementations of new composed service

new service has a replicate of the services involved in the composition by that way it can continues to execute even if services disappear. Nevertheless the replicate may become obsolete as no update mechanisms are provided.

	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 of S1, S2

Table 4.1: Integration techniques for services S1 and S2

Libraries and dependencies

If the employed technique is a composition by replication, the new service needs to have local access to a replicate of the services implementations it integrates (cf. table 4.1). These service implementations are jarred within the new bundle. The class path to the replicates will be specified in the manifest file by the `createManifest()` method. The new service will have the same dependencies as the service it composes. These dependencies are specified in the import and export part of the manifest of the new service.

4.8 The MySIM Registry Service

Registering services This service (cf. figure 4.18) registers all the newly composed services and is responsible of providing application with the best possible services responding to their needs.

EXAMPLE 31 *The new storage service that results from the composition of storage and naming have the same operation signature and non-functional QoS properties as storage but different implementations with properties:* $Pr = \{< storageftp, integrated > (< naming, atomic >)\}$ *and* $Pr = \{< storageLocal, integrated > (< naming, atomic >)\}$ *(listing 4.14).*

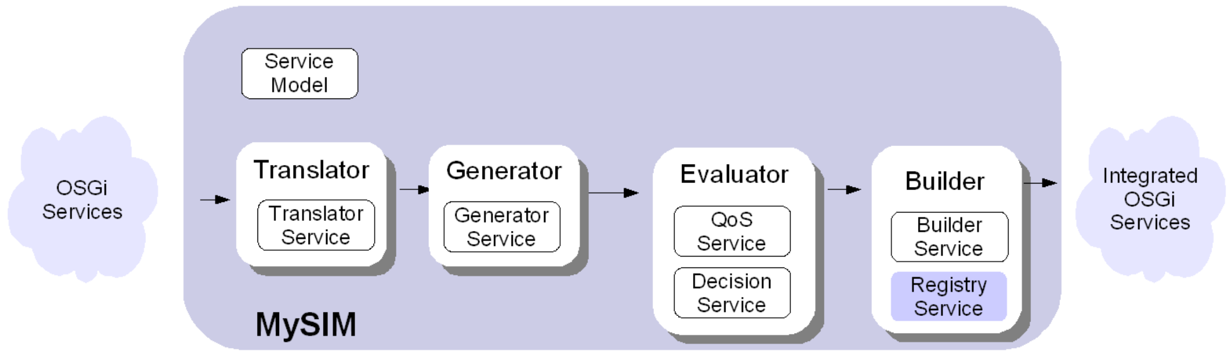


Figure 4.18: Registry Service

```
// Storage service via Ftp
props.put("service", "StorageFtpServiceNamingService");
context.registerService(
cam.interfaces.ifc.StorageIfc.class.getName(), serv, props);
}
// Storage Service
props.put("service", "StorageLocalServiceNamingService");
context.registerService(
cam.interfaces.ifc.StorageIfc.class.getName(), serv, props);
}
```

Listing 4.14: Service storage interface new properties

The new storage service that results from the composition of storage and packaging have the same operation signature but different non-functional QoS properties as storage with properties: $Pr = \{< storageftp, integrated > (< packaging, atomic >)\}$ *and* $Pr = \{< storageLocal, integrated > (< packaging, atomic >)\}$ *(listing 4.15).*

```
// Storage service via Ftp
props.put("service", "StorageFtpServicePackagingService");
context.registerService(
cam.interfaces.ifc.StorageIfc.class.getName(), serv, props);
}
// Storage Service
props.put("service", "StorageLocalServicePackagingService");
context.registerService(
cam.interfaces.ifc.StorageIfc.class.getName(), serv, props);
}
```

Listing 4.15: Service storage interface new properties

The new webcam service resulting from the composition of webcam service and packaging service as specified figure 4.11 keeps its operation signature and non-functional QoS as they respect both services involved in the composition. The new property is $Pr = \{< webcam, integrated > (< packaging, atomic >)\}$ *(listing 4.16.*

```
// Webcam service
props.put("service", "WebcamServicePackagingService");
context.registerService(
cam.interfaces.ifc.WebcamIfc.class.getName(), jmstudio, props);
}
```

Listing 4.16: Service webcam interface new property

The new storage service that results from the composition of storage and naming and Packaging is attached to the new properties: $Pr = \{< storageftp, integrated > (< packaging, atomic >)(< naming, atomic >)\}$ *and* $Pr = \{< storageLocal, integrated > (< packaging, atomic >)(< naming, atomic >)\}$ *and registered as follows (listing 4.17):*

```
// Storage service via Ftp
props.put("service", "StorageFtpServicePackagingServiceNamingService");
context.registerService(
cam.interfaces.ifc.StorageIfc.class.getName(), serv, props);
}
// Storage Service
props.put("service", "StorageLocalServicePackagingServiceNamingService");
context.registerService(
cam.interfaces.ifc.StorageIfc.class.getName(), serv, props);
}
```

Listing 4.17: Service storage interface new properties

After the spontaneous service composition, MyStudio new environment still provides the same services, as it still publishes the same interfaces but it has been extended with several new implementations for these services interfaces. We can distinguish three main functionalities provided by the three services storage, webcam and printing having several possible implementations.

The storage service has two atomic implementations allowing to store images and copy files locally or remotly via ftp. It has also integrated implementations with the naming and packaging services. This variety of available implementations allows the ***Decision Service*** *of* ***MySIM*** *to be able to adapt the application using this functionality to any disappearance that may occurs. If a storage service is no longer available, the search for an equivalent service or almost equivalent one will provide the list of all these storage services with their different implementations and non-functional QoS properties.*

For the webcam service, if two different implementations exist they are completely dependent on the webcam device. If the hardware is no longer reachable it is difficult to find a replacement unless another webcam device is available. On the other hand, the implementations are different and the ***Decision Service*** *can propose to applications one implementation or the other depending on what they need. For applications wishing to manipulate small images the service webcam implementation that is composed with the packaging service fits the most.*

For the printing service, an equivalent service is the printer service with different non-functional QoS properties. The adaptation choice here is not on the functional property as for the webcam service but on the non-functional QoS properties. Depending on what the application needs and prefers, the ***Decision Service*** *chooses the best service that fits its requirements.*

Services adaptation

The `Registry Service` is the one that provides applications with the services they require. We detail in listing 4.18 how the `Registry Service` proceeds.

```
ServiceReference[] refs = context.getServiceReferences(
cam.interfaces.ifc.class.getName(), "(service=prop)");
if (refs != null){
cam.interfaces.ifc ifc =
 (cam.interfaces.ifc) context.getService(refs[0]);}
else {
ServiceReference[] ref = context.getServiceReferences(
  cam.interfaces.ifc.class.getName(), "(service=null)");
        if (ref != null){
        // applying QoSort on the set of founded services
        // profile is the profile of the service required by
        //applications andnot found
                for (i=0; i<ref.length; i++){
                cam.interfaces.ifc ifc[i] =
                (cam.interfaces.ifc) context.getService(ref[i]);
                sortedS = QoSort(profile, ifc[i]);}
        // Choice of the first service of the set
                cam.interfaces.ifc ifc =
                (cam.interfaces.ifc.WebcamIfc) context.getService(sortedS[0]);}
  else {
  // no services available ask Generator Service to
  // search for almost equivalent services, semantic, etc.
  // re-sort the sets following the QoS degree
  // ..
}}
```

Listing 4.18: `Service Registry` adaptation

The `Registry Service` begins by searching for the required service with the specified property. If a service reference is found, the service object is returned to the client.

If no service is found, the `Registry Service` will try to find other services publishing the same interfaces but registered under different properties. If many services are found, it asks the `QoS Service` to calculate the QoS_{degree} relating to the application profile specifying the desired values of non-functional properties and the importance of these properties via wi. Then, the set of services are sorted following this computing and the `Registry Service` returns the first service of the set which corresponds to the best service.

If no services are found, the `Decision Service` is notified and a search for syntactic and semantic equivalent and almost equivalence can start by the `Generator Service`. This is done upon a syntactic and semantic matching via the `Generator Service`. If many services are found, the selection would be done after calculating the QoS_{degree} of these services by the `QoS Service`.

If no services are found, the application call are redirected to a proxy that awaits the appearance of an appropriate service to redirect these calls.

4.9 MySIM Prototype Performance Evaluation

In this section, we evaluate the performance of our MySIM middleware. We begin by evaluating, our service implementation and generation, depending on the employed techniques. Then, we evaluate the syntactic and semantic matching upon the functional services interfaces. The QoS degree function is evaluated depending on the number of services available in the environment. Our service composition and adaptation are finally evaluated using the *MyStudio* use case.

OSGi services implementation and generation

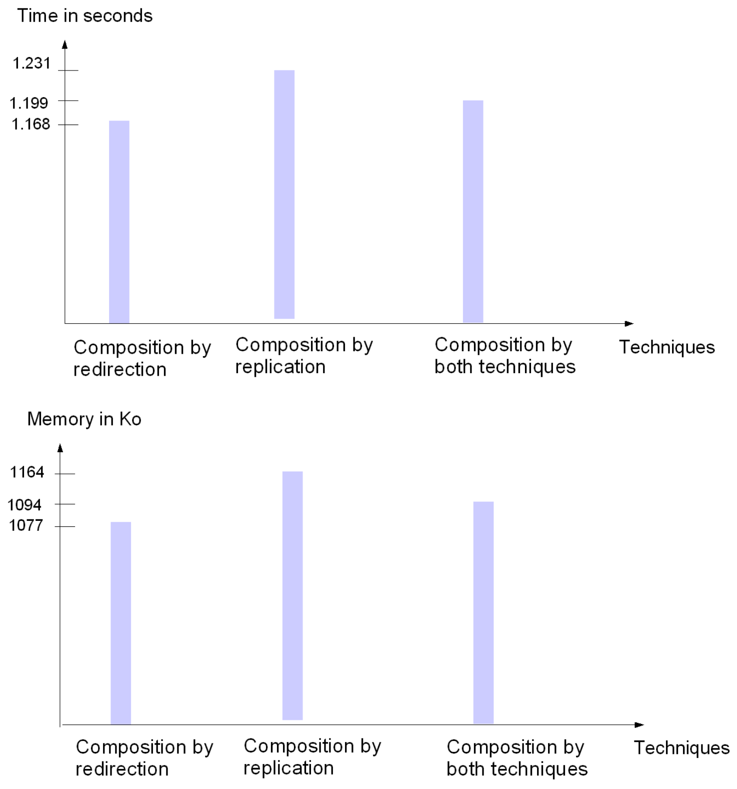


Figure 4.19: Time and memory consumption of two operations composition

The service composition time takes approximatively 1 second to compose two services as shown figure 4.19. It is costly in terms of memory as it uses the **sun.tools.javac.Main** and **sun.tools.jar.Main** classes to build services from scratch. The MySIM service composition implements and generates new services in the environment. If this feature is interesting for the persistence of services in the environment, it is much less interesting in terms of memory consuming than the approaches that compose services on the fly without generating unit of deployment for these services.

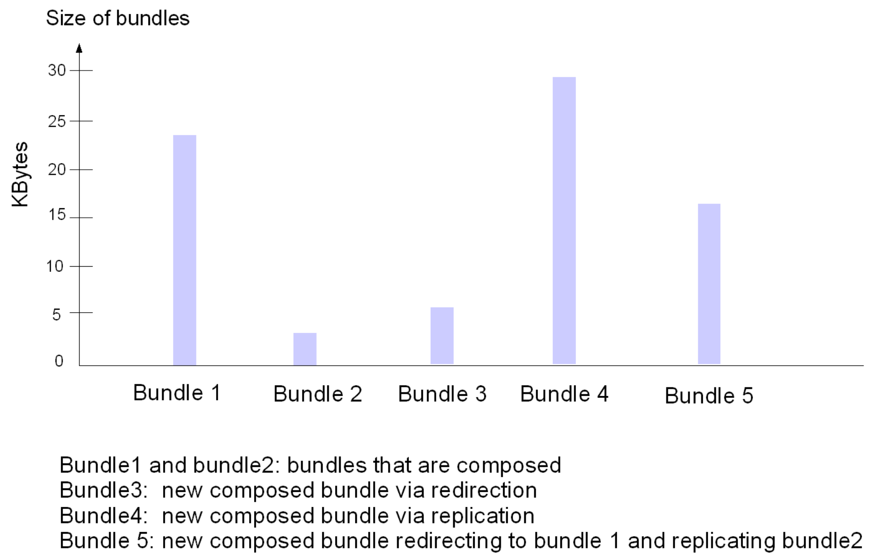


Figure 4.20: Size of bundles depending on the composition techniques

Depending on the employed technique the size of the generated bundle is different as shown (cf figure 4.20). The replication technique generates less network communications but takes much more space than the redirection one, on the device where the new service is deployed. Choosing one or the other technique will have a repercussion on the environment. Indeed, in devices where no space is available, it would be preferable to compose services by redirection. On devices where on the contrary the connectivity is unstable but the space is available the replication method would be preferable.

Syntactic and semantic matching

To evaluate the impact of the syntactic and semantic matching, we evaluated the time execution and memory consuming for our *MyStudio* use case but also for a set of 100 services for services syntactic and semantic matching done by the `Generator Service`. Figure 4.21 gives time and memory values for syntactic service matching done by the `Generator Service` in order to find all the spontaneous composable services in the environment. The results are those applied to *MyStudio*, and to a set of 100 services. The syntactic spontaneous composition takes no time at all, and can be executed over relatively constraints devices.

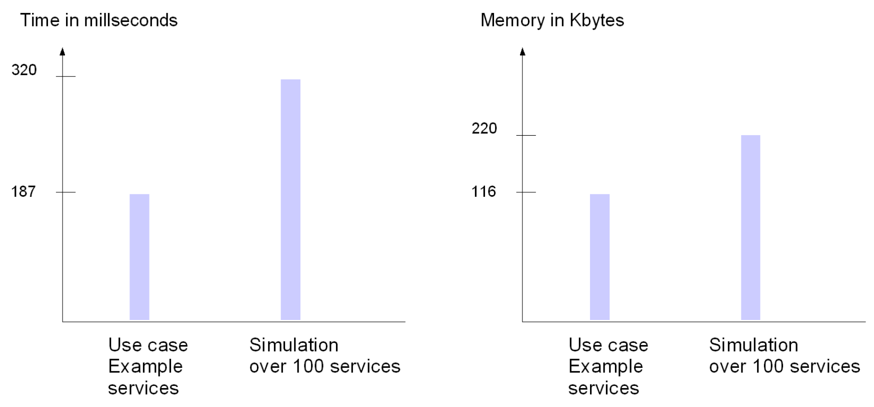


Figure 4.21: Time and memory consumption for syntactic service matching

The semantic matching is much longer than the syntactic one (cf. figure 4.22). The OWL-S API takes about 12 seconds to compare and matches 8 services owl-s descriptions (*MyStudio*) and 55 seconds for about 100 services. The pellet matching engine that reads all the owl-s files by adding them to the reasoner and extracts the inputs, outputs and concepts fields is much slower and much more memory consumer than as simple syntactic matching based on introspection methods provided by the Java language. we conclude that the semantic matching using online semantic reasoning is a very heavy process.

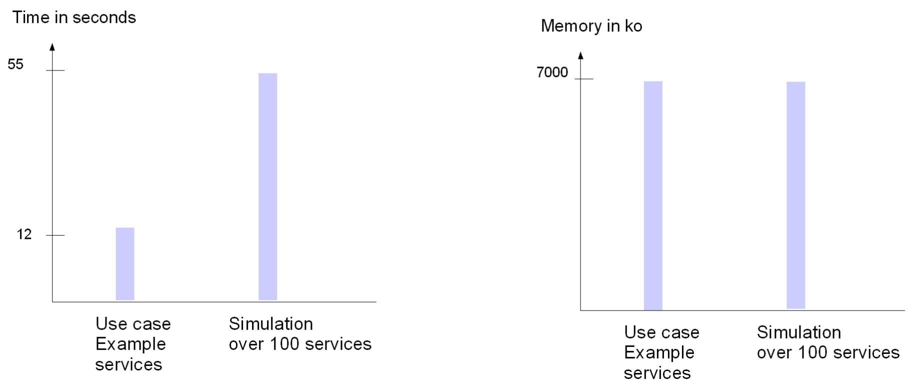


Figure 4.22: Time execution for semantic service matching

The use of a syntactic matching combined with a semantic one, gathers the benefits of both techniques. Indeed, the syntactic matching is quicker and lighter. A semantic verification done upon the resulting services, is faster than if done from the beginning on all the services of the environment. But the semantic matching is important when services are provided in different technologies and for that a more optimized semantic execution engine is needed so that it can better fit pervasive environments.

Handling the QoS non-functional properties

We begin by examining the time execution and memory consumption of the `QoS` function that calculates for a set of non-functional properties the QoS_{degree} related to a predefined service non-functional properties that we wish to replace.

Figure 4.23 gives the time execution and memory consumption for quantitative non-functional properties QoS_{degree} function computing. We suppose that each service has one quantitative non-functional property.

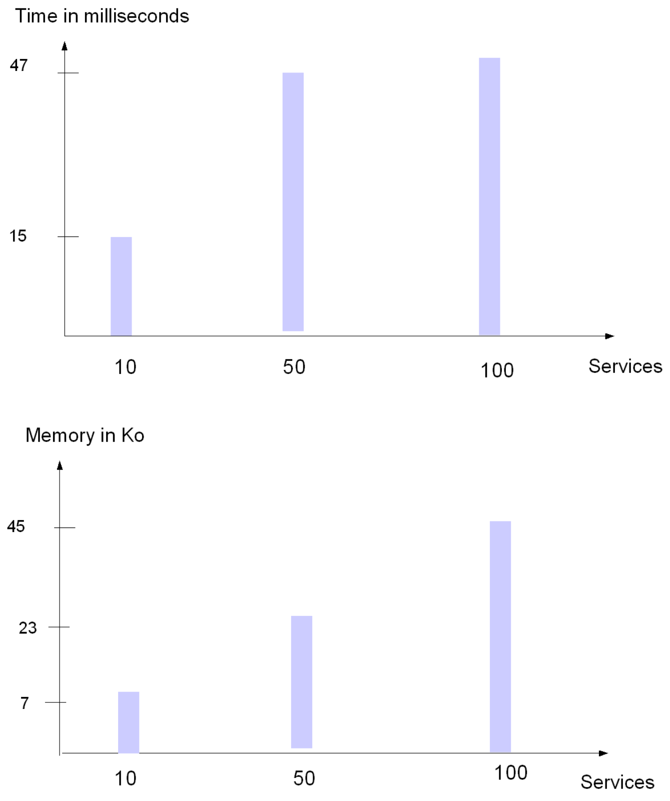


Figure 4.23: Time and memory consumption for QoS degree computing

As we can see the computing time is quick and it does not require much memory. For the qualitative non-functional properties, the concept matching depends on the size of the employed ontology that contains the corresponding concepts. For now no test were conducted to evaluate the time of the execution engine depending on the size of the ontology, but first results show that the time undertaken is much more important than for the quantitative non-functional properties computing.

Once the concept matched, the computing of the semantic distance takes as much time as the one for computing the quantitative non-functional properties QoS degree.

The sorting of the services sets provided by the `QoS Service` is relatively quick as shown figure 4.24.

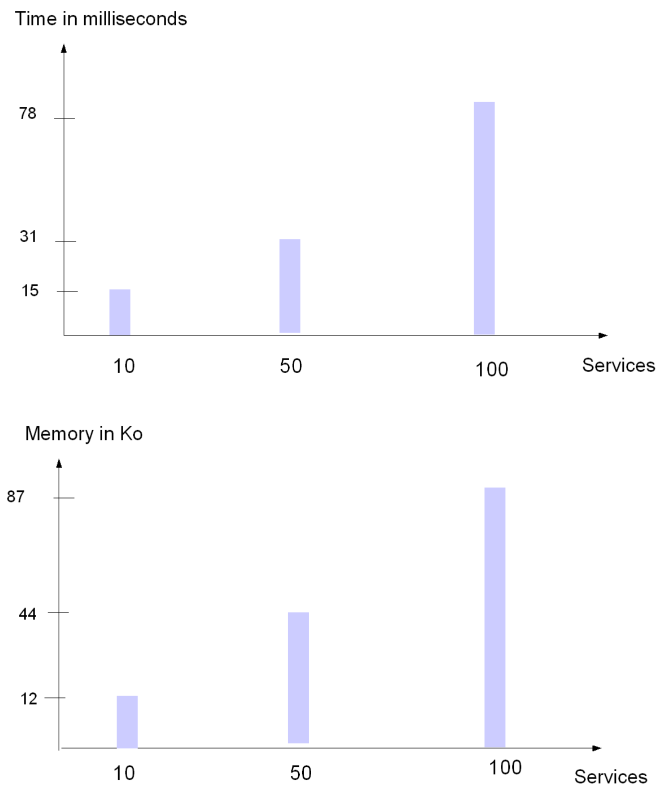


Figure 4.24: Time and memory consumption for services QoS sorting

Spontaneous service composition

The spontaneous service composition time depends strongly on the number of services available in the environment. If the syntactic matching is relatively quick, composing services takes about 1 second for every composition. We can imagine the time needed to compose lots of services. Adding to this the time to semantically match services, we quickly realize that some compositions may occur while the services are no longer available, especially if the environment is highly instable in terms of connectivity.

Our `MySIM` spontaneous composition needs to be applied to relatively stable environments in terms of connectivity, with a good time connection average.

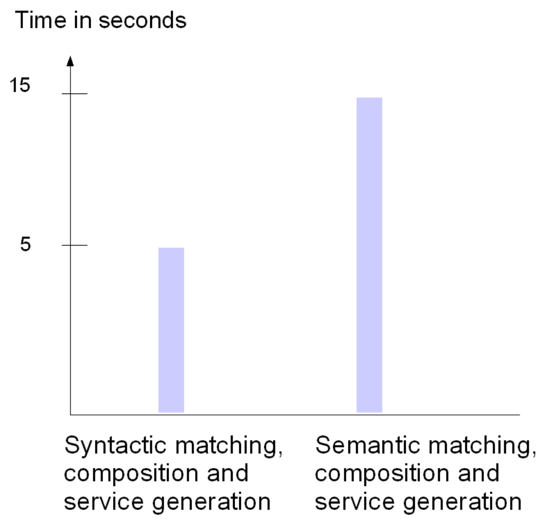


Figure 4.25: Time execution for spontaneous service composition in *MyStudio*

In the `MyStudio` use case, the spontaneous syntactic composition time is about 5 seconds, from the moment the MySIM begins to match service syntactically till the moment it generates and starts all the new services in the context (after two cycle of spontaneous composition). For this use case, this time is relatively good, as we suppose devices can stay connected longer than 5 seconds. If we add a semantic matching, the time goes to 20 seconds because of the cost introduced by the semantic execution engine (cf. figure 4.25).

For environment populated with services, launching a spontaneous composition can be problematic especially if the number of services to compose is important. The spontaneous service composition will need to be scalable to these environments. An idea would be to prioritize the services to compose depending on the connectivity provided by the devices hosting the services. The `Decision Service` may begin by composing the services that may disappear, followed by those providing better connectivity. The `QoS Service` may be used to find these services by analyzing the non-functional property related to connectivity if provided, and if not by querying directly a `Context Service` to get the information from the devices hosting the services.

Spontaneous service adaptation

The `Registry Service` registers the services that appear in the environment and unregisters those that disappear. When an application requires a specific service by specifying its functional interface, the `Registry Service` searches for the reference of the service by searching for a specific property. If the application does not provide any property, the `Registry Service` will find all the available services in the environment that publish the same interface, and upon these apply the `QoS` and `QoSort` functions.

We calculated the time needed for the Registry Service to find a service based on its property and if not to return all the available services in the environment. The results showed that for 25 services in the environment, publishing the same interface, the `Registry Service` instantly found these services and that the adaptation time was reduced to the one of the services QoS degree and services QoS sorting computing defined above.

When a service leaves the environment, the time to adapt to this changes is the time required to compute and sort the QoS degree of available services publishing the same interfaces. If none is available, the time of adaptation will be the sum of time needed to syntactically and semantically find equivalent and almost equivalent services with the time needed to compute and sort the QoS degree over these services.

When a service appears in the environment, the `QoS Service` computes the QoS degree of this services to find if it better suits the applications using equivalent services. If so, the `Registry Service` will propose to applications the new service and the adaptation would be done in no time for the application, as it is a proactive adaptation. For the middleware, the time needed is the time it takes to compute the QoS degree of services.

These different functionalities of QoS degree computing were tested and evaluated just before and we can deduce that the only thing that is time and memory consuming in our implementation are the semantic related aspects due to the chosen execution engine and the time to generate the new services due to our technique based on `sun.tools` libraries.

Chapter Conclusion

In this chapter, we presented the current state of our `MySIM` implementation prototype. For each service we resume what have been already done and what is currently being developed. The `MySIM` project is a private inria gforge project. The `Translator Service` provides manual rules to map an OSGi service model into our SERVICE model. Currently we are improving this module by automating these rules and extending them to other services model for interoperability purposes. The `Generator Service` provides all the matching methods to find equivalent, almost equivalent, and composable services whether syntactically or semantically. For now the syntactic matching is Java related. We aim to extend this matching to other languages such as C++. The semantic matching is relatively slow and costly as it relies on an unoptimized execution engine. Improvement are studied to use lighter engine. The `QoS Service` is in a preliminary development stage, as non-functional QoS are considered as variables instantiated by the operations themselves. No real connection with the context is yet made. Improvement over the prototype are implemented to describe these properties semantically and to provide services and applications that require strong respect of the non-functional QoS in order to execute. The `Decision Service` has an event-based mechanisms to launch the spontaneous service integration once a new service in the environment and to control it in order to stop it when needed (property control). We are studying contextual strategies concerning the spontaneous deployment of the newly integrated services in the environment [Ayed et al. 2008, Ichiro 2005] and a contextual combination of syntactic and semantic matching that provide the best performances depending on the nature of the context. The `Registry Service` is responsible of registering the services in the environment and providing the applications with the best service depending on their needs and what is available in the environment.

Chapter 5

Conclusion

5.1 Contributions

Pervasive computing, the successor of mobile computing, offers challenging problems such as the integration problem. In pervasive environments, functionalities provided by heterogeneous software and hardware resources need to be integrated transparently toward providing users and applications means to execute their daily tasks and activities. In heterogeneous environments where functionalities are provided and described in different models, technologies and languages, integrating these functionalities in the environment is a guarantee for applications and users wishing to use these functionalities in their operations. Indeed, for an entity to be operational in its new environment, it needs to be integrated to this environment. By this way, it can take full advantages of all the possibilities offered by the environment it evolves in.

In this thesis, we focus our interest upon the intersection of two major domains, the service-oriented architectures meeting the pervasive computing, and propose a novel solution to integrate services in a pervasive environment. Functionalities are described and offered in the form of services. Integration is defined by the three operations of transformation, composition, and adaptation. The problem of service integration in pervasive environments is decomposable into the problems of service transformation, service composition, and service adaptation.

The main contributions of this thesis are threefold. They arise from the lacks noticed in current service integration middleware for the pervasive environments. If many middleware dealt with one or more of our service problems - transformation, composition, adaptation - few proposed a unified vision for the service integration in pervasive environment, a management of the functional and non-functional properties of services during the integration, and especially, a spontaneous service integration based on events affecting the environment rather than explicit demands coming from the application layer. This thesis contributed to these three aspects affecting the current service integration middleware.

5.1.1 Unified Vision for the Service Integration

We proposed the SIM model, a Service Integration Middleware model. The SIM model is at a high-level of abstraction, without considering a particular service technology, language, platform or algorithm used in the integration process. The SIM is modeled by four modules: the `Translator`, the `Generator`, the `Evaluator`, and the `Builder`. As an entry point of the SIM are the services of different technologies available in the environment. These services are potential targets for integration. The services are modeled differently as they can be provided from different technology platforms. At the end point of the SIM are the resulting integrated services (by transformation, composition, or adaptation). The `Translator` module is responsible of describing the services available in the environment, the functionalities requested by the applications and users in a system comprehensible language, and for that it transforms the diverse existing technologies that are platform-dependent into one service model independent from the platforms, the SERVICE model. The `Generator` provides new functionalities by composing the available services, or searching for equivalent services for an application adaptation due to the environment changes. It tries to generate one or several composition or adaptation plans with the services available in the vicinity. The `Evaluator` relies on QoS criteria or applications and users profiles to choose the most suitable and realizable integration plan for a given situation. This selection is done from all the plans provided by the `Generator`. The `Builder` executes the selected integration plans and produces several implementations corresponding to the required integrated services.

We proved the genericity of the SIM model by showing that the current service integration middleware, whether the transformational ones, composition ones or adaptation ones can be described by at least one of the SIM modules. Based on this model, we proposed `MySIM` middleware, a service integration middleware for pervasive environments that does the service transformation, composition and adaptation. `MySIM` also deals with the non-functional QoS properties of services and propose spontaneous service integration adapted to pervasive environments.

5.1.2 Functional and Non-Functional QoS Management of the Service Integration

The `MySIM` middleware integrates services based not only on their functional interfaces and operations but also on the non-functional QoS properties they provide or require. For a service composition, a functional and non-functional compatibility need to be verified in order to combine services together. In a service adaptation, replacing one service by another equivalent one at execution time need to take the functional and non-functional properties of services into account, for the replacement to be efficient. We introduced several services relations for equivalence and composition. These relations between services allowed to define and explain the `MySIM` spontaneous service composition and adaptation.

We have first defined a general SERVICE model respecting the service-oriented programming paradigm and general enough to model the current existing service technologies such as the Web services or OSGi service specifications. Based in this model, we distinguished between syntactic and semantic equivalence, functional and non-functional equivalence, and services equivalence and almost equivalence one. We defined service composition for the functional parts of services and stipulated the conditions upon their non-functional properties. We also defined syntactic and semantic composition, as services are not necessarily provided in the same technology languages and the use of semantics is more than necessary to resolve heterogeneity.

5.1.3 Spontaneous Service Integration

Our last contributions dealt with the spontaneous service integration. Spontaneity is defined vis-à-vis to users and applications of the application layer. `MySIM` middleware is capable of integrating services of the environment without external demands coming from the application layers and upon specific events on the functionalities of the environment. The particularity of this spontaneous integration is in the transparency it provides to the application layer. For the users and applications of this layer, the interfaces that are published remain the same even after spontaneous service integrations. When a service appears in the environment, `MySIM` extends the environment with new implementations corresponding to service composition, transformation, or adaptation between services, leaving the services interfaces unchanged. When a service disappears from the environment, `MySIM` adapts transparently the applications execution to the available implementations, hiding from the applications the disappearance of the service.

The spontaneous service integration, is defined using the service equivalence and composition relations. For a spontaneous service composition, services need to be composable and the resulting composition equivalent to a service of the environment. By that the invisibility of the integration for the application layer is respected. A localized spontaneous service composition is provided to fit more with the pervasive nature of the environment allowing two service to compose and produce a service that is equivalent or almost equivalent to one of them. For a spontaneous service adaptation, it can occur upon a service appearance or disappearance. If the current service adaptation mechanisms considered adaptation as a mechanism to mask a loss few propose adaptation as a solution to provide better than what is actually provided. Our service adaptation considers both cases, and replace services by others that are equivalent or almost equivalent. Special attention needs to be given to the non-functional QoS during the adaption process. For a spontaneous service transformation, transformational rules transform from a technology model to our SERVICE model and vice versa. This allows specific services to be used by applications provided in different technology model, or different services from different providers to be composed together.

5.2 Perspectives

We can distinguish two kinds of perspectives, the short term perspectives which are related to improving our `MySIM` middleware, and the long term perspectives which are more related to our integration problem in pervasive computing.

5.2.1 Improvement of `MySIM` Middleware

`MySIM` middleware can be improved and that by tackling several key points. We explain these points and try to propose alternatives to the encountered problems.

SERVICE Model

We have proposed a SERVICE model that respects the SOA and provides means to do service transformation, composition, and adaptation in pervasive environments. This model can be improved as some limitations raised while defining and implementing our `MySIM` middleware. These limitations concern the functional property that is bound to the functional interfaces and that describes its implementation. While composing services, `MySIM` used this property to verify whether two services were already composed and we noticed that the composition description was not well described in the property. Indeed, if a functional property gives information about the services that are composed together, it does not specify on which operations they are composed or even more on which input parameters of these operations. This information is valuable in order to distinguish between two composed operations but not over the same inputs parameters. One idea is to extend the property tuple that contains for now a name and a value to contain also a two dimension vector specifying which operation is being composed in a service and on which parameter. The matching between properties will take into consideration this new element to determine whether two implementation properties of two services are equal or not.

Service Transformation

Our `MySIM` prototype provides a `Translator Service` that only transforms services implemented following the OSGi specification under Felix platform to our SERVICE model. It also provides a `Builder Service` that generates and implements the results of a service composition from the SERVICE model to the OSGi services over the Felix platform. The `Translator Service` needs to be extended with transformational rules that transform from another technology model such as .Net services to our SERVICE model and vice versa. Another way in providing interoperability can be by using interoperable middleware such as middleware developed in Amigo [Georgantas 2005] proposing interoperable mechanisms for services discovery and interaction. The Amigo interoperable middleware core implements middleware-layer interoperability methods so that services of the networked home environment may be discovered and accessed by the other networked services, and conversely, be independent of the service-oriented middleware technology the various networked services are implemented upon.

Service Composition

Our service composition as we propose it, combines service two by two and generates a new service that can also takes part in another service composition. The `Generator Service` as `MySIM` provides it, compares and matches services two by two, and returns all the possible spontaneous service composition from the set of available services. The `Generator Service` is not able to combine more than two services in one composition cycle. The extension to a number n, $n \in \mathbb{N}$ of services is possible, as the condition for the spontaneous service composition remains the same. The services composition relations need to verify the same conditions as in two by two service composition on the parameters outputs and inputs of the n chained services. The non-functional QoS properties of the combined services need to be compatible. Finally, the overall resulting composition service needs to be equivalent or almost equivalent to a given service in the environment for respecting the spontaneity of the service composition. A possible future work is to try and see when can we define spontaneous composition of n services as a k spontaneous two by two services composition.

Service Adaptation

Our service adaptation takes the functional and non-functional properties of a service into account when searching for equivalent services for replacement. The adaptation is seen as a service substitution as in other works such as [Fredj et al. 2008]. One aspect that is not tackled by our `MySIM` middleware is the state of a service that disappears while executing. If a service disappears while executing an application needs, to replace it in a transparent way, `MySIM` needs not only to find equivalent services in terms of functional and non-functional QoS properties but to know from which state to start the execution of the new service, so that the application does not loose what has been already executed by the previous service. Mechanisms of logging and checkpoints need to be introduced at the service execution time level to save the state of a service at runtime. These mechanisms allows our `MySIM` to keep a trace over the state of services and to know when they disappear at which state of execution they were. By this way, once new equivalent services are found, `MySIM` can specify the state from which the services need to start the execution, allowing to the applications and users a transparent replacement of the disappeared services.

Service Composition by Replication

The `Builder Service` provides two ways for composing services, the redirection and replication of services. The replication we propose do not take into account any changes or evolution undertaken by the initial services that have been replicated. A service combining two other services by replication does not care of any evolution affecting the initial services and relies only on the replicates it has. An interesting issue is to provide a composition technique more adapted to the pervasive environment and that by proposing a pervasive replication of services.

These replications need to be aware of any changing affecting the initial services and have to adapt to these changes. Studies should be conducted to know the costs of these 'copies' communications and to see whether it is more interesting to dis-integrate and re-integrate the composed service if changes affect the initial services or just consider the services that have changed as new services launching by that new spontaneous service composition in the environment.

`MySIM` Implementation Prototype

The actual developed prototype is more a proof of concept and is available as a private project on the inria gforge website, and several aspects need special improvements. For the service transformation, the genericity of our SERVICE model may be proved with another service model than OSGi. Taking the web services technology for example and showing that with simple mapping rules, we can pass from the web services model to our SERVICE model is an issue to dig. The semantic interface matching employed uses an unoptimized matching engine based on owl-s API. We can improve the matching time and memory consuming by employing techniques as in PERSE [Mokhtar 2007] that propose efficient semantic service matching using encoding classified ontologies. If the number of services become very important, the spontaneous service integration may cause problems in the network communications, as most of the calls between services are done via the network, but also in the deployment of services as for each composition a bundle is created and deployed. Considering n the number of available services in the environment, the spontaneous integration may propose C_n^2 possible compositions. If n grows, these compositions will generate a huge number of services and hence bundles on devices. An issue is to propose a scalable spontaneous integration by proposing pervasive strategies to adapt the spontaneity of the integration to the resources available in the environment and the quality of the employed networks. Another issue is to decide where to deploy the newly integrated services. For now the deployment is done on devices with sufficient resources, but some propositions about letting the `Decision Service` spontaneously and contextually decide about where to deploy the newly integrated services can be studied.

5.2.2 Integration Everywhere

In this thesis, we were interested in the integration in the middleware layer. One of the long term perspective is to consider the integration not only in the middleware layer but everywhere. Integration needs to be everywhere in a pervasive environment (cf. figure 5.1). Indeed, providing mechanisms to integrate services is not useful if the infrastructures these software resources rely on or if the applications that are constructed upon these services do not also consider the problem of integration.

At the network layer, the integration can be defined as integrating all possible communication technologies (wired or wireless) into seamless platforms that can communicate regardless the mobility of software and hardware resources. If the devices upon which services are deployed are not connected, services cannot communicate in order to integrate, compose or simply interact.

At the application layer, the integration is more user centric and allows users through natu-

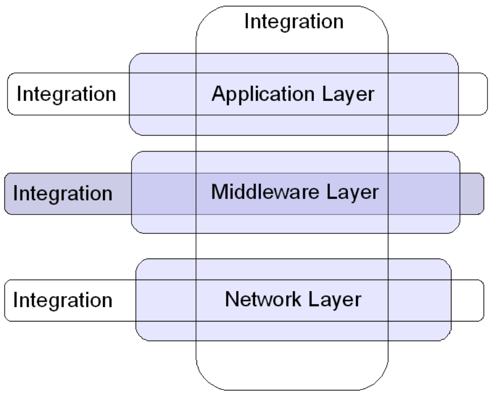


Figure 5.1: Integration every where

ral interfaces, such as speech, vision, and touch to interact wherever he is with the computing infrastructures. Pervasive applications that are user centric are located in the application layer and constructed from services assemblage. The composition and adaptation defined by `MySIM` in the middleware layer are easily mapped and used by applications for composition and adaptation purposes. Every application publishes a profile that depends strongly on the context and allows applications to compose and adapt services that compose them depending on the context these applications evolve in.

A step further is to study a vertical integration related to the three horizontal integration defined at each layer of the computing paradigm (cf. figure 5.1). This integration will tackle the interfaces between each layer and propose solutions to provide integrated environments to people going from the network layer to the application one. The ultimate goal is to develop methods to realize end-to-end Quality of Service as requested by the end-user over multi-domain network infrastructures in a cost-effective manner. To achieve this goal, both horizontal integration over multiple domains and vertical integration over the end user, application/middleware and network layer are researched cf. figure 5.2).

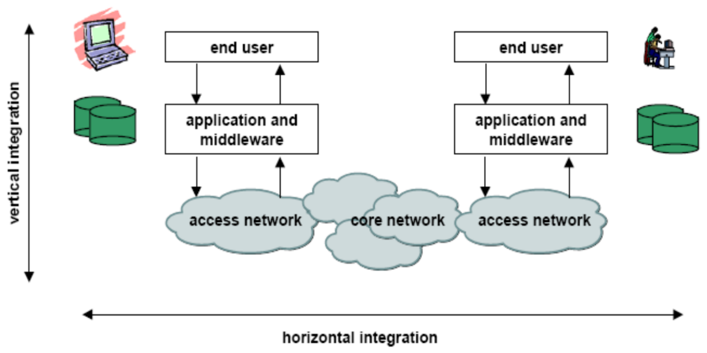


Figure 5.2: Vertical and horizontal integration

5.2.3 From Pervasive Computing to Ambient Intelligence

Satyanarayanan [Satyanarayanan 2001] identified the effective use of smart spaces, invisibility, and localized scalability as the challenges that made mobile computing evolves to pervasive computing. We think that the integration challenge is a first building block among others that make pervasive computing evolves to what we now call Ambient Intelligence (cf. figure 5.3).

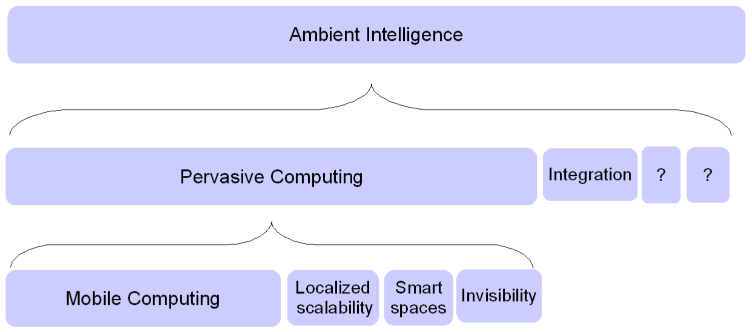


Figure 5.3: Ambient Computing

Ambient intelligence are environments that are sensitive and responsive to the presence of people. Devices work in concert to support people in carrying out their everyday life activities and tasks in an easy and natural way using information and intelligence that is hidden in the network connecting these devices, in the devices themselves and in the software deployed upon them. As these devices grow smaller, more connected and more integrated into the environment, the technology disappears into the surroundings until only the user interface remains perceivable by users. Our spontaneous integration as defined in this thesis provides a building block in integrating smartly the available technology around the user. Possible extensions for our work and that go in the direction of Ambient Intelligence is to take the user into account when we spontaneously transform, compose, and adapt services. Our model is internally controlled (via well defined syntax and semantics) and for now users can not modify or re-parametrize the spontaneity to their ease. One possibility is to redefine the context by taking into account the users and applications profiles and preferences. For now, `MySIM` context is constructed around the services that come and leave, leading to a spontaneous integration sensitive to what is available in term of functionalities but not necessarily in terms of users real needs.

The other building blocks for this evolution are the anticipation and smartness of the computing environment. If nowadays pervasive computing offer to user huge possibilities in terms of hardware or software resources, they very shyly anticipate the user expectations and are not smart enough to adapt to user demands without constantly interrupting him. The ambient intelligence will find its sources in the pervasive computing meeting the artificial intelligence.

Bibliography

[Alliance 2005] OSGI Alliance, *OSGi Service Platform, Core Specification Release 4*, Draft, 07 2005.

[Ayed et al. 2008] Dhouha Ayed, Chantal Taconet, Guy Bernard et Yolande Berbers, ≪ CADeComp: Context-aware deployment of component-based applications ≫, *Journal of Network and Computer Applications*, p. 224–257, vol. 31, n°3, 2008.

[Becker et al. 2004] Christian Becker, Marcus Handte, Gregor Schiele et Kurt Rothermel, ≪ PCOM - A Component System for Pervasive Computing ≫, *the 2nd IEEE Annual Conference on Pervasive Computing and Communications (PERCOM'04)*, IEEE Computer Society, Washington, DC, USA, mars 2004.

[Bellavista et al. 2003] Paolo Bellavista, Antonio Corradi, Rebecca Montanari et Cesare Stefanelli, ≪ Context-aware middleware for resource management in the wireless internet ≫, *IEEE Transactions on Software Engineering*, p. 1086–1099, vol. 29, n°12, 2003.

[Bhagwat et al. 1996] Pravin Bhagwat, Satish Tripathi et Charles Perkins, ≪ Network Layer Mobility: An Architecture and Survey ≫, *IEEE Personal Communications*, p. 54–64, vol. 3, n°3, 1996.

[Bruneton et al. 2002] Eric Bruneton, Thierry Coupaye et Jean-Bernard Stefani, ≪ Recursive and Dynamic Software Composition with Sharing ≫, *Seventh International Workshop on Component-Oriented Programming (WCOP02) at ECOOP 2002*, juin 2002, Malaga, Spain.

[Bruneton 2004] Eric Bruneton, *Developing with Fractal*, The ObjectWeb Consortium, France Telecom (R&D), mars 2004, version 1.0.3.

[Capra et al. 2003] Licia Capra, Wolfgang Emmerich et Cecilia Mascolo, ≪ CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications ≫, *IEEE Transactions on Software Engineering*, p. 929–945, vol. 29, n°10, 2003.

[Casati et al. 2000] Fabio Casati, Ski Ilnicki, Li jie Jin, Vasudev Krishnamoorthy et Ming-Chien Shan, ≪ Adaptive and Dynamic Service Composition in eFlow ≫, *CAiSE '00: Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, p. 13–31, Springer-Verlag, London, UK, 2000.

[Chakraborty et al. 2005] Dipanjan Chakraborty, Anupam Joshi, Tim Finin et Yelena Yesha, ≪ Service Composition for Mobile Environments ≫, *Journal on Mobile Networking and Applications, Special Issue on Mobile Services*, p. 435–451, vol. 10, n°4, 2005.

[Chan et Chuang 2003] Alvin T.S. Chan et Siu-Nam Chuang, ≪ Mobipads: a reflective middleware for context-aware mobile computing ≫, *IEEE Transactions on Software Engineering*, p. 1072–85, vol. 29, n°12, 2003.

[Chetan et al. 2005] Shiva Chetan, Jalal Al-Muhtadi, Roy Campbell et M.Dennis Mickunas, ≪ Mobile Gaia: A Middleware for Ad-hoc Pervasive Computing ≫, *IEEE Consumer Communications & Networking Conference (CCNC 2005)*, Las Vegas, USA, janvier 2005.

[Coalition 2003] The OWL Services Coalition, *OWL-S: Semantic Markup for Web Services*, OWL Services Coalition, 2003, White paper.

[Constantinescu et al. 2004] Ion Constantinescu, Boi Faltings et Walter Binder, ≪ Large Scale, Type-Compatible Service Composition ≫, *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, p. 506, IEEE Computer Society, Washington, DC, USA, 2004.

[Cooperation 2000] Microsoft Cooperation, *Understanding UPnP : A White Paper*, rapport technique, UPnP Forum, 2000.

[Coulouris et al. 2001] George Coulouris, Jean Dollimore et Tim Kindberg, *Distributed Systems Concepts and Design*, ADDISON-WESLEY, 2001.

[dCastro et al. 2006] Valeria de Castro, Esperanza Marcos et Marcos López Sanz, ≪ A model driven method for service composition modelling: a case study ≫, *Int. J. Web Engineering and Technology*, p. 335–353, vol. 2, n°4, 2006.

[dCastro et al. 2007] Valeria de Castro, Juan Manuel Vara et Esperanza Marcos, ≪ Model Transformation for Service-Oriented Web Applications Development ≫, *3rd International Workshop on Model-Driven Web Engineering*, juillet 2007.

[Debaty et al. 2003] Philippe Debaty, Patrick Goddi et Alex Vorbau, *Integrating the physical world with the web to enable context-enhanced services*, rapport technique, Technical report, Hewlett-Packard, septembre 2003.

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

[Fox et al. 1996] Armando Fox, Steven D. Gribble, Eric A. Brewer et Elan Amir, ≪ Adapting to Network and Client Variability via On-Demand Dynamic Distillation. ≫, *Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, octobre 1996.

[Fredj et al. 2008] MANEL FREDJ, NIKOLAOS GEORGANTAS et VALÉRIE ISSARNY, ≪ Dynamic Service Substitution in Service-Oriented Architectures ≫, *IEEE Services 2008-SCC 2008*, juillet 2008.

[Frei 2005] ANDREAS RALPH FREI, *Jadabs - An Adaptive Pervasive Middleware Architecture*, Thèse de doctorat, Swiss Federal Institute Of Technology Zurich, 2005.

[Frénot et al. 2003] STÉPHANE FRÉNOT, ANIS KRICHEN et STÉPHANE UBÉDA, ≪ Light Support for Dynamic and Pervasive Services on P2P Networks ≫, *Special Theme: Applications and Service Platforms for the Mobile User, ERCIM news*, n°54, 2003.

[Fujii et Suda 2005] KEITA FUJII et TATSUYA SUDA, ≪ Semantics-based dynamic service composition ≫, *IEEE Journal on Selected Areas in Communications*, p. 2361– 2372, vol. 23, n°12, décembre 2005.

[Garlan et al. 2002] DAVID GARLAN, DAN SIEWIOREK, ASIM SMAILAGIC, et PETER STEENKISTE, ≪ Project Aura: Towards Distraction-Free Pervasive Computing ≫, *IEEE Pervasive Computing, special issue on "Integrated Pervasive Computing Environments"*, p. 22–31, vol. 21, n°2, 2002.

[Georgantas 2005] NIKOLAOS GEORGANTAS (éd.), *Detailed Design of the Amigo Middleware Core: Service Specification, Interoperable Middleware Core, Delivrable D3.1b, IST Amigo project*, 2005.

[Gomolski 1997] BARB GOMOLSKI, ≪ Messaging Middleware Initiative Takes a Hit ≫, *Computerworld*, vol. 31, n°39, septembre 1997.

[Gu et al. 2004] TAO GU, HUNG KENG PUNG et DA QING ZHANG, ≪ A Middleware for Building Context-Aware Mobile Services ≫, *Proceedings of IEEE Vehicular Technology Conference*, Los Angeles, USA, 2004.

[Hashemian et Mavaddat 2006] SEYYED VAHID HASHEMIAN et FARHAD MAVADDAT, ≪ A Graph-Based Framework for Composition of Stateless Web Services ≫, *the European Conference on Web Services (ECOWS'06)*, IEEE Computer Society, 2006.

[Hoareau et Mahéo 2006] DIDIER HOAREAU et YVES MAHÉO, ≪ Ubiquitous Fractal Components ≫, *5th Fractal Workshop, ECOOP'2006*, juillet 2006.

[Ichiro 2005] SATOH ICHIRO, ≪ Dynamic Deployment of Pervasive Services ≫, *Proceedings of the IEEE International Conference on Pervasive Services (ICPS'00)*, p. 302–311, IEEE Computer Society, juillet 2005.

[Ishikawa et al. 2005] FUYUKI ISHIKAWA, NOBUKAZU YOSHIOKA et SHINICHI HONIDEN, ≪ Mobile agent system for Web service integration in pervasive network ≫, *Syst. Comput. Japan*, p. 34–48, vol. 36, n°11, 2005.

[Iverson 2004] WILL IVERSON, *Real Web services*, O'Reilly, octobre 2004.

[Juric et al. 2006] Matjaz Juric, Poornachandra Sarang et Benny Mathew, *Business Process Execution Language for Web Services (2nd edition)*, PACKT Publishing, 2006.

[Kalasapur et al. 2005] Swaroop Kalasapur, Mohan Kumar et Behrooz Shirazi, ≪ Seamless service composition (SeSCo) in pervasive environments ≫, *International Multimedia Conference, Proceedings of the first ACM international workshop on Multimedia service composition*, p. 11–20, ACM Press, 2005.

[Kalasapur et al. 2007] Swaroop Kalasapur, Mohan Kumar et Behrooz Shirazi, ≪ Dynamic Service Composition in Pervasive Computing ≫, *IEEE Trans. Parallel Distrib. Syst.*, p. 907–918, vol. 18, n°7, 2007.

[Kistler et Satyanarayanan 1992] James J. Kistler et Mahadev Satyanarayanan, ≪ Disconnected Operation in the Coda File System ≫, *ACM Transactions on Computer Systems*, février 1992.

[Kjaer 2007] Kristian Ellebaek Kjaer, ≪ A SURVEY OF CONTEXT-AWARE MIDDLEWARE ≫, *25th IASTED International Multi-Conference, Software Engineering*, Springer-Verlag Berlin Heidelberg, 2007.

[Kumaran 2002] S. Ilango Kumaran, *JINI Technology An Overview*, Prentice Hall PTR, 2002.

[LBM et Satyanarayanan 1995] Maria R. Ebling Lily B. Mummert et Mahadev Satyanarayanan, ≪ Exploiting Weak Connectivity for Mobile File Access ≫, *the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, USA, décembre 1995.

[Le Mouël et al. 2002] Frédéric Le Mouël, Françoise André et Maria-Teresa Segarra, ≪ AeDEn: An Adaptive Framework for Dynamic Distribution over Mobile Environments ≫, *Annales des Télécommunications*, p. 1124–1148, vol. 57, n°11-12, novembre-décembre 2002.

[Loritsch 2001] Berin Loritsch, *Developing With Apache Avalon*, rapport technique, Apache Software Foundation, 2001.

[Medjahed et al. 2003] Brahim Medjahed, Athman Bouguettaya et Ahmed K. Elmagarmid, ≪ Composing Web services on the Semantic Web ≫, *The VLDB Journal*, p. 333–351, vol. 12(4), 2003.

[Milanovic 2006] Nikola Milanovic, *Contract-based Web Service Composition*, Thèse de doctorat, University of Berlin, 2006.

[MIT. Project Oxygen 2007] MIT. Project Oxygen, *Pervasive, Human-Centered Computing*, Website: http://oxygen.lcs.mit.edu/, 2007.

[Mokhtar 2007] Sonia Ben Mokhtar, *Semantic Middleware for Service-Oriented Pervasive Computing*, Thèse de doctorat, University of Paris 6, 2007.

[Mukerji et Miller 2003] Jishnu Mukerji et Joaquin Miller, *Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1*, rapport technique, OMG's Architecture Board, juin 2003.

[Mullender 1993] Sape Mullender, *Distributed Systems*, ADDISON-WESLEY, 1993.

[Munoz et al. 2004] Javier Munoz, Vicente Pelechano et J.Fons, ≪ Model Driven Development of Pervasive Systems ≫, *International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, p. 3–14, 2004.

[Myerson 2002] Judith M. Myerson, *The Complete Book of Middleware*, AUERBACH Publications, 2002.

[Pagels 2005] Manager Michael Pagels, *The DARPA Agent Markup Language*, http://www.daml.org/, 2005.

[Paolucci et al. 2002] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne et Katia Sycara, ≪ Semantic matching of Web Services capabilities ≫, *Lecture Notes in Computer Science*, 2002.

[Pellegrini et Riveill 2003] Marie-Claude Pellegrini et Michel Riveill, ≪ Component management in a dynamic architecture ≫, *Special issue of The Journal of Supercomputing*, p. 151–159, vol. 24, n°2, février 2003.

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

[Puder et al. 2005] Arno Puder, Kay Römer et Frank Pilhofer, *Distributed Systems Architecture: A Middleware Approach*, ELSEVIER, 2005.

[Ranganathan et al. 2004] Anand Ranganathan, Jalal Al-Muhtadi, Shiva Chetan, Roy Campbell et M. Dennis Mickunas, ≪ Middlewhere: A middleware for location awareness in ubiquitous computing applications ≫, *International Middleware Conference (Middleware 2003)*, p. 397–416, Springer-Verlag Berlin Heidelberg, 2004.

[Rao et Su 2004] Jinghai Rao et Xiaomeng Su, ≪ A Survey of Automated Web Service Composition Methods ≫, *the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC*, 2004.

[Raverdy et al. 2006] Pierre-Guillaume Raverdy, Valérie Issarny, Rafik Chibout et Agnès de La Chapelle, ≪ A multi-protocol approach to service discovery and access in pervasive environments ≫, *MOBIQUITOUS - The 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services*, 2006.

[Roman et Campbell 2003] Manuel Roman et Roy H. Campbell, « A Middleware-Based Application Framework for Active Space Applications », *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, p. 997–1013, Springer-Verlag Berlin Heidelberg, 2003.

[Royer et al. 1999] Elizabeth M. Royer, Satish Tripathi et Chai-Keong Toh, « A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks », *IEEE Personal Communications*, p. 46–55, vol. 6, n°2, 1999.

[Satyanarayanan et al. 1994] Mahadev Satyanarayanan, Brian Noble, Puneet Kumar et Morgan Price, « Application-aware adaptation for mobile computing », *the 6th workshop on ACM SIGOPS European workshop: Matching operating systems to application needs*, Wadern, Germany, septembre 1994.

[Satyanarayanan 1996] Mahadev Satyanarayanan, « Fundamental Challenges in Mobile Computing », *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1996.

[Satyanarayanan 2001] Mahadev Satyanarayanan, « Pervasive Computing: Vision and Challenges », *IEEE Personal Communication*, août 2001.

[Sirin et al. 2003] Evren Sirin, James Hendler et Bijan Parsia, « Semi-automatic Composition of Web Services using Semantic Descriptions », *Web Services: Modeling, Architecture and Infrastructure Workshop*, ICEIS, Angers, France, avril 2003.

[Sorensen et al. 2004] Carl-Fredrik Sorensen, Maomao Wu, Thirunavukkarasu Sivaharan, Gordon S. Blair, Paul Okanda, Adrian Friday et Hector Duran-Limon, « A context-aware middleware for applications in mobile ad hoc environments », *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing (MPAC'04)*, p. 107–110, Springer-Verlag Berlin Heidelberg, 2004.

[Terry et al. 1995] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer et Carl H. Hauser, « Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System », *the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, USA, décembre 1995.

[Vallée et al. 2005] Mathieu Vallée, Fano Ramparany et Laurent Vercouter, « A Multi-Agent System for Dynamic Service Composition in Ambient Intelligence Environments », *Doctoral Colloquium - Pervasive 2005*, Munich, Germany, 2005.

[Vallée et al. 2007] Mathieu Vallée, Fano Ramparany et Laurent Vercouter, « Using Device Services and Flexible Composition in Ambient Communication Environments », *1st international workshop on requirements and solutions for pervasive softwares infrastructures (RSPSI)*, Dublin, Ireland, 2007.

[Walsh 2002] Aaron E. Walsh, *UDDI, SOAP and WSDL: the web services specification Reference book*, Pearson Education, avril 2002.

[Want et al. 1992] Roy Want, Andy Hopper, Veronica Falcão et Jonathan Gibbons, « The Active Badge Location System. », *ACM Transactions on Information Systems*, janvier 1992.

[Yang et al. 2005] Zhonghua Yang, Robert Gay, Chunyan Miao, Jing-Bing Zhang, Zhiqi Shen, Liqun Zhuang et Hui Mien Lee, « Automating integration of manufacturing systems and services: a semantic Web services approach », *Industrial Electronics Society (IECON) 31st Annual Conference of IEEE*, p. 6, IEEE Conference Proceeding, 2005.