Semantic Description of Services and Service Factories for Ambient Intelligence

Rémi Emonet

GRENOBLE INP

n° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

Docteur Du Groupe Grenoble INP

Spécialité : Informatique

préparée au Laboratoire d'Informatique de Grenoble

dans le cadre de l'École Doctorale Mathématiques,
Sciences et Technologies de l'Information, Informatique

présentée et soutenue publiquement

par

Rémi Emonet

le 23 septembre 2009

Description Sémantique de Services et d'Usines à Services pour l'Intelligence Ambiante

---

Semantic Description of Services and Service Factories for Ambient Intelligence

---

Directeur de thèse : James L. Crowley
Co-directeur de thèse : Dominique Vaufreydaz

JURY

Pr. Marie-Christine Rousset	Président
Pr. James L. Crowley	Directeur de thèse
Dr. Dominique Vaufreydaz	Co-directeur de thèse
Pr. Gregory D. Abowd	Rapporteur
Pr. David Simplot-Ryl	Rapporteur
Pr. Michel Riveill	Examinateur

Résumé

Ce manuscrit étudie l'adéquation des approches à services, ainsi que leur amélioration, pour l'intégration dynamique et la réutilisation logicielle dans le cadre de l'informatique ambiante. Le dernier chapitre de ce manuscrit est un résumé étendu, en français, du manuscrit complet.

La coévolution du matériel informatique et du logiciel a donné naissance à l'informatique ubiquitaire. Dans leur vie de tous les jours, les gens sont largement entourés par des appareils électroniques dotés de capacités de calcul et de communication. Les gens attendent de plus en plus des manifestations d'intelligence de la part de ce réseau d'appareils. Le domaine de l'intelligence ambiante essaie de satisfaire ces attentes en utilisant ces appareils pour percevoir l'activité des personnes et raisonner pour agir sur l'environnement de manière à rendre un service pertinent à l'utilisateur.

L'intelligence ambiante est un domaine hautement interdisciplinaire et implique de nombreux domaines de recherche. C'est pourquoi les avancées dans ce domaine sont conditionnées par l'interaction et l'intégration entre ces nombreuses disciplines. Ce manuscrit s'intéresse à la problématique de capitalisation des travaux existants, ainsi qu'à l'intégration dynamique de logiciels et d'appareils développés indépendamment les uns des autres.

Notre approche consiste à utiliser des méthodes existantes comme les architectures à services et le web sémantique, et à les adapter à nos besoins particuliers et au large publique des spécialistes des différents domaines. Nous étudions des systèmes existants pour identifier des possibles améliorations du point de vue de l'intégration et de la réutilisation. Nous proposons une intergiciel à services et une interface utilisateur graphique associée qui sont utilisables et extensibles par les différents spécialistes. En combinant et en étendant les approches à services et les principes du web sémantique, nous proposons une nouvelle méthode de conception facilitant l'intégration dynamique et la composition de services pour l'intelligence ambiante. Cette méthode de conception se base en partie sur le nouveau concept d'usines à services qui sont capables d'instantier n'importe quel service issu d'une famille possiblement infinie de services. En support à notre méthode de conception, nous proposons un langage de description ainsi qu'un compilateur et un environnement d'exécution pour notre langage. Nous appliquons notre méthode et nos outils à la reconception de systèmes existants avec un gain en extensibilité et en possibilité d'intégration dynamique pour ces systèmes.

Abstract

This manuscript studies the adequacy and the improvement of the service oriented principles for software reuse and dynamic software integration in the context of ambient intelligence.

The coevolution of computer hardware and software has led to the advent of ubiquitous computing: people are surrounded by networked computing devices. People (end-users) increasingly expect smartness from these networked devices. The domain of Ambient Intelligence (AmI) tries to fulfil these expectations by using these devices to perceive human activity and reason to act in an adequate manner for the user.

Ambient Intelligence is highly interdisciplinary and involves many research fields. Advances in Ambient Intelligence are conditioned by the proper interaction, capitalization and integration of all disciplines. This manuscript tackles the problem of capitalization and dynamic integration of devices and software not initially designed to work together.

Our approach consists in taking existing methods such as Service Oriented Architectures (SOA) and the semantic web, and adapting them to our particular needs and to the specialists from a wide range of domains. In this investigation, we analyze existing systems designed for intelligent environments and examine how they were designed by different specialists. We also study how to adapt existing SOA concepts to make them useable for a wider audience. We create a service oriented middleware, and associated tools, insisting on their adequacy with its target audience. We introduce a design method that reuses concepts from SOA but insists on the usability by non software-engineering specialists. With our method we introduces the concept of a "service factory" that emerges as a necessary construct from our analysis. We propose a language and a runtime execution environment for our method building on top of our SOA middleware. We use our method and this language to redesign existing systems with improved extensibility and dynamicity.

I would like to thank my supervisor Prof. James L. Crowley and my co-supervisor Dominique Vaufreydaz for giving me the opportunity to discover the domain of scientific research. I thank them for providing me with the context and the freedom to explore my ideas. I also want to thank all the members of my jury, Marie-Christine Rousset, Gregory D. Abowd, David Simplot-Ryl and Michel Riveill, who carefully examined my work and provided me with valuable questions and feedback. A special thank goes to Prof. Gregory Abowd for his support before and after my defense.

I also want to express my thankfulness to all my family members, and particularly my parents and my brother, for supporting me during these years: you have literally provided me with the fresh air necessary to fulfill my thesis. I especially want to thank Claire for its patience and its daily support during this thesis. I also really want to thank my distant friends and everyone that supported me but cannot be present at my defense.

During these years, all my teachers colleagues have made my teaching experience really enjoyable and has given me a breath of fresh air during my research work. More generally, anyone who made me who I am deserves my gratefulness: unfortunately, this category includes long lost contacts and many persons that won't read this document.

A last special thank goes to the PRIMA research group for its variety and openness that made my thesis subject possible. More than coworkers, PRIMA members are friends and all stimulate an ambience of creativity and friendliness. Each individual has contributed to the vision of my thesis. I want to thank James L. Crowley who actively acts for the presence of these persons and for the unique ambience in the PRIMA group.

Contents

List of Figures

Chapter 1

Introduction

1.1 Foreword (on presentation and layout)

In this manuscript, margin notes are used to improve readability. Most of the paragraphs are annotated to make it easier to navigate back in the manuscript. Margin notes appear with the same style as the "this is a margin note" text beside this paragraph.

this is a margin note

Transitions between sections appear like the current paragraph. These transitions should help in replacing in context a particular section with respect to the preceding or the next ones.

All through chapter 2, we try to keep the reader in the flow of reading by referencing details given in chapter 3. References are bidirectional: the overview references the detailed section that references the overview back. The reader can thus choose to go deeper in some details right from the beginning or decide to read the details later. To illustrate how this cross-referencing is rendered in this document, we put some more details on the authoring of this manuscript at the end of this chapter.

introduction to "details on demand"

See details about "How this Manuscript is Written"
→ in section 1.6 (page 20)

1.2 Technological Context

Over the last sixty years, the evolution of computer science, physics, microelectronics and other sciences related to computer development has lead to a continual increase in the ratio of devices to the number of persons. At the beginning of computer science, a single building sized computer was shared by many users, each waiting his turn to run his program. At this time, the processing power in a whole building was inferior to what we can have today in a Personal Digital Assistant (PDA) or even in some models of wrist watch. Despite this relatively low processing power, computers were already far faster than humans for a set of simple mathematical operations: this property, totally useless for the average citizen was a great improvements for many scientists requiring large amounts of computation for their research.

computers for scientists

The growing needs for computational power drove a fast evolution of computers and led to an increase in computational power. With this increase in power came a progressive "democratization" of computers. Originally usable by one person at a time, computers begun to accept multiple user tasks to be run at the same time. Further increase in power, miniaturization and production cost reduction brought access to computing to non-scientist as applications were developed for ordinary people. The Personal Computer (PC) era brought a period where each individual could have access to his own computer: size, price and power of computers made it possible to develop personal and entertainment applications and to sell computers to

the personal computer

non-professional customers. Leading applications in the PC era were office applications (e.g. spreadsheet and word processor), multimedia applications (players and editors for audio and video content) and video games (from simple games like *pong* to complex tri-dimensional games).

computers everywhere

The number of computers per user has continued to increase beyond the Personal Computer vision. Human society has increasingly and seamlessly been surrounded by computers. Computing devices are now all around us: examples include mobile phones, PDAs, portable audio and video players, etc. Taken alone, mobile phones already outnumber conventional personal computers and the trend towards emergence of new device classes continues.

with great capabilities ...

The available computation-enabled devices are gaining in power, autonomy and connectivity: many of these new portable pocket devices are as powerful as personal computers were a decade ago. Moreover, they can now often communicate with each other and access the Internet.

... come great expectations

This ever-increasing density of portable devices continues to raise expectations in a manner similar to personal computers. Not only do users want graphical, entertaining, reliable, well-designed applications but they now increasingly expect intelligence to emerge from this network of devices. As a not-so-futuristic example, we can imagine a user having some pictures on his digital camera. With current technologies and existing devices one could expect that, with little or no interaction, the pictures could be displayed on a digital picture frame or sent as Multimedia Messaging Service (MMS) messages through a mobile phone.

1.3 Problem and Approach

ambient intelligence...

With the increased hardware capabilities, users expect networked devices to act intelligently. Implementing such "ambient intelligence" requires the interaction of experts from many active research domains including artificial intelligence, artificial perception, human activity modelling, networking, sociology.

... is intrinsically interdisciplinary

The domain of "ambient intelligence" is fundamentally interdisciplinary: different specialties must work in cooperation to progress efficiently. Ambient intelligence is not just the juxtaposition of all the specialties but rather an interconnection and a mutual enrichment of contributions from these specialties. For example, modelling human activities taking place in a room requires artificial perception such as computer vision but also uses some methods issued from artificial intelligence and machine learning. Software engineering methods can help in the integration and reuse of contributions from different specialists.

problem of integration and reuse

To take advantage of recent advances in software engineering methods, each specialist must do a double technology watch: in his individual specialty and in software engineering. In practice, specialists cannot stay current with both their domain and latest software engineering methods. This can cause problems in the domain of intelligent environments: only a small part of the contributions from different domains are integrated together. The integration is not only problematic between domains but also in time: effective reuse of existing software remains challenging.

approach: adapting existing methods

One of our objective in this investigation has been to propose solutions to problems of integration and reuse in the domains involved in ambient intelligence. Our approach consists in studying the state of the art in software engineering methods and in adapting these to the target audience: a variety of specialists in non-software engineering domains. The adaptation of existing technologies must be guided by two activities: a detailed study of existing technologies and methods, and a deep analysis of the target audience.

service oriented architectures

Service Oriented Architectures (SOA) provide the properties that are necessary to improve interoperability and reuse. SOA also have the advantage of being suited for the high dynamicity

and openness of intelligent environments. We believe that the tools for SOA are often considered too complex for the intended users.

some contributions

In this investigation, we analyze existing systems designed for intelligent environments and examine how they were designed by different specialists. We have also studied how to adapt existing SOA concepts to make them useable for a wider audience. We have created a service oriented middleware, and associated tools, insisting on its adequacy with its target audience. We have also introduced a design method that reuses concepts from SOA but insists on the usability by non software-engineering specialists. With our method we have introduced the concept of a "service factory" that emerged as a necessary construct from our analysis. We have proposed a language and a runtime execution environment for our method building on top of our SOA middleware.

1.4 Experiments and Results

SOA middleware ...

An important criteria for the evaluation of a middleware that targets usability is the extent to which it is adopted. We have introduced a middleware we developed in our research team and observed its adoption. Another important criteria is the manner in which the middleware API ensures good programming practices.

... has good adoption

Our service oriented middleware and its tools have been adopted: the middleware is used by most of the people in our team. Service oriented approach has improved ease of sharing and reuse between team members. The graphical user interface designed for the middleware is also widely used by service designers. More than half (6/10) of the most active service designers using our middleware contributed custom extensions for the graphical user interface.

redesigning existing systems...

We have used our design method and the new concept of service factories to redesign existing systems present in intelligent environments. The two main systems for which we propose a redesign are a 3D tracking system and an automatic cameraman for seminar recording. We have studied the architectural properties of the redesigned systems to evaluate the benefit of our method.

... improved dynamic reuse and integration

Using our method, target systems have been restructured without major constraints. The resulting systems are less monolithic. Splitting the systems using the guidelines provided in our method improves the reusability of each subpart. Our method also facilitates dynamic reuse of existing software components, particularly with factories: service factories remain always started as a representer of the capabilities of the environment.

1.5 Structure of Chapters

chapter 2

Chapter 2 first introduces the context of our study. A brief history presents the evolution of computers and computer usages. Computing devices have changed from occupying a full room to being small enough to be carried in a pocket. Not only are computing devices getting smaller but they are increasingly powerful and have continued to improve their ability to communicate with each others. Chapter 2 introduces how this evolution has led to intelligent environments and ambient intelligence. The democratization of computing devices gave birth to the domain of software engineering: dedicated methods are required to build these more and more complex systems.

Chapter 2 gives an overview of the main domains that have inspired and influenced our approach. It first presents service oriented architectures and describes how they can be used in intelligent environments. It also introduces knowledge representation domains, such as the

semantic web, and explains how they can enrich software architectures. An introduction to service composition is also given in this chapter.

chapter 3

Chapter 3 addresses aspects of the state of the art related to our work that are not detailed in chapter 2. It describes how context aware systems and frameworks approach the problem of ambient intelligence. Service oriented architectures are introduced in more details and we examine how they can evolve in interaction with the domain of knowledge representation. Different approaches to service compositions are also detailed in this chapter.

Chapter 3 presents two kinds of service oriented architectures: web services and micro-services. Web services are used for large distributed applications while micro-services are services implemented at a programming language level such as Java. The convergence of the semantic web and service oriented architectures leads to semantic web services. The goal of semantic web services is to describe, in a formalized way, the functionalities provided by web services and not only their communication interfaces. The two major implementations of semantic web services, OWL-S and WSMO, are also presented. Different approaches for service composition are introduced, the most notable being composition based on workflow and planning based methods issued from artificial intelligence. Chapter 3 finishes with a synthesis of the state of the art introducing our approach presented in following chapters.

chapter 4

The contribution of chapter 4 resides in a deeper analysis of the problems involved in the conception of intelligent environments. The details of our approach are drawn from the analysis conducted in this chapter. We base our analysis on intelligent environments and take as an illustration the example of a 3D tracking system developed in the PRIMA team. A detailed description of this 3D tracking system is given in order to illustrate major problems of reusability.

Chapter 4 exhibits the problem of reuse and integration that slows down the experimentations and evolution of intelligent environments. The root of these problems resides with the monolithic aspect of most systems developed by non-software engineering specialists. Systems must be developed in a more modular way using low coupled components to favor reuse and to ease integration. The analysis opens with the requirement for use of a service oriented approach. Although SOA is necessary for efficient integration, we must go beyond SOA to ease deployment of components and ease dynamic integration between higher and lower abstraction level software. These different objectives exhibited by chapter 4 are tackled in the following chapters.

chapter 5

Chapter 5 presents our contributions towards the adoption of service oriented architectures (SOA) by the various specialists involved in the conception of intelligent environments. The adoption of SOA has been identified as a requirement to improve interaction, integration and reuse of different contributions in intelligent environments. We have designed a service oriented middleware together with an application programming interface (API) tailored for the variety of target developers. Along with this API, we have designed a graphical user interface that helps in monitoring and interacting with running services.

In chapter 5, the new API for the OMiSCID service oriented middleware is introduced and compared to the previous technology-driven API. This API is designed to be as simple as possible while promoting best programming practice: the API makes it easier to do things well. The API is based on few user oriented concepts (service, connector, search filter, etc.). Chapter 5 also introduces the OMiSCID graphical user interface (GUI) that we have constructed to provide extensibility. The core of the GUI provides generic functionalities such as listing services and their interconnections. More specific functionalities are provided by extensions that can be written by any service designer. This requirement for easy extensibility is fundamental to favor interaction between team members and sharing of functionalities (anyone can install and use the extensions developed by others in a few mouse clicks).

chapter 6

Chapter 6 introduces new design concepts for the conception of systems in intelligent

environments. A new design method based on these concepts is introduced and illustrated to redesign two existing systems.

Also in chapter 6, we introduce the concept of a service factory. Service factories are services that are dedicated to the instantiation of others services. "Timers" provide an example of a factory. A "timer factory" is a service that can instantiate "timer" services emitting events at any fixed frequency. A factory represent a possibly infinite family of instantiable services. Factories can be used to express service composition patterns: a "composite translator factory" can compose any two compatible "translator" services to produce a third one. For example, this factory could compose a "French to English" and a "English to Polish" to produce a "French to Polish" translator.

Chapter 6 presents a design method articulated around the concept of service functionality and service factories. This method aims at splitting the systems in reusable and lightly coupled services. The role of these services is then abstracted as functionalities. As in semantic web services technologies, correspondences between functionalities can be used to integrate systems design by different persons. The method is applied in detail for the redesign of a 3D tracking system and of an automatic cameraman for seminar recording.

Chapter 7 presents the User-oriented Functionality Composition Language (UFCL). This language targets the various specialists involved in the conception of intelligent environments. UFCL is a proposal to support the design method presented in the previous chapter and it must be easy to learn, read and write for its target audience. chapter 7

UFCL provides support for our design method and allows the designer to express the concepts involved in our method: service functionalities, functionality correspondences and descriptions of service factories. The proposed language has a human readable syntax inspired from languages like the Structured Query Language (SQL). This chapter details the concepts that can be expressed in UFCL and introduces the associated syntax. It presents, among others, the two major kind of factories: open service factories (e.g. the "timer factory") and composition pattern factories (e.g. the "composite translator factory").

Chapter 8 details the proposed runtime environment to interpret and reason about UFCL descriptions exposed by OMiSCID services. As UFCL is designed to be loosely coupled to OMiSCID, this chapter explains how the runtime integrates UFCL and OMiSCID. This execution environment is based on the gathering of all UFCL descriptions, their automatic compilation into a rule based system and an inference using these generated rules. chapter 8

In chapter 8, we explain how we compile various UFCL constructs into rules and facts in a rule based system dedicated to semantic web technologies called Jena. Due to some limitations in the rule engine, we had to implement backward chaining using forward chaining capabilities. This chapter introduces the principle of our compilation process. Services exposing functionalities are compiled to some facts in the knowledge base. Descriptions of correspondences between functionalities and service factories are compiled to provide backward chaining rules that rewrite queries of the user into new queries.

Chapter 9 contains a critical study of our work and draws some retrospective conclusions. In this chapter we evaluate our different contributions, first in an isolated manner and then more globally. We then summarize what can be learnt from this investigation. We then conduct a reflection about the difficulties of evaluating research in pervasive computing environments and more specifically in software engineering methods for pervasive computing. We evaluate the adoption of our useable middleware and its tools and show an important adoption rate. We study the advantages provided by our design method that we applied in the redesign of two existing systems. Redesigned systems are easier to deploy, integrate and reuse. The introduction of factories, by making explicit the capabilities of the environment, incites programmers to reuse existing services and to integrate with existing systems. The choice we made around UFCL chapter 9

metamodel and syntax make it a good Domain Specific Language (DSL) but that happens to lack debuggability.

With OMiSCID and its Gui, our work illustrated the importance of a complete solution to ensure the adoption of a technology. Our analysis in chapter 9 also shows that our attempt to separate context information (context awareness) from service functionality description is limited by the intrinsic interrelation between functionalities and context. A more promising approach is provided by the integration of context information with service oriented conception. One conclusion is that the developer-user orientation we have advocated in our work is the key to the success of software methods for intelligent environments. This usability of design methods must be integrated together with an approach involving a dynamic knowledge base unifying services, functionalities and context.

1.6 Important Preliminary Notes About the Manuscript

≫ Details about "How this Manuscript is Written"

how to read the first chapters?

Chapter 2 presents the context of our work in the light of various facets of the state of the art. To keep a clear and fluent progression in this presentation of the context, chapter 2 introduces the main elements present in our state of the art. A more in-depth study of each related approach is given in chapter 3 and is linked using the "details on demand" cross-referencing.

our guiding thread

In this manuscript, a guiding thread is used to illustrate the concepts presented whenever possible. The general context of this guiding thread is the case of a virtual personal assistant that knows about its "master" and acts to simplify his or her daily life. The user's virtual assistant is a digital entity holding information about the user and acting to ease different user's activities. We will often consider a restricted use case: the user arrives at home and his personal assistant decides to play some background music. Then the assistant tries to notify the user of an important but non critically urgent mail.

print/screen version

The current version of the manuscript is optimized for a "printing" scenario. Print version will produce better quality printing but can be notably slower to display on screen. Screen version can contain notably degraded figures as it uses bitmap versions of the original vector images. Both print and screen versions are available by querying the author and will eventually be available online.

End of details about "How this Manuscript is Written"
(referenced from page 15)

Chapter 2

The Emergence of Service Oriented Software Architectures

2.1 Structure of this chapter

how to read the first chapters?

This chapter presents the context of our work in the light of various facets of the state of the art. To keep a clear and fluent progression in this presentation of the context, we will only introduce the main elements present in the state of the art. A more in-depth study of each related investigations is given in chapter 3.

2.2 From Computers to Computing Devices

2.2.1 The Evolution of Computer Usage

In the introduction chapter we gave a quick overview of the evolution of computer hardware over time. We highlighted the fact that hardware evolutions, application potential and user expectation affect each others. This section is more centered on the evolution of computer usage, observed and supposed, in the post personal computer era.

Mark Weiser's calm computing

Introduced by Mark Weiser in 1991 in [Weiser 1991], the vision where computation enabled devices are present everywhere and can communicate together is called "ubiquitous computing" or *pervasive computing*. From the user standpoint, one of the biggest advantages of ubiquitous computing is that it could favor a seamless integration of computer services in the user's daily life. Ubiquitous computing makes it possible to eventually diminish the cognitive load of the user in his use of the system: Marc Weiser gave a name to this aspect and designated it as *calm computing* in [Weiser 1996].

ambient intelligence

Building upon this network of devices, *ambient intelligence* (or AmI) tries to make these devices address the user in an appropriate way by making the devices aware of the user's activity: current task, availability, current focus of attention, emotions, etc. The concept of "ambient intelligence" first appeared in 1998 in some presentations given by Eli Zelkha and Brian Epstein (see [Url-a]). Ambient intelligence has since been studied and developed in its technical and sociological aspects as in [Ducatel 2001].

intelligent environments

In a more general sense, environments that sense user activity and act according to it are named "intelligent environments" or "smart environments" depending on the communities. The community around intelligent environments is particularly interesting as it insists on the interdisciplinary nature of intelligent environments as mentionned in various articles such as [Coen 1998], [Brumitt 2000] and [Mozer 1999]. Ambient intelligence requires the system to

be highly aware of the users and of their activities: this requirement places ambient intelligence at the confluence of pervasive computing and Artificial Intelligence (AI).

prolific science fiction

From the early days of AI, science fiction writers already imagined both what intelligent computers could do and what problems it could cause. From this viewpoint, our guiding thread lags behind the services that computers are depicted as providing in science fiction books and movies: fully automated and intelligent homes, buildings or cities; intelligent assistance in accessing world knowledge or in decision making; etc. Authors have also explored the possible negative effects of computer intelligence: artificial intelligence killing humans, controlling them or even growing humans for energy; the risks of using complex artificial intelligence when taking important decisions; etc.

2.3 From Computer Programming to Software Engineering

In previous section, we centered our analysis on the evolution of computers: we mainly looked at the hardware evolution but also on the past and foreseen evolutions in computer usage. Hardware evolutions do not automatically turn into changes in usage. Software, as the mandatory link between the hardware and the user, must also evolve. This section is dedicated to the maturation of computer programming.

2.3.1 From Switches to Software Engineering

punch cards revolution

In the early days of digital computing, the activity of programming consisted in manually reformulating a problem in terms of computer machine instructions. Machine instructions usually take the form of numerical codes. The computer program (the succession of bytes) was input to the computer by an operator that might be the programmer himself but that was often someone else. Entering the program was first done byte per byte by manipulating physical binary switches representing the bits of the current byte. This tedious work was replaced by punch cards: the operator punched holes in carboard cards to represent bytes of the program (each hole or non-hole representing one byte).

birth of compilers

"Materialization" of computer programs using punch cards made it far easier to modify an existing program, execute it with different inputs or reuse parts of it by cutting/copying/pasting. These were huge benefits but the major advantage of this materialization of programs is that it was then possible to write a program that would generate another one. One program could read its instructions and data from a set of input punch cards and punch some cards as an output. The general principle of having a program generating another program was called "metaprogramming". The most common and widespread metaprogramming activity is compilation. Works on compilation and programming languages started at this time after the programs became manipulable entities.

from programs...

... to applications

The emergence of compilers and the progressive spread of computers made writing software easier and accessible to more persons. These facilities caused a huge increase in the number and the variety of programs and the birth of applications. We can define a software application as a software program that is designed to help a human user in fulfilling a particular task. The most impacting change was the introduction of interactive applications where the application is manipulated by a human user (i.e. not only loaded by an operator or programmer). From the moment the program became an application and started to address human users that were not programmers, the range of application domains and the potential volume of computer users made a huge jump. Applications quickly grew both in size and number, became graphical and more interactive to eventually reach the complexity of today's desktop applications running on personal computers.

With the increased size and complexity of applications, it is now impossible for any real size application to be developed by a single person. Today, building a software application is not a one-person activity. It is a complex process involving various skills brought by many different people with different skills: psychologists, analysts, ergonomists, software designers, programmers, visual designers, software testers, etc. *Software engineering* aims at studying the process of building software and proposing methods and solutions to problems found in this process. Even if software engineering can anticipate incoming methodological and organizational problems, it is never until the problems are present that people *begin* to use solutions or even to look for them. The consequence of this continuous evolution of software conception is that software engineering practices are always one step behind the reality of problems.

software engineering

2.3.2 Software Engineering and Ambient Intelligence

In previous section, we explained how considering software programs as an object of study made it possible for software programming to grow considerably. We will underline major software engineering problems that ambient intelligence is facing and how it is solved in other domains.

Examining software evolution in the light of the new paradigms of ubiquitous computing and ambient intelligence shows how taught software engineering methods lag practice. In addition to the skills required to build any kind of software, ambient intelligence involves experts from many additional complex domains: applied mathematics, statistics, machine learning, signal processing for acoustic and visual perception, dynamic software architectures, etc. We can illustrate this interdisciplinarity using our guiding thread that involves most of the cited domains:

... and the lack of it in AmI

- Statistics and machine learning are required to learn and refine user preferences.
- Dynamic software architectures are necessary as the agent must properly adapt its behavior to a dynamic software environment that changes as the user moves or some devices are turned on and off.
- Visual perception is important to locate and identify the user and to recognize activity.
- Acoustic perception is required for vocal interaction with the system and to obtain clues about current activity.

Specialists involved in the software development do not have the time to be up-to-date both in their individual specialty and in the domain of software engineering. In addition to this necessary double technology watch, designing intelligent environment is still not an industrial activity: the market is not mature enough and a notable engineering effort would be required to bring research works to market. This explains why the application of software engineering methods is delayed in the context of research in various domains around intelligent environments. This last point makes the software engineering situation even worse in the domain of research in ambient intelligence: ambient intelligence is both a highly computer science oriented domain and a highly technical domain.

impossible double technology watch

The problem of reuse and interoperability of software exists since the early day of software engineering. Even if it is occurs in any software engineering activity, this problem of reuse and interoperability heavily affects intelligent environments where many domains meet. To overcome this classical problem, many programming styles have been created and tried in the building of software. One major aim of all these methods is to build software as an assembly of elements with lower coupling, better re-usability, facilitated replacement, etc. Procedural functionality decomposition, modular programming, Object Oriented Programming (OOP), component based programming are all representatives of this continuous quest for better software separation. When distributed computing started to be more common, evolutions of these paradigms appeared including distributed OOP, distributed components and services.

general lack of software reuse

SOA

Recently, Service Oriented Programming, also known as Service Oriented Architectures (SOA), has emerged as a solution to many interoperability problems. The fundamental idea behind SOA is to split the system into distinct parts representing a functional decomposition of the original system. These parts are called services. Splitting a system into services aims at having a clean separation of functionalities that should favor reuse of these services to build or improve new applications. The fundamental concepts in SOA are that services are like objects or component. We can list three important differences between services and objects:

- services are always defined by their interface: the implementation is totally hidden to the service consumer;
- services are designed to be found and composed at runtime, dynamically;
- services are often accessible on a network (but not always).

web services and micro-services

Many technologies can be used to implement a SOA: we can briefly give two categories of implementations. Web Services implement a distributed SOA using web technologies such as the HyperText Transfer Protocol (HTTP). Various mechanisms around the Java Virtual Machine (JVM) implement SOA but put the emphasis on the dynamic availability of services.

See details about "Services and Service Oriented Architectures"
→ in section 3.2.1 (page 36)

2.4 Service Oriented Architectures for Intelligent Environments

In section 2.3, we studied the evolution of computers and software engineering. We terminated by a brief introduction of Service Oriented Architectures (SOA) principles. SOA is an architectural solution to contemporary problems of software decoupling, interoperability and reuse. In this section we will study SOA in the context of Intelligent Environments.

adequacy of SOA

Service Oriented Architectures (SOA) provide solutions for the interoperability and reuse problems posed by pervasive computing and intelligent environments. Research on software architectures for pervasive computing is already experimenting with SOA. This architectural style is however not "democratized" and is not used by the various specialists involved in intelligent environments.

2.4.1 Acceptance and Acceptability of Service Oriented Solutions

supposed impact on performance

Most specialists, working in other domains than software engineering, do not implement service oriented architectures. Many solutions such as Web Services are perceived as having a considerable overhead which is a real barrier for people writing highly technical code where performances and optimizations are often key points. This perceived overhead exists but, depending on the application, it can be unacceptable or negligible. The "web" connotation of Web Services makes many people think that these technologies are only applicable for web applications.

unclear return of interest

The necessity to learn a new method is also slowing down SOA adoption. The priority of researchers and specialists is on their respective domains: learning SOA takes time on other higher-priority tasks. Most people do not take time to learn SOA because they see no advantages in using it in comparison to the learning time. The low perceived benefit of SOA prevents low level component designers from adopting it. We call *low level component* a component that, from an architectural point of view, has no dependencies and is only used by other components.

perceived complexity

For most specialists, the real gain of SOA is indirect and hidden: integration and sharing is not the main objective when doing research. Most developers flee SOA as soon as they hit the complexity of existing solutions. There are more and more tools around SOA to help the

users. However, users often feel assisted and dread an underlying complexity that they do not understand.

restricted development environment

Some SOA solutions, such as OSGi service platform, also confront the user with a totally different constraint: a restriction on the programming language. Even if OSGi services can interact easily and in an efficient way, the designer of OSGi services is forced to use Java. This limitation of available language is a real barrier for people working in domains where implementations are mostly in C++ or Matlab. OSGi can be used as a service framework within the JVM but has to be running on a single virtual machine. Depending on the application domain, the limitation to a single JVM can be an even bigger limitation than the restriction to the Java language.

2.4.2 Solutions to the Acceptance Problem

In previous section, we identified three main barriers to SOA adoption in the context of intelligent environments: perceived resource overhead of SOA, unclear return of interest for researchers and limitations in development environment (programming language, platform).

easy access to SOA

From the different barriers identified in the previous section, we can take a positive point of view and extract some requirements for a more acceptable SOA solution:

- SOA implementations must be very easy to learn by the target audience
- in the first steps of the learning process, it must be illustrated through examples what is the return of interest for the audience
- SOA implementations must be easy to install
- SOA implementations must be available in any language susceptible to be used by the target audience
- development of services must integrate in the target audience's development process
- SOA implementations must be efficient, must communicate about it and should prove it (using benchmarks)

Even if we talk here about SOA, most of these points can be applied to the design of many software targeting developers: libraries, build tools, unit testing tools, ... In fact, we can apply these guidelines later in this manuscript when talking about semantic web services.

previously identified requirements

Prior to the start of this thesis, part of this reflection had already been taken in our team and the conclusions were that no existing solution fits the requirements for ambient intelligence. At this time, the identified requirements were mainly: "having a cross-platform, cross-language middleware to make it possible to declare, discover and interconnect services in a efficient way". These requirements came from the usage of C++, Java and scripting languages (mostly TCL at the time) within the team. The performance requirement came from the need to transfer and process important data volumes such as uncompressed live video streams recorded by cameras but also from the use of this middleware in the human machine interaction loop. Communications between services must support both high bandwidth and low latency to ensure its usability in the target context. This middleware got a first implementation for each language of interest and started to be used in the team.

2.5 Mixing Services and Knowledge Representation Methods

In previous sections, we have introduced Service Oriented Architecture (SOA) as a proper architectural solution to improve decoupling, favor reuse and increase interoperability of parts of software systems. We also discussed difficulties have hampered the adoption of SOA in the creation of applications in ambient intelligence. In this section we will show the limitations to interoperability that classical SOA solutions provide and how these limitations can be overcome.

2.5.1 From Implementations Decoupling to Design Decoupling

implementation decoupling

using interfaces

Service oriented computing aims at separating the use of a software functionality from its underlying implementation. As we have seen above, this decoupling is a major objective of many current software design methods. For instance, object oriented programming hides implementation details by making part of a class totally inaccessible to the code using this class (private content). The use of interfaces (or purely abstract classes) is a step forward in this direction: the real functionality is only described and might be provided by various providers transparently. Using interfaces properly, an implementation of a software element is only dependant on the interfaces of other software element and is fully decoupled from their implementations.

integrating one design into another

a priori agreement from scratch

We think that this objective is necessary but not sufficient as it gives no easy way to enable an "a posteriori" interoperability between system parts designed by different vendors. Currently, when two entities (persons, teams, companies) design software that will interoperate, two cases often occur. In the first case, one (or both) entity(ies) already has all or part of its software implemented and the second entity will adhere to the conventions of the first: it will use the same communication technology and will design its parts according to the interfaces exposed by the first entity's code. In this case, programming often consists in writing some wrappers around legacy code. This can be problematic because of the highly probable architectural mismatch between the two projects [Garlan 1995]. The second possible solution to integrate skills from two entities is to start a cooperative project from scratch and agree on some choices. The most visible choices are at the level of the technology being used for interoperability.

interaction protocols

From our point of view, the most important choice is the selection of the interfaces of the different software building blocks. Having chosen a given technology, and for example a particular SOA implementation, is far from sufficient to enable true interoperability. Given a service interface, what we call an *interaction protocol* is the set of sequencing constraints of service invocations: an operation must be invoked before another one, an operation cannot be called twice, delay before timeouts, etc. If interoperable services are desired, the entities must specify their services, their interfaces and their interaction protocols.

SOA standardization

Integrating with another design is always a difficult task that requires fully understanding the concepts involved in the design and to build adapters to insert between the designer's own elements and existing elements. The vision behind decoupling provided by services is to make it possible to have a spontaneous integration between services designed independently by different vendors. A first requirement is to settle down on some interoperable SOA technologies. Given the variety of SOA use cases, this requirement is hard to fulfil but many efforts are made in this direction: we are heading toward a standardization of multiple sets of technologies for SOA, web services being a good example of it.

impossible standardization of design

To integrate two designs, not only must there be an agreement on the used technology but a second standardization effort is required. This second aspect of standardization is quite bigger and is impossible with todays simple SOA approaches: having spontaneously interoperable services would require worldwide standard on service design. Not only the technologies used would have to be the same but so would the way of thinking and of formulating the problems. Considering that software designers are human, this view of the world is unlikely. Even if

possible, this would mean that there is no more creativity in software design and architectures: either the domain is "solved" or humanity has lost its capability to be creative.

2.5.2 Design as Knowledge

The previous section has argued that an agreement on a particular SOA technology is not sufficient to provide spontaneous interoperability. The most commonly used methods to enable interoperability between two designs falls in two categories: make both design at the same time with the same persons, and, bend (or break) one design to make it compatible with the other. In this section we will show that a parallel can be done between, on one side, designs alignment and architectural mismatch, and on the other side, the problems of knowledge representation and fusion.

To summarize the argument of the previous section we can say that seeing standardization for design is utopic. In this statement, we are not talking about *design methods* such as UML but rather about design as a way of *representing things* and linking concepts. Our conviction is that the objects of the design must be materialized and manipulable. Spontaneous interaction between alien designs should be provided by some design alignment methods rather than done by manually modifying existing software elements. We would then have mixed systems composed of components using some interfaces and interaction protocols, and of adapters between this protocols.

adaptation layer

To illustrate this proposal on our guiding thread example, we could take the case of the user that wants to control, using his wifi or bluetooth-enabled mobile phone, the media player that runs on his home computer. As this is a functionality that was not included in the original design of the phone, we need to enable a posteriori interoperability between the phone and the music player. The first solution is to modify an existing application on the phone to build a new application that can send commands to the music player such as "play/pause" or "volume up". This solution requires some deep understanding and some modifications of existing software.

e.g. manual integration

The second solution consists in having an adapter layer that would convert "key press" events, sent by a generic service on the phone, to some commands for the music player. Even in the case where the adapter has to be written and deployed, it still has advantages over the first solution: we only need to understand the interface of a simple service rather than all its internals, and we do not necessarily need to deploy the adapter on the mobile phone nor alter the mobile phone setup. Materializing the adaptation at runtime is even better as we can imagine that the mapping between keys and commands to the music player can be dynamically generated, perhaps even by the user's intelligent assistant. Interacting with the user, and learning from the user's habits and feedback could make it possible for the assistant to evolve this mapping automatically.

e.g. adaptation

The reflection we had about software modeling and design is the software architecture counterpart of the one that is present in knowledge representation communities. Designing software and representing knowledge are both modeling exercises and share many properties. To insist on this statement, we can reformulate it as: software design and architecture falls into the range of modeling tasks, and software modeling can learn from state of the art knowledge representation methods.

similarity with knowledge representation

2.5.3 Knowledge Representation and the Semantic Web

The previous section has underlined the similarities between a posteriori interoperability in software and knowledge representation problematics with knowledge aggregation. We insisted on the fact that software interoperability could learn from the solutions that have been developed by the knowledge representation communities.

In this section, we will present some knowledge representation methods and we will particularly study the semantic web.

the semantic web

In knowledge representation domains, there is an agreement on the formats used to represent knowledge such as Resource Description Framework (RDF) and its extensions (RDFS, OWL). These technologies are fundamental to the "Semantic Web". The semantic web places itself as the successor of the existing web and seeks to represent information in a manner that is more structured and understandable by computers.

current web

Currently, information available on the web are often mixed with presentation information: there is no real separation between what is the information and how it is displayed on screen. Even if some technologies such as various stylesheet languages (CSS, XSLT, etc.) can separate content from presentation when used properly, the knowledge is still expressed in natural language with all associated ambiguities, inexactness and implicit context.

current web search

Current web search engines are very efficient for simple information retrieval: they compensate for the intrinsic natural language difficulties by profiling the mass of users and ranking web pages based on their popularity. Many other parameters and automatic machine learning methods influence this ranking. The consequence of this popularity mechanism is that as long as search seeks popular targets, an appropriate result is returned. However, the popularity factor is of no use when you are doing an uncommon search.

semantic web search

The idea of the semantic web is to represent information in a more formal, structured representation using languages such as RDF and OWL. The main advantage is to allow semantic search to give meaning to the terms used in the queries. A semantic web page is seen as a set of concepts and relations between them, all of this being linked to the information available on other pages. Seeing information as interrelated concepts makes automatic reasoning and inference easier and more powerful compared to todays situation where search engines consider a web page as a simple "bag of words".

e.g. search scenario

Imagine the simple task of finding all (or a given number of) programming languages that have an "elsif" keyword. Given the current traditional way of representing knowledge, it is possible but far from easy to find a list of programming languages and to search their specifications for the "elsif" keyword. Human intervention is necessary to discover if a given language has this keyword: it could happen that the word "elsif" appears just as an illustration of its absence (e.g. "elsif is not a reserved word in X language").

e.g. semantic search

In a more structured representation of knowledge, each programming language would be represented by a concept that would be in relation with all its keywords. Each language would also be in relation with other elements like other languages. This could be used to produce a genealogy of languages such as in figure 2.1). In the case of the problem of finding languages with an "elsif" keyword, an unambiguous request could be formulated as follows: "find all X1, X1 is of type language, X1 has keyword X2, X2 has spelling 'elsif'".

See details about "From the Classical Web to the Semantic Web"
→ in section 3.2.4 (page 42)

concept alignment

The principle in knowledge representation is to reuse existing concepts whenever possible, to create new concepts when needed and to express correspondences between concepts when they have similar meanings. This principle may be linked with the concept of adapter as presented in section 2.5.2. In knowledge representation, putting similar concepts in relation is called an "alignment" of concepts. Alignment is mainly done by adding subsumption (specialization, subtype) and equivalence relations between concepts created independently.

See details about "Ontologies and Ontology Alignement"
→ in section 3.2.4 (page 43)

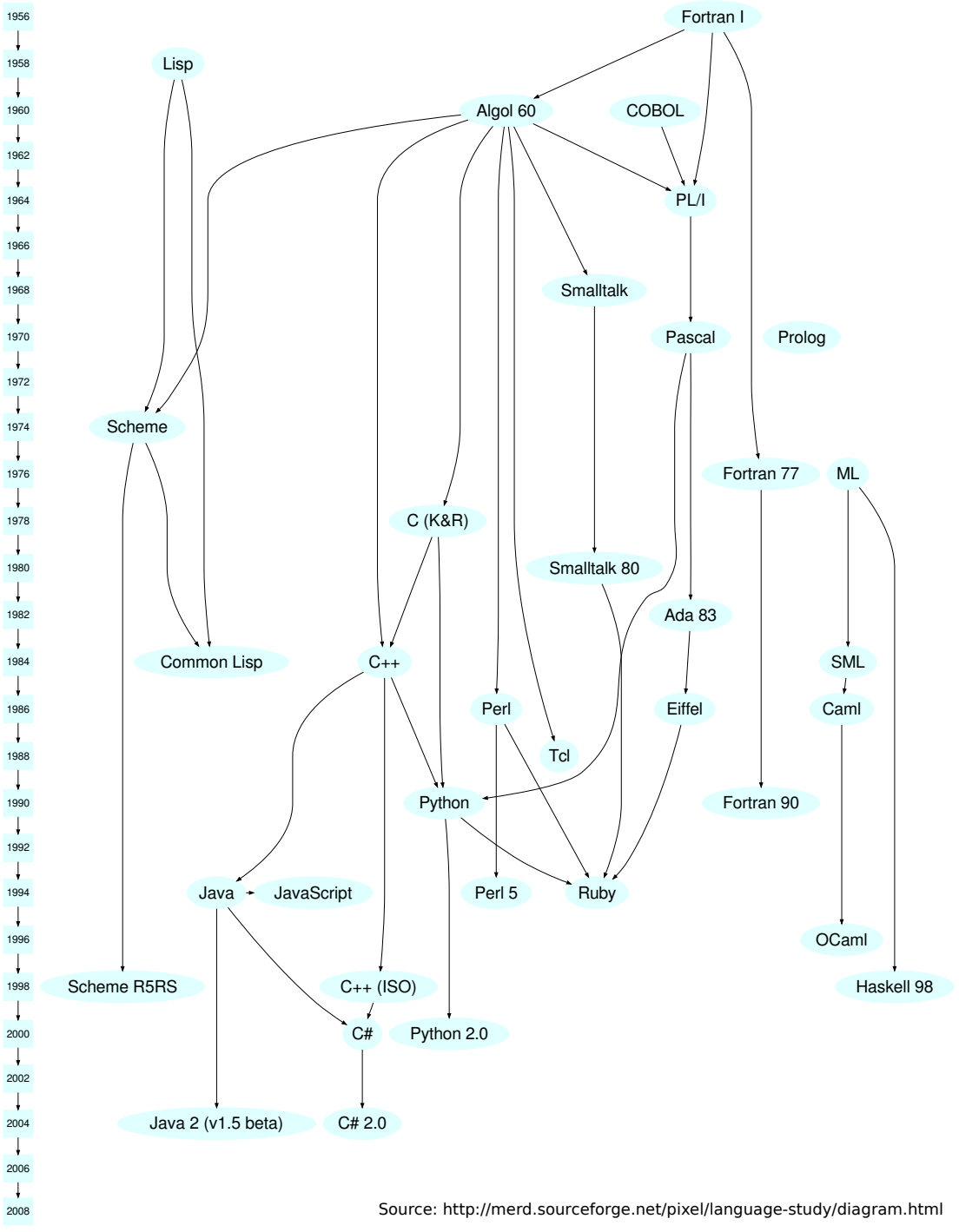


Figure 2.1: *Advantage of Structured Information. This programming language genealogy can be automatically generated. A link represent an influence from one language on the creation of another one. Such information, already available in natural language, has been formalized in a more structured way, allowing its automated use.*

2.5.4 Semantically Described Services

After introducing the similarities between software design and knowledge representation, the previous section has presented semantic web technologies and tools. In this section we will describe existing efforts and standards that aim at applying semantic web methods and tools to Service Oriented Architectures.

functionality descriptions

Semantically describing functionalities is in fact a materialization of the design decisions and makes it possible to manipulate these descriptions and express relations between them. Having materialized the description of a **bluetooth phone** functionality, we can easily say that this **bluetooth phone** concept is a specialization of the **remote control** concept described by other people previously (e.g. created originally for an infrared remote control).

semantic web services

Mixing semantic knowledge with service functionalities descriptions is not a new idea as it is exactly the foundation of *Semantic Web Services* (SWS). SWS technologies add a semantic description of the functionalities provided by classical web services. A semantically described service has not only a proper interface but it also has a meaningful description of its functionality.

OWL-S and WSMO

There are two major proposal for SWS technologies: OWL-S (Web Ontology Language for Services) is one of them. In OWL-S, a classical web service can be semantically described on various aspects ranging from the general "capabilities" it provides, to the detail of its computational process, and to how these abstract concepts are grounded on web service operations. Another set of technologies for the design of semantic web services is the Web Service Modeling Ontology (WSMO) and the associated technologies.

low adoption rate

Both technologies for SWS (OWL-S and WSMO) have their advantages. Even if the possibilities of SWS in general are broad and impressive, there is still a lack of tools that are both efficient and easy to master by most developers. There have been many efforts put on tools around SWS but the relative lack of adoption makes these efforts decrease and most software has not been updated for a few years. For most developers, teams or companies, adopting semantic web methodologies currently results in a cost that is too important compared to the expected return of interest.

See details about "Semantic Web Services"
→ in section 3.2.5 (page 44)

need for usability

Despite the relatively low adoption on the side of SWS, much effort is still ongoing for both classic web services and the semantic web (pure knowledge without services). Following the reasoning about barriers to SOA adoption we had at the beginning of this chapter in section 2.4.2, we can express the same guidelines for SWS technologies. The emphasis should particularly be put on the ease of use and the ease of the learning process for the target audience.

2.6 Composition of Services

Section 2.5 has examined a first approach to services. Borrowing from the domain of knowledge representation, this approach consists in semantically describing service functionalities to improve potential interoperability. A second interesting approach is based on the composition of services and will be studied in this section.

SOA for dynamic composition

Separating different aspects of software in lightly coupled building blocks makes it possible to build systems as an assembly of these building blocks. Using services or any other technology such as objects, components, modules or plugins, this composability property comes directly from the effort of specifying a clean interface to interact with building blocks. What SOA brings from this point of view is that it facilitates composition that is dynamical and open.

2.6.1 SOA and Service Composition

services are composable

Service Oriented Architecture enables better separation of concerns, better isolation and looser coupling between software elements. Such approaches have a great advantage: once software elements are well defined and well separated it becomes possible to compose them to build new applications or new functionalities. The most common illustration for any programmer using object oriented programming is the possibility to reuse and combine existing classes to build new ones through aggregation. Another common illustration of this composability is the component based approach ([Brown 1996], [Heineman 2001]) in which the applications are build only by reusing, configuring and assembling existing fully operational components.

abstraction of the implementation

Most service oriented environments are distributed. Like other distributed architectures such as distributed components or distributed objects, services can be easily composed to build distributed applications without being overloaded with the implementation details of every service.

dynamic composition

runtime service selection

The main particularity of SOA is that components are designed to operate in a dynamic context where services can appear and disappear at anytime. Services must be aware of these changes in service availability and must behave accordingly. With this dynamicity comes a constraint: some choices about how to assemble services have to be made at runtime. In a less dynamic context, the designer of the application was the expert responsible for taking these decisions about composition. Taking runtime composition decisions can be done automatically or by asking the final user. Any intermediate solution can be imagined and implemented ranging from choosing everything automatically (even randomly when no pertinent information about the decision is available) to asking the user for everything.

e.g. intelligent assistant

In the context of intelligent environments, we advocate the presence of an intelligent assistant representing the user, learning preferences and habits, and taking simple composition decisions on his behalf as in various systems such as [Zaidenberg 2008] and [Garlan 2007]. When the user's assistant has no information to take a decision, it may fall back to a default behavior: random choice or query to the user depending on the user activity and interruptibility.

2.6.2 Enabling Methods for Service Composition

Previous section have introduced the main difference between services and components: services are designed to be composed dynamically, at runtime and according to context whereas components are not. In this section we will give an overview of some languages involved in the description of services composition.

Service composition is interesting for both the industry and research. For industry, composition is an enabler for massive reuse and thus can provide important productivity gains. For research, automatic service composition uncovers many interesting challenges and research problems.

choreography and orchestration

This general interest from various communities has driven the setup, reuse and maturing of standards. These standards aim at making it possible for system designers to express mainly two concepts that are called "service choreographies" and "service orchestrations". A service choreography defines the expected behavior of a set of services. A choreography expresses how the interactions should happen, while the services themselves are eventually responsible for the execution of the choreography. As a description of what should happen, a choreography can be easily but optionally used to monitor a running service assembly. A service orchestration is an executable process: it describes what the conductor should ask the others services to do. To come to life a service orchestration has to be executed by an entity (program) present at runtime and giving orders to others services.

associated languages

Languages such as the Web Services Choreography Description Language (WS-CDL) described in [Kavantzas 2005] can be used to describe choreographies. Most of the service orches-

tration languages are business process description languages such as Business Process Execution Language (BPEL) and its variants. A business process is a set of tasks that have to be done to achieve a particular goal in a company. The tasks composing a business process are usually temporally related through a workflow. BPEL is, in fact, a shorthand for BPEL for Web Services (BPEL4WS) and its later renaming to Web Services BPEL (WS-BPEL) for name uniformization purposes. Figure 2.2 gives an example of visual representation for an example business process.

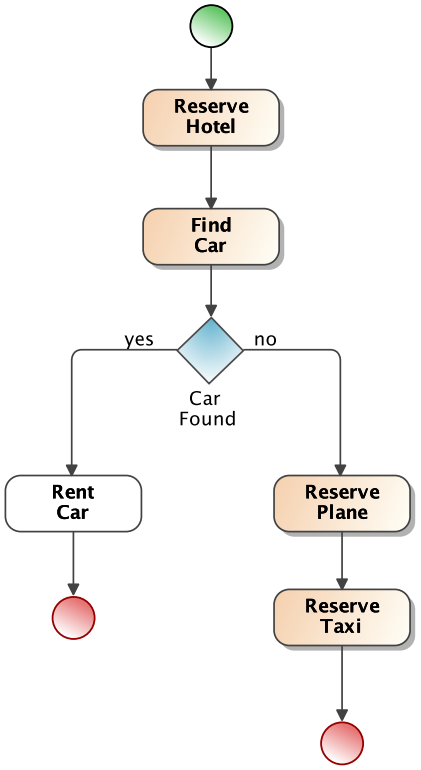


Figure 2.2: *Visual Example of a Business Process for a mid size trip. One must first reserve a hotel, then try to find a car or by default reserve a plane and a taxi.*

programming using orchestration

Most use cases of BPEL show only its use at design time: the designer uses BPEL to specify how to properly call an aggregate of some existing web services to implement a new functionality. The BPEL process can then be executed by a runtime orchestration engine to obtain the new functionality. One good habit encouraged by BPEL is to expose the composed functionality as a service. BPEL processes can be composed transparently with other services allowing hierarchical composition.

orchestration as composition pattern

Even if a BPEL process definition is expressed at design time, it references services abstractly: the services are effectively used and discovered only at runtime. This late binding is the only real abstraction in the definition of BPEL processes. Once the task of specifying how we can compose existing services to produce another one is done, this information can be used in ways that are even more interesting than to just start a service by executing the BPEL process. It is possible to interpret business processes not only as an executable specification of a composite service but also as a composition pattern: given some existing services with particular properties, we can instantiate a new composite service with some properties derived from the component's properties. This vision is unfortunately underexploited as may be see in detail in following sections.

2.6.3 Reasoning and Planning for Service Composition

In previous sections, we introduced some existing languages to describe service compositions but they are often seen as a programming language to control existing services. In this section, we will explore the approaches were reasoning and planning is preponderant.

composition for spontaneous interaction

Seeing business process descriptions as composition patterns is an interesting approach for intelligent environments. Ambient intelligence requires a spontaneous interaction between services designed independently. Reasoning about the composability of services can improve spontaneous integration significantly. Imagine a case where some services are running in an environment and a user arrives accompanied by a personal intelligent assistant that can reason about what services are presents, what services it brings in the environment and what compositions could be made to fulfill the user needs.

Despite the particularity of services of being composed at runtime and the potential behind formalisms such as BPEL, reasoning about service composition pattern is underexploited. We can classify automatic service composition usage in 3 categories:

- The simplest level of composition aims at building an application by filling holes in an application pattern at runtime. This composition is only a form of dynamic linking: the concrete services used in the composition are chosen at runtime. We use the term of application as the only objective of this service composition is to fulfill the final user need.
- A minor improvement consists in exposing the composition as a service itself. This way, the composite service that is produced can be reused latter, for example in another higher level composition.
- In the two previous cases, only one composition pattern was "tested" at a time to start a particular service. A more interesting form of composition consists in reasoning about all composition patterns altogether. This makes it possible to share composition patterns (i.e. knowledge about composability) rather than sharing only the result of an effective composition.

process calculus based composition

Many research works aim at reasoning on the composability of services based on the definition of service composition patterns. Depending on the approach, these definitions are usually highly formal such as finite state automata, petri nets, π-calculus or BPEL which is a form of process calculus. A good insight of these methods can be found in [Berardi 2005]. Good results can be obtained with these methods as long as the operational process of services is described with sufficient details: basically, composition patterns must be fully "programmed" using these languages to ensure composability.

2.6.4 Composition of Semantically Described Services

Sections 2.5 and 2.6 illustrate the gains provided by semantic description of services for composition and by reasoning about composability. This section will introduce some ongoing investigations that combine these improvements and approach real spontaneous interaction between services.

Most of the initiatives aiming at describing interactions between services and thus enabling the creation of simple composite services are done on classical web services. This means that none of the particularities and difficulties of semantic descriptions are taken into account but that none of their advantages are present either. Only recently, investigations have appeared concerning composition of semantically described service composition patterns. The most promising and interesting article we have found is about BPEL for Semantic Web Services (BPEL4SWS), an adaptation of BPEL to semantically described web services.

See details about “Service Composition”
→ in section 3.3 (page 47)

2.7 Concluding Remarks and Wrap Up

This chapter gave an overview of the main domains that inspired and influenced our approach. Different aspects of the state of the art are detailed in chapter 3. A synthesis of the state of the art, replaced in context, can be found as a closing of chapter 3 in section 3.4. Section 3.4 also introduces our approach in the light of the state of the art.

Chapter 3

Approaches to Service Functionality Description and Composition

Chapter 2 has explained the context of the research presented in this manuscript. The objective of chapter 2 is to introduce the motivations of our work in the light of existing research and technologies. The current chapter aims at detailing the state of the art of some aspects that arose in chapter 2 and at giving a more comprehensive introduction to important technologies.

3.1 Pervasive Computing Environments Become Context Aware

Building on top of ubiquitous computing (or pervasive computing), ambient intelligence needs to have an understanding of what happens in the environment. This awareness of the application about its environment includes a wide range of elements. The simplest and most common context element is the user location in the environment. More complex context information may involve current activities, preferences and the schedule or various environmental elements such as the current time and weather or computing resources available in the environment. *Context Aware Applications* are applications that can act according to their context of execution in a broad sense.

context awareness

Context aware computing is a very active research domain. The notion of context awareness has been introduced by Bill Schilit in [Schilit 1994] and described as software that "adapts according to the location of use, the collection of nearby people, hosts, and accessible devices, as well as to changes to such things over time". Major contributions on context aware computing are presented by Anind Dey in [Dey 2001] were context is defined as "any information that characterizes a situation related to the interaction between humans, applications and the surrounding environment".

definitions of context

Many frameworks, middlewares and toolkits have been designed to help in building context aware applications. A variety of surveys analyze and compare these different toolkits: a particularly detailed analysis is given in [Baldauf 2007] which presents the Context toolkit, Gaia, CoBra and other frameworks. One common aspect of such frameworks is the separation of context producers and context consumers. In the model of the "Observer" design pattern introduced in [Gamma 1995], this proper separation allows the addition of new context aware applications without major modifications.

context frameworks

An aspect distinguishing different context acquisition frameworks is the model used to represent context information. The underlying representation used by most frameworks can be considered as tuple spaces:

- Frameworks such as CoBra presented in [Chen 2004] use the Web Ontology Language (OWL) having an underlying model in the form of triples.
- Gaia uses 4-ary predicates composed of a "context type", a subject, a predicate and an object, as explained in [Román 2002].
- Frameworks such as the Context Toolkit presented in [Dey 2001], Hydrogen described in [Hofer 2003] and frameworks based on OWL indirectly use representations based on objects with valued attributes.

representation of context information

Approaches such as Gaia and the Context Toolkit accept any kind of context information while approaches based on predefined fixed ontologies add constraints to the nature of the context. Both approaches have their advantages and drawbacks. A fixed ontology provides a structure and a meaning to context information but it limits the expression of dynamic "unexpected" kind of context information. The ideal solution enables different context providers to describe their own ontology of context information and have some alignments and integration between these ontologies. This principle is underlying the concept of ontologies as we detail in further sections of this chapter.

context and services

Service oriented architectures are mentioned in relation with some of the major context frameworks but it is seen as an enabling technology for the implementation of the framework.- Web services are for example used in CoBra ([Chen 2004]) as remote sources of context information. The survey mentioned above also recommend the use of web service technologies for the standardization of context aware toolkits.

As formulated in [Coutaz 2005] "Context is key". Context should be a first class consideration for intelligent systems. However, we also believe that systems can also benefit from recent software engineering methods such as service oriented computing. More details about service oriented architectures are provided in the rest of this chapter.

3.2 The Convergence of Service Oriented Architectures and the Semantic Web

3.2.1 Definitions of Service and SOA

≫ Details about "Services and Service Oriented Architectures"

terminology

Service Oriented Architecture (SOA) is the architectural part of the service oriented approach.- The term "SOA" is often used to represent the service oriented approach as a whole. A service oriented approach can be applied anywhere in the software engineering process from analysis to implementation.

service orientation

The fundamental idea behind *service orientation* is to split the object of analysis (e.g. a domain model or a system) into distinct parts representing a functional decomposition of the system. This decomposition in parts called *services* aims at having a clean separation of functionalities that should favor reuse of these services to build new applications or improve existing ones. There is no unique definition of service or SOA and this section will try to give a clearer insight of what these concepts involve.

a pragmatic definition of services

As there is no universally recognized definition for service, we decided to start with a citation from *The Pragmatic Programmer: From Journeyman to Master* by A. Hunt and D. Thomas [Hunt 1999]. This choice is driven by the fact that this book is not about service oriented architecture but rather about general best practices in software design, analysis and programming. Hunt and Thomas extract a set of tips all along their book and the 40^{th} of them is entitled as follows: "design using services". This advice emerges after an example of application architecture that exhibits important decoupling and reuseability properties. Together with their tips, the authors describe their architecture choice in this way:

> *Instead of components, we have really created services: independent, concurrent objects behind well-defined, consistent interfaces.*

We propose to use their definition to underline some important element of definition for services:

distributed, reusable, loosely coupled entities

- Services can be considered as improvements over objects or components.
- Services are loosely coupled and autonomous entities.
- Service functionalities are defined by a clear interface or contract.
- Services encapsulate their implementation details and can be used in an abstract way without any knowledge of the implementation.
- Services run concurrently and can be distributed if necessary.
- Services, like component and objects, can be reused and composed to simplify the creation of new applications.

An additional aspect, not considered in *The Pragmatic Programmer* is the dynamicity of service interconnections:

dynamic discovery and composition

- Services are designed to be composed at runtime.
- Services find each others using a discovery mechanism: service providers declare the services they provide, service consumers describe the services they require.
- Services run in a dynamic software environment where other services can become available or be retracted at any time.

Service discovery is an important aspect of service oriented architecture. Service discovery mechanism is often assured by a service repository as presented in figure 3.1: new services declare their presence to the service repository and service consumers consult the repository when they require a particular service. A service provider publishes in the repository a description of the provided service together with a reference to itself telling how to access this service. A service consumer queries the service repository by describing the services it desires; in response it receives one or several references to the appropriate service providers.

service advertisement

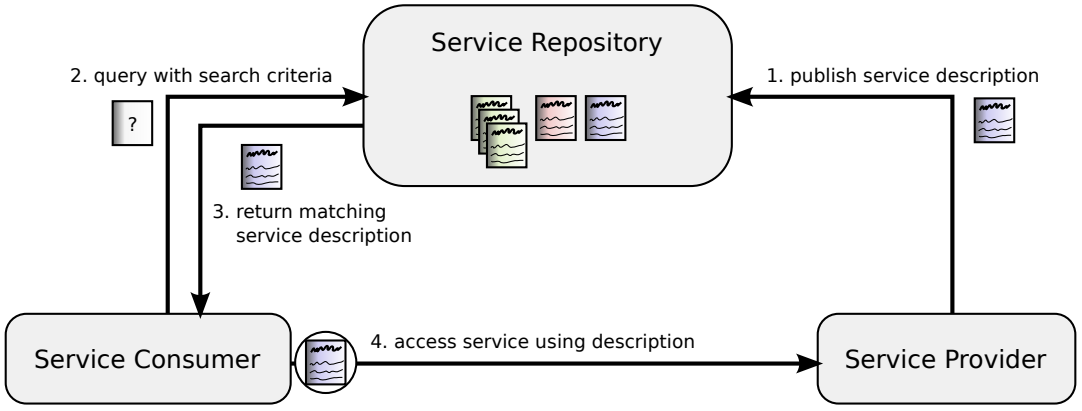


Figure 3.1: *Service Discovery Using a Service Repository. Service providers register the description of their services in the repository. Service consumers obtain references to services matching their research criteria.*

The core principles of SOA listed above are mainly aiming at improving the interoperability and reuseability of elements constituting a software system. The objective of SOA is to facilitate the communication and the integration of business processes between companies. It is also used within companies to facilitate internal integration of various business entities.

SOA objectives: interoperability and integration

End of details about "Services and Service Oriented Architectures" (referenced from page 24)

3.2.2 The Example of Web Services

The previous section has introduced the principles and motivations behind Service oriented Architectures (SOA) but has not referenced any particular technology or existing implementation. This section will detail the Web Services technologies that constitute one of the most used and developed supporting technologies for implementation of a service oriented architecture.

web services (WS)...

The designation of Web Services is misleading and the first reaction to this name is to think that web services are only useable on the web. In fact, the best way to both define *Web Services* and insist on the naming problem is as follows:

- Web Services are services built using web technologies such as HTTP (HyperText Transfer Protocol)
- Web Services are not necessarily available on the web

... services using web technologies

In a sentence, Web Services are not services on the web but services using web technologies. Web Services communication is using HTTP as a transport protocol and formats used by web services are based on XML (eXtensible Markup Language).

A web service can be seen as an object that can be accessed using networked communications. The implementation of a web service is hidden and the caller only knows about the interface of the web service that is composed of a set of operations. Service interfaces are defined using the Web Services Description Language (WSDL) which is an XML based format. WSDL is used both by service consumers to describe the service they require and by publishers to declare the service they provide. In the case of a service provider, the WSDL description must contain more information: the provider has to tell not only what service it provides but also how to access this service.

service interface (WSDL)

WSDL is traditionally used in conjunction with the Simple Object Access Protocol (SOAP) that is a message format designed for Remote Procedure Call (RPC). SOAP is the most widely used communication protocol between web services. As WSDL documents can be somewhat verbose and hard to read, we will review the data contained in a WSDL file before providing the reader with a full example.

service communication SOAP+HTTP

SOAP is based on the XML-RPC and represents method calls and return values using an XML format that can be extended. SOAP implements Remote Procedure Call (RPC) making abstraction of the transport protocol it uses. In most cases, SOAP uses HTTP as its transport protocol: because it was designed for the web, HTTP is the most "open" protocol. HTTP has the advantage of being useable in many organizations: even when they block most network traffic, organizations often allow communications through HTTP proxies. Because SOAP is decoupled from its possible transport protocols, there is no tight integration between HTTP and SOAP. All information composing SOAP requests and responses is contained in the "body" of the HTTP messages.

service address: URL

In a WSDL description, a service is composed of ports on which operations can be called. When used in conjunction with SOAP, each port has an associated address in the form of an URL where the web service can be found. This URL is interpreted as a classical URL by a web browsers and an HTTP request is sent to the web server with content expressed using SOAP. The following example illustrates the use of HTTP containing a SOAP request for the web service at the address http://remi.emo.net/MyWebService/AccessItUsingSoap. To invoke the web service, the following request could be sent to the network host named "remi.emo.net":

```
1  | POST /MyWebService/AccessItUsingSoap HTTP/1.1
2  | Host: remi.emo.net
3  | Content-Type: application/soap+xml; charset=utf-8
4  | Content-Length: ...
5  |
6  | <?xml version="1.0"?>
7  | <soap:Envelope
8  |         xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
9  |         soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
10 |    ...
11 | </soap:Envelope>
```

basic HTTP request

We can see that the service address, that is given in the WSDL service description file, is used to find the service at lines 1 and 2 of the HTTP request. The previous example also shows that SOAP uses the possibility of HTTP for a request to contain some content (lines 6 to 11) to send the SOAP request to the web service (contained in a `soap:Envelope`). In a WSDL service descriptions, a service port may have a SOAP address but can also have a binding to some operation. In WSDL, an operation has a name and some input and output parameters that can be typed using XML Schema (XSD), a language used to define grammars for XML formats.

XML extensibility

WSDL is an illustration of how XML can be used as a base format to provide extensibility and interoperability. Not only does a WSDL description reuse XSD for its message format description but, even if WSDL has been designed to work with SOAP, it is not dependent on it. WSDL is highly extensible and providing SOAP related information in a WSDL document is only optional and rely on the capability of XML to mix namespaces. The primary drawback of XML usage is the verbosity of the formats it produces.

WSDL verbosity

While XML and its extensibility is well suited as a replacement for most binary and non extensible formats, it is not always easy to read or write by humans. It is possible to build a human oriented language based on XML but this is a difficult task and requires some compromise. The most human oriented XML format is probably HTML (XHTML) and many of its element names (e.g. “p”, “em”, “h1”, “br”) are abbreviations to make it faster to write content. WSDL is not designed to be written or read by humans and the best illustration is to eventually give a full WSDL example in this manuscript.

```
1  | <?xml version="1.0" encoding="UTF-8"?>
2  | <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
3  |      name="StockQuote"
4  |      targetNamespace="http://example.com/stockquote.wsdl"
5  |      xmlns:tns="http://example.com/stockquote.wsdl"
6  |      xmlns:xsd1="http://example.com/stockquote.xsd"
7  |      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
8  |     <types>
9  |         <schema targetNamespace="http://example.com/stockquote.xsd"
10 |                         xmlns="http://www.w3.org/2000/10/XMLSchema" >
11 |             <element name="TradePriceRequest">
12 |                 <complexType>
13 |                     <all><element name="tickerSymbol" type="string"/></all>
14 |                 </complexType>
15 |             </element>
16 |             <element name="TradePrice">
17 |                 <complexType>
18 |                     <all><element name="price" type="float"/></all>
19 |                 </complexType>
20 |             </element>
21 |         </schema>
22 |     </types>
23 |     <message name="GetLastTradePriceInput">
24 |         <part name="body" element="xsd1:TradePriceRequest"/>
25 |     </message>
26 |     <message name="GetLastTradePriceOutput">
27 |         <part name="body" element="xsd1:TradePrice"/>
28 |     </message>
29 |     <portType name="StockQuotePortType">
30 |         <operation name="GetLastTradePrice">
31 |             <input message="tns:GetLastTradePriceInput"/>
32 |             <output message="tns:GetLastTradePriceOutput"/>
33 |         </operation>
34 |     </portType>
35 |     <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
36 |         <soap:binding style="document"
37 |                       transport="http://schemas.xmlsoap.org/soap/http"/>
38 |         <operation name="GetLastTradePrice">
39 |             <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
40 |             <input><soap:body use="literal"/></input>
41 |             <output><soap:body use="literal"/></output>
42 |         </operation>
43 |     </binding>
44 |     <service name="StockQuoteService">
45 |         <documentation>This service ...</documentation>
46 |         <port name="StockQuotePort" binding="tns:StockQuoteBinding">
47 |             <soap:address location="http://example.com/stockquote"/>
48 |         </port>
49 |     </service>
50 | </definitions>
51 |
```

mixing XML namespaces

The preceding WSDL file could describe a web service accessible using SOAP and implementing a "StockQuoteService" functionality with a single operation (at line 30). The operation has an input and an output format and is bound to a SOAP operation (lines 39 to 41). It is mainly composed of XSD elements (in the `types` element, lines 9 to 21), SOAP binding elements (`soap:*` elements on lines 36 to 47) and of WSDL elements (rest of the structure).

web service registry

Web service advertisement and discovery is provided by a Universal Description Discovery and Integration registry (UDDI). As previously introduced in figure 3.1, service providers publish their description into the UDDI registry that processes queries from service consumers. WSDL is used to describe web services in the registry. Web Service technologies are the base of Semantic Web Services that we present in section 3.2.5.

3.2.3 Micro Services and Dependency Injection

Web services presented in the previous section are the more commonly used implementation of Service Oriented Architecture (SOA). SOA is also widely used in other contexts including the .Net and the Java platforms.

micro services

In parallel to web services providing cross-platform and cross-language interoperability, a service oriented approach may be applied at a smaller scale. The domain of micro services is very active and there are an important number of frameworks and research experiments around these platforms. Because they work in a more constrained environment, these frameworks are easier to put into practice and to experiment with. Two main design patterns are used to achieve low coupling provided by service orientation: the service locator pattern and the dependency injection pattern. In this section, we will present these two patterns. An introduction and an interesting discussion about these patterns can be found in [Fowler 2004].

service orientation in OOP

In an object oriented programming environment, the key advantage provided by the service locator pattern or by dependency injection is the decoupling of a service client from a particular service implementation. A class Client using a particular service will only depend on the interface Service that defines the service. A concrete instance implementing the Service interface will be dynamically provided to the Client. The client is not responsible for the instantiation of the implementation of Service: a third party is responsible for this instantiation and for providing the instance to the client.

service locator

The *service locator* pattern uses a ServiceLocator class responsible for the discovery of existing services. The Client queries the ServiceLocator to obtain one or all implementations of a particular Service interface. Implementations of the Service interface can be provided by any other part of the system. Figure 3.2 illustrates the dependencies between different classes.

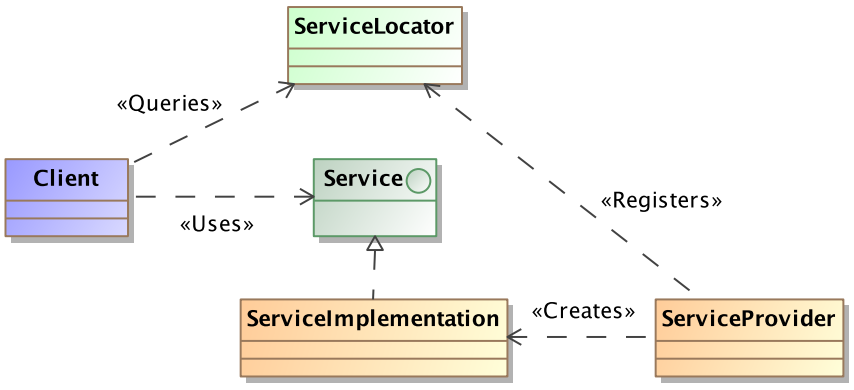


Figure 3.2: *UML Class Diagram: The Service Locator Pattern.*

Java services

The service locator pattern is used by default in the Java platform in the form of a ServiceLoader class. The ServiceLoader plays the role of a service registry and can be queried for any type of services. Any "jar" archive (the packaging unit for Java libraries) can contribute implementations of any type of service using the "manifest" file describing the jar. These facilities are used to provide implementations for image loader, database drivers, etc. The pattern is also present in Java dynamic modules systems like OSGi presented in [Marples 2001] and the Netbeans module system (see [Boudreau 2007]).

dependency injection

Dependency injection applies the "inversion of control" (IoC) principle by externally providing their dependencies to objects. If a Client object requires a particular Service, a particular instance of this service will be provided to the Client by a third party, either at construction

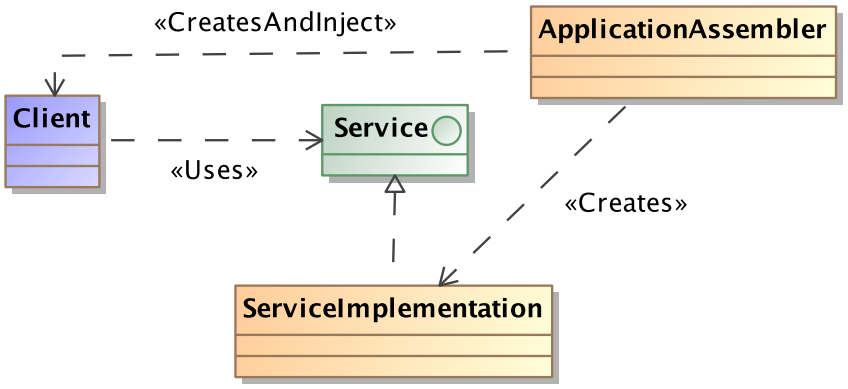


Figure 3.3: *UML Class Diagram: Dependency Injection Principle.*

time or later using a dedicated method. Figure 3.3 illustrates the dependencies between different classes.

injection of service implementations

Dependency injection is used to design application building blocks decoupled from each others. The application is defined as an instantiation and an assembly of these building blocks: objects are instantiated by the application entry point and they receive references to each others through injection: abstract references are passed as parameters of the constructor or using "setter" methods. Various frameworks aim at maximizing the usability of the approach by providing declarative descriptions of the injections. An example of a state of the art framework for dependency injection, based on OSGi, is the iPOJO framework presented in [Escoffier 2007a] and [Escoffier 2007b].

3.2.4 The Semantic Web

≫ Details about "From the Classical Web to the Semantic Web"

semantic web

The underlying principle of knowledge representation methods is to represent knowledge in a structured way, as a network of concepts and relations. The "semantic web" is a vision where most information in the world would be represented in such a structured way. On the current "normal" world wide web, information is mainly expressed using natural language and is hard to understand automatically. Expressing knowledge in a more structured way removes language ambiguities and makes automatic interpretation easier. A more structured form of knowledge causes search to be faster and more accurate.

lexical databases

Different approaches exist towards the realization of the semantic-web vision. One approach consists in building structured knowledge bases based on the expertise of people. Such knowledge bases include "lexical databases" like Wordnet presented in [Miller 1995]. Lexical databases are used as support for other systems and in research, for example to refine classical web search or improve indexing as in [Hotho 2003].

data mining

Another approach stimulating many research works consists in using automatic text analysis to feed semantic networks of concepts. By parsing and doing natural language processing on the world wide web, underlying concepts and relations can be extracted. Examples of research extracting knowledge from Wikipedia are numerous, including for example [Iftene 2008] and [Zaragoza 2007] or the DBpedia project presented in [Auer 2008] or at [Url-b]. Methods working on dynamic high quality databases, such as Wikipedia, benefit from the activity on these databases. Asserting the accuracy of knowledge is a problem inherent in automatic knowledge extraction.

Other initiatives try to stimulate contributions from people by proposing seamless semantic adaptation of existing tools and languages. One example is the RDFa (Resource Description Framework attributes) language that enriches classical web pages with semantics (see [Url-c]). Other examples retain usual web pages made of narrative-style natural language but retrieve most of the underlying knowledge from the semantic web. An illustration of this interleaving between natural language and semantically described knowledge is the Semantic MediaWiki (SMW) project [Url-d]. The SMW project aims at extending MediaWiki (the wiki system on which wikipedia and many other wiki are based) to make it possible to add semantic informations to the pages and to easily query this information to dynamically build page content.

adding semantic to wikipedia

This trend is an illustration of the fact that technology must be useable to have a chance to be used. There are tools for semantic knowledge authoring such as Protégé [Gennari 2002] [Url-e]. These knowledge authoring tools are interesting but, in spite of their quality and usability [Noy 2000], they still may seem over-complicated for the average user [García-barriocanal 2005]. A more accessible approach, where semantic information is progressively inserted into traditional information representation such as in the Semantic MediaWiki (SMW), is a better choice from the acceptability and dissemination point of view. While pure semantic web has not yet come into common use, "tagging" systems and initiatives such as SMW are quickly accepted and more promising. Tagging systems are those systems where users can add some tags of their choices to various documents and media (pictures, movies, etc.).

lessons on acceptance

success of progressive insertion

End of details about "From the Classical Web to the Semantic Web"
(referenced from page 28)

Introduction to Ontologies

≫ Details about "Ontologies and Ontology Alignement"

The term "ontology" is increasingly used to represent sets of interrelated concepts. In informatics, an *ontology* is "an explicit specification of a conceptualization" as introduced in [Gruber 1993], or in more details "a formalization of a shared conceptualization of a domain of discourse". Various formalisms exist to represent an ontology, a common one is the Resource Description Framework (RDF) that is used in this manuscript. In an ontology, concepts can be linked by various relations: subsumption (one concept is a subconcept, a specialization, a particular case of another), type (a concept property as a given type that may itself reference any concept), etc.

ontologies

The Resource Description Framework (*RDF*) is one of the simplest formalism used to represent knowledge. An RDF model is composed of triples of the form "subject predicate object" where "subject" and "object" are two resources and "predicate" is the name of a relation linking these two resources. RDF is the base for most used ontology languages like RDF Schema (RDFS) and the Web Ontology Language (OWL).

RDF

One principle in knowledge representation and sharing is to reuse existing concepts whenever possible and create new concepts when needed. *Ontology alignment* consists in putting in relation concepts from different ontologies. Ontology alignment is accomplished mainly by adding some subsumptions and equivalence relations between concepts created independently. Automatic and assisted ontology alignment is an active research field as illustrated by recent publications including [Choi 2006], [Doan 2003], [Shvaiko 2008] and [Euzenat 2007]. Ontology alignment can be used to enable interoperability in pervasive context-aware applications as detailed in [Euzenat 2008] or as we have presented in section 3.1. In our investigation, we will build upon the use of alignment of software design to enable interoperability.

ontology alignment

End of details about "Ontologies and Ontology Alignement"
(referenced from page 28)

3.2.5 Semantic Web Services

The previous sections have described how the domain of knowledge representation can contribute to software interoperability. This section presents Semantic Web Service technologies that are at the convergence of knowledge representation and service oriented architectures.

≫ Details about "Semantic Web Services"

functionality level service interoperability

Descriptions of non-semantic web services (WSDL files) provide information about the protocol and syntax used to access services and developers have to understand what functionalities are provided by existing services. As introduced in chapter 2, Semantic Web Services add semantic interoperability to web services: a description of the functionality of each service is provided in a machine readable form. Considering these functionality descriptions as live objects of the design makes it possible to build automatic tools and reason about these functionalities.

two technologies: OWL-S and WSMO

Two major set of technologies are used for semantic web services: OWL-S and WSMO. Both technologies remain active. An analysis in [Lara 2004] places WSMO as more promising but less mature than OWL-S; since this analysis was written, WSMO has evolved and matured. We will briefly describe these two frameworks in this section.

description of OWL-S

Built on top of Web Ontology Language (OWL), OWL-S (OWL for Services) can be used to semantically describe services and their functionalities. OWL-S is an OWL ontology that describes three facets of each service:

- One service profile used to describe the functionalities of the service
- One service process model describing how the service works
- One service grounding describing how to concretely access the service

OWL-S is a W3C Member Submission accessible at [Url-f].

OWL-S service profile

IOPE

Figure 3.4 describes the service profile and is extracted from the OWL-S specification available at [Url-f]. A service profile describes both functional and non-functional (name, description, etc.) properties of a service. Functional properties are articulated around the transformation of both data and state. The transformation applied by the service on its data is described by its set of *inputs* and *outputs*. The *conditions* express the required state of the world for the service to be executable. The effect of the service execution on the state of the world is modeled by its *result*.

Having each service registered against a taxonomy of functionalities makes it possible to reason about equivalence and subsumption of functionalities. With inputs, outputs, preconditions and effects (IOPE), programs have access to even more details for the manipulation of services.

OWL-S process model

service interaction sequence

An OWL-S process model describes the correct sequence of messages that a client can receive and send in its interaction with a service. This can be described as a set of atomic processes orchestrated using various control flow operators including sequence, condition, parallelism, iteration. The service profile expresses how the input, output and parameters of atomic processes are related to each others: "variables" are used to express constraints between different elements (as in rule languages). Through the grounding of each atomic process, the process model can be seen as a description of an orchestration pattern.

OWL-S grounding

Service grounding is a mapping of any semantic element onto a concrete representation required to interact effectively with the service. Grounding can for example include a mapping of input and output types to concrete message formats. OWL-S is often used in conjunction with WSDL: atomic processes used in the service profile are mapped to some WSDL operations.

reference to the implementation

(end of details for bulleted list)

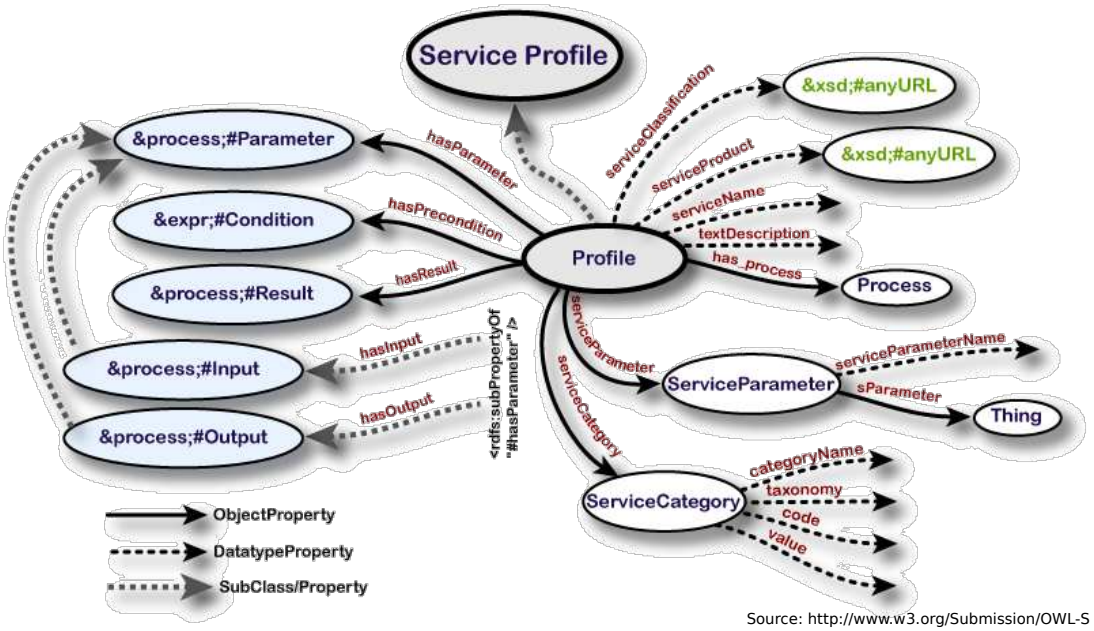


Figure 3.4: *The OWL-S service profile metamodel.*

The Web Service Modeling Ontology (WSMO) is one of the technologies developed by the ESSI WSMO working group ([Url-g]). WSMO is a model for semantic description of web services. The WSML (WSM Language) is one of the possible syntaxes for describing services using WSMO and proposes a choice between multiple tradeoffs on expressive power and reasoning capabilities. WSML is a human readable language aimed at being simple to read and write. Many articles describe WSMO and associated technologies: [Roman 2006] gives a brief introduction while more details can be found for example in [Roman 2005].

description of WSMO

Like most other service technologies, WSMO properly decouples interface from implementation: this is done at an upper level where an interface is the semantic description and an implementation is a web service interface. The most important aspect of WSMO is that it favors strong decoupling by including the concept of a mediator in its model: WSMO services have no strong dependency between each others and they communicate and interoperate using mediators at different levels (ontology, communication, etc.). WSMO is articulated around four points that John Domingue, chair of the WSMO working group, calls the "holy cross" in [Url-h]:

decoupling using ontologies and mediators

- Ontologies
- Goals
- Web Services
- Mediators

Ontologies

All WSMO aspects are described in an ontological description framework. Ontology fragments are provided in the form of a set of concept having attributes and relations. Relations between concepts include elements such as subsumptions (sub typing), symmetry or transitivity. More specific constraints can also be expressed using logical expressions.

Goals

Goals in WSMO correspond to what will be called functionality in other chapters of this manuscript. The concept of goal properly separate an objective provided by a requester (human or machine) from the existing services. Service discovery and reasoning can then be used to automatically fulfill the goal of the requester based on the capabilities provided by web services.

Web Services

Web Services are described in WSMO in terms of the capabilities they provide and of the interface they expose. WSMO can do direct references to grounding information such as Web Service Description Language (WSDL) descriptions. Grounding is not limited to WSDL and can be done using other methods, for example using the Semantic Annotations in WSDL and XML Schema (SAWSDL) presented in [Kopecky 2007].

Mediators

Different types of mediators are defined in WSMO to enable interoperability on the different aspects of WSMO. Ontology to ontology (OO) mediators do integration between heterogeneous ontologies. Goal to goal (GG) mediators interrelates goals. Web service to web service (WW) mediators adapt web services interfaces to make them interoperable. Web service to goal (WG) mediators maps web services capabilities to goals.

(end of details for bulleted list)

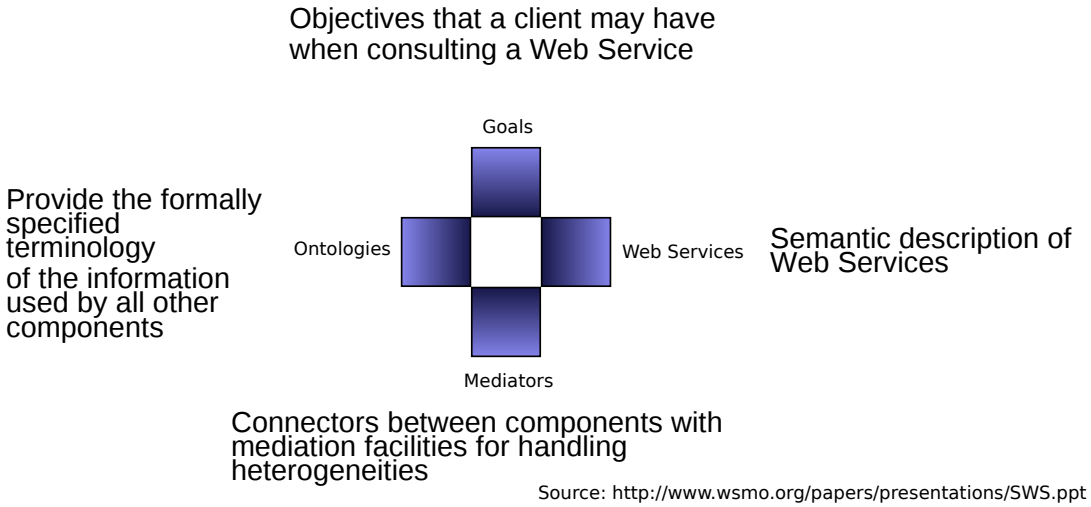


Figure 3.5: *The four facets of WSMO services.*

Wrap up on semantic web services

SWS = SW + WS

The motivation of Semantic Web Services (SWS) is to enable machine interpretation of data and functionalities that are shared on a network. SWS seeds from the observation that Web Services (WS) technologies enable interoperability at a syntactic level only: machines have no information about the functionality provided by a web service. SWS is the fusion of WS technologies and Semantic Web (SW) technologies to describe web services functionalities in a machine interpretable format.

technologies remain complex...

The presentation about both OWL-S and WSMO that best fits the context of our work can be found at [Url-i]. The reasoning about capabilities provided by SWS inspired our approach to enable dynamic spontaneous interoperability in intelligent environments. Even if WSMO provides a human friendly language for writing various descriptions, semantic web services frameworks are not directly applicable to our problem for two major reasons:

- SWS are complex for the non-initiated developer.
- SWS work on web services without notion of connection.

... and web-service oriented

In our context, the building blocks of the applications need to be assembled and remain connected during the time the application is running. SWS technologies concentrate on modeling atomic invocations of web services and ignore the notion of connection between services.

End of details about "Semantic Web Services"
(referenced from page 30)

3.3 Approaches to Service Composition

All software methods aiming at favoring the design of reusable building blocks suppose that new applications can be built by composing existing building blocks. In this section we classify and present various approaches for the composition of services.

≫ Details about "Service Composition"

service composition

As introduced in chapter 2, section 2.6, composition is not specific to service oriented architectures. Over objects or components, services open the possibility to do composition in an automatic and dynamic way. Service composition range from assisted manual composition to fully automatic composition with complex reasoning. Composition of services can be done to produce applications or new services.

We classify composition methods in three top level categories:

- Manual composition of services.
- Automatic composition based on workflow methods.
- Automatic composition based on artificial intelligence (AI) planning methods.

A review of existing automated composition methods, that corresponds to our last two categories, is proposed in [Rao 2004]. The following sections detail these three different composition approaches. Figure 3.6 sums up this classification of service composition methods.

End of details about "Service Composition"
(referenced from page 34)

3.3.1 Manual Service Composition

assembling services

The standardization of web services (or other service technologies) provides a clear definition of the interface of a service while abstracting its implementation details. Using service discovery, tools can list existing services and help a programmer in assembling them to build a composite application. Tools can be graphical and, by reasoning on services inputs and outputs, can efficiently filter possible service compositions. The use of services with well defined interfaces allow powerful and task-efficient tools for interactive service composition and analysis.

automatic orchestration

From descriptions interactively written by the developer, automatic execution and orchestration of services can be performed. This is made possible by the description of service interface and communication protocols. Some tools provide a way to monitor and interfere with a running composite application. Tools based on services and allowing composition are numerous: examples like JOpera and Self-Serv can be found in [Pautasso 2005b], [Pautasso 2005a] and [Benatallah 2003].

human operator is required

The composition based on simple services has limited application as it requires human interaction to build composition applications or services. The objective of semantic web services and other language for service functionalities description is to enable automatic composition of services. To make automatic composition possible, more information on service inputs and outputs and on service functionalities must be provided. The following sections describe existing approaches for automatic service compositions.

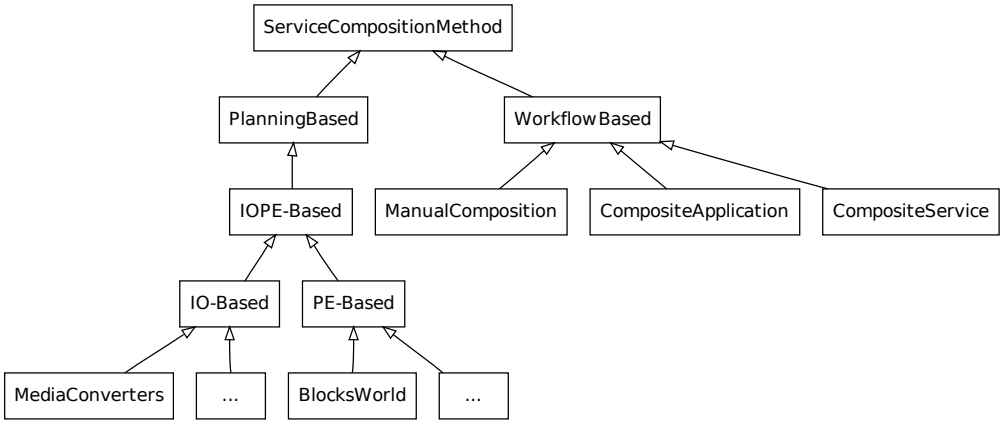


Figure 3.6: *Classification of Service Composition Methods.*

3.3.2 Workflow Methods for Automatic Service Composition

abstract assembly

Workflow based methods for automatic service composition can be seen as an extension to the manual composition method. Instead of assembling existing running web services, the developer assembles abstract services or functionalities. The assembly is done by interrelating different abstract services with workflow operators including sequence, parallelism, conditions, etc.

runtime service selection

In the simplest case, the designer describes an application as a workflow using invocations to abstract (and possibly concrete) services. When the application is executed, service discovery is used to find a concrete service implementing each abstract element in the workflow. Using an abstract workflow makes it possible to automatically adapt the application to the operating condition and the available services. When an abstract element in the workflow needs to be filled

with a concrete running service, there may be some cases where multiple services are available. Different alternatives in such case are to query the user of the application, to choose the "best" implementation based on predefined criteria or to choose an implementation at random.

hierarchical composition

Considering an application composed using workflow as a service makes it possible to construct services in a hierarchical way. A service can be composed of other services, some being atomic services, others being described themselves as a workflow. One of the most widespread workflow language is the industry standard Business Process Execution Language (BPEL) which is discussed for example in [Khalaf 2003]. Many editors are available in the industry for the visual design of workflows, particularly in the context of web services.

3.3.3 Planning (AI) Methods for Automatic Service Composition

Planning methods for service composition evolved from planning in Artificial Intelligence (AI). These methods underly the descriptions of semantic web services described in section 3.2.5. At a general level, these methods are based on the IOPE presented above: inputs, outputs, preconditions and effects.

reaching the goal

In planning based methods, services are represented as transformation of data, through their input and output, and/or as transformation of the state of the world, through their preconditions and effects. A good detailed application of this approach can be found in [Tecnologica 2004]. The objective of planning based methods is to find a sequence of service invocations that transform the current state in the goal state. We can categorize some of the methods on the criteria that they are working mostly on input/output transformation or only on world transformation.

successive format conversions

The problem of planning with only inputs and outputs can be reformulated as finding a succession of compatible converters from a source format to a target format. Planning then consists in chaining service invocations. An applied example of such chaining framework is the Microsoft DirectShow environment that can automatically build graph of "filters". As explained in [Layaida 2005], filters can be of three types: source, transformer and renderer. A sequence of transformers can be created automatically by DirectShow to meet a particular requirement (e.g. low quality rtp streaming of an high quality mpg video). Same architectures can be found in other multimedia frameworks such as the Java Media Framework (JMF).

blocks world

graph search

The other category of service planning only reasons on preconditions and effects of service invocations. This corresponds to the classical blocks world of artificial intelligence (see [Url-j] for details). Each service invocation is an action that can be executed only under certain conditions and that transforms the world state. Planning then consists in finding a sequence of service invocations that will successively transform the initial state of the world into a state that fulfils an objective: a state in the set of goal states. Solving this planning problem requires to explore the graph of possible succession of actions. Different search strategies exist, including forward chaining, backward chaining and heuristic based graph search.

planning for SWS

Semantic web services descriptions frameworks such as OWL-S or WSMO allow both aspects of transformation to be expressed: both inputs/outputs and preconditions/effects are supported. The semantic web frameworks are perfectly suitable for the descriptions of services that can be composed using planning methods. Unfortunately, existing execution environments for these framework concentrate on efficient discovery by only adding semantic subsumptions to classical web service functionalities discovery as in [Srinivasan 2006] and [Shafiq 2007].

3.4 Wrapping It Up in Context

In chapters 2 and 3, a vast range of subjects were treated. This section tries to sum up the major points to remember about these chapters and to briefly introduce our approach.

Interdisciplinarity of intelligent environments

The domain of ambient intelligence, even from a research point of view, requires state of the art software engineering method to evolve effectively. Intelligent environments are interdisciplinary and the cooperation of specialists from various domains is fundamental.

importance of usability

In this context, improving integration can be done by proposing software engineering methods that are easily understandable and applicable by all the specialists involved in the conception of intelligent environments. Any software design method or architecture proposed for intelligent environments should insist on usability and acceptability: a method should ensure the quality of the result it produces but should also maximize its acceptability by its target audience.

on context orientation

Making applications sensitive to their context of use is also a fundamental element for making applications look intelligent. Research around context awareness is very active. Most of the proposed solutions are architectural: context frameworks give a structure to the overall context aware system. From our analysis, service oriented computing is an adequate tool for the application developer and has to be integrated with information about of context. As presented in chapter 4, we envision services as first class software element manipulated by application designers to improve integration and reuse. Depending on the approach, the structure imposed by a service oriented architecture and a context framework may conflict. In our contribution, we tried to separate the service oriented architecture from the context awareness.

on functionality alignment

The domain of knowledge representation tackles the question of the integration of knowledge coming from independent sources: this integration is called "ontology alignment". Real interoperability of software components can be achieved with a vision that is similar to that of knowledge representation. Software services must be semantically described in terms of the functionalities they provide. This semantic description of provided functionalities should be neither at the implementation level nor at the interface or communication protocol level. The description of service functionalities should be conceptual and should allow functionality alignment.

example functionality

As an example of services, consider a "printer" service: what should be described semantically is the fact that a printer service can print a file that is passed to it, has some properties such as the ability to print colors, an indicative printing speed, accepted formats, etc. Lower level information such as the underlying service oriented technology (e.g. Web Services using SOAP) or the method sequencing and signatures (e.g. "`boolean printFile(String fileName)`") are "implementation details" that constitute grounding. *Grounding* expresses how an abstract functionality is grounded in a concrete implementation.

semantic web services

Semantic Web Services (SWS) place themselves at the confluence of service oriented architectures (using web services) and knowledge representation domain. The goal of SWS is to describe web services in a sufficiently structured way that will make it possible to do automatic service composition. Two major classes of approach exist for automatic service composition: workflow methods and planning methods.

on workflow-based composition

The designer of a service can program it by defining a workflow of invocations to abstract services. When the service is executed, service discovery is used to find a concrete service implementing each abstract element in the workflow. One interesting aspect of workflow methods is that they are able to describe not only sequence of service invocations but also wiring of services (dataflows). In our context, we are trying to compose services to build applications: we need to establish connections between services that will continuously send messages to each others. This adequacy of workflow methods inspired us to use compositions patterns as building blocks for our approach as presented in chapter 6. We believed that this is an original approach for reasoning about the interaction of different service composition patterns as we propose.

composition patterns

on planning methods

Semantic web services allow service descriptions that make planning-based approach possible for service composition. Semantic web services frameworks, however, concentrate on efficient discovery. This concentration on efficient discovery is in agreement with their objective to provide semantic discovery for the huge amount of web services available on the web. In our context of intelligent environments, we need planning based composition to happen to enable spontaneous interaction of services. The number of services present in an intelligent environment is small compared to the number of services available on the entire web: a powerful planning based reasoning is more adapted to our context than "simpler" efficient subsumption-based discovery present in current implementations of semantic web services frameworks.

As we will detail in 4, our approach takes inspiration from the domains and methods presented above. An important particularity of our contribution is the concentration on the usability aspect that is key in the context of intelligent environments design. Success of intelligent environments come from effective interdisciplinary integration and we aim at making methods usable by the variety of specialists involved in the development task.

Chapter 4

Identifying Integration Problems in Intelligent Environments

4.1 Motivation and Contribution Overview

sharing is the key

The conception of intelligent environments is highly interdisciplinary, involving many specialists with different backgrounds and cultures. We are convinced that the success of research in intelligent environments is tied to the effective communication, interaction and sharing between these communities.

To improve the synergy between these different domains, we propose to analyze the state of the interactions between different specialties. In this chapter, our analysis will be illustrated by a distributed 3D tracking system developed in the PRIMA group.

analysis objectives

The goal of our analysis is to identify weak points that penalize experimentation and advances in intelligent environments and their applications. This analysis has been done to obtain more precise directions on the requirements for dedicated software design methods and tools. Weak points are opportunities for evolutions: we aim at extracting, from the identified weak points, concrete requirements and coarse grained directions that guide the rest of the thesis.

chapter contributions

In the light of our analysis we are able to examine integration problems in intelligent environments. We find that the move towards distributed computing is an occasion to introduce new software design methods and architectures. Service oriented computing and architectures are well suited for intelligent environments: they provide an approach to solve integration problems while providing a means for distributing the applications across multiple computers. Further analysis efforts show that we must go beyond service oriented principles to enable real dynamicity and spontaneous service interactions in open systems.

4.2 The Intelligent Environment Landscape

4.2.1 Many Specialties Involved In Intelligent Environments

interdisciplinarity in intelligent environments

Following our definition, an intelligent environment must be capable of modeling and recognizing a users' activity to provide the user with contextually pertinent services. These requirements imply that the system must have capacity for both perception and action. Building intelligent environments thus requires specialists from many domains ranging from artificial perception to machine learning to artificial intelligence to robotics and man-machine interactions. This list could be considerably enriched with entire domains such as computer and sensor networks or with very active sub-domains such as natural language understanding.

We started our analysis by identifying and classifying existing research and engineering contributions around intelligent environments in our research team. We propose a categorization along 4 main axes:

- Fundamental perception
- Integrated perception to autonomic computing
- Activity modeling and recognition
- Human-computer interaction

Fundamental perception

We call "fundamental perception", research and system design on artificial perception that can be separate from interacting with upper levels of a system. Fundamental perception algorithms and components are often developed, tested and evaluated off-line, on recorded databases. These systems are designed to be parametered and controlled by upper levels in the system. Intelligent environments are at the confluence of many domains that are still highly active. Whether in computer vision or acoustic perception, much research is concerned with solving problems that are already difficult without the constraints and specificities of intelligent environments. Working in these domains includes object recognition, face detection, visual person identification and voice recognition.

unsolved problems

Integrated perception to autonomic computing

"Integrated perception" includes research on artificial perception that concerns operation in real world situations. This category includes work on visual or multimodal tracking systems.- The "integrated" part comes from the fact that the final product is usually a perceptual component that can provide a packaged service in a wide range of scenarios. Taking works from "fundamental perceptions" and optimizing them to run in real time is the "simplest" form of integrated perception. In addition to the real time aspect, integrated perception often has to operate in less constrained environments which generates important specific challenges that might be considered uninteresting for fundamental perception: easy or automatic configuration, automatic adaptation to changing environments, etc. Insisting on these problematics leads to the domain of autonomic computing that aims at building systems that can adapt to their environment and continue working.

robust perception

Activity modeling and recognition

"Activity modeling and recognition" aims at studying, describing and observing human activity. On the one hand, it consists in finding formalisms to express and describe these activities. These works are heavily using perception capabilities of the environment and are used to provide services to the users (e.g. in [Essa 2000], [Christensen 2002]). Modeling user activity or group activity is a key requirement to provide them with context aware services: not understanding what users do makes it impossible to avoid unnecessary distractions or to provide additional assistance at the right time. Even the users are not directly in interaction with the system, recognizing their activity can still be central in some applications such as automatic recording of meetings and seminars ([Metze 2005]), or ambient monitoring and assistance to favor elderly people living in their personal apartment ([Url-k], [Cook 2006]).

recognizing human activity

Human-computer interaction

An intelligent environment aims at providing the user with useful context dependent services and thus how the user interact with the system is a fundamental aspect. In an environment where standard screen/mouse/keyboard disappear to be replaced by more diverse and spontaneous interaction forms (gesture, speech, portable and wearable devices, etc.), the interaction paradigms have to be reinvented, tested and improved.

ergonomics

Apart from the ergonomic aspect of the new form of interactions, new engineering methods also have to be found. The interactive capabilities of an intelligent environment are continuously evolving: a user brings his laptop or cell phone, this adds many perception and action capabilities; the ambient noise is too loud, speech recognition and acoustic feedback will not be of much use; etc. Given these variations on the environment capabilities, interactive system design must be general: the major parameter in the current devices is the screen size and it has only a limited impact on the design. Todays interactive systems are designed for a small family of hardware configurations. Given the increase in user mobility and in the number of devices, future systems will have to increasingly address an open environment containing devices that are not known at design time, neither in number nor in variety.

dynamic interaction context

(end of details for bulleted list)

The involvement of a wide spectrum of scientific communities is necessary for the creation of intelligent environments. Their interaction and cooperation is the key to the realization of the ambient intelligence vision. This interaction already shows to be a hard task in the context of our team where different specialists have different cultures and objectives. We observe that this aspect takes an increased importance with ambient intelligence systems.

inter-cultural interactions

4.2.2 Problems With Capitalization and Reuse

The problem of reuse and conservation of legacy software often manifests when designing applications at higher abstraction levels. For example, applications involving activity modeling or learning of user habits and preferences must have perceptive capabilities. Reusing off the shelf perceptual components being the fastest alternative, designers often follow the following steps:

attempt to reuse

- study the perceptive functionalities that existing perceptual components provide and directly reuse it when possible
- else, find a backup solution among the following
  - implement a component that has the same functionality but uses a simplified and constrained method (e.g. people localization using an RFID tag reader)
  - implement a component that is based on the methods used by perception people but with the proper packaging for its use in higher level systems; this usually ends up with a poor quality and poor performance component as the expertise is missing from its design
  - decide to work on purely simulated data supposing that one day the perception will do exactly what they want

fallbacks

All the enumerated situations can be observed depending on the context. The first situation involving reuse brings a mutual benefit to the stakeholders: perceptual components designers benefit from having their components used and tested in various contexts and, at higher abstraction level, application designers can build bigger systems faster and safer. The reuse is a virtuous circle: continuous reuse and integration makes further integration easier.

virtuous circle: continuous integration

The lack of software integration, from perceptual components to higher level applications, heavily penalizes higher abstraction level efforts. Software designers have to do substantiate

higher levels penalized

additional development, often to obtain only low quality stubs of perceptual components for their experiments.

lower levels also

At lower abstraction levels (e.g. perceptual components), the lack of integration and reusability hinders the ability to test algorithms and systems. The lack of integration also becomes a limitation considering two current evolutions of perceptive systems:

- perception systems are increasingly distributed,
- perceptual components increasingly interact.

perception systems are increasingly distributed

example of vision systems

To illustrate the tendency to distributed processing in perception systems, we can take a simple example of a tracking system working at the scale of a building. In this example we will restrain the sensors to cameras covering a meeting room, an office and a corridor. Even if we could use video or network cables long enough to wire all the cameras to the same computer, it would not be able to support the acquisition and processing or transmission of all the video streams. This first constraint explains the initial requirement for distributed processing of sensor data: all the processing necessary for the perception cannot happen in a single computer. Even the 3D tracking system that may be run on the cameras (usually 3 to 5) in the office require to be distributed to be properly operating at the time of its creation. The problem of building and defining architectures for perception systems is an active area of research as illustrated in [Jovanovic 2007], in [Crowley 2007] or in [Lömker 2006] and [Wrede 2006].

This kind of distributed processing is different from grid computing: it is not only a question of dividing a computing task in part and scheduling their execution on a computer grid. In the case of perceptual systems, computers are not interchangeable: sensors such as video camera, microphones, RFID tag readers or bluetooth adapters are connected to particular computers.-

localization of data

Even if data can be transmitted to be processed in another computer, network bandwidth limitations still prevent some sensor data types to be systematically transfered. Lossless video stream transmissions quickly break any existing network down: a camera such as [Url-l] doing acquisition of 1600x1200 images at 30 Hz with a digitization of 12 bits per pixel ($1600 * 1200 * 30 * 12 = 691.2 Mbits/s$) already uses a dedicated gigabit ethernet connection to communicate with its host computer. Image acquisition technology currently evolves as fast as networks and we can expect this tendency to continue: cameras are present in many devices, are increasing in resolution and represent a major industrial market.

perceptual components increasingly interact

limited reliability of perception

Each perception technique has its intrinsic limits that can be overcome only by getting feedback from other perception systems or from higher level systems. No perception method is 100% reliable and most of domains tend to converge to a limited reliability. Only a few tenth of percent improvement is achieved each time a method is improved.

merging perception methods

The tendency to solve the intrinsic limitations of each perception method is to merge multiple methods that are all imperfect but complementary. The fusion of methods is inspired by natural perception. In the case of human perception, it has been shown that there is a continuous integration and feedback between different perception modalities as illustrated for example in [Giard 1999]. Without the interaction of these perceptions, humans perception is like computer one: inaccurate and imprecise. A concrete example is the use of lips reading when trying to understand speech in noisy environments as studied in [Erber 1975]. This multimodal computer perception can be done directly at the signal level, by considering all the low level signals (audio, video, ...) as a whole, or at higher level by reasoning about the output of low level perception process (tracking, voice recognition, ...).

4.3 Symptomatic Example: the 3D Tracking System

Section 4.2 presented the variety of specialties involved in intelligent environments. In section 4.3, we present an existing 3D tracking system developed in the PRIMA team. This tracking system will serve as the main illustration in our analysis.

tracking system as an integrator

The 3D tracking system presented in this section is a major element in many intelligent environment scenarios. A 3D tracking system has the potential to integrate a considerable variety of contributions including image processing, audio localization, localization using wifi or GPS, person identification, etc.

the seed of our contributions

The description given is this section is fundamental as it illustrates and guides our analysis of the conception of intelligent environments. All the contributions presented in the rest of this manuscript (chapters 5 to 8) are motivated by this analysis presented in following sections.

4.3.1 General Principle of the Tracking System

PRIMA's tracking system

We will use a 3D tracking system developed in the PRIMA group (there are equivalent problematics in other works and in other teams) as an illustration of perception systems. This 3D tracking system is initially an improvement over an existing 2D tracking system. The basic principle of this 2D tracking system is to detect and follow visual targets in the image, making an efficient use of computing resources.

detection regions

The tracking system has a set of current targets and a set of detection regions. A *detection region* is a rectangle in the image were the system will look for the appearance of new targets. Detection regions are usually fixed on entry points (doors, steps, etc.) or can be randomly moved at each frame to statistically cover all the image (in a few frames).

image based detection

Both detections regions and the regions around tracked targets are used for detection. A *detection* is an affectation to each pixel (of the region of interest) of a probability of target presence. The detection can be performed with various image processing algorithm. For example, one method is to compute an image difference between the current image and a reference background image.

2D tracking process

The 2D tracking system operates as follows:

- Predict the new position of each target: based on the position, size and speed of each target in previous frame, we predict in which region of the image it should be at the current frame.
- Estimate the new position of each target.
  - A detection is performed in the region of interest obtained by the prediction of the next position of the target.
  - From this detection image, the new position and size of the target is estimated.

  At this step, "split" and "merge" can also happen based on various criteria. A "split" is the replacement of a single target by two targets: this happen when the detection image seems to contain two distinct targets. A "merge" is the opposite: two targets become too close or overlapping, the system replaces them by a single one.
- Do a detection for new targets in each of the detection regions. A new target is created on the detection region if the detection has a sufficient "energy"; that is if the overall probability of target presence in the detection region is sufficiently high.

Using this principle, image processing is only done where necessary: in the detection regions and around the targets.

4.3.2 From 2D to 3D Tracking

$n * 2D = 2.5D$

The first attempts to move to 3D tracking were done by running multiple 2D trackers and interpreting their results. Supposing the targets are evolving in a plane in the 3D space, we can convert a position in the camera image to a position in a 3D reference frame. Doing so for each target on each camera, we obtain targets in the 3D space. The 3D tracking system then tries to associate targets from different cameras together. This "simplest" approach is only a kind of 2.5D tracking: the tracking itself is done in each camera and the occlusion problems are not better solved than in 2D tracking. This approach was only briefly experimented and a full 3D tracking was quickly implemented to robustly solve occlusion problems.

real 3D tracking

Full 3D tracking consists in tracking targets directly in the 3D space. Compared to the 2D tracking case, both the set of detection regions and the set of targets are defined in the 3D space; target prediction is also done in 3D space. The detection of a 3D region of interest (predicted target or detection region) is done by projecting it on each of the camera 2D image, doing the detection on the projected region on each camera and then merging back the probability of target presence issued from the 2D detections.

distribute for performances

Being it 2.5D or 3D, the tracking system started to use 3 cameras and to require much computing resources. Depending on the size of the images, the number of cameras and the number and size of the targets, the tracking rate was not following the framerate of the cameras anymore. This performance problem was the first motivation to split the system and distribute it among multiple computers.

a tracker + one component per camera

This initial split led to the tracking system as it was at the beginning of our investigation.- This initial redesign of the system consists in having a single 3D tracking component for the system and an additional component for each camera. The "3D tracker" is responsible for holding the set of detection regions and the set of current targets. When it has to do a detection, the 3D tracker will delegate to each camera the task of doing a detection in the 2D image space. Each component responsible for this 2D detection is called "detector estimator".

3D tracking process

When it has to do a detection, the 3D tracker will pass necessary information to each detector estimator, receive back the result and merge the informations from all detector estimators. The cases of detection regions and targets are a little different:

- For detection regions, the tracker projects the 3D detection region to a 2D region of interest on each camera using the camera calibration (a transformation from the 3D space to 2D image coordinates). This 2D detection region is passed to the detector estimator that is responsible for doing the 2D detection (e.g. background subtraction) and summarizing it as 2D gaussian estimated from the 2D detection image. This gaussian representation of the 2D detection is transmitted back to the tracker that fuses it with the other same informations coming from other 2D detector estimators.
- For 3D targets, the 3D region of interest is projected on each camera as for detection regions. In addition to this information, the 3D prediction is also projected to each camera space. This 3D prediction is a 3D gaussian, and, when it is projected, it is approximated as a 2D gaussian (mathematically, the projection or marginalization of a 3D gaussian is not an exact 2D gaussian). Receiving a 2D region of interest together with a 2D prediction of the target position, the detector estimator does the detection and fuses it with the provided estimation. The 3D tracker then fuses the result of all detector estimators to obtain the current target position.

a necessary system review

The conversion from 2D to full 3D tracking required to split the system and distribute it across multiple computers. Initially done only for performance concerns, the separation of the system is minimal and not necessarily done in the best way. The example of the evolution of the tracking systems perfectly illustrates that systems need to be reviewed when they need to be distributed. This brings two remarks that will impact our approach:

- the conversion to distributed computing should be relatively easy to make it acceptable for the various specialists and,
- the necessary rework of most of the contributions is the occasion to introduce simple but state of the art methods of software development.

4.3.3 Distributed Tracking Using BIP

from ad hoc communications...

The 3D tracker and its detector estimators, as presented in previous section, are different processes that must be able to run on different computers. Communication between this processes were first performed in an ad hoc manner by specifying at startup time a port for each detector estimator and a configuration for the 3D tracker containing the list of hosts and ports were detector estimators are running. All network communication details and interpretation of the stream were provided by the tracker and detector estimator code.

... in each project...

We illustrated the need for easier interconnection and communication between components of perception systems through the case of the tracking system. At this time, this need was present in almost every application and research work done in the team: automatic adaptation and error recovery in the context of tracking systems, activity recognition based on a situation model, automatic learning of situation models, etc. Basically, any component or application that required to consume the output of the core perception components (tracking systems, voice detection, etc.) had the need for easier connection and communication with these components.

... to a shared communication library

BIP

To provide a simplification of network communications, a dedicated library was designed at the same time as the 3D tracker described above. At the time, the library for network communications was called "BIP" like the underlying protocol it used: the Basic Interconnection Protocol. BIP requirements were:

- Avoid having to specify host and port for interconnections.
- Have interoperable implementations in at least C++, Java and Tcl.
- Have a lightweight communication protocol that does not become the bottleneck of perception systems.
- Enable interoperability between major operating systems (Windows, Linux, MacOS).

service oriented middleware

BIP was an early version of a library for service oriented architectures. BIP made it possible to define communication channels of 3 types: input, output, bidirectional. The protocol used splits into messages the communication stream that occurs through a channel. The communication channels can then be added to a "service". In addition to its communication channels also called "connectors", a service has a name and a set of state variables. The most important aspect of BIP services is that they can be published and discovered using the concept of zero configuration networking (Zeroconf): it makes it possible for components to advertise their presence and look for other ones based on their names and their properties rather than their host and networking port.

discovery using Zeroconf

With Zeroconf, services are registered using the same form as in the standard Domain Name System (DNS). Classical DNS constitutes a directory that is distributed and replicated on a relatively small set of servers. Zeroconf uses multicast UDP packets to implement a fully distributed naming system on Local Area Networks (LAN). This DNS implementation over multicast UDP is called Multicast DNS or mDNS. On top of DNS, DNS based Service Discovery (DNSSD) only consists in advertising in a DNS repository services such as available printers, ftp servers, ssh servers. BIP services are advertised using DNSSD and are self described making it possible for other BIP peers to find their connectors and their variables.

naming

As the name "BIP" was confusing and only described the protocol, it was renamed to "O^{3}MiSCID" standing for Object Oriented Opensource Middleware for Service Communication Inspection and Discovery. For typographical simplification, it is often named "OMiSCID" or "Omiscid" by stripping inconvenient exponent.

necessity dynamic deployment

Having a system composed of multiple distributed components adds some deployment work. For the 3D tracker to find its detectors estimators, they must have been already started on the proper computer. Some work has been done to automate this deployment using the Jade multi-agent platform as detailed in section 6.2.2. One problem of tying the deployment of all the part of the applications is that the application is entirely configured by the deployment descriptor. Having such definitions of the application does not encourage developers to build applications with a dynamic architecture.

4.4 Obstacles to Software Capitalization and Sharing

4.4.1 The Computer Mouse: a model for intelligent environments

necessary abstraction

Perception is massively used at higher abstraction levels. From the higher abstraction levels, perception should be accessed in abstraction of the implementation details. Choosing how to abstract lower level processes is delicate: the abstraction must be neither too generic nor too specific.

a successful abstraction: the mouse

A simple illustration of a well defined abstraction is the computer mouse. From higher abstraction levels, a mouse is a device that can input relative movements and some button presses. This abstraction can be used directly, for example to navigate in an immersive 3D environment or still at a higher abstraction level by considering the mouse cursor that maintain a "position" state. The lower level perception components implementing the functionality of the mouse can be diverse:

- ball mice and trackballs use two notched wheels to capture two dimensional movements
- optical mice use image processing to track table movements under the mouse
- touchpads and touchscreens may use pressure or capacitance of fingers to detect pressed location
- many other pointing devices exist and have their particular 2D movement perception: pen tablets, Wiimote (Nintendo Wii remote), keyboard-based mouse emulation, etc.

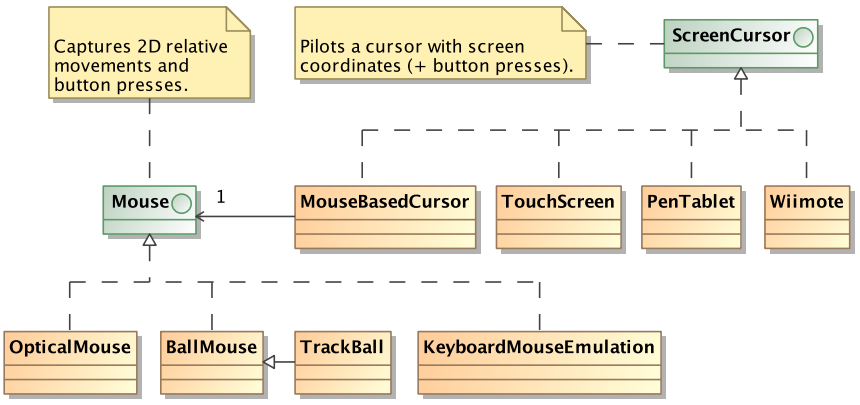


Figure 4.1: *UML Class Diagram: computer mouse abstraction.*

Figure 4.1 underlines the concept layering involving the mouse and the cursor and gives a classification of mouse input devices. The example of the computer mouse illustrates the fact that abstractions can be used by both higher level "clients" and higher level abstractions (the cursor). The concept of computer mouse has contributed to the rapid development of visual desktop environments by abstracting this fundamental interaction device.

profitable abstraction

Perception used in intelligent environments is more complex, various and evolving than the domain of pointing devices. It would however be desirable to generalize the mouse principle to various concepts in intelligent environments. This abstraction can be profitable to have for sensors but also for actuators.

finding abstractions for intelligent environments

Different perception concepts could be layered as with the computer mouse and the cursor. Lower perception usually directly operates on images, video sequences or audio streams. These lower level components produce higher level perception such as target positions, target shapes or speech activity. These higher level perception can usually be both used to implement higher level tasks and, at the perception level, to produce other information such as person's posture or person's attention.

layering perception abstraction

4.4.2 SOA Adequacy and Advantages

Even if it does not ensure them, Service Oriented Architecture (SOA), introduced in section 3.2.1, favors some good software design practices:

SOA benefits

- clear affectation of responsibility between system parts,
- better separation of concerns,
- sharing and reuse of existing software,
- encapsulation of functionalities,
- independence to implementation and distribution topology.

These aspects are clearly missing in the 3D tracker we presented beforehand. To give an example of a possible problem and improvement, we can take a look at the "detector estimator". Already doing a simplification, we can list four major functionalities embedded in the detector estimator: doing image acquisition from the camera, having the knowledge of the background, doing the detection, doing the estimation using the detection and the received prediction. Even if performances could impose some restrictions on the split of the system, in this case, the "estimation" part could be separated from the "detection" part. The acquisition of images from the video camera could also be separated from the rest without major impact on performances as shown in chapter 9.

separating responsibilities

Separating the system in parts makes these parts easier to test, more reusable, distributeable and more robust as the crash of a part does not bring all the system down with it. Chapter 6 covers these advantages in details by rearchitecturing the 3D tracking system. The rearchitectured system is more extensible and some of its parts, including the images acquired from the camera, can be shared and reused at runtime.

foreseen improvements

SOA has the qualities and potential to improve integration and reuse in intelligent environment.- To provide an effective solution, SOA must be widely adopted: isolated creation of services has only a limited impact. We can set an objective to have most of the actors involved in the team (and in the development of intelligent environments) fully embrace the concepts of SOA.

objective: SOA adoption

It is important here to take conscience of the gap there is between the old architectures with at best clearly specified modules linked together and the new service based architecture. The whole team "management" idea is here to take advantage of the need for distribution to jump

opportunity: jumping directly to SOA

to a state of the art software architecture method. Looking at the domain of vision systems, we see that they are advancing slowly compared to the gap that we try to cross. Vision systems are moving to distributed systems but they still remain in a software environment that does not evolve at runtime.

improved dynamicity

We want to exploit the fact that the distribution method that is used by the tracker is implementing a service oriented architecture to impose all the associated practice in common usage. To explain the difference here, the main particularity of the service oriented approach is that services are discovered dynamically at runtime. Services must thus be designed in a way that makes them robust to the absence or disappearance of a particular service, or to the dynamic apparition of a new service (e.g. a new detector estimator could be started when the 3D tracker is running).

action: pushing SOA

Intelligent environments require more than distributed perception systems: the systems must be designed to handle the dynamic nature of the environment with sensors being brought in or out of the system at any time. The adequacy of SOA for the intelligence environments is the motivation to trying to push the full adoption of SOA among developers. Having identified this major objective toward better development methods for intelligent environments, we will present in chapter 5, what should be done to facilitate SOA adoption and what we did for it.

4.4.3 Problems Beyond SOA

remaining problems

Continuing our analysis shows that SOA is a good first step but is not sufficient to solve the problem of software integration in dynamic intelligent environments. Other problematics could be summarized as follows, and chapter 6 tries to give methodological and conceptual answers to these problematics:

- Deployment is unnecessarily repetitive during system conception.
- Using a service requires to know, at design time, the functionality of the service.
- Using a service requires to know, at design time, the data model of the service.
- Higher level processes have no way to abstract the constant evolutions happening at the perception level.

Deployment is unnecessarily repetitive during system conception

When testing systems involving multiple services with some dependence on their configurations, it is often required to stop and restart services. To reuse our 3D tracker example, we can precise that detector estimators are started with some parameters that are also part of the 3D tracker configuration: this is the case for the kind of detection algorithm that is used. Doing detections with a background subtraction method or with a consecutive image difference (motion detection) does not lead to the same interpretation in the 3D tracker. Thus, when the designer want to test different methods, detector estimators have to be restarted and this is mostly done manually. Even if it only consists in aborting a program and restarting it with some modified parameters, this task is tedious and error prone.

Using a service requires to know, at design time, the functionality of the service

When writing a service that will look for another, one must specify how to find this other service. This specification is primarily done by providing a service name that will be looked for. This kind of service discovery is limiting in the sense that the name of the service must represent its exact functionality. To quickly illustrate this limitation, we can take the case of

a sound player that is designed to play the sound that a "microphone" service is outputting. If we want to listen to a "radio" service that is outputting sound at the same format as the microphone, we will have to change to sound player to handle services named "radio". Another solution would be to externally force the connection of the sound player to the radio service.

Using a service requires to know, at design time, the data model of the service

In the same manner a service has to know the name of the services it will use, it also has to know the "format" of the messages it will exchange with these services. What we call the "data model" is this format that describes how information is structured in the messages sent between services. BIP allows any messages to be sent: messages are plain binary data. Except for high bandwidth tasks (image streaming), message are in text formats and mostly in XML formats. Even if we suppose an XML formatting of the messages, it is still required to have a description of what tags are used in the messages and what they represent. What we call the "protocol" is the convention on what exact form messages take and when they are sent. As an example, we can give a simple output protocol for the 3D tracking system:

- interactions: one message is output at each cycle of the tracking loop;
- content: an output message has a timestamp and a list of targets with, for each, its identifier, position and size;
- format: messages may be formatted in XML with for each target a "target" element (tag) containing attributes for the identifier "id", the position "mx", "my", "mz", etc.

Only with such information, is it possible to interact with an existing service. Such information makes it mandatory to patch an existing service consumer to make it understand another newly integrated service that would provide the same information but with a different "protocol".

Higher level processes have no way to abstract the constant evolutions happening at the perception level

To do experiments in real conditions, higher abstraction level research works need to use real perception and action system. These research works include activity modelling or user modelling, like the personal assistant we used as a guiding thread in chapter 2. Modelling the user's activity, learning user's habits and interacting with the user necessarily involve the perception of this user. Such higher abstraction level research suffers from the continuous evolutions made to perception systems: evolutions bring many performance and accuracy improvements to the perception system but are often accompanied by slight changes in the usage.

To build themselves efficiently, the research domains placing themselves as a client of the perception processes need to have an abstraction of the perceptual functionalities provided by the environment. There is a delicate choice for this abstraction: it must be a real abstraction over the exact perceptual services, but must not be too coarse as higher level processes want to keep a certain control on what is perceived.

4.5 Opening on Other Contributions

In this chapter, we exhibited the major integration and interoperability problems impacting the conception of intelligent environments. In intelligent environments, most building blocks depend on some others. Designed by specialists from various domains and communities, these building blocks are rarely interoperable. Interoperability problems leads to a lack of integration between contributions from different specialists.

integration problems...

... cause capitalization problems

The lack of integration causes most of the software contributions to be lost as soon as their author leaves or changes research domain. In some domains, like "fundamental perception", this lack of capitalization over previous works as only a few consequences: theories, algorithms, methods and concepts are the core matter and are capitalized with scientific publications. For intelligent environments, however, the absence of software integration and reusability is catastrophic.

... solved by SOA

In our analysis of the integration problem we observed a tendency of different systems to get distributed. We identified that the requirement for distributed processing is a unique opportunity to introduce Service oriented architectures (SOA) in the development of components for intelligent environments. SOA fulfills the requirements for distributed processing but also provides better software integration, reuse and dynamicity. We formulated the adoption of SOA as a major goal for intelligent environments. Chapter 5 explains our actions toward the completion of this goal.

... and more

We also identified a set of major obstacles that remains even if SOA is used. These obstacles mostly concerns service deployment and runtime integration of existing services. Chapter 6 details some methodological and conceptual answers to these problems.

Chapter 5

OMiSCID: a Usable Middleware for Service Oriented Architectures (SOA)

5.1 Motivation and Requirements for SOA Adoption

need for capitalization

Chapter 4 introduced the particularities of software design for intelligent environments and underlined problems with reuse and capitalization. Problems of integration and capitalization penalize experimentation and advances in the domain of intelligent environments. Service Oriented Architectures (SOA) encourage improved code interoperability, integration and reuse. Chapter 4 identified the use of SOA as an enabler to handle integration problems and dynamicity of intelligent environments.

need for distribution...

... an opportunity for SOA

From our analysis, we have seen that perception systems tend to expand in complexity and are increasingly distributed. The distribution of perception systems often implies that the developers learn a new tool: methods used by developers rarely handle distributed system design. Not only service oriented principles can solve integration and capitalization problems but they are also one possible solution for the distribution of perception system. We believe that using a proper service oriented solution as a distribution mean will simplify further integration and reuse of heterogeneous components. To make SOA used as a distribution mean, we need to make service oriented design as easy or easier than custom distribution methods.

BIP middleware

At the start of this thesis work, we adopted a middleware used by the group to implement service oriented architectures and named BIP (Basic Interconnection Protocol). The possibility to properly implement services revealed a problem: the complexity of the implementation was repulsing potential users and not favoring the use of services as a design element. At this time, the implementation was the result of a bottom up technology-driven design: the API (Application Programming Interface) was obfuscated by implementation details. The OSGi version of BIP was already an abstraction over the Java version and influenced the conception of the user oriented API we propose in this thesis.

pushing service orientation

from BIP to OMiSCID

Our study has led to a complete redesign of this middleware to make it more accessible to potential users. Given the tendency at this time to use the middleware only for distribution and not for real service oriented designs, this appeared to be necessary to ensure our other contributions on service oriented architectures have an impact. The new implementation has exploited the existing BIP codebase to provide solutions to many technical problems. During redesign, BIP has evolved into OMiSCID (Opensource Middleware for Service Communication Inspection and Discovery). In this manuscript and as a simplification, we will call BIP the previous implementation and OMiSCID the new redesign.

SOA blockers

Through interviews and discussions with potential users, we have identified the most important aspects that inhibited the use of SOA in general and BIP in particular. We can summarize these problems as follows:

- Some users did not see a clear return of interest in implementing SOA. This was argument invoked mostly (but not exclusively) by persons working on lower level components (with few dependencies to other components).
- In the conceptions of some people, the term of "Services" was associated with "Web Services" and was thus for "those who build web applications".
- Users tended to find service design too complicated: for example too much source code had to be written with BIP.
- Service oriented architectures were perceived has having bad performance. Splitting systems into parts and adding communication was perceived as having a major impact on software performance by most people.

These points are the main problems that came out of our analysis of people feeling about the use of services in their applications.

With the design of OMiSCID, we sought to solve these problems. We recognized, moreover, that some of these problems cannot be solved by any implementation as they are at the communication and at the tool level. Rather than directly produce pure implementation specifications for OMiSCID, we first enumerated a set of guidelines to maximize the adoption of SOA by a family of users. From our point of view, most of these guidelines can be generalized to the adoption of other methods and other users.

guidelines

We propose the following guidelines for the adoption of service oriented architectures by engineers and designers involved in the conception of intelligent environments:

- SOA implementations must be very easy to learn by the target audience
- In the first steps of the learning process, the return of interest for the audience should be illustrated through examples
- SOA implementations must be easy to install
- SOA implementations should be available in any language susceptible to be used by the target audience
- Development of services should integrate in the target audience's development process
- SOA implementations must be efficient and should communicate about this efficiency (using benchmarks)

Applying these guidelines in our context leads to three kind of actions:

- designing a user-oriented service oriented middleware,
- designing tools to integrate service design in the software development process,
- communicating about service oriented middleware.

Our actions concerning these three aspects will be detailed in the following sections.

5.2 Implementing OMiSCID, a Usable Middleware for SOA

The new implementation of OMiSCID was based on the BIP code base to maximize the reuse of already written code. This has the advantage to keep all the lower level conventions and thus to have a runtime compatibility between OMiSCID and BIP services (even if this is not used anymore). Our contributions around OMiSCID falls in 3 categories: designing a brand new user oriented API (Application Programming Interface), improving the existing implementation and expanding the language availability.

5.2.1 User Oriented API

a plain library

The form of a library, already adopted by BIP, was totally appropriate for the target audience: developers are used to using external libraries. Graphical environments purely dedicated to a new development method also tend to be rejected by programmers. This tendency is natural and there is no good reason for requiring developers to change habits to adopt service oriented architectures.

favoring proper design

When designing the new API, we listed use cases from a user standpoint. Given our objective to promote service usage, we have to make a compromise between pure use cases (that rarely use real service oriented concepts) and service oriented principles. The objective is to have an API that fulfills use cases, that is simpler than other methods for the users and that lead to a proper service oriented design.

more concise API

A major area for improvement of BIP was to reduce code that needed to be written and to limit the number of concepts and classes involved in the API. We can give some quick facts about this new API, more details (including source code example) can be found in chapter 9. Starting a service with a variable and a connector, find another service and connecting to it requires:

- With BIP
  - 6 BIP classes with their full implementation details, most of them requiring to be instantiated and managed by the developer,
  - 25 different methods names involving all the nifty-gritty details of BIP implementation and the manipulation of many pointers.
  - $\Longrightarrow$ 45 lines of codes purely induced by the use of BIP.
- With OMiSCID
  - 2 classes and 4 “concepts”, the total number is the same as for BIP but the complexity is greatly reduced and the abstraction higher,
  - 10 method names,
  - $\Longrightarrow$ 15 lines of easily maintainable code purely induced by the use of OMiSCID.

fewer concepts

In the OMiSCID API, the fewest possible concepts are exposed to the users to simplify their understanding. The user manipulates Services that represent services created by the application, ServiceProxys that represent services discovered through OMiSCID service discovery. By manipulating a Service, the user can add some connectors and variables to it, start and stop it, and manage connections between its connectors and other OMiSCID peers. The user can register listeners on Service connectors to be notified of received messages and the same on local Service variables (that can be modified by other services) and remote ServiceProxy variables.

understandable service declarations

We simplified the creation of services and particularly the addition of connectors and variables, but also the interconnection of services. With OMiSCID, the user can express in a more concise way how he wants to set up his service. Simplifying this part of the process clears

a barrier for new users: code examples that starts services are readable and developers can more easily understand the basic meaning of them.

better discovery paradigm

Another tedious process with BIP was the discovery of remote services to obtain ServiceProxys. With OMiSCID, looking for remote services can be done punctually through a Service or continuously by attaching a ServiceRepositoryListener to a ServiceRepository. A ServiceRepository monitors the apparition and disappearance of services in the environment; each listener gets automatically notified of these event and receives an associated ServiceProxy.

declarative service search

The key simplification of the service discovery API comes from the introduction of declarative ServiceFilters. Using BIP, the specification of what are the remote services of interest was done by implementing a callback function. The user had to write this function and handle all the details of finding if the service matched a his criteria. With OMiSCID, the user can use logical combinations ("and", "or", "not") of predefined services filters to describe the search criteria. For example, a ServiceFilter can be expressed like this in OMiSCID:

```
1 |   And( NameIs("MovieMaker"),
2 |        OwnerIs("Bob"),
3 |        HasConnector("Command", AnInput))
```

What is straightforward to write and read in OMiSCID would require writing a function of at least 10 lines of code with BIP.

custom filters

As with BIP, highly specific filters can also be written by implementing the ServiceFilter interface. This use case is extremely rare as, to our knowledge, this possibility has never been used. In the case a specific filter is implemented, we can expect its author to follow the good practice that is exampled in the API and to write atomic, combinable and reusable filters.

By making service declaration and discovery far easier than in BIP, the user oriented API for OMiSCID makes the use of service oriented principles simpler than a custom implementation of network communications. In our context, the simplest way for the average to distribute an application is to do it using services.

common API

One last benefit that this new API brings is that it unifies the API between the two languages of implementation (at this time): C++ and Java. BIP implementation were runtime-compatible meaning that services written in C++ could communicate with service written in Java. BIP API for Java and C++ were however very different. With OMiSCID, an effort has been made to have very similar APIs in all languages. Some differences in language features imposes some slight difference in APIs but basically, someone who has written a service in C++ can easily understand how to write one in Java and vice versa.

5.2.2 Corrective Maintenance and Evolutions

In previous section, we introduced the new OMiSCID API that we designed for user friendliness. The manipulation of the BIP code base and its wider use through the new API was the occasion to improve the hidden part constituted by the implementation.

testing and bug fixing

Providing an implementation for the newly created OMiSCID API was done by modifying the existing BIP middleware. Defining a new API, we also did thorough testing of the proposed API. Manipulating the code and increasing test coverage revealed a set of bugs and "not yet implemented" features. While we had the hand in the code, we fixed these bugs and implemented the missing features.

reactive support

The improvements to OMiSCID and particularly its API caused its overall use to increase. This increased use revealed some remaining bugs, mainly race conditions. Reactively fixing these bugs has been a requirement to the adoption of OMiSCID. This can be retained as a

lesson for the launch of any software aiming: near real-time handling of user's requests is a critical factor of success. User support does not take a huge amount of time but it requires constant availability to quickly provide the users either with bug fixes, workaround for bigger problems or supplementary information when documentation is insufficient.

We also made evolutions on the API due to queries and observations of usage. These evolutions mainly consisted in adding some methods that we had forgotten when designing the API and in creating new helper methods for most observed use cases. To illustrate the kind of modifications that this represents, we can mention the addition of an optional ServiceFilter when registering a listener to a ServiceRepository. At the beginning, ServiceRepository were notifying their listeners of all service apparition and disappearance and often, almost systematically, users were holding a service filter against which they tested each service that appeared.

improved documentation

Another area of improvement we worked on during this reimplementation was the installation of OMiSCID. We made OMiSCID as easy as possible to install and documented the necessary steps. Depending on the language, the installation and use of OMiSCID library varies slightly. As an example, a Java developer can learn how to develop and start an empty OMiSCID service in less than 3 minutes by watching some short screencasts (available on the OMiSCID website [Url-m]).

Implementing a new API unified between all languages was the occasion to reassess our choices about what languages should be supported by OMiSCID. BIP services could be designed using C++, Java or Tcl. Due to the deprecation of Tcl in the team, no Tcl implementation was provided for the new API. The Java version of OMiSCID can be still be used in Tcl by using the pure Java implementation of Tcl (Jacl, [Lam 1997]). Only two "native" implementations of OMiSCID were thus developed: a Java version and C++ version, both fully operating under the three main operating systems (Windows, Mac OS, Linux).

Python and Matlab support

With its use by new users, we identified that OMiSCID should also be available to programmers developing in Python and Matlab. These two environments can be targeted by adding a layer of adaptation over the core implementations. Python can use OMiSCID C++ library through some wrappers that are mostly autogenerated. Some subtleties involving differences in memory allocation and threading between C++ and Python complicated the work of writing these wrappers. For the other case, Matlab being written mainly in Java, it is easy to use a Java library. Only a few code had to be written to properly handle callbacks from OMiSCID to listeners written in Matlab: when a service started from Matlab receives a message, it is likely that the user wants to process the message in Matlab too. The additional classes have to be compiled against the Java Matlab Interface (JMI) but given the lazy nature of Java class loading, it can still be included in the Java distribution of OMiSCID.

5.3 Building Tools for SOA

When considering the usability of a technology, the core technology cannot be separated from the associated tools. For example, the usability of a programming is dependent on the availability of dedicated editors, good libraries, debugging tools, etc. Previous section presented OMiSCID, a solution to implement Service Oriented Architecture (SOA). This section discusses the tools required to make our solution more usable and introduces graphical environment we propose to manage services.

5.3.1 Which Tools?

Visualization tools?

Having a library that can declare, discover and interconnect services without any tool to visualize services is like driving by night in a high-tech car with no headlights. With BIP already, it was identified that the service designers and consumers need a way to visualize running services. Together with the Java implementation of BIP, a graphical user interface (gui) was initially designed.

Like BIP itself, the existing gui was very good as a prototype but was exposing many low level details of BIP implementation: it was designed as a tool for the creators of BIP. Some interesting features were however already present in this first gui. The most interesting was that it was extensible: extensions could be written in Java even though none were written due the underlying complexity.

generic service management

The developers and consumers of services want to be able to visualize what services are present and what are their connectors and variables. Some use cases are generic enough to consider them as part of the service oriented library. These generic requirements mostly include 3 kinds of operations: listing of all running services, accessing the details about a service and visualizing the interconnections between services.

service-specific extensions

In addition to the basic visualization of the services and their interconnections, users often want other tools to visualize and interact with their services. Services can be diverse and some of these tools can only be useful to the developer himself or to a small community of developers. Even for these highly specialized tools, there is a need for sharing: when a service consumer wants to use a service you designed, he can often take advantages of the visualization tools you developed around for your service. When we talk about visualization tools here, we include both tools used to observe the service and tools used to interact with it.

As far as the highly specific visualization and interaction tools are concerned, we cannot and should not implement them in place of the concerned developer. What we should ensure in the design of a visualization tool is that it must be extensible and that extensions should be sharable and easy to install.

Code generation tools?

avoid wizards...

Another kind of tools that could help in the development process are code wizards. A "wizard" interactively ask the user for some main parameters and automatically generate some, usually long and verbose code. One problem with wizards is that to be comfortable with them, the user has to understand all the code that is generated (see [Hunt 1999]). If the generated code is not understood by the user, this means it is an abstraction for the user and that the details should be hidden from the user, for example encapsulated in a library. We prefer to design a clear and simple API than providing wizards that would generate huge amount of code. We followed this principle with the new OMiSCID API. From a user standpoint, one advantage of wizards is that it makes it possible to author code without actually writing code: wizards are often more declarative than standard programming.

... prefer good APIs

XML service descriptor

To fulfill the need for a yet more declarative way of declaring services, we made it possible to describe a service with its name, its connectors and variables using an XML service descriptor file. As the XML descriptor is interpreted directly by the OMiSCID library, this approach has the advantage of not generating any code. By modifying an XML description, it is possible to change a service by just restarting it, without the need for a recompilation. A replacement of such an XML description can be to use dynamic programming languages such as Python or any scripting language running on the Java virtual machine and able to access any Java library.

5.3.2 The OMiSCID Graphical User Interface

Design Alternatives and Decisions

new Gui from scratch

We weighed the two possibilities we could imagine to provide the user with a visualization tool for their services. We could either start from the existing code knowing that most of it would require to be rewritten or we could start from scratch with a clearer design and better requirements. We decided to consider the existing gui as a prototype and to implement a brand new gui from this experience and the requirements we identified. This decision made it possible to reconsider some technical choices that were done and most significantly the fact that no library or application framework had been used.

Rich Client Platforms

We looked for a portable desktop application framework built for extensibility. Two main "platform" fulfilled our requirements: the Eclipse Rich Client Platform (RCP) and the Netbeans Platform. Both platforms provide about the same functionalities: extensibility through plugins, easy update via update sites, reusable core application framework, dockable windowing system, portability brought by the Java virtual machine, etc.

Swing / SWT

Even if, at least at the time, Eclipse and its plugins were more popular and many rich client applications were build on the Eclipse platform, we decided to settle down to the Netbeans platform. The first criteria was that Netbeans is based on Swing, the standard while Eclipse is based on the "Standard Widget Toolkit" (SWT). SWT was created as the toolkit for the Eclipse IDE: the competition with Swing drove both toolkits forward. Being in the core Java library since Java 1.2 (1998) and having a fixed API since then, Swing tends to be learned by more people and was also more mature at the time.

Netbeans reputation: easier to install

Our second criteria was a purely psychological one: Eclipse was accumulating reports of users having problems with the installation of plugins. Eclipse has long been more open than Netbeans and thus has attracted more contributions. These contributions produced many plugins for a wide variety of task but these plugins were of unequal quality. Even if the main plugins were properly designed and fully functional, some additional plugins were conflicting when installed together or simply bugged. Users that wanted to use Eclipse for different tasks often ended up with having multiple Eclipse installation to avoid conflicts between plugins. For a user having lived such an experience, talking about an application that uses the eclipse platform for its extensibility will usually induce a negative prior.

Gui guidelines

Having decided to build the OMiSCID Gui as a Netbeans Platform application, we started the implementation following some principles about the gui:

- it must include a list of running services,
- it must allow the user to manage variables (view and set their values),
- it must allow to manage connectors (receive and send messages),
- it must be easily updated with new plugin versions,
- it must be easily extensible with plugins written by domain experts,
- it should not include any domain specific plugins,
- it should include a view of service interconnections,

Based on these principles and trying to be efficient in the implementation, we designed the gui with first all the "must" and then adding the visualization of service interconnections. The result constitutes the core of the OMiSCID Gui that is presented below.

OMiSCID Gui - Core

This section details the core components of the OMiSCID Gui. These are the components installed by default in the Gui and usable with any OMiSCID service.

listing services

The OMiSCID gui revolves around a main graphical component: the service browser. This service browser lists all running services and displays their variables and connectors. When starting a bare installation of the gui, only this tree will initially be displayed. By default, the service browser only displays a tree composed of services and their components as illustrated in figure 5.1 but the user can decide to add some columns giving details about services, the tree is in fact a "tree-table" as shown in figure 5.2.

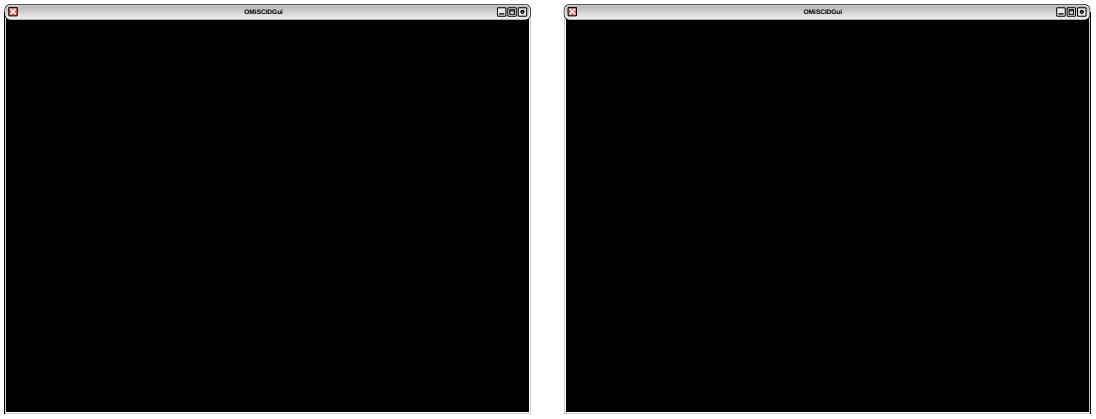


Figure 5.1: *The OMiSCID Gui showing only the service browser. On the left: with no service. On the right: with three services, one being unfolded.*

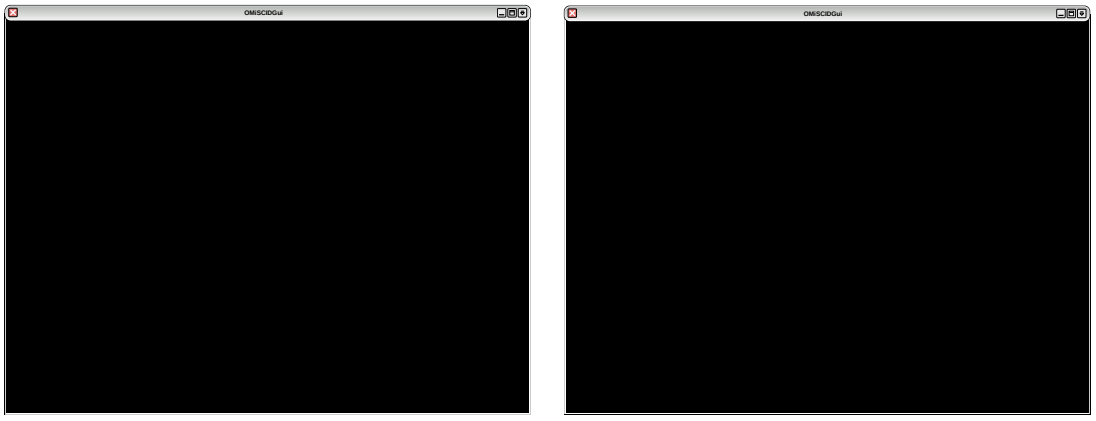


Figure 5.2: *The OMiSCID Gui showing only the service browser with some different user-selected columns selection and placement.*

service browser = extension point

The service browser is the entry point for most of the plugins developed for the OMiSCID Gui. Plugins register themselves as participating in the context menu of the service browser. The end of current section explains the underlying principle for the extension of the gui.

variable and connector manipulation

The core gui includes a variable manager and a connector manager. Both can be opened through a context menu entry that is present only when the selection is pertinent (contains a variable for the variable manager, etc.). The variable manager is used to monitor the changes in service variable value and, if the variable has remote write access, to query for a modification.

Figure 5.3 shows the variable manager in action. In the same way, the connector manager can be used to connect to a service connector, display the messages sent by this service and interactively send messages to services. Figure 5.4 shows the connector manager in action.

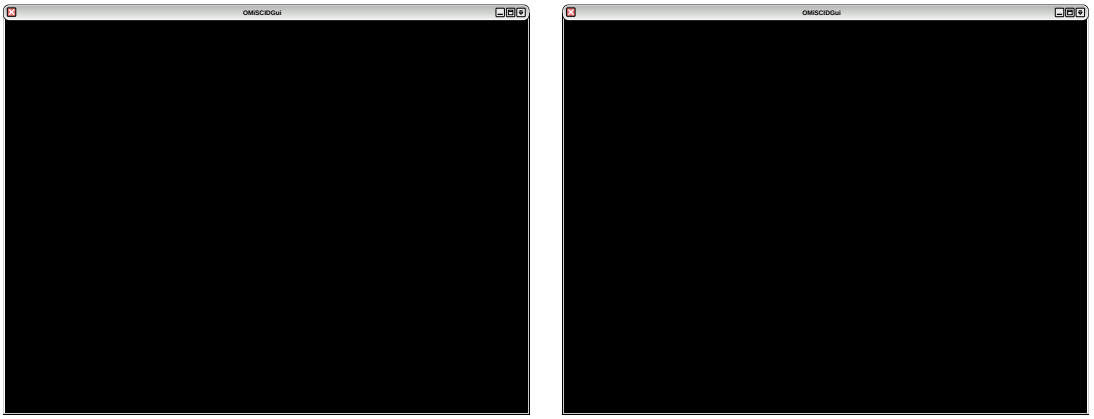


Figure 5.3: *The OMiSCID Gui showing a variable manager component. On the left: monitoring a read only variable. On the right: monitoring a variable with write access.*

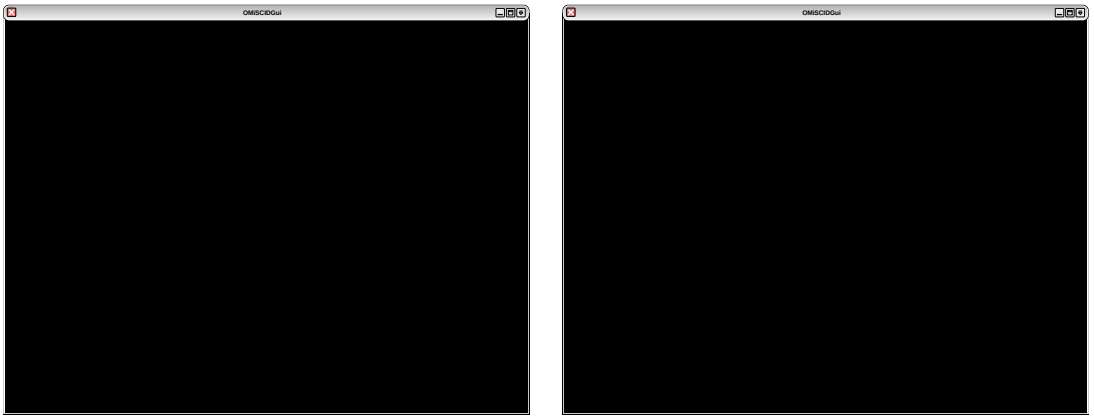


Figure 5.4: *The OMiSCID Gui showing a connector manager component. On the left: monitoring an output connector. On the right: monitoring an "inoutput" (bidirectional) connector.*

service interconnections

The next interesting feature present in the core gui is the possibility to visualize the interconnections between services. This visualization uses remote introspection of OMiSCID services to build a graph that is automatically laid out but can be tuned by the user. One connector of a service can be connected to other connectors of other services. It is also possible to have OMiSCID peers that are only clients and are not declared as services: these appear without name on the service interconnection graph. Figure 5.5 gives an example of service interconnections.

automatic update and extension

The last functionality provided with the core gui can be considered as the most powerful one as it is a meta-functionality. The gui comes preconfigured to use a default "update site". An update site is a website hosting updates for the core modules of the OMiSCID Gui and its extensions. The principle of the update sites is built in the netbeans platform and this functionality "comes for free" with our choice of using this framework. An update site for the OMiSCID Gui makes it possible to release bug fixes and enhancement to the OMiSCID Gui core but also to publish new extensions that people want to share. The OMiSCID Gui automatically checks for updates and prompts the user if he wants to install these updates automatically: this update process and installation of new extensions is very simple and straightforward for the user.

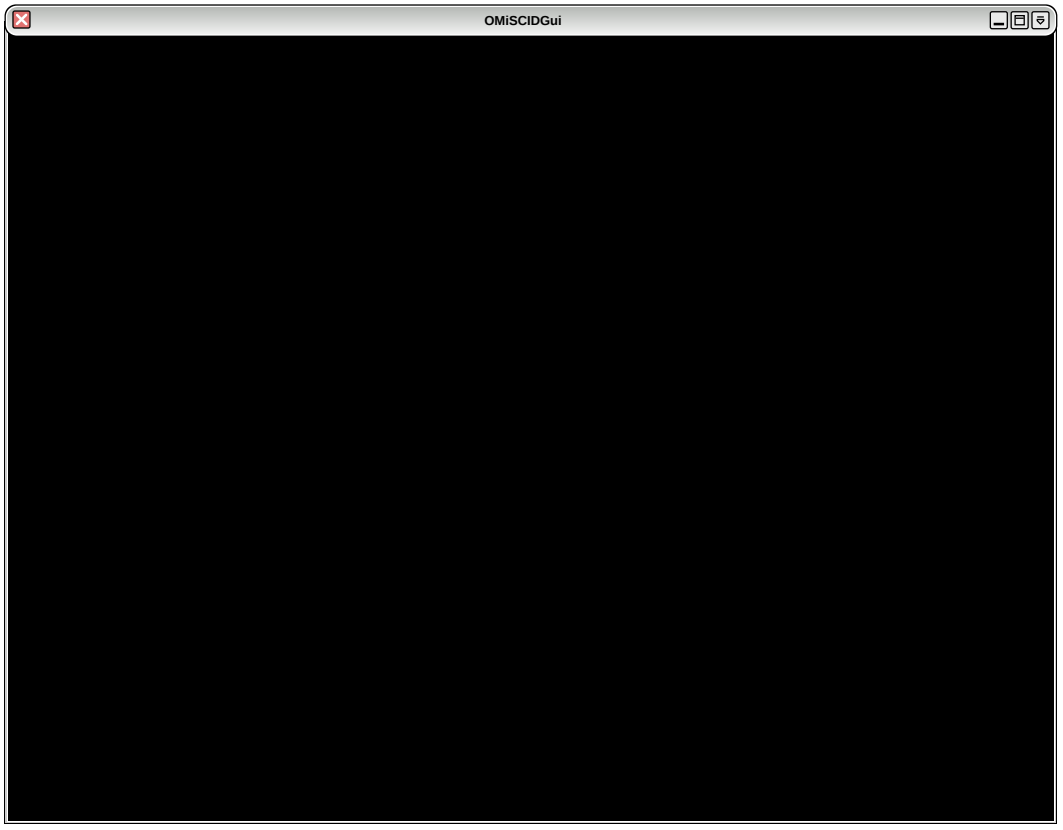


Figure 5.5: *The OMiSCID Gui showing the service interconnections graph. Non-service peers appear with no name (like* `f6627a00`*). Services can be folded, hiding their connectors and variables (like* `RandomIncrementalVariable`*).*

OMiSCID Gui - Extensions

Gui modules/plugins

The core gui provides only the most widely used functionalities. The gui is designed to be easily extensible and more specific needs can be relatively easily implemented by the authors of services. The extensions to the gui can be called plugins or modules: a module is a unit of deployment and update in the netbeans platform and a plugin is a module or set of modules that bring a functionality to the user. The distinction between plugin and modules is only important to consider in some focused discussions and most users use the two indifferently.

existing plugins

A set of plugins has been designed by different persons in the team for their particular needs. These plugins are used by the other people through the automatic update site. We will showcase only a few plugins but plugins have also been designed for other tasks: 3D visualization of a 3D tracking system, plotting variable value evolutions, remote control cameras, skinning of the gui interface, listening to audio sources, visualizing audio signal and its FFT (fast fourier transform), etc. We also wrote a plugin that is useful for writing documentation. This plugins takes vectorial screenshots of the gui by leveraging Java2D and the Batik SVG library. All screenshots present in this manuscript have been generated using this plugin.

laser telemeter

We first take an example with a quite simple device and its associated plugin. This device is a laser telemeter that measures distance of objects in a plan by scanning almost 360 degrees around itself. The telemeter outputs a list of distances for a sampling of its field of view; the angular resolution is around half a degree. The outputted distances are not easy to exploit by just reading the outputted messages.

We show, in figure 5.6, the gui with two views of the output of the telemeter: the generic connector manager and a visual view provided by a plugin dedicated to the telemeter service. The plugin was specifically written to view the telemeter. This plugin registers an action in the context menu of the service browser: this action only shows up when a Telemeter service is selected and, when executed, it displays the graphical component that we observe on the figure.

In our first example, we basically have one plugin for the Telemeter. Our second example of extension involves more plugins that interact together. These plugins range from highly task-specific to very generic. One contributions provided by the plugins (the "addin manager") are now integrated in the core gui even if they were not presented in previous section. These functionalities are easier to understand in the use case we present now than out of context and were initially developed outside the core gui, it is thus better to present them as such.

3D tracking

Our second use case shows the tools used during the design (and the demonstrations) of the 3D tracking system. The 3D tracking system involves many services including some cameras, some image processor and the 3D tracker itself. During the development of the 3D tracker, it is often executed on some recorded videos instead of using live video. Working on recorded videos make it possible to pause the system and also to rewind the video and replay a time interval to study the behavior of the system at this time.

controlling image sources

Figure 10.1 shows a typical situation where the gui is used to manage and visualize the 3D tracking system. On the upper left corner of the gui, the classical service browser is displayed, listing the running services. On the lower right corner, a dedicated panel is used to control the camera service that are reading recorded images from the hard drive. This controller is in charge of synchronizing the playback from various cameras but it also makes it possible to control the replay speed, rewind the record or pause the playback. This controller is created by selecting some camera services in the service browser and using the action that then appears in the context menu.

visualization of output

Other elements are used to visualize the images from the cameras together with the boxes representing regions of interest that are used by the image processing services to do person detections. One plugin provides the component that is able to display the image received from

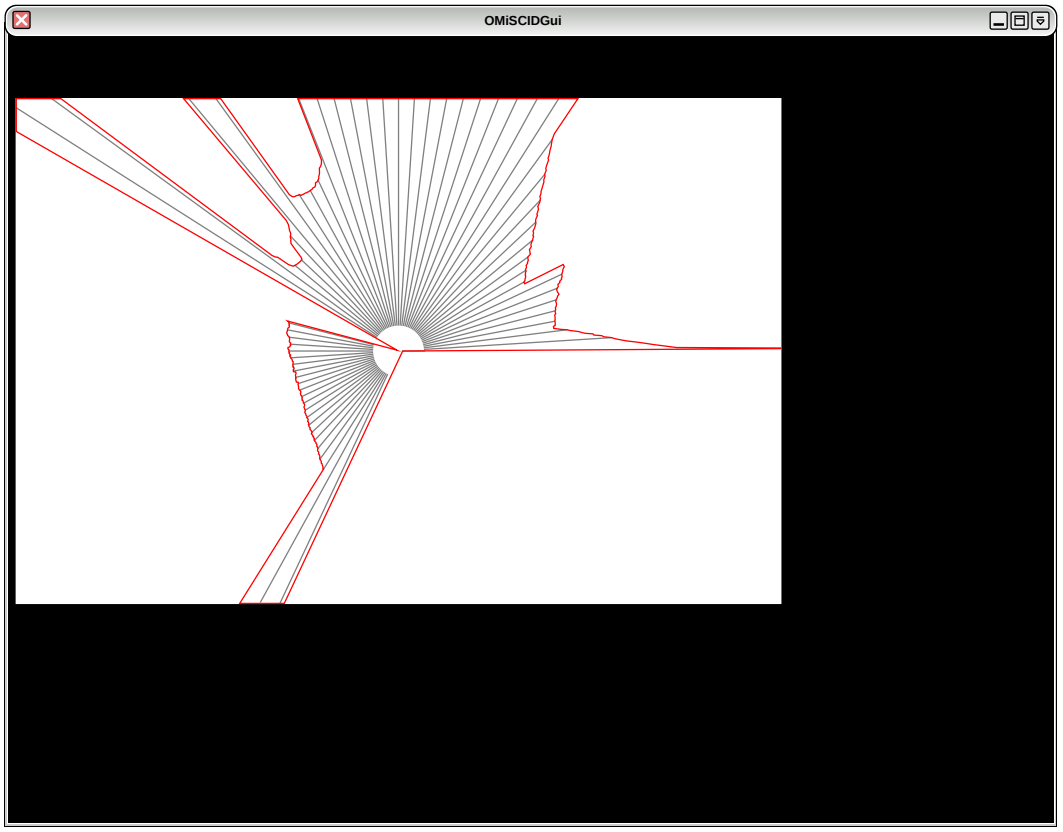


Figure 5.6: *The OMiSCID Gui with a plugin showing a graphical representation of a Telemeter service output. Laser rays are represented diverging from the position of the telemeter and a shape is reconstructed from the impact distance of these rays. For example, the two obstacles at north-west of the telemeter correspond to the feet of a standing person.*

the camera. Another plugin is responsible for the connection the image processor and the display of the regions of interest as rectangles. Both of these elements are injected in the J2DViewer, provided by a third plugin and proposing a generic panel to draw using Java2D API. Each plugin that adds the image or the rectangles to the J2DViewer depends on the J2DViewer but they do not depend on each other. It is perfectly possible to visualize an video stream without having an image processor and vice versa.

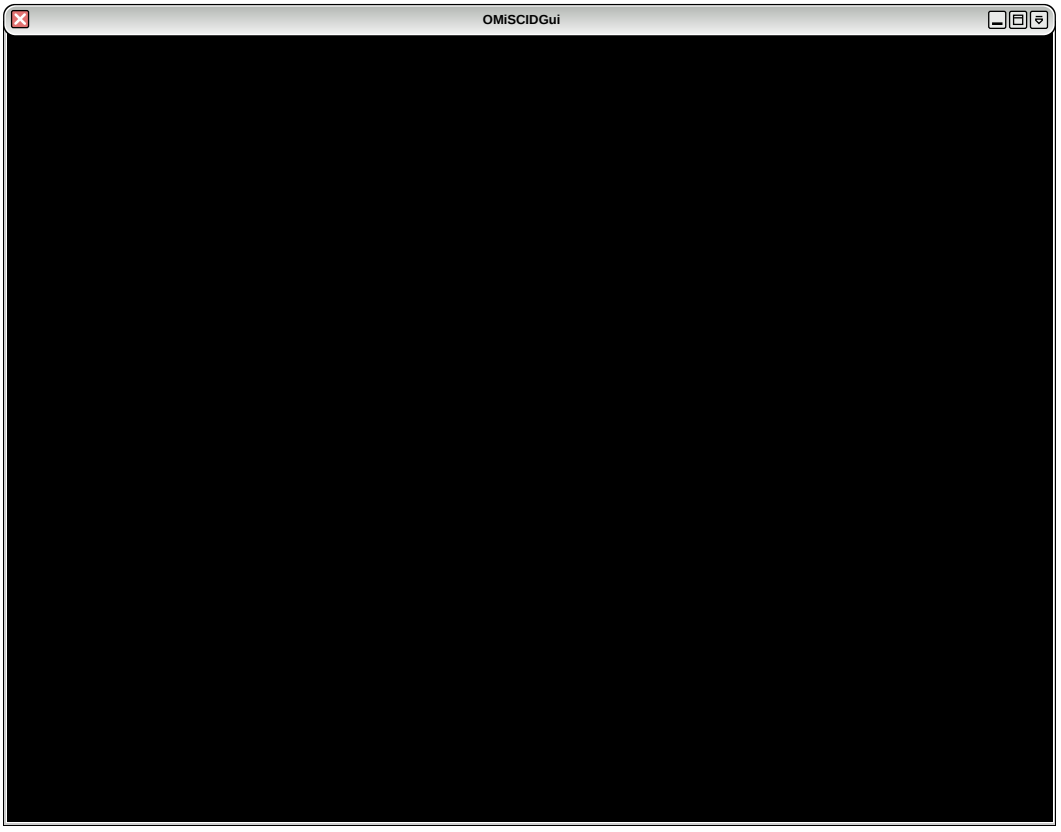


Figure 5.7: *The OMiSCID Gui used to monitor and control a running service assembly that constitutes a 3D tracking system. Multiple plugins interact to provide this overall view (detailed in the manuscript).*

Gui elements life cycle

To explain the last panel, on the lower left corner, we first take back the case of components such as a connector manager. When a connector manager is opened, it connects to a connector and displays a visual component. This visual component can be closed by the user with the implicit semantic that it should close the connection and free any allocated resources. The user can manage the life cycle of the connector manager using the visual component that was created. Other elements such as the variable monitor, the service interconnection graph and the telemeter output viewer share this property of being manageable by the user.

unmanageable element = addin

Not all actions triggered by plugins can propose an interface with which the user can manipulate the instantiated objects. For example, the "listen to audio stream" action only starts the playback of a current audio source in the gui. There is no particular interface associated with it and, at first, the solution was to create an empty component that stopped playback when it was closed. Another example is the one of the image viewer and region of interest viewer presented in figure 10.1. Both of them are injected in a same J2DViewer component and they cannot be managed by the user. This elements that are not manageable by the user, we call them "addins". This name comes from the fact that these elements are usually not standalone and are "added" to a container: the audio mixer, the J2DViewer, a 3D viewer, etc.

The "Addin Manager" shown in the lower left corner of figure 10.1 has now a clear functionality: it lists all running addins and makes it possible for the user to manage them. The main action that is done on addins through the manager is to close them but common action are also to deactivate and reactivate a particular addin. Addins can provide any actions they desire to the user through their context menu in the addin manager. Closing and deactivating/activating an addin are the only core actions that are defined for all addins.

integration into core modules

The concept of addins together with the addin manager have originally been designed to handle addins in a 3D visualization tool. In this tool, there is a 3D canvas that displays a smart environment with some graphical objects provided by different services: tracker targets, sensor positions, etc. Given the generic nature of the concept of addin and the reusable functionality provided by the addin manager, these modules have been added to the core gui.

OMiSCID Gui - Extension Mechanism

Extensions to the OMiSCID Gui are provided by writing modules for the Netbeans platform. To make the packaging transparent, it is recommended to use the Netbeans IDE or the Apache Maven (a project management and build system for Java available at [Url-m]). Extensions can be written by contributing to the OMiSCID service browser that is part of the core gui. Plugins mainly register to the service browser two kind of contributions, selectors and actions, that are used to build the contextual menu presented to the user.

extension points

We will take the example of the "manage connector" action that shows up in the context menu when the user selects a service connector. The presence of this action in the context menu is due to two extensions to the service browser that are contributed by two plugins from the core gui:

- an action that declares itself as being able to handle all tasks of type OmiscidConnectorTask, and that contains the code that opens the connector manager visual component,
- a selector that tells that when the service browser selection is made of a connector, then an OmiscidConnectorTask is available.

Using these two information sources, the service browser is able to display a contextual menu that completely depends on the current selection.

selectors, tasks, actions

A selector is basically an object implementing a particular programming interface: for a given selection and a given task type, a selector must return the tasks of this type that it can instantiate from the selection. A task has no particular API and is basically plain Java class, usually just an aggregation of properties. An OmiscidBrowserAction is just a plain Swing action with an additional method listing all the types of tasks that this action is able to handle.

improved reusability

A simpler solution would have been to use only actions: an action would be executable and either enabled or not (based on the current selection in the service browser). The main motivation of our complex solution is to be able to reuse existing actions without having to copy and modify them. If the actions were written to directly look at the service browser selection, it would be impossible to easily extend the range of operation of the action. We can take the example of a video stream viewer action: it is designed to be run on a service having a connector named "rawVideoStream". If we want to use it on a new service that outputs an image stream, for example an automatic cameraman service, but on a different connector says "montageVideoOutput", we would have to modify the action to accept this new output. With the separation of selector and action, we only have to write a new plugin that adds a selector that handles this "montageVideoOutput" case.

sharing around an "audio" task

Another example of the separation of selector from action could be taken with the AudioStreamTask. This task contains the necessary format informations together with the reference to a service and its connector. An AudioStreamTask can be produced using a microphone

service, a radio service or even a diapason service. An AudioStreamTask can be used by a player, an audio signal visualizer or a writer that write `.wav` files. With this simple example we already illustrate that instead of writing 9 highly redundant actions, we only 3 selectors and 3 actions. Also, adding new selectors and actions does not involve any code rewrite or cut and paste.

Concluding Remarks

Offering tools around a software method is fundamental for the method to be adopted. In the context of SOA, developers require a way to visualize and manipulate their services during their conception. We implemented a graphical user interface for this purpose and insisted on some requirements specific to SOA. Specific services require specific interfaces and thus the graphical user interface must be extensible and easy sharing of extensions must be easy. The graphical user interface has been successfully used and extended by members of the team since its initial release.

5.4 Communication On SOA

In previous sections, we presented how we designed OMiSCID and its extensible graphical user interface. Having a useable solution for the development of service oriented computing for intelligent environments, we decided that it was important and necessary to diffuse this software and make it visible and attractive. This section present the communication efforts that we did around service oriented computing and more specifically around OMiSCID.

We believe that the use of service oriented principles in the design of dynamic software system is a key point. With OMiSCID, we propose a middleware that is easy to use for non software engineering specialists. Our main objective is to favor the use of good software development practice by various developers. Toward this objective, we want our middleware to be used as widely as possible.

Licensing and Release

One non-technical aspect that often is the first criteria when choosing what software library or method will be used in a project is the license. The choice of an open source license was a requirement to make the distribution process easier. Closed source libraries and software pose a problem of maintenance and durability: there is no guaranty for the client that the software will still be available some years later. Given the current offer of open source software and the risk of depending on closed source software, an important proportion of the software developers, organizations and companies tend to prefer open source software for their infrastructure (operating systems, databases, etc.).

Another aspect about the license is that, even in the open source landscape, there are two main kind of alternatives: copyleft licenses and permissive licenses. The probably most widely known copyleft license is the GNU General Public License (GNU GPL). The principle of copyleft licenses is that anyone can reproduce, adapt or distribute the software as long as all the modifications and redistributions are released using the same copyleft licensing scheme. The idea behind copyleft licenses is to produce a maximum quantity of free software by inciting people to share their contributions (if they want to reuse any contribution from someone else).

Copyleft license have a great success and the current outstanding status of open source software is in part due to this licensing scheme. In some cases however, copyleft licensed software can be a blocker to the adoption of a software. A simple example is the one of a company that wants to reuse such software but want to keep its added-value code private. Copyleft licenses

prevent "pillaging" by unscrupulous companies but unfortunately, at the same time, it denies interoperability with all non open source companies. Taking the example of our service oriented middleware, a company that would like to expose some of its private value-added functionality as a service would not be able to do so with a copyleft license (and thus would not do so).

MIT License

Given our alternatives and arguments about licensing, we decided to do an official release of OMiSCID and its tools. The license we chose is a derivative of the MIT License which has the following properties: it is open source, it allows everything (non copyleft), it is short (under 200 words) and easy to understand compared to some other licenses.

make installation easy

We release OMiSCID in the form that makes it the simplest to use depending on the platform. The Java version comes with its dependencies to minimize the installation time. The C++ version follows the policy of different platforms. For examples, it is bundled as sources to create statically linked executables under Windows. For Linux, we provide sources to build OMiSCID dynamic libraries and install them together with the include files. We also provide packages that can be used with both debian and ubuntu based distributions.

Publications, Presentations and Diffusion

academic publication

We described OMiSCID in the academic domain by describing its objectives and features in the workshop on Services Integration in Pervasive Environments held during the IEEE International Conference on Pervasive Services in [Emonet 2006]. This presentation was done during the early days of the new OMiSCID API and we had a basic ontology based description of services that has been deprecated since then. Chapters 4 and 6 to 8 of the present manuscript can be considered as a deeper analysis of the requirements at this level of semantic description and a proposal to fulfil these requirements.

presentations

We also presented, in a less formal way, the OMiSCID middleware, the OMiSCID Gui and, depending on the audience, how to extend it. These were presented first in our research team but also to some other groups either involved in some collaborative research projects and/or working in computer perception and activity recognition. Most of these people see an interest in OMiSCID and it has been adopted in our collaborative projects anywhere it can apply. Some external research groups are thinking of switching to OMiSCID for their developments. As these groups want to do it all-in-once and "force" everyone to do the switch at the same time, this switch is currently delayed.

on the world wide web

To support our communication effort and give some visibility to OMiSCID, we set up a website to hold information, download files and documentation for OMiSCID. The website provide the necessary information for anyone who wants to learn about OMiSCID: installation documentation, tutorials, API documentations, download section, ... A team member contributed the visually attractive web design that is used at the time of writing. This website, together with a unique name, makes OMiSCID easy to find on the web. During the time of our work, even Google taught itself that this is a new word and not a typo.

Chapter 6

Concept and Method for the Design of Open Dynamic Systems

This chapter can be seen as following chapter 4 and proposing solutions to the problems exhibited. We propose design concepts that are easy to manipulate and we illustrate their use in the redesign of existing systems.

6.1 Motivation and Contribution Overview

requires chapter 4

This chapter build upon the analysis done in chapter 4. Reading chapter 4 is a requirement for easily understanding this chapter. In particular, this chapter supposes that the reader section 4.3 describing an existing 3D tracking system.

problems beyond SOA

In chapter 4, we identified some requirements to make system parts more interoperable and more open to dynamic evolution. One major step forward and an enabler for other advances, is the use of Service Oriented Architectures (SOA). Our contributions toward adoption of SOA have been presented in chapter 5. In this chapter, we build on the use of SOA and propose methods to solve other problems mentioned in 4 and particularly in section 4.4.3. We can recall these problems here:

- there is a poor separation of concerns and responsibility allocation in existing systems,
- some deployment steps are unnecessarily repetitive, particularly during system conception,
- communicating with another service requires a service to know, at design time, the exact interface and data model of the other,
- higher level processes need a way to abstract the constant evolutions happening at the perception level.

stagnation of higher levels

All these problems complicate deployment of intelligent environments by limiting the sharing and reuse of existing components and thus complicate integration.

service factories

To simplify the deployment and remove the duplication of information that often leads to incoherences in deployment, we will introduce the concept of service factories. Service factories are services that can instantiate other services on demand, given a specific specification for a service configuration. We will also describe efforts to formulate our problem as a distributed system deployment problem and examine the limitations this formulation can bring.

semantic service functionality description

An important part of this chapter consists in the introduction of the concept of service functionality to add a layer of semantic description of service functionality to OMiSCID (or other) services. The basic idea is that services will not depend directly on each other and on

their protocols but rather will use some functionalities. These semantic functionalities can be aligned to bridge and make interoperable concepts from different contributors. Applying this principle of semantic description of functionalities to factories opens interesting possibilities: expressing factories in term of functionalities makes it also possible to describe abstract composition patterns. The introduction of this concept of functionality will be illustrated through the redesign of existing systems, mostly the 3D tracking system.

proposed design method

In the rest of this chapter, we propose an overall method for the design (or redesign) of systems. We use what can be called a bottom-up-down approach to this presentation: we progressively lead the reader to the method by using the tracking system as an example. After having formalized the method, we go back down to its concrete application on another example: an automatic seminar recording system.

6.2 Simplifying Deployment: Service Factories

6.2.1 Introduction to Service Factories

restarting services is tedious

To illustrate the problem of deployment, we can first sketch a quick view of the 3D tracking system at the time we started rearchitecturing it. There is a 3D Tracker service and one detector-estimator service per camera (see section 4.3). The use of OMiSCID here makes it easy for the tracker to find and connect to the detector-estimator services. Nonetheless, two important problems remain: when started, the 3D tracker is configured with an exact description of the detector-estimators it will use. Each time the tracking system is to be restarted, for example during its development, the safest solution is to stop and restart all detector-estimators and the 3D tracker service. This solution is the one used in practice. Restarting everything ensures that configuration files, usually passed as command line arguments to the detector-estimators, are properly updated and thus fully synchronized between the 3D tracker and its detector-estimators.

duplicated configuration

The need for restarting four or five services each time on multiple computers is tedious, but it is only due to the dependency between the configuration of the 3D tracker and the ones of its detector-estimators. Part of this configuration dependency comes from improper allocation of responsibilities. Both the 3D tracker and a detector-estimator require camera calibration. A *camera calibration* is a change of reference frame from the 3D space to the camera image 2D space. Other dependencies are better justified given the used architecture. For example, the configuration of what image processing algorithm is used by the detector can influence how the 3D tracker should interpret its result. In such a case, it is justified that both the detector-estimator and the 3D tracker have information about the configuration.

introducing service factories

To remove the duplication of configurations and the restart of all services, we introduce the concept of service factories. A service factory is a service that is capable of instantiating other services given a configuration. In our example, instead of having to manually restart each detector-estimator to maintain the coherency between configurations, we propose to have a factory deployed for each camera. When starting, the 3D tracker sends to each factory the configuration of the corresponding detector-estimator, the factory instantiates the detector-estimator service and sends back a reference to it (its service identifier). The 3D tracker can then use the detector-estimator as in the old scenario and when it disconnects from it, the detector-estimator stops automatically. Using factories as proposed, the configuration is centralized in the 3D tracker and is spread to the detector-estimator services that are automatically created each time the 3D tracker is started.

factories benefits

We believe that the usage of service factories can greatly reduce the repetitive deployment task when designing a system. This benefit also remains "in production" as it minimizes the number of services to deploy manually. Factories can be reused both in variety and in time:

- in variety, two systems can use the same factory to instantiate their detector-estimators;
- in time, for example to redeploy a system like a 3D tracking system with an updated configuration.

more in this chapter

Factories have been quickly presented here as a solution to simplify deployment. Factories are not limited to starting simple services: for example, they can be used to represent composition patterns. This concept of factories will be used and detailed in the rest of this chapter: applying factories on concrete examples illustrates their potential and their role in the design method we propose.

6.2.2 An Excursion into Deployment Frameworks

generic factories

We presented a factory that can instantiate only detector-estimators but it could be tempting to have a fully generic factory. An all-in-one factory could be deployed on each computer, executing any arbitrary command, even administration commands (to install or update applications and services). Apart from the security issues that arise from this approach, we think it looses many important aspects of the service factory approach has:

- localizing a sensor (e.g. a camera) is not possible anymore, in our case the service factory is a representer of this sensor
- all the abstraction is lost and we are back to the "ssh/rlogin" deployment method where the client knows every details about the application.

Some approaches use this kind of generic factories and add some description of computer capabilities to handle heterogeneity of computers in a grid as in [Flissi 2008].

deployment for the 3D tracker

In a collaboration, we experimented the interaction of these approaches with our service oriented approach. The collaboration consisted in using a deployment framework based on the Java Agent DEvelopment (JADE [Bouchenak 2006]) framework to automatically deploy the 3D tracking system. Once deployed, experiments with the dynamic reconfiguration of the deployed system were conducted. The experiments proved to be conclusive and the 3D tracking system was properly deployed using JADE and simple reconfiguration cases were put in practice.

deployment descriptor

For this successful deployment, OMiSCID services had to be wrapped as deployable elements. In addition to this wrapping, the only requirement is to deploy the JADE runtime on each computer (like a generic factory would be). The most important piece of the automatic deployment is a deployment descriptor that tells where to start what system parts. This descriptor contained all the configuration that were then passed to the service executables as they were started.

redundancy between deployment and SOA

A deployment based on a descriptor like this one, makes it necessary to write a new descriptor to properly execute in totally new conditions. Knowledge such as presence of a camera on a particular computer has to be expressed: this information is here in the deployment descriptor while service oriented methods recommend the declaration of a service. Having such a deployment is redundant with the approach of using SOA. As an additional illustration of this redundancy, it would be possible, using the deployment framework to specify communication ports for the detector-estimators and to automatically pass these ports to the 3D tracker.

better dynamicity handling by SOA

Even if we can make them work together, using both a SOA and a sophisticated deployment framework is not a real synergy. We prefer a service oriented approach that handles dynamicity in a better way in our opinion. Obviously, service oriented solutions are more complicated to deploy than a dedicated deployment framework. That is why we propose the concept of specialized factory as a tool to simplify repetitive deployment. As we will present in the rest of this chapter, factories have an important potential and can be used as a support for service composition.

6.3 Reallocating Responsibilities in Tracking

6.3.1 Overview of Responsibilities in 3D Tracking

duplicated knowledge

Starting from the 3D tracking system presented in section 4.3, we separated it in multiple services with a better separation of concerns. At this time the tracking system was just split in a single 3D tracker service and one detector-estimator service per camera device. Information was duplicated, detector-estimators were combining at least 2 functionalities and algorithmic effectiveness was even penalized by these choices.

identifying responsibilities

We first try to identify the different responsibilities present within the system. Most responsibilities can be classified either as knowledge holding or as processing. The distinction between these two kind of responsibilities is most of the times easy to do but it can still get complicated with lower level systems. We can consider that a knowledge responsibility consists in providing data or holding information used by other elements in the system. A processing responsibility aims at transforming data in the system.

separating concerns

The 3D tracking system can be "exploded" by isolating the independent responsibilities. At a coarse grain we identify the following knowledge in the system: the camera calibration information, the list of cameras that are usable by the 3D tracker, the captured images and the background image used by each camera for detection based on foreground/background image difference. On the processing side, we find the tracking in the 3D space, the conversion between a 3D space and the 2D spaces of each camera, and finally the detection based on 2D image processing. We could also consider the image grabbing as a processing: doing acquisition from camera device often involves notable processing involving image decompression or color conversions.

resulting architecture

We propose a new architecture and organization of the 3D tracking system. This redesign is based on the identified responsibilities and is expressed in term of services. Figure 6.1 pictures an example of these services in action. Obtaining the following system state requires the 3D tracker to invoke factories to create product services. We will detail what is the role and responsibilities attached to each service involved in the system in following sections.

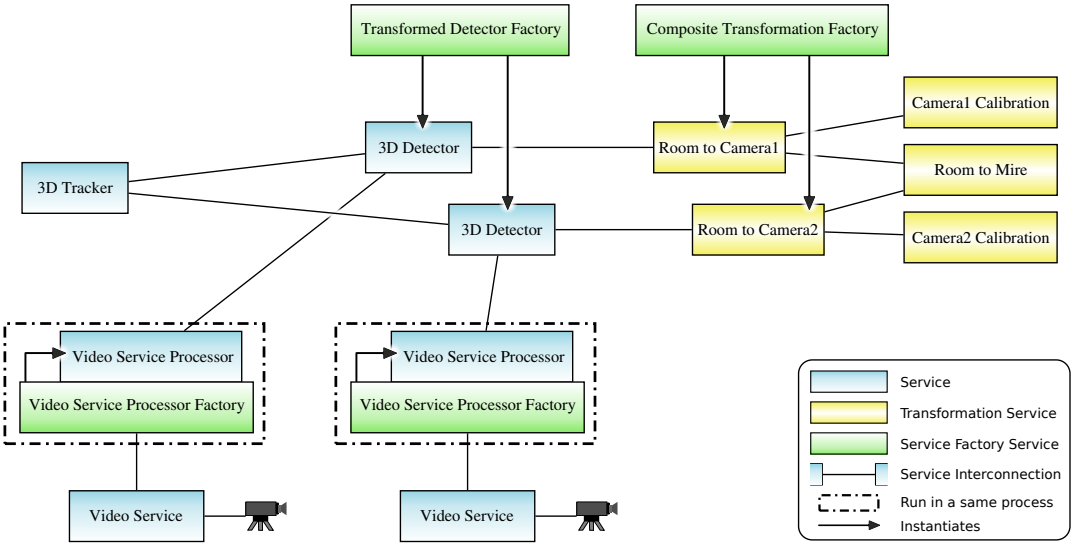


Figure 6.1: *Split 3D Tracking System. The 3D tracker is separated from its 3D detector components. A 3D detector is obtained by the composition of a camera calibration with a 2D detector. Dynamic addition of calibrations and cameras is made possible.*

6.3.2 Architecture at the 2D Image Processing Level

splitting the detector-estimator

The first action was to remove the all-in-one detector-estimator and create a 2D detector. Each 2D detector is only responsible for the image processing tasks. Given a set of points or a region of interest, a 2D detector is responsible for evaluating the probability of object presence for each of these points or pixels in the region. The detection is done purely in 2D and is based on the observations constituted by the images captured from the camera. A detector does not have any knowledge of camera calibration nor it requires it to derive any higher, more elaborated, information than the simple probability of object presence.

factory to handle configuration

To handle the configuration aspect of the detector, we use a factory to produce detectors. We generalize the concepts and see image based detectors as a particular video processor. We introduce a factory that generates video processor from a particular configuration. The configuration consists in the definition of a chain of image processing modules. For the case of a simple detector, a background image subtraction module is used and the result is output using OMiSCID. We replace the need for a reference background image by using an adaptive background model whose parameters are passed to the factory within the processor configuration.

performance optimization

Having introduced a factory makes it possible for any other client to use a detector at the same time as our tracking system. Even if both the factory and its products are exposed as services, they run in a same process to maximize performances. From the outside, this decision is not visible: one cannot tell that two services are in a single process.

camera service

It is possible not only to have multiple image processor operating on the same image at the same time, but also multiple arbitrary clients using the images captured from the camera. The camera service provides this functionality. The processors factory will connect to this camera service to obtain images and at the same time any other client can still access these images. This can be useful for a visualizer showing both the images captured from the camera and some information provided by the 3D tracker as shown in figure 10.1.

alternative split

Another solution would have been to merge the video service and the processors factory to prevent image transfer between these two processes. We have not done so to preserve the independence between image processing and image acquisition. Our solution decouples the image source from the processor and thus makes it easier to handle new sources such as videos read from a streaming source or videos generated using image synthesis. Each solution has its advantages and can be equally used. One constraint must be respected: the camera device must be exposed as a service even if it is embedded in the processors factory process. Exposing the camera is capital to properly reflect the capability of the environment to any potential service consumer.

still expose capabilities

6.3.3 Architecture at the 3D Tracking Level

separate tracker from detectors

At the 3D level, we split the system in two main processing entities. A 3D tracker is responsible for holding and updating the list of current targets and detection regions. It predicts the future possible position of targets based on their past evolution and delegates the detection to some 3D detectors. One 3D detector is present for each camera and is responsible for projecting the 3D region of interest (or a set of points) and triggering the actual detection to the 2D detectors.

from 3D to 2D detectors

These 3D detectors are created by a factory that composes a camera calibration with a corresponding 2D detector. The factory receives a change of reference frame in the form of a 4x3 matrix projecting points from the 3D space to the 2D reference frame of the image. A composite 3D detector basically uses the matrix to project the 3D space query (a list of 3D points or a 3D region of interest) into the 2D image. The projected query is transmitted to the 2D detector and the reply is transformed back to 3D by the composite 3D detector.

dynamic configuration

3D tracking occurs in a particular 3D reference frame. The tracking may for example be done in an orthonormal reference frame having its origin in a corner of the room. At the 3D level, detections are expressed in this reference frame. To convert coordinates from this 3D reference frame to the 2D reference frames of the images, transformations are necessary. This information must be provided dynamically to the system to allow addition of new cameras at runtime.

calibration procedure

From a practical point of view, a calibration procedure provides a transformation from a reference frame attached to an object in the room Ro to the reference frames of each camera image being calibrated during the procedure. A calibration procedure involving two cameras present in the same room would provide 2 transformations: Ro↦Ri and Ro↦Rj, Ri and Rj being the the reference frame of the images. Another calibration procedure may involve other cameras and a different calibration object and would for example provide Rc↦Rk. The knowledge of a transformation from the tracking reference frame (Rr) to the various objects used for calibration (Ro, Rk) is necessary to allow a camera to be used for tracking. Such transformation are in our example Rr↦Ro and Rr↦Rc where Rr is the tracking reference frame (attached to the room).

dynamic calibration manipulation

Different transformations can be provided dynamically, independently and by different means. Calibration procedure may involve giving manually the position of the calibration object relatively to the room. This relative position can also be evaluated using computer vision to perceive the room walls, etc. The overall tracking system must be able to use any camera for which a transformation from the tracking space to the camera space exists. Information dynamically provided by different means must be composed to obtain the required reference frame. For example, to obtain Rr↦Rk, Rr↦Rc and Rc↦Rk must be composed.

factory for calibration combination

We have a factory that does the composition of transformations: it basically accepts two transformations and provides the composition of transformations. At this step, we expose transformations as dedicated services. The factory will take two such services and produce a third one. As soon as one of the two services disappears, the product is also stopped.

improved reusability

With the new separation in services, the tracking system is more adaptable and its parts can be reused. A designer can now easily reuse each subpart of our system: previously limited to using the output of the 3D tracking system, he can now reuse intermediate functionalities such as images from cameras or detections in the image. One can also extend the 3D tracking system by writing new services with the same interface and communication protocol such as a new 3D detector.

6.4 Reasoning in Term of Functionalities

In previous section, we split the system in multiple services with well identified independent responsibilities. In this section, we generalize the functionalities provided by services and raise a little the level of abstraction.

6.4.1 Service Functionalities for Reasoning-Based Integration

design time reuse

Having split the system in well chosen services increases the number of entry points for reuse and extension. All this reuse and integration is limited to services designed after the 3D tracking system: services must be designed to be integrated with the 3D tracking system. We are convinced that real intelligent environments require spontaneous integration of services not especially designed to work together.

enabling spontaneous interaction...

To allow spontaneous integration of services designed in total ignorance of each others, we add a semantic description of the functionality provided by the services in the 3D tracking system. Services describe their functionality using abstract terms. These terms are put into

correspondence to make runtime integration possible. We also describe the capabilities of service factories using these functionality level descriptions. All this information is present at runtime and can be dynamically enriched to enable integration. Reasoning about functionalities opens interesting combinations: functionalities can be provided directly by services, indirectly from functionality correspondences or by potential calls to service factories.

... with functionality descriptions

Once a functionality is found to be useable by a service, a possible protocol adaption can be required. For the initial conception of the system, no adaptation is required as the different elements are created together.

protocol adaptation

We propose to add a level of semantic description of the functionalities provided by the services. The architecture of the 3D tracking system with functionalities is depicted in figure 6.2. This diagram shows the system once it is running and this state is reached by calling factories to create different services. On the contrary to the case where functionality descriptions were absent (section 6.3), all calls to factories can be automatically inferred from the descriptions present. We will now detail semantic descriptions of service functionalities involved in this architecture.

tracker with functionality descriptions

6.4.2 Introducing Abstract Functionalities

The first kind of semantic functionality description that is present in figure 6.2 is the simple labelling of services with a simple functionality. Descriptions of this kind are the core of our approach to functionality description. For the tracking system, we use mainly three kind of functionalities with which we decorate our services:

functionality descriptions...

- Noted "Detector(Ref)", the functionality represents a detector in a particular reference frame. A detector is responsible for affecting a probability of object presence to any point, expressed in the "Ref" reference frame, that is passed to it. This functionality is used both for 2D and 3D detectors as they conceptually provide the same functionality but in different spaces.
- Noted "Tr(A↦B)", the functionality represents a transformation from A to B. Such a transformation holds the knowledge of how to convert a point expressed in the frame reference A to a point expressed in the frame of reference B. As with detectors, this functionality is used for both 3D↦3D transformations and 3D↦2D projections.
- Noted "ImageSource(Ref)", the functionality represents an image producer and attaches it to a particular reference. Our video services that grab images from the video device are such image producers but other image sources can be used as a replacement: synthetic images producers, recorded video sequences, etc.

As stated before, describing service functionalities in more generic terms and in abstraction of the concrete implementation (communication protocol) makes it possible to dynamically accept new services that provide a required functionality. Services use each others through this functionality layer and they do not depend directly on each other.

... to abstract concrete implementation

6.4.3 Functionality Correspondences and Factories

A second kind of knowledge is the correspondence between two types of functionalities. An example of such knowledge can be found in the architectural diagram (figure 6.2) on the left in the middle. We noted it as in "VSP(C, detectConf) $\ggg$ Detector(C)". This expression means that a VideoServiceProcessor (VSP) functionality having the particular configuration "detectConf" can be seen as a detector having the same reference frame as the processor. In this case, "C" is any arbitrary reference frame that is propagated from the VSP functionality to the Detector one. On the contrary, "detectConf" represents a particular configuration that should be used for a VSP to be considered as a Detector.

functionality correspondences

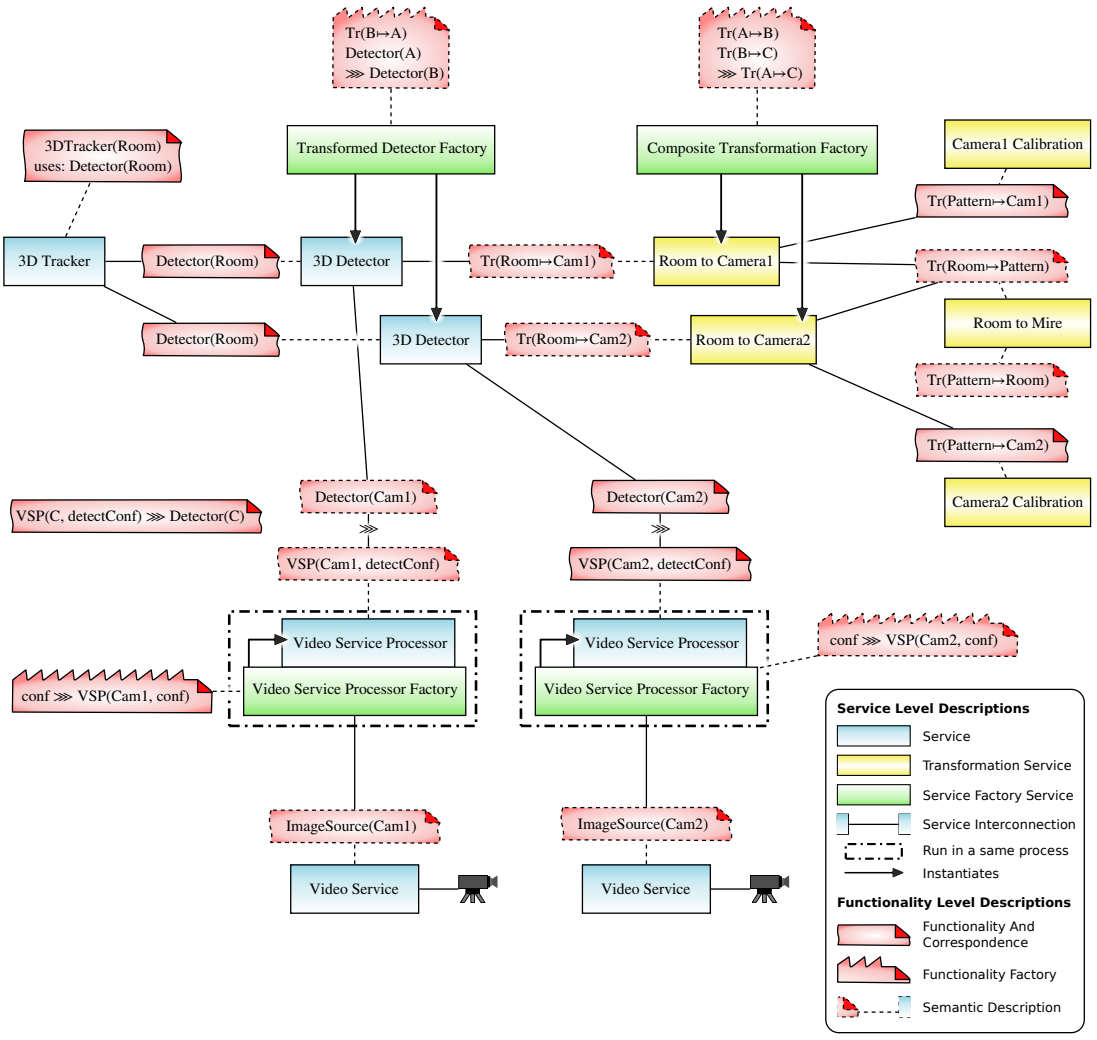


Figure 6.2: *Split 3D Tracking System With Functionality-Level Descriptions. Functionality descriptions has been added, in red, to the architecture from figure 6.1. Semantic level descriptions are used for normal services, for service factories and for functionality correspondences. Services now depend on functionalities rather than on other services.*

The "VSP(C, detectConf) ⋙ Detector(C)" knowledge is linked to nothing in the diagram. It can be exposed by any mean and is not describing a particular service. Expressing this knowledge can be done by the designer of the 3D tracker or by an integrator (human or automated). Basically, this knowledge tells which video service processor configuration is suitable to create a detector. This knowledge is necessary for the 3D tracking system to integrate the video service processor as a detector.

= integration knowledge

Functionality factories are the last kind of functionality-level descriptions used in the presented architecture. We have three kind of factories in this architecture, all of them being described in term of functionalities:

functionality factories

- The Composite Transformation Factory was originally taking two compatible transformation services and composing them into a new transformation. In the new version, it is expressed at the functionality-level to document this capability and allow reasoning about it. We note this capability as follows: "Tr(A↦B) Tr(B↦C) ⋙ Tr(A↦C)". The constraint on the compatibility of the two transformations to compose is implicitly expressed by the use of "B" in both "Tr(A↦B)" and "Tr(B↦C)".
- The second factory on the top of the diagram is described using the same principle and composes a transformation B↦A with a detector in A to produce a detector in B. The direction of the required transformation, B↦A, often looks wrong at the first sight. To produce a detector in B, we need to be able to associate a probability to any point in B. Receiving a set of points in B, we apply the transformation B↦A on them to obtain points in A. For each of these points, the detector in A returns a probability that we can affect to the original point in B.
- The last factories are the Video Service Processor factories. They are described in term of functionalities, using for example "conf ⋙ VSP(Cam1, conf)". This description is rather generic, telling that given any configuration, the factory is capable of creating a VSP functionality with this configuration. It also adds the information that the produced processor will be working in the Cam1 frame of reference.

The state previously presented in figure 6.2 is a picture taken of a running system. A part of the involved services and descriptions can in fact be produced automatically using other services and descriptions. Particularly, reasoning about what functionalities are instantiable by factories and what are the possible functionality correspondences opens many possibilities. Figure 10.2 shows the same architecture but we grayed out everything that can be derived from the core descriptions and services. These elements, not grayed out in the diagram, are the one that have to be created by the designer, everything else is a derived product of its design.

automatically derived elements

6.5 Step-By-Step Design Method

Using the concrete example of the 3D tracking system, we introduced the underlying concepts of the software design method we propose. This section presents this method more generally.

To bridge the gap between pervasive computing and intelligent environments, we propose a design method for systems. This method improves extensibility, interoperability and dynamicity of software systems. Our method favors the creation of systems with profitable characteristics:

method objectives

- systems are made of parts that can be dynamically reused by other systems,
- systems can use new parts added dynamically.

The method we propose can be applied to both the redesign of existing systems and the design of new systems. Redisigning existing systems can involve substantial work on the implementation as it tends to generate more services to improve reuseability and extensibility.

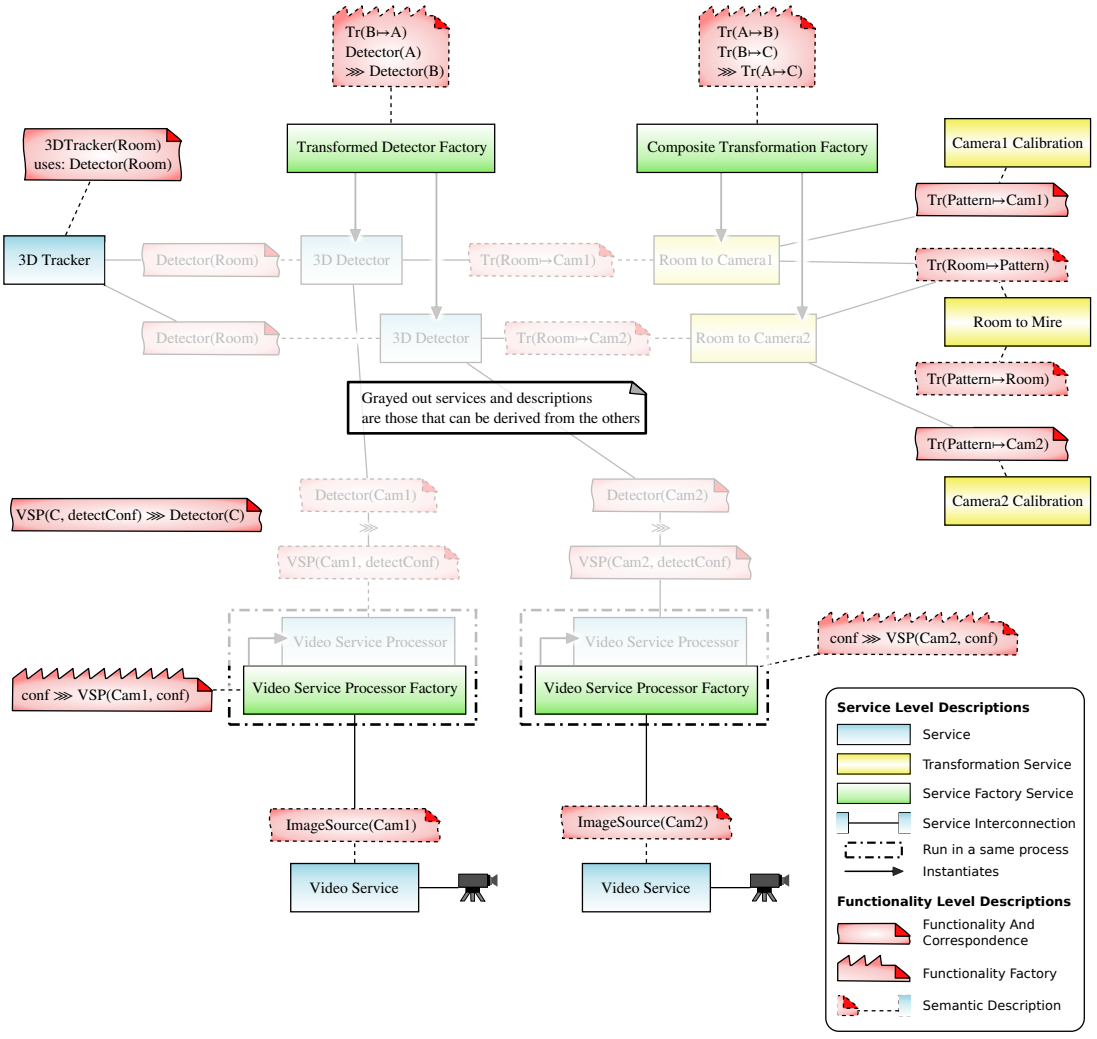


Figure 6.3: *Split 3D Tracking System: What Can be Derived. Same architecture as in figure 6.2. Lighter parts of the schema are all the elements that are automatically derived and instantiated. Darker elements are the ones provided by the various designers.*

Our method is composed of 6 steps. These steps are designed to be executed in sequence but some reordering and interleaving is possible: once the steps are well understood, they can be seen as best practices or guidelines. The method can be summarized as follows:

6 steps

- Identify responsibilities within the system
- Split system into parts
- Introduce factories where possible
- Abstract each part as a functionality
- Share the functionality
- Integrate

Identify responsibilities within the system

The first step to a proper separation of a system into parts is to identify various responsibilities that are often hidden in the original design or in our first conception of the system. These responsibilities are of two main types: information and processing. Another equivalent formulation would be to consider what are the data involved and what are the functions present in the system. Some examples of system parts holding an information responsibility are the camera calibration, and the camera itself as an image acquisition device. Concerning the processing responsibility, examples such as the image processor or the 3D part of the tracker are good examples.

information and processing

Identifying different information responsibilities in the system mainly consists in exhibiting what constitutes the input of our system and its parameters. This identification of parameters and inputs may get complicated: many of these are implicit, imposed by the context. For example in the 3D tracking system, this can be the case of the image source that can easily be forgotten as a source of information in the system. The complexity of involved methods and algorithms makes the presence of "magic" parameters quite common: some of the parameters are manually set, directly in the program source code, to a value that is appropriate. The appropriateness of the parameter may be total in a particular context but its embedding in the code forbids any adaptation to some different conditions.

implicit and "magic" parameters

A functional decomposition of the main functionality of our system makes it possible to identify processing responsibilities. An effort is often required to properly separate different aspects of the processing. Specialists tend to see their system as a whole and the first split that comes to their mind is the succession of steps to create their systems (e.g. select training set, do machine learning, start my service). Identifying different responsibilities involved in their system when it executes is an unnatural task for the specialists but it can become a habit with some practice.

a habit to acquire

Split system into parts

Having identified the different responsibilities involved in the system, the second step consist in splitting the system accordingly. As the first tendency is to build monolithic applications, a simple methodology could be to process as follows: first start the modeling with one independent system part for each identified responsibility and then merge these parts when required.

start with small services

Merging responsibilities should be done only to solve evident overdesign issues and to prevent obvious performance problems. In order not to fall in the trap of premature optimization, one must be very careful with the performance concern. As an leading example, we can take the example of the camera image acquisition and the image processing algorithm that composes

then merge

the detector. The idea of having two services with all images going from one to the other would be seen as a performance problem by most specialist: measuring the performance of such a solution and particularly when the services are on the same computer makes this solution fully acceptable and operational as presented in chapter 9.

performance is rarely a problem

Eventually, performance issues are rarely encountered and the main work while splitting the system, is to choose a proper level of granularity for services. We can efficiently transmit an image between two services but doing a functionality decomposition down to the pixel processor and having one service per pixel would clearly be an overdesign. By the way, it would also probably impact performance but the problem in the design is even more penalizing.

Introduce factories where possible

evaluate each part

Having split our system in multiple parts, we can evaluate the introduction of service factories at each level of responsibility. Introducing a factory where appropriate simplifies deployment and favors potential reuse. Service factories are not silver bullets and replacing all system parts by factories would not be a much help. When designing a system, one must take the time to evaluate, for every service in the system, whether using a service factory would be pertinent.

identify parameters of services

The first case a factory is often appropriate is when the service has some instantiation parameters. Care must be taken on the nature of the parameters. If parameters can arbitrarily be chosen by the designer and in a wide range of values, we recommend to use a service factory. When many different parameters can be chosen by the service creator, exposing a service factory gives the opportunity to others to reuse the factory with their own parameters. The fact that the set of plausible parameters is huge or infinite is a good indication of the benefit of using a service factory: it would be impossible to start all the possible services, the factory does it on demand.

avoid hardware specific parameters

Not all service instantiation parameters are indicating that the introduction of a service factory is a good design decision. Some instantiation parameters are reflecting some constraints of the execution environment and are have a role of configuration of the executable to its particular operating conditions. Such parameters cannot be arbitrarily chosen by the service creator, the service must have proper values to run. A perfect illustration of this case where factories are of no use is the case of an image grabber that does image acquisition for a camera device. An image grabber requires a path to the camera device on the system to run properly: we can obviously try to start a grabber with any parameters but the grabber will be eventually started only if a valid video device is passed to it. Independently from the presence of instantiation parameters, when a service represent a capability of the physical environment, it should be directly exposed and not provided by a service factory.

generic composition as factory

Another situation where factories are interesting to introduce is when a service is orchestrating or interfacing other services. When a service composes and enriches the functionality provided by one or more other services, it usually looks for the required services before declaring itself. In cases where this composition of services (or enrichment) is sufficiently generic, a factory can be used to represent the composition pattern. The factory will be provided with the service it has to enrich or compose and also with necessary additional parameters. For these kind of factories also, the decision to replace a single service by a factory must be done based on the potential reusability of the composition pattern. If the range of possible services to compose and composition parameters is sufficiently wide, a factory will improve reusability and spontaneous integration.

Abstract each part as a functionality

Following the previous steps, we have a system that properly allocates responsibilities to distinct software elements or services. To facilitate deployment and improve the virtual range of available services, factories are used where it is justified. The architecture we produced involves concrete services, for example OMiSCID services: they all have names, connectors and variables.

raise abstraction

This step consists in extracting from these concrete services the functionality that they implement. Choosing the proper functionality is a delicate exercise that must be guided by keeping the purpose of "functionalities" in mind. The definition of functionalities must be at a higher level than the definition of services. While a service is a concrete software elements, a functionality is an abstract and more general description of the capability of the service. This distinction can be seen as the distinction between an interface and a class in an object oriented modeling, when these concepts are properly used.

even for factories

The aim is to express the requirements of services using functionalities. A same functionality will be provided by multiple services and spontaneous interaction will become possible. We can add new services implementing a functionality in a novel way. Services that required this functionality will spontaneously, without particular modifications, be able to use the new service. To maximize the gain of expressing functionalities, factories should also be described in term of the functionalities they use and the functionality they can provide by instantiating their product.

review protocols

For both services and service factories, this addition of functionality based description comes in addition to the choices of previous steps. Service names stays the same, factories continue to operate in the same way, connectors remain untouched, etc. The only element which can be usefully reviewed at this step is the communication protocol used by services. The first conception of a communication protocol is often highly specific and having taken the time to reason about the more abstract functionality provided by a service can help in writing more generic and often simpler communication protocols.

Share the functionality

consider your application as a service

The idea of this step is simple: do not forget to do the same as you did for each subpart of your system but with the final functionality you are implementing. Even if you first imagine your final product as a final application, we recommend to consider the overall system as a potential service. If the system has the potential to provide a functionality for another application, it must be exposed as a service and described in term of the functionalities it provides.

or as a factory

Pushing this vision forward, the designer must consider whether a factory would be adequate to instantiate this service. If a factory is introduced, the final application will be separated from the core service and will ask the factory to create the service when needed. This implies the reusability provided by factories but also allows our application to work with another service provider than this particular service factory.

ease access to your functionality

A last aspect of sharing concerns the source code. Even if the whole code may not be candidate for an open source release, it is interesting to share any tool that facilitates the access to the designed services. Having exposed a functionality as a service makes it easy to use, providing a simple library to access the service makes it trivial. We recommend service creators to extract from their code base and openly provide any tools that simplify the access to their services. A simple example can be the one of the video service: it outputs images using both a particular lossless encoding and a jpeg encoding. Receiving and interpreting images requires some code that the service designer obviously writes to test its service, at least in one language. Cleaning, documenting and providing this image interpretation code makes the

reception of images trivial for the service consumer that does not have to know anything about the specific encoding: he can concentrate on important features of the images such as its capture time, its dimensions and content.

Integrate

integrate with others...

The last step in our design method consists in integrating the newly designed system with other existing systems. With properly designed services, the integration can profit to both systems. Parts of the newly designed system can provide functionalities required by another system and can also reuse parts from existing systems.

... at the functionality level

We recommend to do this integration at the functional level. Modifying a service or its communication protocol can still be a viable solution at the early stage of development. More generally, the integration must be done by expressing two categories of mapping: some correspondences between functionality types and some communication protocol adapters.

integrate functionality and protocols...

Functionality correspondence and protocol adaption are two distinct aspects. If the functionalities have been defined properly in the associated step of our method, they should be placed at conceptual level. Functionalities should be totally independent from the concrete communication protocol used. Given this independence, it is natural to express independently functionality correspondence and protocol adaptation.

... independently

While functionality correspondence is relatively easy to describe but hard to detect automatically, protocol adaptation description can be tedious and verbose but can be done automatically. The separation between functionalities and protocols is important not to limit the designer in expressing functionality correspondence. When a functionality is found to be matching a request but no adequate protocol adapter is present, the system can notify the designer or log the fact that a protocol adapter was needed but not found. Protocol adapters can then be written only when necessary.

6.6 Application to an Automatic Video Composition System

starting at step 3

To illustrate our design method, we will apply it to PRIMA's automatic cameraman. This system makes a live montage of a seminar to produce seminar recording that captures the important events for a seminar. It has been effectively used to record seminars and we will work on its architecture to make it more open to dynamic evolutions. The architecture of this system already features a good separation of responsibilities. As a result, the presentation of the current system implementation covers almost the entire two first steps of our method: the identification of responsibilities and the split of the system.

cameraman principle

The central service composing the automatic cameraman is the Meeting Director service. Internally, the meeting director holds the expertise about how to choose the best image and audio source based on contextual informations. The meeting director uses a rule system to implement activity recognition based on the information perceived by other services. Based on the perceived situation, the meeting director selects the best video source among three: a video of the speaker, a video of the audience or a capture of the current slide (more precisely of the image projected by the speaker).

existing architecture

The running system is depicted in figure 6.4 that presents the Meeting Director service with all services involved in its operation. To improve readability, we laid out the services in 3 layers. We will present the responsibilities of the services composing the system.

responsibilities

The meeting director is responsible for the orchestration of lower level services. The beginning of the recording and its end are controlled by a graphical user interface that exposes a

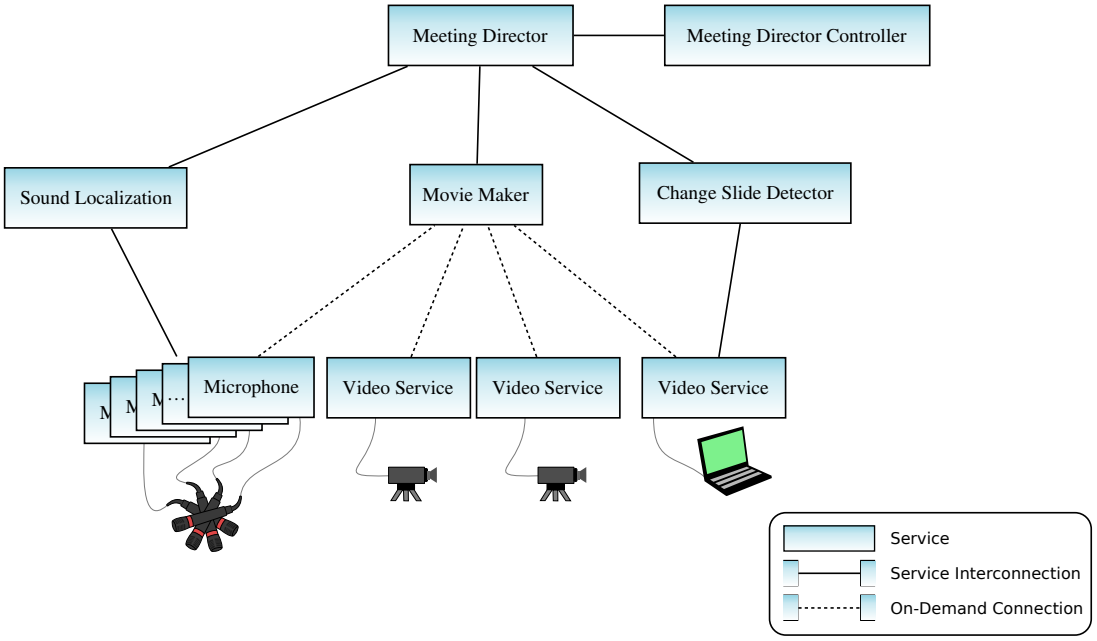


Figure 6.4: *Architecture of the automatic cameraman before its redesign. The cameraman is not monolithic and uses a layered, service based architecture.*

Meeting Director Controller service. The director connects to the controller and waits for its events to start and stop the recording. The controller has a simple interface and can be easily replaced by some other services. The user interface can be replaced by a remote control or a simple switch and the director controller can even be automated: for example, a service can automatically detect the bluetooth phone of the speaker and send start and stop events when the speaker enters or leaves the room.

visual input

The first input used by the meeting director to recognize current activity is a slide change detector service: it is responsible for telling when the speaker changes slide. Slide change detection is based on simple image processing that tries to identify important changes in the screen image. Screen images are continuously grabbed from the speakers computer by a video service. Depending on the seminar, a full software screen grabber or a hardware VGA recorder can be used. Both solutions expose as a video service and the slide change detector can use these images to detect slide changes, appearance of new items in a slide or animations.

audio input

The other input used by the meeting director for activity recognition is an audio localization system. This localization system uses a set of microphones to detect speech activity and evaluate the direction of this activity. The localization is provided as an angle around the antenna of microphones: the system covers the full 360° range of directions. This range can be restricted, for example to 180°, when the antenna is located against a wall or an obstacle.

role of the director

The director holds the knowledge of the relative position of the speaker and the audience with regards to the microphone antenna. It also knows which service video can be used to record the speaker, the audience or the current slides. Using its model of the current activity and this knowledge, the director pilots a service dedicated to the video montage and the addition of "special effects".

previous microphone selection

The architecture presented here and in figure 6.4 already results from a slight change in the structure of the system. Originally, the choice of the camera to record was done by the meeting director but the choice of the microphone was done by a dedicated service called an audio router. The meeting director had no control over the choice of the microphone and the coherence of

the recording was conditioned by the fact that the meeting director used audio information to select the video source. This choice was historical and motivated by performance and separation of concerns: audio acquisition and processing for all microphones were performed by a single service.

policy based microphone selection

To provide the meeting director with a choice of what to record but still keep a smart adaptive microphone selection, we propose to integrate the functionality of the initial existing audio router service in the movie maker itself. The meeting director then uses policies to express high level queries of what sound the movie maker should record. An example policy could be: "record the two microphones with best signal to noise ratio among microphones 3 to 6". Another solution would be to keep the audio router service and have it controlled by the meeting director using policies.

video montage

The audio video montage is provided by a Movie Maker service. This service generates an audio video media file. Based on the order it receives from the director, the movie maker combines the proper audio and video sources. The movie maker handles the connections to video and microphone services.

startup configuration

The presented architecture of the automatic meeting recorder has the advantages of respecting a proper separation of concerns. At this point however, the overall system must be set up for the specific environment in which it will operate. This configuration consists in starting the proper services and setting up a configuration file for the meeting director. The association of camera identifiers to particular roles (speaker, audience, slides) is part of such configuration. This configuration is necessary and doing it in a configuration file passed at the startup of the meeting recorder is better than having it hard coded in the meeting director itself. The same kind of information should be provided concerning the microphone: what angle provided by the audio localization system correspond to the speaker or the audience.

lack of dynamicity

From an architectural point of view, the recording system was not monolithic but still suffered from a lack of dynamicity: the meeting recorder expected particular services and is fully configured at startup time. The architectural style was based on distributed components than on services. This component orientation limited the dynamicity and the possible spontaneous interaction of the recording system with other elements. Following our design method, the next steps are those which effectively bring dynamicity and service orientation to existing systems. These next steps tell us to evaluate the possibility of using service factories in our system and to use abstract functionalities to describe both our services and their requirements.

redesigned cameraman

The whole running system with services, factories and semantic functionality descriptions is depicted in figure 6.5. As with the 3D tracking system, semantic descriptions tend to replace configurations and an important part of the descriptions and the services can be derived from other descriptions and from the factories. Figure 6.6 underlines what needs to be put into the system and what can be automatically derived.

abstracting: director

At the upper level, we abstracted the requirements of our meeting director services as functionalities. The meeting director requires four main functionalities: a speech detector for the speaker, and one for the audience, a slide detector notifying of slide activity and a movie maker dedicated to the meeting. A meeting controller is also required but no real abstraction is added: we just bring the service into the functionality domain for consistency. These abstract requirements are indirectly implemented by low level services: the latter exposes their raw functionality. Some integration descriptions are added to do the mapping between these functionalities and the requirements of the meeting director. Most functionalities are scoped by the location of interest, here "Room", or by an identifier associated to the current meeting recording "Meeting1234". This precaution allows multiple systems to run at the same time without interference.

abstracting: slide change detector

The slide detector, on the right part of the figure, can be integrated within the architecture involving the video service processors presented with the 3D tracking system. The meeting

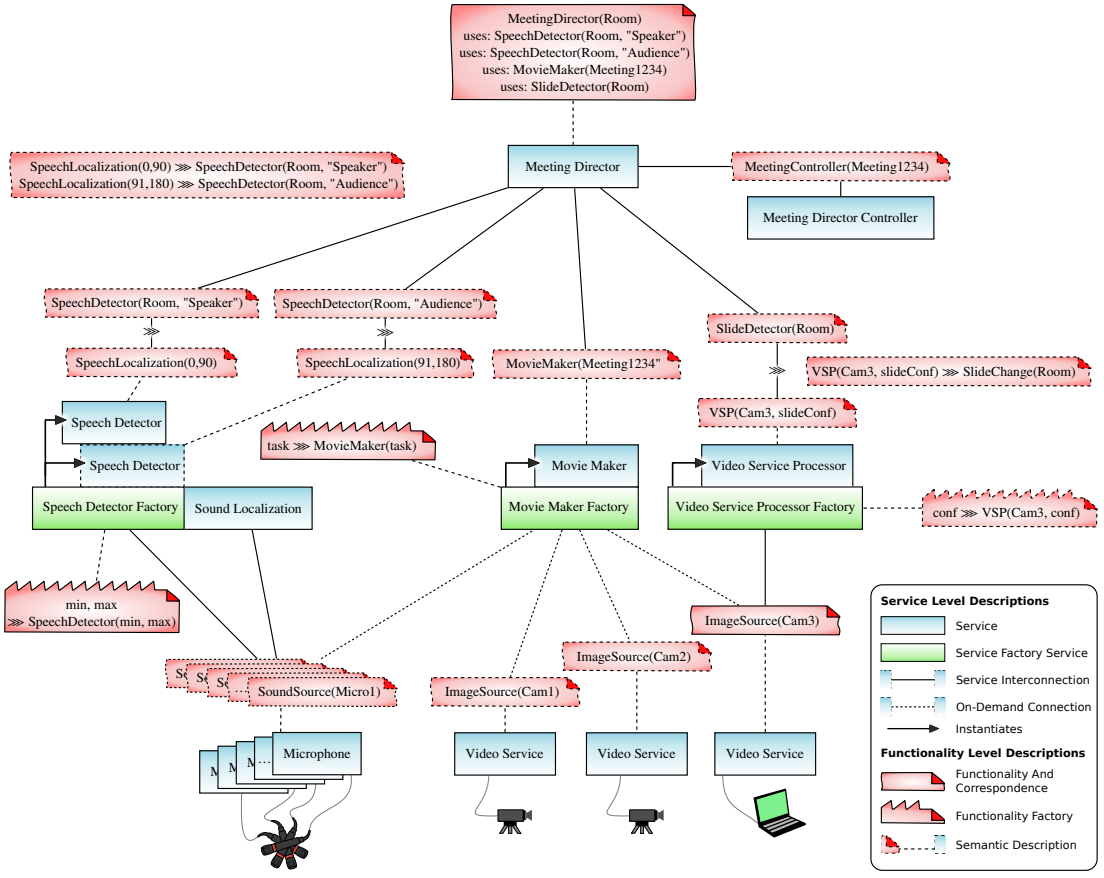


Figure 6.5: *Automatic Cameraman with Functionalities and Factories. Result of the application of our design method on the cameraman from figure 6.4. Factories have been introduced, in green. Functionality level descriptions have been added to all elements, in red.*

director can use a video service processor factory started on the video service recording the presenter screen. With the proper configuration, a video service processor can be started by the factory and can detect slide changes. An example configuration is simply to do motion detection on the video stream and if motion energy is above a threshold, emit a slide change event. The knowledge of this configuration was already present in the initial system in a completely ad hoc form: a dedicated service has been written and hard coded with this configuration. Using semantic functionalities correspondences and the generic processor factory, this knowledge can be provided anytime by a configurator or an integrator. This knowledge is present in the architectural diagram as "VSP(Cam3, slideConf) $\ggg$ SlideChange(Room)". This configuration knowledge explicitly expresses that "Cam3" is the camera recording speaker's screen and that the appropriate processor configuration for change detection is "slideConf". This exact configuration needs to be defined somewhere, for example directly in the description in place of "slideConf".

factory: movie maker

Considering the movie maker service, we only introduce a factory for consistency. The original movie maker was already capable of handling simultaneous recording of multiple independent recording. This handling of multiple recording was done by a kind of session management where the client started a session and then worked within this session. This use of sessions had exactly the same role as service factories and we prefer to introduce a factory. With the movie maker services factory, each session is materialized as a movie maker service. This approach is more consistent with the rest of the architecture and allows the factory to declare itself as producing movie maker functionalities. To distinguish movie makers functionalities from each other, each has a property that we called "task" in the architectural diagram.

factory: configurable speech localization

As far as the speech detection and localization are concerned, we have separated the detection of the speaker and the audience. Originally, the sound localization service was directly used by the meeting director. This service provides the angle at which it perceives sound together with an information about the nature of this sound, voice or non-voice. The configuration of the meeting director was basically telling what ranges of angles corresponded to the speaker and the audience. We enrich the sound localization service and make a factory out of it. The speech localization factory can produce speech detectors based on a range of angles. The configuration of the meeting director is externalized as functionality correspondence and thus can be provided anytime, even when the system is already started. Telling that the speaker area corresponds to the angles from 0° to 90° can be expressed by the simple correspondence "SpeechLocalization(0,90) $\ggg$ SpeechDetector(Room, "Speaker")". Materializing this configuration makes it both easier to change during the lifetime of the system and reusable by other systems: the knowledge of what range of angles correspond to the speaker is not specific to the meeting director and it can be of interest for other services.

sharing: director factory

Our method has two last steps: "share the functionality" and "integrate". Sharing our meeting director functionality consists in evaluating whether having a factory of meeting directors is pertinent. In this case we can perfectly introduce a factory of meeting directors based on the requirements we identified. Figure 6.7 presents the factory we can introduce together with its functionality level description. In this figure, the factory has instantiated two meeting directors for two distinct rooms. In the description of the factory, we used "meetingNum = unique()" to express the fact that a unique identifier should be generated, for instance based on the identifier of the client of the factory.

integrated with processors

Concerning the integration of the meeting director into other systems, the last step of our method, we will not go further than what has already been done. We already integrated the meeting director with the video service processors infrastructure for the implementation of the slide detector. Further integration can be done afterwards by introducing new functionality correspondences.

dynamic configuration

Redesigning the automatic meeting recorder using functionalities makes it more flexible and dynamic. Using functionalities, system parts can easily be replaced by some other services providing the same functionalities. Functionality correspondences are also used to replace startup

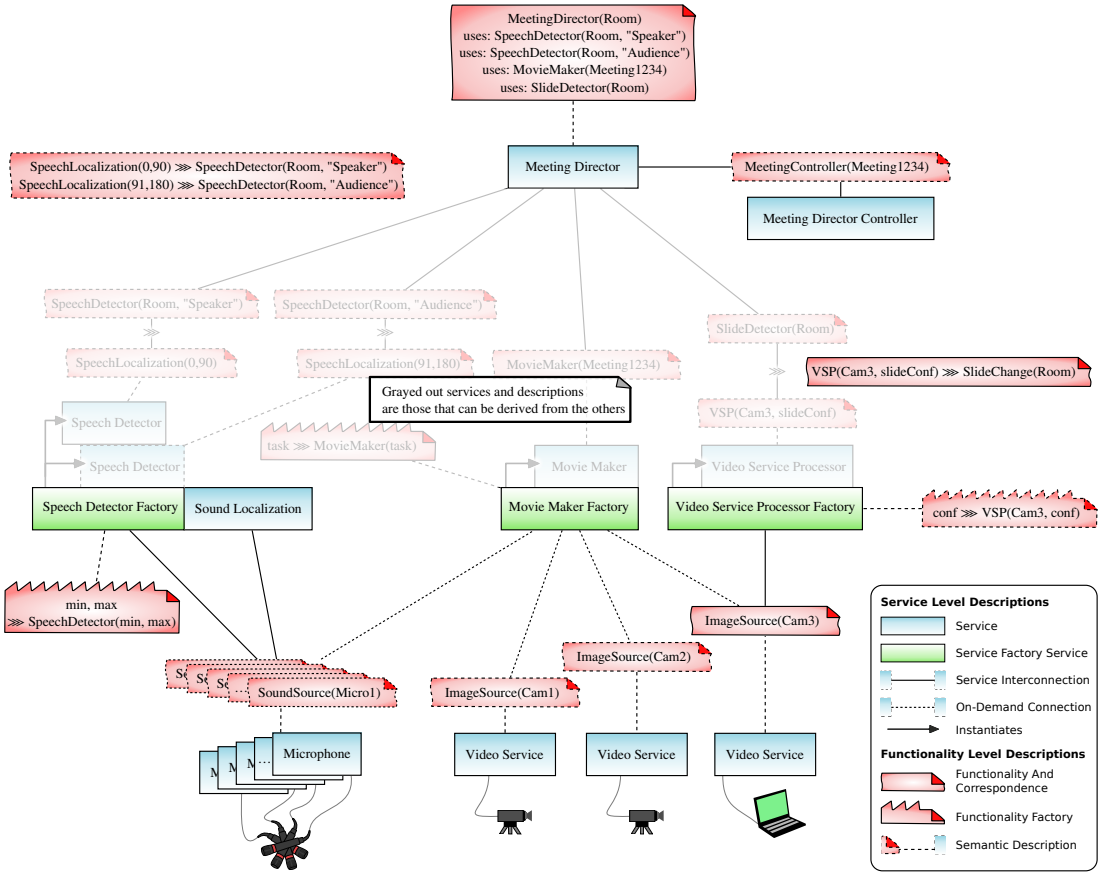


Figure 6.6: *Automatic Cameraman: What Can be Derived. Same figure as 6.5 but showing what has to be provided by designers. Lighter parts of the schema are all the elements that are automatically derived and instantiated.*

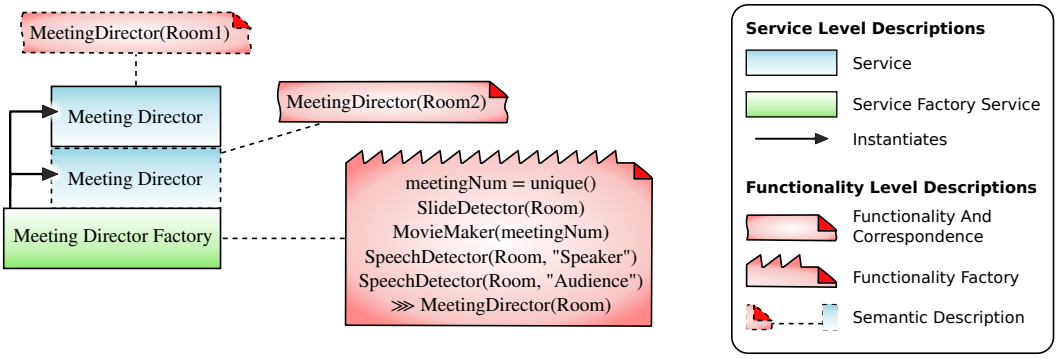


Figure 6.7: *Automatic Cameraman: Sharing the Functionality. Applying the "share the functionality" step of our design method incites to expose the meeting director as a factory and as a functionality. In this example the factory has instantiated two meeting director services.*

time configuration by dynamic configuration expressed as knowledge useable by any service. Effective benefits of our design method will be presented in chapter 9 that presents the final results of our work on the presented systems.

Chapter 7

UFCL, a Language for Semantic Service Description

7.1 Motivation and UFCL Positioning

new design method

In chapter 6, we proposed a design method to overcome reuse and interoperability problems identified in chapter 4. The presented design methods involved semantic descriptions of service functionalities to allow dynamic integration and spontaneous interaction between services. The presentation of the design method has been done in a total abstraction of the technologies that can be used to implement services or describe their functionalities.

UFCL to support our method

In this chapter, we introduce the User-oriented Functionality Composition Language (UFCL) that is specifically created to support our design method. UFCL uses a SQL-like syntax and allow the designer to express descriptions of three kind: service functionality descriptions, functionality correspondence descriptions and descriptions of service factories.

7.1.1 UFCL as a Simple Design Tool

obstacles to enabling concepts

We have identified three main principles for improving the design and architecture of systems in intelligent environments. All three of these principles encounter obstacles. In order of difficulty these are:

SOA

- Adoption of SOA: many middleware and tools are available to implement service oriented architecture but most of them have some problems. This includes difficulties in accessibility for average developers, performance problems for high throughput communications, and important restrictions on the language or platform that can be used.

functionality descriptions

- Semantic descriptions of service functionalities: although there are multiple semantic description languages, work on semantic descriptions is generally done by researchers from the knowledge representation field. These language are well designed but they address experts from that domain. Most of these languages propose little or no abstraction between pure generic knowledge representation and target user task consisting in the expression of service functionalities.

  Semantic web services (SWS) propose to describe service functionalities using semantic web technologies. SWS are not a suitable alternative in our case. First, persistent connections with high bandwidth streaming are required. Second, SWS technologies require the user to have a broad knowledge of many technologies (XML, XSLT, OWL, ...) which are long to master for any developer.

new concept of factories

- Open service factories and composing factories: to our knowledge, no existing research work or technologies propose a way to describe service factories. In chapters 4 and 6, we identified service factories as a major element supporting our design method. As there is no existing model for it, describing service factories will have to be done in an ad hoc manner.

SOA: chapter 5

We worked on overcoming these identified obstacles and on allowing average developer to build applications using the design concepts of SOA, semantic descriptions of functionalities and service factories. First obstacle concerns SOA adoption and has already been addressed as described in chapter 5.

framework for functionalities and factories

Concerning the two other obstacles, we propose a simple conceptual framework for the representation of service functionalities and service factories. This framework is directly inspired from our design method presented in chapter 6. The framework remains simple by semantically describing service functionalities independently from underlying service oriented middleware. Our conceptual framework allows semantic description of functionalities provided by services but it also includes the concepts around service factories including composing factories.

supporting language

UFCL

Based on this easily understandable framework, we have designed a language to write expressions in this framework. This language has been named Usable Functionality Composition Language (*UFCL*). The design of both the conceptual framework and the language were driven by the requirements identified in chapters 4 and 6, and oriented by the target audience: developers with some programming and modelling skills but no particular XML and semantic web ones.

7.1.2 UFCL as a Hub in Ambient Intelligence Engineering

Chapter 6 illustrated how we can redesign an existing complicated system and what interesting design tools could be useful in the developer's toolbox. The previous section gave an overview of the framework and the language (UFCL) that we propose for the design of intelligent environments and introduced how these can be used as a design tool. In this section we will illustrate how UFCL can be used by different actors in the design, maintenance and evolution of an intelligent environment.

integration is necessary...

As mentioned previously in this manuscript, the design of intelligent environments involves many specialists. An intelligent environment must be able to perceive, understand, learn, act and communicate with the user. Different specialists are working on these aspects and their contributions must interact to form the intelligent environment. The simplest solution that comes to mind is to say: "let the specialists do their work and have an integrator handle the packaging of the parts and their interoperability". Given the complexity of each specialty, this simple solution consumes much manpower: the integrator has to understand all domains to do his job. Understanding sufficiently each complex domain requires a considerable amount of work and the integrator cannot handle many domains at once.

... but too costly

continuous integration

We are convinced that, for a real interoperability and for a durable reuse of components, specialists must cooperate and favor the continuous integration of their systems. UFCL and our functionality centered framework aims at being a hub for the integration of individual contributions in intelligent environments.

implementer, consumer, integrator

We can categorize the actors involved in the conception of intelligent environments in 3 categories: functionality implementers, functionality consumers and integrators. This separation is oversimplified and depending on the level of abstraction that we consider, one actor may be for example both an implementer and a consumer. We will use this simplification in 3 categories to underline the advantages of a cooperative integration over an integration that is based on an integrator. At the scale of an intelligent environment, these categories can be the following:

- the functionality implementers could be the specialist implementing artificial perception: computer vision, acoustic signal processing, statistical object recognition, etc.
- the functionality consumers could be the specialist in activity modelling and user assistance
- the integrator is the engineer that currently has to understand the functionalities that are implemented and used, and who adapts them to have them operate together.

classical integration

Figure 10.3 illustrates the case where the integration is done by an engineer responsible for the interfacing of most software elements. The work of the integrator is essentially a "compile-time" work where he must adapt parts provided by different specialists so they fit. On the other hand, figure 10.4 shows the case where the concept of functionality is used as a central concept and every specialist uses it to express his requirements and what his system part provides. UFCL and the central concept of functionality is the base for interoperability.

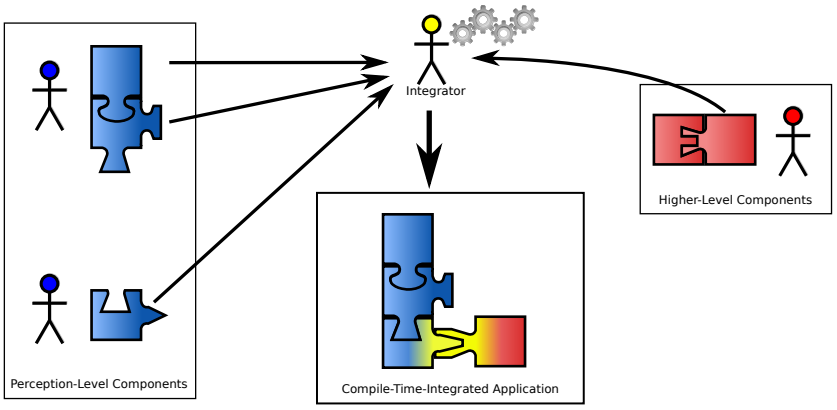


Figure 7.1: *Classical Integration Task. The integrator has to understand and modify most of the components that needs to be integrated. Components are modified at design time to produce an application.*

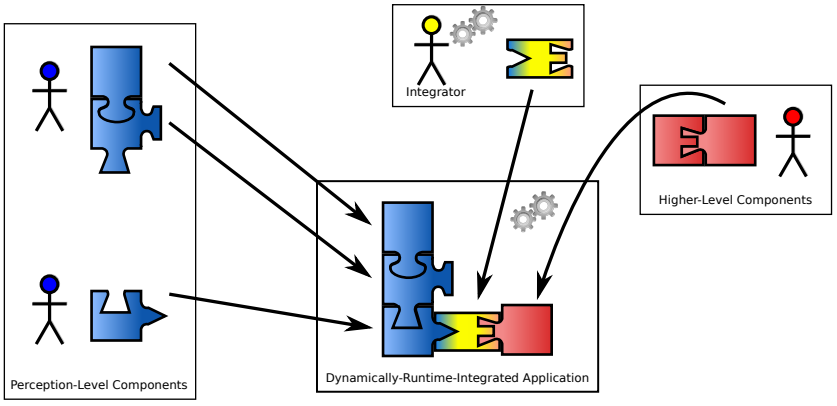


Figure 7.2: *Integration Task with UFCL. The integrator has to understand only the functionality provided and requires by components. Functionality correspondences and transformations are introduced without changing existing components to produce an application.*

continuous integration with UFCL

Using UFCL, the role of the integrator is transformed: formerly having to dive in the implementation details of each system part and adapt it, he can first look at the involved functionalities and see if he can express some correspondences between these functionalities. Two functionalities could be totally identical (by pure luck or if there was a prior agreement

between the provider and the consumer) or they can be slightly different, requiring some syntactical transformations (message formats, etc.) or they can be almost compatible, requiring some adaption code to be written.

dynamic integration

The principle of service oriented computing is already to hide implementation details to facilitate interoperability. Articulating the interoperability around the concept of functionality greatly simplifies the integration task in most of the cases and frees up some time to integrate more things or improve other aspects. In addition to simplify the task of the integrator, it also makes it possible to express functionality correspondences dynamically at runtime: integration in the form of functionality correspondences takes the form of some knowledge that can be provided at any time.

In the rest of this chapter, we will present the main concepts involved in our conceptual framework. Each concept will be explained and will be immediately illustrated by an example expressed using the proposed language syntax.

7.2 Services and Functionality Facets

7.2.1 Metamodel for Functionality Facets

concrete services have functionalities

The entire conceptual framework revolves around the concepts of *services* and *service functionalities*. The term of *service* is used in the sense attributed by the middleware community. Service represents a software entity (or software representing some hardware) that is discoverable and introspectable independently of its particular implementation. Details about what exactly constitutes a service in the middleware we have used can be found in chapter 5, section 5.2.1. To improve extensibility, our framework is designed to have no strong dependency to the middleware itself.

functionality = type+properties

In the knowledge framework we proposed, a service can expose any number of service *functionality facets*. A *functionality facet* has a functionality type and some associated properties. We chose this simple "type+properties" as it is a natural way of describing objects for human (and thus service designers): it is close to human mental models. For the same reason, this "type+properties" framework is the model underlying object oriented methods (modeling and design) and is close to knowledge representation formalisms.

detailed model

To detail all the possibilities offered by our framework at the level of functionality facet description, we can present the complete metamodel of our framework:

- A service has 0..* functionality facets.
- A functionality facet has
  - a functionality type,
  - 0..* valued properties.
- A functionality type (or functionality for short) is a semantic name representing a kind of functionality that services can provide.
- A valued property is composed of
  - a simple name,
  - a property value.
- A property value can be of different types

- simple literal (string, number, ...),
- a reference to a running service,
- a reference to a facet exposed by a running service,
- a reference to an arbitrary resource (e.g. a particular room, a unit).

Figure 7.3 gives a representation in UML of the metamodel presented in the previous bulleted list.

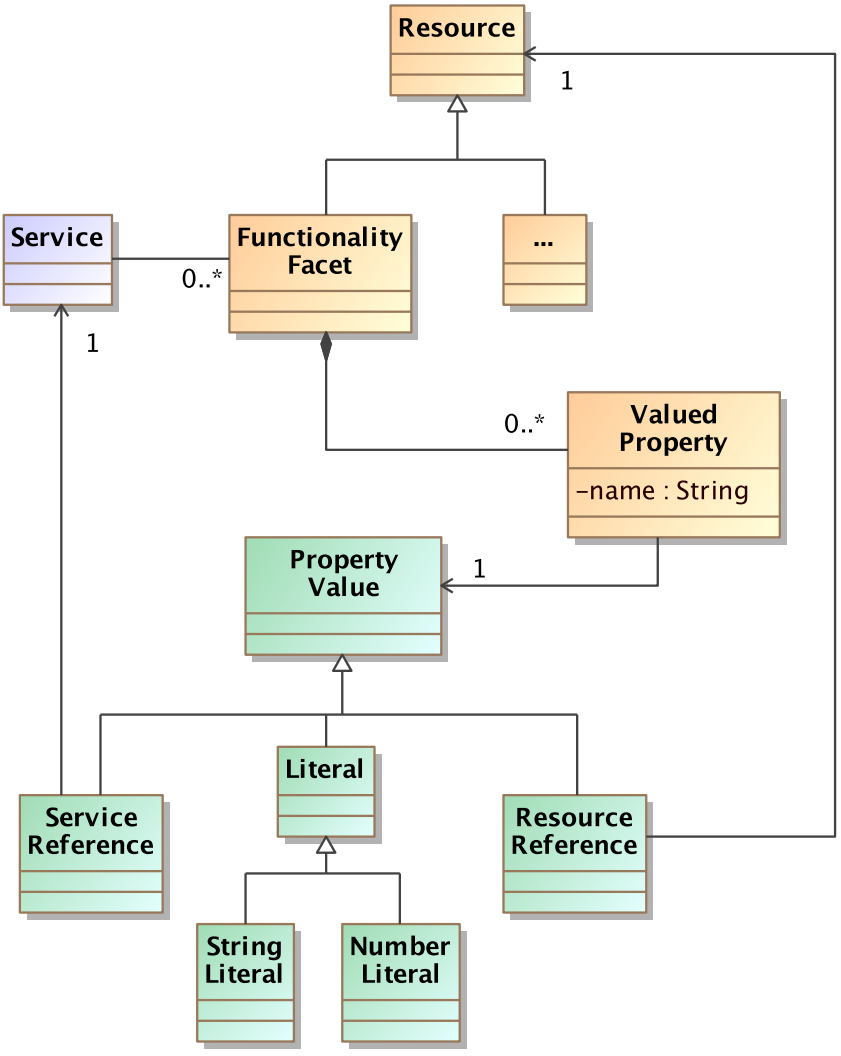


Figure 7.3: *Metamodel of Functionality Facets expressed in UML. A service may expose any number of functionality facets. Each facets has some named valued properties.*

avoid conflicts using namespaces

To avoid having two people use the same name for two different concepts (functionality types), the functionality type is a semantic name. Each knowledge writer has to specify the namespace or namespaces that he wants to use. Specifying namespaces must be mandatory and relatively easy for the service designer. We decided to reuse the solution chosen by XML (and all knowledge representation languages based on it) as specified in [Url-n]. Our semantic functionality names are represented by URLs and the designer can define short aliases for namespaces. Examples about namespaces are given in next section together with the UFCL syntax examples.

7.2.2 Expressing Functionality Facets in UFCL

Our language makes it easy to express that a service exposes some functionality facets. We can express such knowledge in UFCL as shown in the following snippet.

```
1 | namespace is http://emonet09#
2 | this isa Timer
3 |   with freq = 2
4 |   with grounding = "C(tickTwice)"
```

This UFCL expression simply declares that the current service is exposing a **Timer** functionality with a frequency of 2 (Hertz). As a first remark on UFCL syntax, we can state that whitespace and indentation are unimportant as they are in most of the programming languages. This choice has been made for the sake of familiarity: most modern programming languages are insensitive to indentation and whitespace.

namespace

In this declaration, line 11 gives a default scope for semantic names such as `Timer` at line 22. Given the namespace declaration, `Timer` will be interpreted as `http://emonet09#Timer`. Our language allows writer to use multiple namespaces and give them aliases using a syntax inspired by classical XML namespaces (e.g. `myNS:Timer`). There is no absolute default namespace: writers are forced to declare a namespace for their descriptions. Without this obligation, description writers would be tempted to write most of their descriptions in the absolute default namespace and all the advantage of namespacing would be lost. When everything is defined in a unique namespace, name collisions are likely to happen: a name can be used by two persons to represent different concepts.

"this"

At line 22, `this` is a special identifier that implicitly represents the service exposing the UFCL description. The identifier `this` could have been equivalently replaced by the exposing service identifier. Using `this` is a convenience for service designers as it allows instance-independent descriptions to be written. It is helpful not to use `this` in the case where a service describes another one. The need for describing another service may arise when a service has been deployed with incomplete or no description.

semantic of "isa": implements

Also at line 22, UFCL keyword `isa` introduces the functionality implemented by `this` service. We allow one service to implement more than one functionality. To reflect that, we refer to this implementation as a functionality facet: in the current example, we can say that `this` has a `Timer` functionality facet. One can note that `isa` could be replaced by `implements` which would be semantically better; however, in object-oriented design and languages, the implementation is associated to interfaces or purely abstract classes which are concepts that not all developers fully master.

reusing ontologies

At line 22, `Timer` (standing for `http://emonet09#Timer`) is simply a functionality name that may but does not need to be used or defined elsewhere. In fact functionality names are Unique Resource Identifiers (URI) like the one used in Resource Description Framework (RDF) and Web Ontology Language (OWL). This is intended to keep the door open to reuse possibly existing ontologies when describing implemented functionalities.

properties and grounding

Lines 33 and 44 affect values to two properties of the functionality facet, `with` being only a separating keyword. In this example, one of the two properties (`freq`) is a simple property of the `Timer` functionality facet. The other does not particularly concern the timer facet. The special `grounding` property attaches the defined functionality facet to an existing software service running in the environment. In this case, it tells us that the `Timer` facet is implemented using the single OMiSCID connector called `tickTwice` (`C(tickTwice)`) to be found on the service exposing the description.

grounding in another service

In the presented example, the grounding implicitly refers to a connector on the service implementing the functionality facet, `this`. Rare are the cases where one functionality facet

should reference variables and connectors from services other than the facet owner but it is still allowed by UFCL grammar. This possibility could for example be used to expose a `Radio` functionality extracted from an existing audio-video acquisition device. The radio service would then reference directly the audio connector of the existing service. The following code snippet uses the grounding expression `C(12340000:audio)` to explicitly reference the `audio` connector on the service having the identifier `12340000`.

```
this isa Radio
  with ...
  with grounding = "C(12340000:audio)"
```

To underline the semantic of `isa`, we can imagine that our service, that has been declared to be a Timer with a frequency of 2 is also a Timer with a frequency of 1, tick events being sent on a connector named `tick`. In this case, we would have another Timer facet declaration for the same service:

multiple implementation...

```
namespace is http://emonet09#
this isa Timer
  with freq = 2
  with grounding = "C(tickTwice)"1 |  namespace is http://emonet09#
this isa Timer
  with freq = 1
  with grounding = "C(tick)"
this isa Timer
  with freq = 1
  with grounding = "C(tick)"
```

... disallowed in object oriented modeling

This mechanism would not be permitted by a simple class belonging or classical interface implementation principles. In classical object-oriented modeling, inheritance and interface implementation mechanisms make it possible for an object to be of several types. Object-oriented modeling disallows an object to implement several times the same type. It is only through multiple inheritance that an object can be twice of a given type and these are clearly corner cases of object-oriented usage.

... and in plain OWL

Similarly, the semantic of RDF-based ontologies ignores the fact that a resource is linked to another one twice with a same predicate. This RDF semantic makes it impossible to model multiple implementation by using directly the rdf:type predicate. OWL inherits this limitation from RDF and its application for the description of semantic web services in OWL-S faced the same problem of multiple functionality implementation.

reifying as in OWL-S

The concept of functionality needs to be reified to fit in classical ontology description frameworks. In OWL-S, a service resource is not directly the subject of a `rdf:type` as we introduced the concept of functionality facet, they introduced the concept of capability. An OWL-S service has a set of capabilities, each of an arbitrary type and with some properties: no restriction prevents a service from having multiple capabilities of the same type. As we will see in chapter 8, we convert our language to an RDF representation and we adopted the same reification approach as in OWL-S.

7.3 Functionality Correspondences

7.3.1 Concept of Functionality Correspondences

e.g. equivalence of "timer" and "metronome"

Having described functionalities exposed by our services makes it possible to do service selection based on these functionalities but it is still required to have an a priori agreement on what functionalities will be used. Reconsider our **Timer** functionality introduced before: it only sends events at a fixed rate. In a different context, for example in a music oriented application, the designer could have named this functionality **Metronome**.

spontaneous interaction

We would like an application designed to work with `Timer` to spontaneously be able to use a `Metronome` service that would be present in the environment. To allow this integration, two kinds of knowledge are required: the knowledge that `Timer` and `Metronome` are equivalent concepts (modulo the name and unit of frequency) and the knowledge of how we can convert metronome's messages and communication protocol to timer's. These are two concerns that can be separated and we concentrated on the first one (chapter 9 discusses the protocol adaptation).

required correspondence information

Integrating a metronome in an application that expects timers requires a knowledge about this concepts correspondence. Either written by a human (the integrator from previous section) or generated using an automatic method, this knowledge is mandatory. This knowledge is basically composed of 3 parts:

- A functionality facet type that can be considered as another one (e.g. **Metronome**)
- Some constraints on the functionality properties (e.g. the number of beats per minutes (bpm) cannot be higher than 240)
- A functionality facet of correspondence with some values for its properties (e.g. a **Timer** with a frequency of bpm/60)

7.3.2 Expressing Functionality Correspondences in UFCL

UFCL has a dedicated construct to describe the equivalence between functionality facets. As detailed in the section 8.3.5 about the compilation of functionality correspondences (in next chapter), this syntax is just a shortcut for a more generic syntax based on factories. In the following snippet, lines 85 to 118 use this simplified syntax. Lines before line 85 are present only to help understanding: knowledge expressed in UFCL can be fragmented and provided by different sources at different time.

```
1 | namespace is http://emonet09#
2 | this isa Timer
3 |   with freq = 2
4 |   with grounding = "C(tickTwice)" 1 | namespace is http://emonet09#
2 | this isa Timer
3 |   with freq = 1
4 |   with grounding = "C(tick)"
5 | this isa Timer
6 |   with freq = 1
7 |   with grounding = "C(tick)"
8 | a Metronome
9 |     having bpm = ?f
10 |   isa Timer
11 |     with freq = ?f / 60
```

mapping of functionalities and properties

This UFCL fact expresses that a `Metronome` functionality facet can be used as a `Timer` functionality facet. Lines 85 and 107 put the two concepts in relation while 118 tells that the `freq` property of the produced timer has to be set to "?f / 60". `?f` is a wildcard defined in line 96 as having the value of `bpm` property in the Metronome facet (bpm stands for beats per minute) and has to be divided by 60 to be converted into a frequency with the unit expected in Timer facets. Globally, these 4 lines state that any service having a `Metronome` facet (with a `bpm` property but no particular constraints) will also have a `Timer` facet with its `freq` property set to 1/60th of `Metronome`'s facet property `bpm`. Correspondences between functionalities can be seen as a kind of ontology alignment, putting into relations concepts introduced by different designers. The `grounding` property that is defined for all functionality facets is automatically propagated in case of functionality correspondences. In this example, the `Timer` facet will inherit from `Metronome` facet's grounding.

adaptation of protocols

The second kind of knowledge required in functionality correspondences concerns message

format and communication protocol adaptation. It is not described by UFCL functionality correspondence. However it is an important problem that drives many research efforts, we do not handle this part of format and protocol adaptation, we just provide this clear architectural separation between description of semantic functionality correspondences and adaptation of communication formats. In our current grounding, formats and protocols are stored in service description: OMiSCID middleware associates to each service connector or variable a description and a format description. Using this middleware, these descriptions are the base information for protocol and format adaptation. In a fully operational system, we could allow services to expose knowledge about protocol and format adaptation, for example using text manipulation scripts and/or XSLT as it is done in OWL-S. In addition to these simple descriptions, we should also allow dedicated services to convert messages on demand (format and protocol adapter services).

7.4 Service Factories

7.4.1 Concept of Service Factories

describing service factories...

Both constructs presented in the previous sections are classical in existing semantic service description using ontologies. Through a simple syntax, UFCL aims at bringing these concepts to the average non semantic web specialist developer. In this section, the mechanism presented is more original and consists in describing service factories. The role of a service factory is to instantiate other services on demand.

... of two types

There are mainly two kinds of factory for two kinds of instantiations:

- parameterized service instantiation: given desired parameters, we can produce a service instance with those parameters. Our Timer facet is a good example of this kind of instantiation; we could easily write a factory that would instantiate any new Timer service given the desired frequency.
- composite service instantiation: given some references to existing services, we can produce a new service by composing their functionalities. We can illustrate this by the *translators* example. We could write a factory that would use any translator from language A to language B together with one from language B to language C and produce a translator from A to C.

One can imagine having a factory that both uses one or more services and accepts some free parameters.

infinite families of services

We claim that factories are a useful extension to existing service oriented architecture that uses service repositories. The timer/metronome example is underlining this point: it would be impossible to advertise the presence of all possible timers as there is an infinity of them but factories allow to instantiate any required timer. Factories have to advertise what family of services they can instantiate and under which conditions.

7.4.2 Expressing Service Factories in UFCL

e.g. "timer factory"

UFCL handles declaration of service factories we just presented. A first example is the one concerning `Timer` factory that would expose this UFCL description:

```
12 | composing
13 |   grounding "C(start)"
14 |   format "<run f='?pFreq'/>"
15 | gives a Timer
16 |   with freq = ?f
```

semantic of the description

Lines 15 and 16 express the fact that this factory produces services implementing `Timer` functionality and that the `freq` property takes the value of the `?pFreq` wildcard. No constraints are expressed on this `?pFreq` wildcard in the rest of the expression. It is completely free and any `Timer` can be produced. Any service consumer understanding this UFCL expression and requiring a Timer with a specific frequency would be able to ask the factory for the needed `Timer`.

concrete access to the factory

Lines 13 and 14, after `composing`, set some properties for the factory and may define required facets for composition (none here). While the `grounding` property works like its homonym in functionality facets, the `format` property tells the factory client how to format instantiation requests. In this example, to ask for instantiation of a Timer having a frequency of 7 Hz, one should send, on connector `start`, a message like "`<run f='7'/>`".

a factory for composition

An optional section, absent from this example, allows the factory to express constraints on used wildcards and required facets. It is illustrated in the following example on **Translator**s composition factory:

```
17 | composing
18 |   grounding "C(start)"
19 |   format "<c a='?t1#' b='?t2#'/>"
20 |   a Translator ?t1
21 |   a Translator ?t2
22 | having
23 |   ?t1.to = ?t2.from
24 | gives a Translator
25 |   with from = ?t1.from
26 |   with to   = ?t2.to
```

constraining using wildcards

This example contains many references to wildcards `?t1` and `?t2` defined at lines 20 and 21. These two wildcards are each representing one `Translator` functionality facet. At line 23, in the `having` section, is defined the only constraint between these two functionality facets: to be composable, destination language in `?t1` must be the same as source language in `?t2`. As in the previous example, last section at lines 24 to 26 expresses which functionality this factory produces. These lines tell us that this factory can produce a new Translator with the same source language as `?t1` and the same destination language as `?t2`. This example is particular and uses three times the same functionality (`Translator`) but any functionality or combination of functionalities can be used.

referencing services in grounding

Lines 18 and 19 give all the necessary information to ask the factory to create a product. First property, `grounding`, is used to locate the factory itself. In our case, like in previous examples, we are grounded into OMiSCID and the grounding expression contains a reference to an OMiSCID connector. Only the `start` connector name is given here so the grounding implicitly references the `start` connector on the service exposing the UFCL description. The second property, `format`, tells how to build the instantiation query message to the factory. The format expression is a plain string with some special expressions enclosed in curly braces. Here, format uses a particular expression "`?t1#`" that references the existing Translator functionality that matched `?t1` and more precisely its grounding (using the "`#`" suffix). In our case, services are represented by some unique numeric identifier and a formatted message accepted by this factory could look like "`<c a='C(12340000:tr)' b='C(98760000:tr)'/>`" where "C(12340000:tr)" references the `tr` connector (translate) on service with identifier "12340000".

no dependence from factories to UFCL

An important remark at this point is that factories receive messages containing only grounding information and no facet references. Grounding information is service level information and can be easily interpreted by a service even if it has no knowledge about UFCL and existing facets. This is made to decouple the factory implementation layer from UFCL : a factory does not need to gather UFCL descriptions nor to understand them nor to do reasoning about it. One can even build a factory service or use it without any exposed UFCL description.

The given examples do not feature combination of facet wildcards (like `?t1`) and parameter wildcards (like `?pFreq`). UFCL allows such combinations and a single factories description can use both references to services and open parameters.

hybrid factories: composition and parameters

7.5 Special Constructs to Make Designer's Life Easier

To simplify the authoring of UFCL descriptions, special "shortcuts" have been added to the language and we will present the two most important. These shortcuts are the result of the study of different use cases and are relatively important for the usability of the language. The first shortcut we will is a "syntactic sugar" that simplifies the definition of some classes of functionalities. The second one tries to follow the principle of "least surprise" and it impacts the semantic so as the reasoning on functionalities is closer what developers expect.

syntactic sugar and least surprise

The first shortcut allows the UFCL writer to easily create resources that are attached to a service. For instance, when declaring a **Camera** functionality, we want to be able to say that this camera has a 2-dimensional reference frame (the reference frame of its image) and a 3D reference frame (the reference frame of the camera itself in the 3D space). This can be easily done using the special construct that uses a "|" character. An example of an UFCL expression exposed by a camera service could be:

resources "within" services

```
1 | this isa Camera
2 |   with imageRef  = this|image
3 |   with cameraRef = this|camera
```

With this same UFCL description, each camera service will define two resources representing its reference frames. This "|" construct provides a kind of namespace that is local to a service. The namespace defined with this construct is visible from any other UFCL expression. Given a camera service that would have the unique identifier "12340000", another service such a person detector service could expose a functionality like this one:

service local "namespace"

```
1 | this isa Detector
2 |   with referential = #12340000|image
3 |   with modality    = modality:Vision
```

The second principle has no visible effect on the syntax but can be summarized as follows: "a reference to a service is considered as a reference to all of its functionality facets". An example where this shortcut is useful appears when we consider a camera calibration that has to reference a camera. Here is an example of an UFCL description illustrating this case, it is detailed just after:

a service is all its facets

```
 1 | this isa ExternalCameraCalibration
 2 |   with camera = #12340000
 3 |   with object = J113Room
 4 |
 5 | a ExternalCameraCalibration
 6 |     having camera = ?c
 7 |     having object = ?o
 8 |   isa ChangeOfReferenceFrame
 9 |     with from = ?o
10 |     with to   = ?c.cameraRef
```

Here an **ExternalCameraCalibration** references a **Camera** and knows a referential in which the calibration has been done. Conceptually a camera calibration contains the information to pass from the reference frame attached to an object used for the calibration procedure to the reference frame of the camera. This equivalence of knowledge is expressed in the functionality correspondence in the second part of the UFCL snippet. In this subsumption, `?c.cameraRef` is used. If we look in details, `?c` is a reference to the service with identifier "#12340000" but

avoid sneaky problems

it is its facet (and not the service) that has a cameraRef property. The fact that "a reference to a service is considered as a reference to all of its functionality facets" makes `?c.cameraRef` behave as expected.

How to Use these Descriptions

wrapping it up

Systematically advertising and looking for services with semantic functionalities together with the presence of service factories bring a lot of flexibility and make it easier to adapt to changes in software environment. One can easily integrate into an already running application an alien service and have it interoperate with existing services. However, to make this spontaneous interoperability come to life, we must be able to manipulate and reason about all these semantic descriptions. In the following chapter, we will present a compiler from UFCL to some rules that makes it possible to take advantage of the presented constructs.

Chapter 8

Runtime Framework Over UFCL Descriptions

8.1 Objectives and Design Decisions

8.1.1 Bringing UFCL to life

In previous chapter, we presented a framework for semantic description of service functionalities. We presented the motivations of this framework and proposed a language, UFCL, that aims at being simple and usable by the average developer. When faced with such descriptions, we, as human beings, naturally infer what factories and services can be used and composed to obtain a given functionality. Even if it may seem trivial for simple small size cases, making a computer do this reasoning rigorously is a complicated task.

rigorous inference is required

We propose a runtime that, given the exposed UFCL descriptions, reasons about these descriptions and is capable of answering a query from a "client". Here, the client is an application that is looking for a particular functionality. The aim is that developer of this application just have to express what functionality type and which properties are needed and the runtime will do all the rest.

... to fulfill queries from clients

We can reformulate the task of our runtime as "finding a plan that will provide the desired functionality". A plan may involve some queries to service factories and it is ultimately a grounding to a functionality facet fulfilling the target functionality. All exposed UFCL descriptions may interact in a complex way in solving this problem.

... by finding a composition plan

Performance-wise we would have liked to be able to formulate this problem as an optimization problem. This would also have made it simple to integrate some costs to different plan elements such as cost of composing two particular services, cost of starting an new service, etc. We have not found a proper way to reformulate our task as the resolution of an optimization problem but this direction is still to explore in depth as it has many advantages. The main obstacle to viewing this problem as an optimization one, was the concept of service factories and particularly open service factories. Some factories can produce any functionality among a possibly infinite family of services.

optimization made difficult by factories

Two other directions for solving such problems have appeared promising: multi-agent systems and rule based systems. Both of these paradigms are hard to master and can become tricky when used intensively. Rule-based reasoning is well suited for problems where many independently designed sources of knowledge must spontaneously interact. Given this adequacy and having no experience with multi-agent systems and limited experience with rule-based systems, we decided to explore the usage of rule-based systems.

falling back on rule based systems

8.1.2 Using a rule engine and backward chaining

facts and rules

A rule based system is composed of a base of facts and some rules. The base of facts is often called a working memory. A rule is composed of some premises and some conclusions and can be seen as an if-then rule. For example, a rule can express something like "if it looks like a duck and it quacks like a duck then it is a duck". In this example, "it looks like a duck" and "it quacks like a duck" are the premises while "it is a duck" is the unique conclusion. We can see that "it" is not clearly defined and appears in multiple parts of this rule. "it" is a variable that can represent anything and, appearing multiple times in the rule, it adds some constraints on the facts that can be matched or will be produced by the rule.

forward chaining

Based on its working memory and on its rules, the rule based system can have two main inference policies: forward chaining and backward chaining. The case of forward chaining is the easiest to imagine: rules that have their premises fulfilled by the facts in the working memory are fired (executed) and generate new facts in the working memory. In some systems, a priority can be set for each rule in order to control the firing order when multiple rules can fire at the same time. The user of the rule based system can inspect the working memory and search for particular facts of his interest.

... can easily loop forever

One problem of forward chaining is that it "blindly" applies rules and generate facts. This can cause problems in case of infinite rules. To take a simple example, we can consider a context where we have a string of a given length and want to use it to measure a smaller distance. One can imagine a rule like "if we can measure a distance d then we can measure a distance $d/2$": as soon as a string can measure a distance d, we can fold it in two to measure a distance half as long as d. With such rule, having even a single string measuring a distance d, we can use it to measure any distance of $d/2^n$. If the production rule is applied "blindly" and with no stop condition, it can get stuck into trying to generate all these possibilities ($d/2$, $d/4$, ...) and never apply some other rules that are necessary to solve the problem.

... for example with factories

Affecting priority to rules and telling the production algorithm to be breadth first can be a solution in this example. However, forward chaining cannot tackle the case of open rules where a rule can generate infinite products without need for recursive application. In our context, an example of open rule can be the example of a **Timer** functionality: we can easily create a timer having any (reasonable) frequency and we would want a rule that represents that. This requirement of ours makes the use of simple forward chaining inappropriate to solve our problem.

backward chaining

The alternative to forward chaining is backward chaining. Compared to forward chaining, rules and facts stay the same but the inference mechanism is different. Backward chaining uses the user request as a basis and rewrites this request using existing rules. This rewriting process progressively builds trees of requirements until the facts present in the knowledge base can fulfill a tree. Rewriting the goal expressed by the user can fall into the problem of infinite loop as much as the forward chaining solution. Using backward chaining, it is possible to properly apply open rules such as the **Timer** functionality rule: as soon as a particular **Timer** will be required, the rule will be fired and will create the proper **Timer**.

... is guided by client request

choosing Jena

We chose the Jena Semantic Web Framework [Url-o] [Url-p] to run our rule based system. This choice was driven by the open sourceness of this project, its high activity and its features. Jena offers the possibility to do both forward and backward chaining. Jena also proposes a hybrid inference engine based both on forward and backward chaining. In this hybrid mode, each rule is either backward or forward; all applicable forward rules are first executed and then backward inference is used to fulfill user query.

... with limited backward chaining

Our requirements clearly state that we need backward chaining. We discovered, only during the implementation, some limitations in Jena's backward chaining support. We reimplemented a custom backward chaining algorithm, reusing only Jena's forward chaining support. Section 8.2.2 details the principle our custom backward chaining implementation.

This custom implementation of backward chaining was an unexpected and significant extra work over what we expected from reusing Jena. Having to implement backward chaining gave us a clearer insight of how backward chaining works and a better control over it. It also made our implementation portable to any basic forward chaining inference engine.

... caused extra work

8.1.3 Do not let the user write rules

We chose rule-based systems as a tool to solve our problem but not as a front-end to our target users (average developers). When people decide to use rule-based systems they often fall in what we consider as a common trap: they let the user "free" to write custom rules. In our case, we could be tempted to let user write rules that would interact with the rules generated by the compilation of UFCL. However, with great freedom comes complexity: letting the user write the rules delegates all the complexity to the final user. This complexity causes the user to have to learn all the tricks of designing rule-based applications. In addition to this complexity, the fact that all rule engines are different forces the user to adapt to the particular rule engine being used.

with freedom (to write rules)...

... comes complexity

Our usage of rules can be compared to classical assembly language. Each UFCL expression can lead to the generation of many complicated rules and it would not be helping to let the user write custom rules. We use UFCL as a high abstraction, domain specific language that is designed to be productive and that is compiled to some set of coherent interacting rules: some of the generated rules heavily rely on the structure of the facts generated by other rules. We can compare UFCL to languages like C++ that gets compiled down to a possibly huge amount of assembly language statements that all need to be coherent (call conventions, structure of virtual tables, ...). For example, in C or C++, the definition of a function is compiled to some assembly code that has a particular structure: it starts with a label, it expects a particular state of the stack that reflects the value of the parameters of the function, it ends with a return assembly instruction (`ret`) and has to free any allocated stack space. The exact structure of the generated code for C functions is the call convention and both the code of the functions and the code of the callers must stick to this convention to have the program run properly. To go back to the case of UFCL, letting the user write custom rules would be equivalent to letting the user mix assembly language with C++ (which is by the way possible but used only in rare highly specific cases).

rules as the assembly language for UFCL

do not mix assembly and higher level languages

8.2 Introduction to the compilation mechanisms

8.2.1 The Jena semantic web framework

As explained in previous section we decided to use Jena as our rule engine. We had to implement a custom backward chaining engine using forward chaining capabilities of Jena.

Jena is designed for the semantic web and it uses RDF as its fact representation format. Where some others knowledge representation system use any arbitrary length tuples, RDF is using only triples. RDF triples are of the form "subject predicate object" or "(subject, predicate, object)" using a tuple notation. When in some other systems such as CLIPS we would represent a person having a birth year and a height with a tuple like "(Person Bob 1981 1.65)", Jena's limitation forces a reification into triples: multiple triples will be used to represent a single tuple. In this example we could use three triples to represent the same knowledge: "Bob rdf:type Person", "Bob yearOfBirth 1981" and "Bob height 1.65".

Jena facts: RDF triples

This example gives an occasion to introduce another concept in RDF: apart from the expression of knowledge using triples, RDF also brings some predefined predicates and resources. Among these elements defined in the RDF specification is the rdf:type predicate which is used to associate a type to resources from the knowledge base. This rdf:type predicate is one that

existing RDF predicates

we reuse intensively in our descriptions. Any element in a triple can be written using a prefix followed by a colon. This prefix references a namespace defined in the knowledge base. In previous example, we used "rdf:type" that, given the definition of the rdf prefix stands for "http://www.w3.org/1999/02/22-rdf-syntax-ns#type". To simplify the notation in this section, we will consider that any element with no prefix (such as "Bob") is in a custom namespace.

An RDF model composed of triples can be represented in various ways. In this manuscript, we will use two main representations: a simple textual syntax with some colors to improve readability and a graphical syntax. Using the textual notation, we could express that some knowledge about the previously cited "Bob" and his boss "Harry" as follows:

textual notation for RDF...

```
1 |   (Bob rdf:type Person)
2 |   (Bob yearOfBirth 1981)
3 |   (Bob height 1.65)
4 |   (Bob hasBoss Harry)
5 |   (Harry height 1.9)
6 |   (Harry yearOfBirth 1592)
```

... and a graphical one

In addition to this textual notation, we introduce here a graphical notation. In this graphical notation, each resource is represented using an ellipse and properties are represented using edges between these ellipse-shaped nodes. Figure 8.1 is an example of the graphical representation of the previous triples.

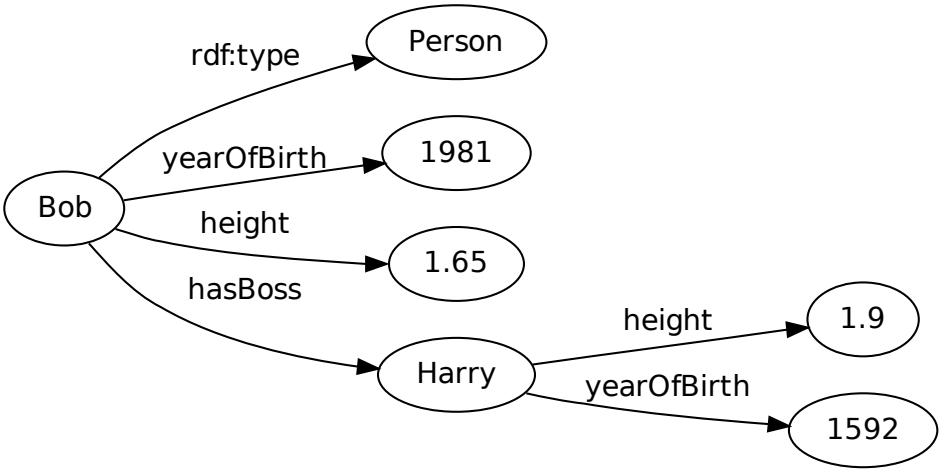


Figure 8.1: *Graphical representation of an RDF model*

In addition to manipulating some RDF models (set of triples), we have to declare rules that will be applied on these models to generate new triples. Jena is extensible and allows any custom syntax to be used to define rules. It also proposes a built-in syntax for rules that is perfectly suitable for our usage. As we are autogenerating rules, exact rule format is not very important to us, expressive power is far more important. All rules through this manuscript will be written using the rule syntax used in Jena, syntax that we introduce here:

rule syntax...

```
1 | [
2 |   (?this soundsLike ?that)
3 |   (?this looksLike ?that)
4 | ->
5 |   (?this isClassifiedAs ?that)
6 | ]
```

The tuples appearing before the arrow at lines 2 and 3 are the premises, the ones after the arrow are the conclusions (line 5). Elements starting with a question mark are variables and are used to write more generic rules and to impose constraints on the matching and produced triples. Additional rule constraints can be expressed in the premises as in the following example:

... with variables...

```
1 | [
2 |   (?this looksLike ?that1)
3 |   (?that1 looksLike ?that2)
4 |   notEqual(?this ?that2)
5 | ->
6 |   (?this looksLike ?that2)
7 | ]
```

Jena proposes other built-in functions to do comparison, arithmetic operation, test for predicate presence and absence, new node creation, etc. All these built-in functions can be called using a syntax similar to the one used for "notEqual".

... and built-in functions

Variables appearing in theses built-in function calls can be of two types: bound or unbound. Bound variables are variables that are constrained elsewhere in the premises like *?this* and *?that2* in the example. Unbound variables are variables that get affected by the built-in function. The following rule illustrates these unbound variable together with the `makeTemp` built-in function that is used to create a new node:

bound vs unbound variables

```
1 | [
2 |   (?box1 weight ?w1)
3 |   (?box2 weight ?w2)
4 |   sum(?w1 ?w2 ?total)
5 |   lessThan(?total 900)
6 |   makeTemp(?container)
7 | ->
8 |   (?container rdf:type OneMetricTonContainer)
9 |   (?box1 isIn ?container)
10 |  (?box2 isIn ?container)
11 | ]
```

In this example, the rule groups any two boxes that weigh less than 900 kg and puts them in a dedicated container. At line 4, the unbound variable *?total* gets bound to the sum of *?w1* and *?w2*. In the knowledge base, the resource representing the container is created by the rule at line 6 and "affected" to the *?container* variable.

creating resources

8.2.2 Compilation overview and introduction

This section explains the basic principles behind the compilation of knowledge expressed in UFCL to some set of triples and rules. We need to compile 3 kinds of UFCL constructs: simple facet implementations, simple subsumptions and service factories descriptions. Only complex UFCL constructs such as subsumptions and service factories descriptions will lead to the generation of rules. We will first give an introduction to how simple facet implementation are compiled.

3 kinds of UFCL descriptions

The compilation of a simple UFCL facet implementation construct will produce only a set of triples. These triples will feature a resource representing the implementing service and a resource

simplifications for explanations

for the facet; the facet will have a type and some properties. To give an idea of the generated triples, we can take a look at the knowledge generated by the compilation of a simple UFCL statement. This section is just an introduction and thus some simplifications in the notation are made to improve understandability and conciseness. These notation simplifications concern how we represent literals from basic datatypes in RDF (strings and integers here) and on the form of autogenerated tokens (temporary names).

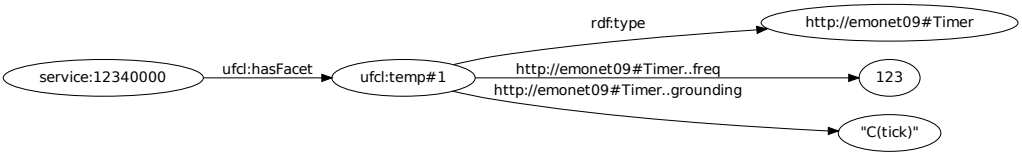
We consider the following UFCL description:

```
1 | namespace is http://emonet09#
2 | this isa Timer
3 |   with freq = 123
4 |   with grounding = "C(tick)"
```

functionality descriptions produce facts

The true semantic of the previous UFCL expression depends on the service that expose it (through "this") and we make the supposition that the exposing service has the unique identifier 12340000. Under this supposition, the compilation of this UFCL statement will produce a set of triples that use the RDF namespace (rdf) and a custom namespace (ufcl) for UFCL compilation elements. These triples are the following:

```
1 |  (service:12340000 ufcl:hasFacet ufcl:temp#1)
2 |  (ufcl:temp#1 rdf:type http://emonet09#Timer)
3 |  (ufcl:temp#1 http://emonet09#Timer..freq 123)
4 |  (ufcl:temp#1 http://emonet09#Timer..grounding "C(tick)")
```



subsumptions and factories...

The global principle for the compilation of facet implementation is presented below. We will first introduce the principle that is used by service factories and subsumptions that have a tighter dependence on the need expressed by the user (the developer or program looking for a particular functionality).

... rewrite user need

When the user searches for a service implementing a given functionality, the system monitors the presence of a matching functionality facet. In the case the facet is directly implemented through a simple declaration, it will easily be found. The case of factories and subsumptions are more complicated as they work in a backward manner and rewrite some needs. To enable backward rules to work, when the user searches for a particular functionality facet, this search is added in the knowledge base as a need.

example of successive need rewriting steps

Factories and subsumptions are based on rewriting an existing need to one or several other needs; where all needs generated by a factory are fulfilled, it generates a fulfillment for the original need. To illustrate the principle guiding the rule generation process, we take an example with simple concepts (trivial UFCL facets with no properties). A service is exposing a functionality A, and some subsumption knowledge is exposed: "a B isa C" and "a A isa B". When the user searches for a functionality C, some triples will be added to tell that a C is needed (step 1 in figure 10.5). One rule produced by "a B isa C" is a need rewriting rule and will then generate a triple telling that a B is needed (step 2). In the same way, a triple telling that an A is needed will be generated because of "a A isa B" (step 3). This last need for A can be fulfilled directly by the service present in the environment (step 4). A second rule generated by the compilation of "a A isa B" is a fulfillment rule and will propagate the fact that the need for A is fulfilled as the fulfillment of the need for B (step 5). In the same way, a fulfillment of

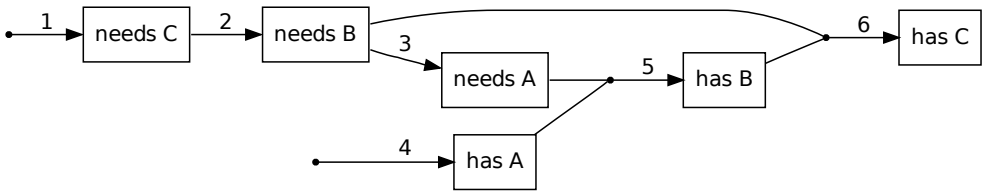


Figure 8.2: *Example steps of need rewriting and fulfillment*

the original need for C will be generated due to the rules issued from compilation of "a B isa C" (step 6).

The presented principle of need rewriting and fulfillment propagation is the core of our generated code. Around this principle, some additions are required to have this reasoning working effectively. These additions mainly aim at preventing infinite recursions when doing inference with our generated rules. Trying to avoid infinite recursion can lead to a problem of incomplete inference: if not designed carefully, our system can easily miss some services that could possibly fulfill user needs.

avoiding infinite loops

Section 8.3 gives an in-depth description of the generation mechanisms, first for simple descriptions and then for factories. Compilations of subsumptions have been reduced to a particular case of service factory to avoid code duplication and thus they will be presented at the end of the section on composing factories.

subsumption as a special case of factory

8.2.3 Automatic Inference of Implicit Constraints

Previous section presented the general principle of our compilation process that generates rules implementing backward chaining. This section explains a mechanisms that is orthogonal and happens during compilation: from the constraints expressed in an UFCL description, we automatically infer derived constraints.

Writers of UFCL descriptions can express constraints between different elements. We can consider an UFCL description for a factory that composes translators, with two expressed constraints:

UFCL expression with...

```
1 | composing
2 |   a Translator ?t1
3 |   a Translator ?t2
4 | having
5 |   ?t1.to != ?t1.from
6 |   ?t1.to  = ?t2.from
7 | gives a Translator
8 |   with from = ?t1.from
9 |   with to   = ?t2.to
```

In the given UFCL description, two constraints appear in the "`having`" section: `?t1.to != ?t1.from` and `?t1.to = ?t2.from`. Two other explicit constraints are present in the given UFCL description. If we call `?prod`, the unnamed product of the composition, these two constraints are: `?prod.from = ?t1.from` and `?prod.to = ?t2.to` (present in the "`gives a`" section). From these four constraints, three others can be derived: `?t1.from != ?t2.from`, `?prod.from != ?t1.to` and `?prod.from != ?t2.from`. Figure 8.3 illustrates explicit and implicit constraints in this case.

... explicit constraints...

... and derived ones

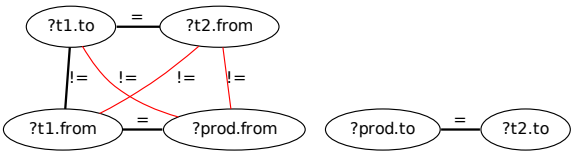


Figure 8.3: *Constraint Inference Example. From explicit constraints (in bold black), implicit constraints are inferred (in red).*

Section 8.3.5 also details the rule generation process for an example involving arithmetic inference. The UFCL expression used in this example is the following subsumption:

```
1 | a Metronome
2 |     having bpm = ?f
3 |   isa Timer
4 |     with freq = ?f / 60
```

constraint inference is mandatory...

In this example, a single explicit constraint is given: we can express it as `timer.freq = metronome.bmp / 60`. From this constraint, the constraint `metronome.bmp = timer.freq * 60` is automatically inferred. This inversion of the constraint is necessary to do proper backward chaining: rules should rewrite a need for a **Timer** (with a given frequency) to a need for a **Metronome** (with the proper "bpm"). To be able to rewrite a need for a **Timer** into a need for a **Metronome**, the system must know what "bpm" value to search for (in function of the "freq" of the **Timer**).

... and independent

Constraint inference is totally orthogonal to our backward chaining fulfilling a query for a functionality. Inference of new constraints is done when an UFCL expression is compiled. The derived constraints are generated and added to the original definition.

rule-based implementation...

We have implemented constraint inference using rules. In this case again, we used Jena as our rule engine. These rules handle inference over equality (transitivity), inequality (e.g. `a = b and b != c` implies `a != c`) and inversion of operations (e.g. `a = b / 60` implies `b = a * 60`).

... with limitations and evolutions

Our current implementation handles all equality and inequality inference, and most common use cases around operations. To handle more complicated cases, we could use and embed a real equation system solver (starting our search from [Url-q]).

8.2.4 Integration on top of OMiSCID middleware

Previous sections gave an overview of how UFCL knowledge get compiled to enable inference. The compilation and reasoning described in previous section is independent from the OMiSCID middleware. This separation is the result of a design effort in this sense. Before detailing the compilation process (in section 8.3), we will see how the integration of OMiSCID and the reasoning system is done to combine their functionalities.

distributed dynamic UFCL knowledge base

The first objective is to make it possible for any service to expose knowledge expressed using UFCL. By convention, we propose that UFCL knowledge would be located in a dedicated variable of services called "knowledge". The particular choice of "knowledge" is a little general and a more specific name should be chosen in the case of a diffusion to a wider community. The combination of all "knowledge" variables from running services constitutes an open distributed database of semantic descriptions. OMiSCID is a convenient way to implement this dynamic distributed database of UFCL descriptions: it handles dynamic discovery of knowledge sources

(services) and adding a dedicated variable is a non intrusive way of providing the description of services.

library to publish descriptions...

A service designer can fill the "knowledge" variable using standard OMiSCID APIs. A library aiming at simplifying the integration of OMiSCID and UFCL has been designed. This library provides helpers to make it simpler to write UFCL knowledge and to set knowledge variables. More importantly, this library encapsulates all the process consisting in gathering and reasoning about the knowledge from all running services. From the user standpoint, only a desired facet type and some valuation for the facet properties have to be provided to the library tools, everything else being done automatically.

... retrieve them...

The library provides the user with the possibility to create a KnowledgeGatherer object that is dedicated to the optimization of the knowledge gathering and reasoning process. A knowledge gatherer listens to the apparition and disappearance of OMiSCID service in order to maintain a union of the UFCL information exposed by running services. When a new service appears, the knowledge gatherer updates its overall model with the new information provided by this new service: it retrieves the value of the "knowledge" variable (if any), compiles the UFCL expression to a reasoning model composed of triples and rules, and inserts this model into the local union model it maintains. In the same way, when a service disappears, its model is retracted from the union model by the knowledge gatherer. Using OMiSCID facilities, the knowledge gatherer can also register to "knowledge" variable modifications to update its model when a service updates its UFCL knowledge.

... and fulfill user need by reasoning...

When the user specifies the functionality facet he or she is looking for, the knowledge gatherer generates the associated triples (see section 8.3.2) and adds them in the union model it is maintaining. From the moment the user has specified his functionality requirement, the knowledge gatherer runs the inference process on its union model. When some new knowledge is added in response to a service apparition, the inference process is resumed with the additional knowledge information. As soon as a service providing the facet required by the user is found, the knowledge gatherer returns the associated grounding (a reference to the OMiSCID service).

... and invoking factories

The concrete service implementing a functionality can be directly present in the environment but it can also be issued from a more complicated plan containing some queries to service factories. If the knowledge gatherer finds a virtual implementation of the functionality facet required by the user, it automatically applies the associated plan by formatting and sending instantiation query messages to the factories involved in the plan. All these calls to factories are transparent for the user and he or she eventually gets the desired functionality facet implementation if it is present or instantiable.

local reasoning on a dynamic distributed database

Leveraging the dynamic nature of OMiSCID, the knowledge gatherer is able to access a dynamic distributed database of semantic description: the knowledge gatherer monitors service apparitions, modifications and disappearance and updates its internal model accordingly. Even if the knowledge is distributed, the reasoning is centralized and duplicated for each knowledge gatherer. Performance-wise, it could be tempting to have an inference process that would be shared by all the users and even distributed. Our case is not well suited to sharing as our reasoning highly depends on the need expressed by the user: different needs lead to very different inferences.

motivations of local reasoning

Duplicating the reasoning and making it local has several advantages:

- it makes it possible for a user to do some reasoning that mixes the knowledge exposed by running services with some knowledge that should not be published: internal "private" knowledge, uncertain "prospective" knowledge, etc.
- it makes reasoning more robust as we only rely on other services to provide their descriptions: busy services will not impact the speed of the reasoning process

- it keeps the possibility to deploy, on a classical computer, an inference proxy that can be used by resource constrained devices (like a PDA).

8.3 Detailed compilation of UFCL constructs

8.3.1 Compiling simple descriptions

The previous section gave an overview of how UFCL can be compiled to give RDF triples and rules. It also described why some mechanisms must be set up in order to cope with problems such as an infinite generation process. Compared to the generation process described in the overview, the only element that we add to the compilation of simple description is an "age".

avoiding loops with ageing

We added an ageing process to our inference mechanism to ensure inference process would not loop infinitely. The ageing is done by affecting an age at each facet implementation in our knowledge base. Facet implementation directly exposed by services will be inserted in our knowledge base with an age of 0 and facets generated by our inference process will get older and older. As an illustration, we can take an example of facet implementation like this one:

```
1 | namespace is http://emonet09#
2 | this isa ChangeOfReferenceFrame
3 |   with from = J113
4 |   with to = Calibrator
5 |   with grounding = "C(transform)"
```

representing age

We make the same supposition that the exposing service has the unique identifier 12340000 and also we simplify the names of temporary resources. The compilation of this UFCL statement will produce the following triples:

```
1 |  (service:12340000 ufcl:hasFacet ufcl:temp#1)
2 |  (ufcl:temp#1 rdf:type http://emonet09#ChangeOfReferenceFrame)
3 |  (ufcl:temp#1 http://emonet09#ChangeOfReferenceFrame..from http://emonet09#J113)
4 |  (ufcl:temp#1 http://emonet09#ChangeOfReferenceFrame..to http://emonet09#Calibrator)
5 |  (ufcl:temp#1 http://emonet09#ChangeOfReferenceFrame..grounding "C(transform)"^^xsd:string)
6 |  (ufcl:temp#1 ufcl:hasAge "0"^^xsd:int)
```

exact RDF representation

The notation `"C(transform)"^^xsd:string` is the one used to declare literals in RDF. Any type from the XML Schema Definition specification (XSD) can be used to type RDF literals. We can emphasize that the code we generate for the facet properties implementation contains predicates such as `http://emonet09#ChangeOfReferenceFrame..from` with not only the name of the property (`from`) but also the name of the facet type (`http:...ChangeOfReferenceFrame`). The motivation of this repetition is to facilitate debugging and understanding of the generated model: there is no homonymy between properties and thus it gets simpler to look for a particular facet property.

8.3.2 Asserting user need

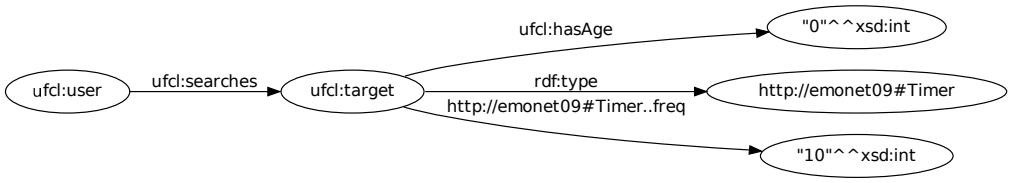
As introduced in section 8.2.2, as soon as a particular functionality facet is required, a corresponding need is inserted in the knowledge base. The triples issued from the compilation, like those presented in the previous section for a **ChangeOfReferenceFrame**, will immediately fulfill this need if they match it. Not only the generated triples can match directly a need but they can also be used by other rules for more complex cases like UFCL factories and subsumptions that will be described in following sections.

The principle of our inference is to do backward chaining. Backward chaining makes the presence of an explicit goal mandatory to trigger inference. When the user searches for a particular functionality facet, we insert triples corresponding to his query into the knowledge. This

asserting the user need...

initial need is expressed using the invariant triple `(ufcl:user ufcl:searches ufcl:target)` where `ufcl:target` get described as a classical facet implementation. For example in the case where the user is looking for a **Timer** with a frequency of 10, we would insert these triples in the knowledge base:

```
1 |  (ufcl:user ufcl:searches ufcl:target)
2 |  (ufcl:target ufcl:hasAge "0"^^xsd:int)
3 |  (ufcl:target rdf:type http://emonet09#Timer)
4 |  (ufcl:target http://emonet09#Timer..freq "10"^^xsd:int)
```



All rules generated by the compilation of UFCL descriptions will basically work on these triples representing a need. The initial need is issued from the query expressed by the user as in the **Timer** example above. This initial need can possibly get rewritten by the rules that we will present in the following sections. Rewriting a need produces a new set of triples having the same form as the one representing the initial need.

. . . that will get rewritten

8.3.3 Compiling open factories

In section 8.3.1, we presented what our system generates for UFCL facet implementation statements. In this section, we will detail what is generated for other statements such as simple subsumptions and various factory descriptions. These generation mainly produces rules and we will see how these rules interact with knowledge generated by simple declaration and with each others.

In this manuscript, we call factory a service that is designed to instantiate other services. We exhibited two main kind of factories: composing factories that aim at producing new services by composing existing ones and open factories that can instantiate any service drawn from a particular parametered family of services. As presented in section 7.4.2, declaring that a service is an open service factory can be expressed in UFCL with declarations of the following form:

two kind of factories

```
1 |  composing
2 |    grounding "C(start)"
3 |    format "<run f='?pFreq'/>"
4 |  gives a Timer
5 |    with freq = ?f
```

Details about the semantic of this description are given in section 7.4.2. Such an open factory will generate only one rule in the knowledge base. The generated rule will process any need for a **Timer** with a given frequency to a fulfillment of this need together with the information required to instantiate the new service implementing the **Timer**.

open factory: one rule

Given this way of expressing user need in the knowledge base, the previous UFCL snippet describing the open timer factory will produce the following rule, expressed using Jena's syntax for rules:

```
1  | [
2  |   (?any ufcl:searches ?prod)
3  |   (?prod rdf:type http://emonet09#Timer)
4  |   (?prod http://emonet09#Timer..freq ?prod_freq)
5  |   makeTemp(?compose)
6  |   makeTemp(?end)
7  |   makeTemp(?endf)
8  | ->
9  |   (?end ufcl:action ?compose)
10 |   (?end ufcl:hasFacet ?endf)
11 |   (?endf rdf:type http://emonet09#Timer)
12 |   (?endf http://emonet09#Timer..freq ?prod_freq)
13 |   (?compose ufcl:parameter-f ?prod_freq)
14 |   (?compose ufcl:hasGrounding "C(11223300:start)")
15 |   (?compose ufcl:hasFormat "<run f='?pFreq'/>")
16 | ]
```

principle of the generated rule

The premises of this rule basically tell: when anyone searches for a product of type Timer with a given frequency, create 3 intermediate nodes that will be used in the conclusions of the rule. Execution of this rule will insert some triples representing predicates and properties to express the fulfillment of the requirement. What will get generated is a new virtual service *?end* that will have a functionality facet *?endf* of type Timer with the required frequency. Additional decoration is added to the virtual service to express how a real service can get instantiated. This information takes the form of the newly generated node stored in the *?compose* variable. This information is called *instantiation information*.

instantiation information

The *?compose* variable holds all information necessary to instantiate a real service corresponding to the virtual service *?end*. This information includes the properties of the factory that generated the rule and an assignment to the free variables present in the UFCL description of the factory. In our particular example we have the two properties of a factory (grounding and format properties) and only one free variable assignment (assignment of *?f* to the desired frequency).

fulfilling any matching need

The consequence of adding this rule in the knowledge base is that anytime an implementation of a **Timer** facet will be required, one virtual service will be declared to fulfill this requirement. This rule will apply for both an initial need expressed directly by the user and for a need that could get generated by another rule present in the knowledge base.

8.3.4 Compiling composing factories

Previous section illustrated how open factories definitions get compiled from UFCL down to some rules. Open factories are factories that can instantiate any services from a particular family of services. This section explains the compilation of composing factories. This compilation process differs notably from the one for open factories.

factories that compose services

This section is dedicated to the compilation of the factories that produce other services by composing existing ones. Compiling these composing factories descriptions is the most complex and difficult part of the overall compilation process. Open factories (having free parameters) described in previous section and composing factories are not disjoint concept: it is possible for a factory to be both composing and open. Openness and composition are two facets of the factories that are handled differently and thus we will first present the composing factory independently of the open factory concept.

To illustrate the composition of composing factories, we will study the compilation of the following UFCL description which corresponds to a purely composing factory:

```
composing
  grounding "C(start)"
  format "<comp corf='?transformation#' det='?detector#'/>"
  a ChangeOfReferenceFrame ?transformation
  a Detector ?detector
having
  ?transformation.to = ?detector.frame
gives a Detector
  with frame   = ?transformation.from
```

a factory composing N services...

Basically, this description tells that composing a change of reference frame and a detector can produce another detector operating in a different reference frame. By the way, the functionality of a Detector is to affect a probability of object presence to any point that is passed to it. If we want to do some detection in a reference frame R1, we can transform our points in R1 to their equivalent in R2 and then use a detector in the R2 reference frame. The transformation from points in R1 to points in R2 can be done using a change of reference frame R1↦R2.

...generates N+1 chained rules

The code generation process will generate a number of rules that depends on the number of services that get composed in the service factory description: in the previous example we have *?t1* and *?t2* appearing in the service factory description. For cases like our example, one rule more than the number of involved services is generated. One rule for each involved service will rewrite previous need (original need for the first rule) to the need for the next involved service. A last rule will wrap everything up and convert the fulfillment of the last need of this chain to the fulfillment of the original need. In our example, we have two services involved in the composition and thus we will have 3 rules:

e.g. change of reference frame and detector

- one rule rewrites the need for the product **Detector** to a need for *?detector*, a **Detector** with a different reference frame of operation,
- one rule rewrites the fulfillment of this need for *?detector* to a need for **ChangeOfReferenceFrame** (*?transformation* variable),
- one last rule produces a fulfillment of the original need for **Detector** when the need for *?transformation* is fulfilled.

Figure 8.4 illustrates how the rules interact with each other to obtain a detector in R1 in the presence of a detector in R2 and a transformation R1↦R2.

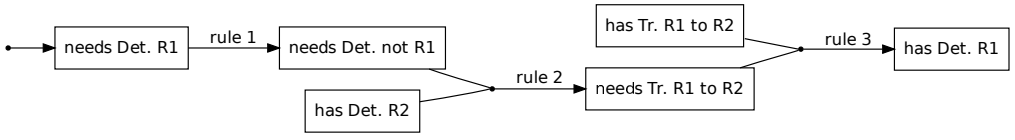


Figure 8.4: *The $n + 1 = 3$ rules for a factory involving $n = 2$ services. Rules working on needs and exposed functionalities to compose a detector (Det.) with a transformation (Tr.).*

any resource can become a reference frame

In this example, the first rule is responsible for the rewriting of the original need for a Detector to a need for any detector in any reference frame. The reference frame of the new detector we are searching for is not bound and can take any value. As we are in an open world and as new functionality facet implementations can get instantiated by service factories, we have to consider all existing resources as a potential reference frame for a potential detector. We introduce the concept of `ufcl:candidate` to represent the set of resources that have to be tried when search is totally unconstrained.

rule 1...

The first Jena rule generated for the previous UFCL description is the following:

```
1  | [
2  |  (?any ufcl:searches ?prod)
3  |  (?prod rdf:type http://emonet09#Detector)
4  |  (?prod http://emonet09#Detector..frame ?prod_frame)
5  |  (?any ufcl:hasAge ?firstAge)
6  |  sum(?firstAge '1'^^http://www.w3.org/2001/XMLSchema#int ?newAge)
7  |  lessThan(?newAge '10'^^http://www.w3.org/2001/XMLSchema#int)
8  |  (?subjectunboundframe ufcl:candidate ?unboundframe)
9  |  makeTemp(?newSearcher)
10 |  makeTemp(?detector)
11 | ->
12 |  (ufcl:factory-1 ufcl:hasSearchInstance ?newSearcher)
13 |  (?newSearcher ufcl:searches-prod ?prod)
14 |  (?newSearcher ufcl:uses ?detector)
15 |  (?newSearcher ufcl:searches ?detector)
16 |  (?detector ufcl:isSearched ?newSearcher)
17 |  (?detector rdf:type http://emonet09#Detector)
18 |  (?detector http://emonet09#Detector..frame ?unboundframe)
19 |  (?newSearcher ufcl:hasAge ?newAge)
20 | ]
```

This rule matches any need for a Detector within a given reference frame, adds one to the age of this need and continue only if this new age is less than a limit (here 10). This rule applies for any candidate frame and this is expressed using the triple `( ?subjectunboundframe ufcl:candidate ?unboundframe )` that will match any candidate resource.

... in details

For each of need and each candidate, a new search instance is created: it is composed of the newly created nodes *?newSearcher* and *?detector*. To keep track of which rule generated this search instance, the *?newSearcher* is declared as a search instance of an automatically generated resource representing this rule: in this example `ufcl:factory-1`. The suffix number is automatically increased during the compilation process each time a new rule creation requires it. To have a generation process more generic and to have enough information for further rules, the *?newSearcher* gets linked to its target product *?prod* and gets bidirectionaly linked to the new facet that it requires: *?detector*. The link from *?newSearcher* and *?detector* is duplicated using `ufcl:uses` and `ufcl:searches`: the first one is a way to keep information about the searcher and the second is there to trigger other rules that can rewrite or fulfill this new need. By the way, all rules aiming at fulfilling or rewriting a need are triggered on the condition that a someone "ufcl:searches" something. More information about the generated needs is also generated including the properties of this needed facet and the newly calculated age.

rule 2...

The second rule for our previous UFCL description will aim at rewriting the fulfillment of the need generated by the previous rule to a need for the second involved functionality facet of type ChangeOfReferenceFrame. The second generated rule is exactly this one:

```
1  | [
2  |   (ufcl:factory-1 ufcl:hasSearchInstance ?searcher)
3  |   (?searcher ufcl:uses ?searched)
4  |   (?searched http://emonet09#Detector..frame ?searched_frame)
5  |   (?searcher ufcl:searches-prod ?prod)
6  |   (?prod http://emonet09#Detector..frame ?prod_frame)
7  |   (?owner ufcl:hasFacet ?detector)
8  |   (?detector rdf:type http://emonet09#Detector)
9  |   (?detector http://emonet09#Detector..frame ?detector_frame)
10 |   equal(?searched_frame ?detector_frame)
11 |   (?searcher ufcl:hasAge ?firstAge)
12 |   (?detector ufcl:hasAge ?tmpAge)
13 |   sum(?tmpAge '1'^^http://www.w3.org/2001/XMLSchema#int ?secondAge)
14 |   max(?firstAge ?secondAge ?newAge)
15 |   lessThan(?newAge '10'^^http://www.w3.org/2001/XMLSchema#int)
16 |   makeTemp(?newSearcher)
17 |   makeTemp(?transformation)
18 |  ->
19 |   (ufcl:factory-2 ufcl:hasSearchInstance ?newSearcher)
20 |   (?newSearcher ufcl:searches-prod ?prod)
21 |   (?newSearcher ufcl:searches-detector ?detector)
22 |   (?newSearcher ufcl:uses ?transformation)
23 |   (?newSearcher ufcl:searches ?transformation)
24 |   (?transformation ufcl:isSearched ?newSearcher)
25 |   (?transformation rdf:type http://emonet09#ChangeOfReferenceFrame)
26 |   (?transformation http://emonet09#ChangeOfReferenceFrame..from ?prod_frame)
27 |   (?transformation http://emonet09#ChangeOfReferenceFrame..to ?detector_frame)
28 |   (?newSearcher ufcl:hasAge ?newAge)
29 | ]
```

The first part of the premises (line 2 to 9) imports a need that has been generated by the previous rule: it matches a search instance of `ufcl:factory-1` (the autogenerated resource representing the first rule). All information that is necessary to this second rule is imported with the first 4 triples matchers. Apart from applying the ageing process (line 11 to 15), the rest of the premises aims at matching a Detector that fulfills the need represented by the *?searched* facet. All constraints that can be inferred from the UFCL description are put in this rule. In our case, the only constraint is a direct one and the rule just has to check that the detector's reference frame is the one we are looking for in the searcher (line 10).

... in details

On the same model as the first rule, this second rules generates a new need: this correspond for the last need for a **ChangeOfReferenceFrame**. At this point, the Detector that is needed (*?prod*) is known to be operating in the *?prod_frame reference* frame. The possible other Detector that could be used (*?detector*) is also known and operates in the *?detector_frame* reference frame. To complete the composition, the **ChangeOfReferenceFrame** required is fully specified: *?prod_frame* $\mapsto$ *?detector_frame* The conclusions of this rule are only asserting the need for such a **ChangeOfReferenceFrame** and decorate this new need with informations that will be used by the fulfillment rule.

accumulating constraints in the process

The last rule that gets generated by our compiler for the previous UFCL statement uses the result of previous rules to enrich the knowledge base with a virtual service fulfilling the original need. This virtual service will appear in the knowledge base as a classical service implementing a functionality facet and will be able to be used by other rules requiring this functionality facet. The asserted virtual service has some decorations containing the necessary information to effectively instantiate this service. The rule is longer than the previous ones but is basically working on the same principle. It is shown in full length here:

rule 3...

```
[
 (ufcl:factory-2 ufcl:hasSearchInstance ?searcher)
 (?searcher ufcl:uses ?searched)
 (?searched http://emonet09#ChangeOfReferenceFrame..to ?searched_to)
 (?searched http://emonet09#ChangeOfReferenceFrame..from ?searched_from)
 (?searcher ufcl:searches-prod ?prod)
 (?prod http://emonet09#Detector..frame ?prod_frame)
 (?searcher ufcl:searches-detector ?detector)
 (?detector http://emonet09#Detector..frame ?detector_frame)
 (?owner ufcl:hasFacet ?transformation)
 (?transformation rdf:type http://emonet09#ChangeOfReferenceFrame)
 (?transformation http://emonet09#ChangeOfReferenceFrame..to ?transformation_to)
 (?transformation http://emonet09#ChangeOfReferenceFrame..from ?transformation_from)
 equal(?searched_to ?transformation_to)
 equal(?searched_from ?transformation_from)
 equal(?transformation_from ?prod_frame)
 equal(?transformation_to ?detector_frame)
 (?searcher ufcl:hasAge ?firstAge)
 (?transformation ufcl:hasAge ?tmpAge)
 sum(?tmpAge '1'^^http://www.w3.org/2001/XMLSchema#int ?secondAge)
 max(?firstAge ?secondAge ?newAge)
 lessThan(?newAge '10'^^http://www.w3.org/2001/XMLSchema#int)
 makeTemp(?end)
 makeTemp(?endf)
 makeTemp(?compose)
->
 (?end ufcl:action ?compose)
 (?end ufcl:hasFacet ?endf)
 (?endf rdf:type http://emonet09#Detector)
 (?endf http://emonet09#Detector..frame ?prod_frame)
 (?compose ufcl:hasGrounding 'C(11223300:start)')
 (?compose ufcl:hasFormat "<comp corf='?transformation#' det='?detector#'/>")
 (?compose ufcl:parameter-detector ?detector)
 (?compose ufcl:parameter-transformation ?transformation)
 (?endf ufcl:hasAge ?newAge)
]
```

same kind of premises...

This rule is built exactly on the model of the previous rule as far as the premises are concerned but the conclusions differ as they assert a new functionality facet implementation instead of a new need.

...with more constraints

Premises import requirement information from the previous rule starting from any search instance of `ufcl:factory-2`. They also require a proper **ChangeOfReferenceFrame** (*?transformation*) to fulfill the last need for composition. All constraints expressed between this *?transformation* and other elements taking part in the composition (*?searched* and *?prod*) are expressed in the premises. As introduced in 8.2.3, these constraints are issued from constraint inference to maximize reasoning efficiency.

conclusions create a virtual service...

... with instantiation information

When everything required to fulfill the composition pattern is found, the conclusions of this last rule assert a new functionality facet *?endf* provided by a new virtual service *?end*. The produced functionality facet implements the original need this factory has previously rewritten in two steps. As this produced functionality facet may eventually get used to fulfill the very need of the user, the necessary information to instantiate a real service from the virtual one must be accessible from the virtual service. This information is contained in the `ufcl:action` of the generated virtual service that is represented by the resource *?compose*. This instantiation information contains the grounding and format string that tell how to instantiate the service: where to find the factory and how to format the message we send to it. As the format string may contain references to the variable used in the UFCL description of the factory, instantiation information also keep some references to the facets matching these variables (*?transformation* and *?detector*). The section dedicated to the integration with OMiSCID middleware details how this instantiation information is used. .

8.3.5 Compilation of Simple Subsumptions

subsumptions in UFCL...

As presented in chapter 7, it is possible to use UFCL to express that a functionality type is a subtype of another one provided some transformations are applied to its properties. There is a dedicated construct in UFCL to do so. As an example, declaring that a Timer can be derived from any Metronome can be done using the following UFCL declaration:

```
1 | a Metronome
2 |     having bpm = ?f
3 |   isa Timer
4 |     with freq = ?f / 60
```

... as need rewriting rules

The rule generation process for such a subsumption is similar to the one for factories. Every need for a Metronome gets rewritten to a need for a **Timer** by a first rule. A second rule transforms any presence of a Timer to an implementation of a Metronome facet.

limited syntax

From the user point of view, the syntax dedicated to subsumptions is in fact a little bit too limited: in the previous example, the Metronome facet is unnamed and thus it cannot be referenced in when expressing constraints. The code `having bpm = ?f` implicitly references the "bpm" property of the Metronome but it is impossible to express constraints between two properties of the Metronome. Writing `having bpm = bpm` would mean: "bpm property is equal to `http://emonet09#bpm` (resource in the current namespace)".

reuse factory syntax and semantic

This need for a more powerful syntax for some use cases and the similarity of the rule generation process compared to the service factories guided us to propose an additional syntax to express simple subsumptions in UFCL. The previous code snippet can be equivalently replaced by the following one that uses a factory like syntax:

```
1 | composing
2 |   a Metronome ?m
3 |   product "?m"
4 | having ...
5 | gives a Timer
6 |     with freq = ?m.bpm / 60
```

This notation allows for a greater freedom as it makes it possible to express constraints on the parent facet as it is named (`?m` in this example). The special "product" property is used in place of the "grounding/format" property pair to tell that no instantiation query is necessary and that one of the components can be used directly.

The compilation of a subsumption is similar to the one of a factory that "composes" a single service. This compilation produces two rules: a first rule rewrite the need for a **Timer** in a need for a **Metronome**, a second rule fulfills the original **Timer** need if a matching **Metronome** is found. The first rule is strictly identical to the rule of a factory:

```
1  | [
2  |   (?any ufcl:searches ?prod)
3  |   (?prod rdf:type http://emonet09#Timer)
4  |   (?prod http://emonet09#Timer..freq ?prod_freq)
5  |   (?any ufcl:hasAge ?firstAge)
6  |   sum(?firstAge '1'^^xsd:int ?newAge)
7  |   lessThan(?newAge '10'^^xsd:int)
8  |   product(?prod_freq '60.0'^^xsd:float ?tmparith)
9  |   makeTemp(?newSearcher)
10 |   makeTemp(?m)
11 | ->
12 |   (ufcl:factory-1 ufcl:hasSearchInstance ?newSearcher)
13 |   (?newSearcher ufcl:searches-prod ?prod)
14 |   (?newSearcher ufcl:uses ?m)
15 |   (?newSearcher ufcl:searches ?m)
16 |   (?m ufcl:isSearched ?newSearcher)
17 |   (?m rdf:type http://emonet09#Metronome)
18 |   (?m http://emonet09#Metronome..bpm ?tmparith)
19 |   (?newSearcher ufcl:hasAge ?newAge)
20 | ]
```

This particular case illustrates the arithmetic transformation that is done by constraint inference as presented in section 8.2.3. We can see that the original constraint `freq = ?m.bpm / 60` has been inverted to `?m.bpm = freq * 60`. The effects of this inference are visible on lines 8 and 18 of the generated rules. The rest of the rule is identical to the first rule of a factory. The second rule differ from the one of a factory:

```
21 | [
22 |   (ufcl:factory-1 ufcl:hasSearchInstance ?searcher)
23 |   (?searcher ufcl:uses ?searched)
24 |   (?searched ufcl:isSearched ?searcher)
25 |   (?searched http://emonet09#Metronome..bpm ?searched_bpm)
26 |   (?searcher ufcl:searches-prod ?prod)
27 |   (?prod http://emonet09#Timer..freq ?prod_freq)
28 |   (?owner ufcl:hasFacet ?m)
29 |   (?m rdf:type http://emonet09#Metronome)
30 |   (?m http://emonet09#Metronome..bpm ?m_bpm)
31 |   equal(?searched_bpm ?m_bpm)
32 |   (?searcher ufcl:hasAge ?firstAge)
33 |   (?m ufcl:hasAge ?tmpAge)
34 |   sum(?tmpAge '1'^^xsd:int ?secondAge)
35 |   max(?firstAge ?secondAge ?newAge)
36 |   lessThan(?newAge '10'^^xsd:int)
37 |   (?end ufcl:hasFacet ?m)
38 |   makeTemp(?endf)
39 | ->
40 |   (?end ufcl:hasFacet ?endf)
41 |   (?endf rdf:type http://emonet09#Timer)
42 |   (?endf http://emonet09#Timer..freq ?prod_freq)
43 |   (?endf ufcl:hasAge ?newAge)
44 | ]
```

This second (and last) rule has exactly the same premises as the last rule of a factory until line 36: premises match a facet that fulfills (lines 28 to 36) the last asserted need (lines 22 to 27). The rest of the rule is different from and simpler than the case of a factory. The subsumption has only to create a new facet (line 38), fill it (lines 41 to 43) and attach it to the original **Metronome** facet owner (lines 37 and 40).

Most of the rules fragments generated for subsumptions are identical to the case of factories. We integrated the compilation of subsumptions in the factory compilation process to reuse efficiently code for factory compilation.

8.3.6 Compilation of Special Constructs

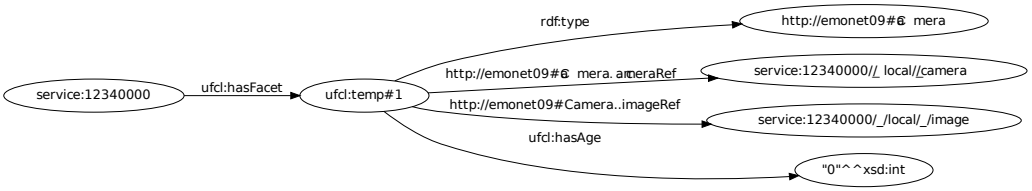
In section 7.5, some UFCL constructs where presented that help designers in writing knowledge. This section illustrates these constructs by giving the result of their compilation.

The first special construct presented in section 7.5 makes it easy to define resources that are attached to service. This construct uses a "|" for its syntax and can be used as in this example:

```
1 | this isa Camera
2 |   with imageRef  = this|image
3 |   with cameraRef = this|camera
```

In this example again, the exact generated triples depend on the unique identifier of the OMiSCID service exposing the knowledge (we consider that the identifier is 12340000). For an expression like `this|image`, our compilation process will generate resource named `image` "within" the resource of the service. To implement this behavior, we use a convention to represent resources "within" another: we added `/_/local/_/` as a separator between the container and the inner resource.

```
1 |  (service:12340000 ufcl:hasFacet ufcl:temp#1)
2 |  (ufcl:temp#1 rdf:type http://emonet09#Camera)
3 |  (ufcl:temp#1 http://emonet09#Camera..cameraRef service:12340000/_/local/_/camera)
4 |  (ufcl:temp#1 http://emonet09#Camera..imageRef service:12340000/_/local/_/image)
5 |  (ufcl:temp#1 ufcl:hasAge "0"^^xsd:int)
```



Summary and Conclusions

In chapter 6, we introduced a method based on semantic description of services and service factories. In chapter 7, we proposed a description language to express these various semantic descriptions. Automatic reasoning about these descriptions is necessary to get some benefits in terms of dynamic integration and spontaneous interaction. In current chapter, we explained how we can transform UFCL descriptions to make reasoning possible and allow automatic service discovery based on service functionalities. Our proposed implementation handle all UFCL constructs from simple service functionality descriptions to more complex service factories descriptions.

reasoning about semantic descriptions...

In this chapter, we detail the compilation of UFCL to some facts and rules in the Jena semantic web framework. We had to re-implement a custom backward chaining engine on top of Jena which constituted an important workload. The complexity of this task supports our argument about the final user not writing rules. Any potential user would probably change method if the design method requires to manually write such complex rules. In our case the front-end is the UFCL language and no rules are to be written by the user: user can take advantage of the inference without being overwhelmed by its complexity.

... using a rule-based engine...

... under the hood

Chapter 9

Critical Evaluation and Perspectives

9.1 Content and Structure of This Chapter

Chapters 4 to 8 presented our contribution and their motivations. In this chapter, we evaluate our contributions and conduct a critical study of our investigation. From the a posteriori analysis of our work, we identify strong and weak points of our approach and propose future research directions.

targeted evaluations

In section 9.2, we try to evaluate each aspect of our work. There are several contributions but not all are easy to evaluate. We will discuss the problem of evaluation: how people evaluate different approaches, what is difficult to evaluate and what just cannot be evaluated. In this first section, the quality of each contribution is evaluated as independently as possible to judge the effectiveness of each action.

global analysis

In section 9.3, we take a step back and analyze how our work articulates with other aspects of software engineering for intelligent environments. We particularly underline the fact that we proposed a complete design framework for real dynamic architectures for intelligent environments. We study the interactions between our method and other key aspects of intelligent environments such as context modeling on the one hand and sensor heterogeneity on the other.

retrospection and conclusions

Finally, we conclude this chapter by some lessons learned from the retrospective analysis of our work. These lessons suggest future research directions, from simple evolutions over our work to completely new ways.

9.2 Direct Evaluation of Contributions

9.2.1 Summary of Our Contributions

In chapter 4 we identified problems with integration and reuse of software in intelligent environments.-

actions to solve integration problems

We stated that reuse and integration problems can be solved by the implementation of proper software engineering methods: semantically described services. The acceptance of new software engineering by such a various audience is conditioned by its simplicity of use and its perceived advantages. Apart from our analysis and illustration of the problems slowing down the advent of intelligent environments, we can categorize our contributions in 4 points:

- Promotion of service oriented architectures
- Proposal of a design method for inter-operable services

- Proposal of a usable language supporting this method
- Implementation of a runtime for this language

Promotion of service oriented architectures

simple SOA solution

Our approach to promote SOA has been to make conception tools usable and tailored for the target users. We proposed a simple library, OMiSCID, that can be used to publish, discover and interconnect services. This library proposes an API that is uniform between various languages and that is written to maximize ease of learning, usability and quality of the most intuitive code. To help with the development of service oriented applications using this library, we provided an easy to extend graphical user interface. These contributions are presented in detail in chapter 5. Section 9.2.4 presents the evaluations dedicated to this contribution.

Proposal of a design method for inter-operable services

method inspired from semantic web services...

Based on our usable service oriented library, we proposed a design method that can be seen as a set of guidelines to follow to make individual software elements easier to integrate and make them interoperable. Chapter 6 details the motivation and principles of the method we proposed. This design method leverages the use of a service oriented architecture and introduces additional practice that we have identified as mandatory to obtain a real integration and make dynamic interoperability possible. Part of these practices are concepts currently used in the semantic web service community: services are described in terms of the functionalities they provide. These functionalities are expressed at a semantic level where automatic reasoning about equivalence of functionalities is possible.

... with additional service factories

In addition to semantic descriptions, we introduced a new design element, service factories: service factories are services that can instantiate other services. With service factories, it becomes possible to represent some infinite families of services and to simplify repetitive deployment of application. Service factory descriptions are part of the reasoning process at the functionality level and can serve to implement automatic service composition. From state of the art methods for semantic service description, we chose a subset of most important concepts enabling dynamic interoperability. Section 9.2.4 studies the properties of the systems designed using our method.

Proposal of a usable language supporting this method

domain specific language

We proposed a language to express semantic service functionality descriptions together with service factory capabilities. Providing a new language was motivated by the new concept of factories on the one side and by the requirement for simplicity on the other. Our method aims at being used by most specialists working in the conception of an intelligent environment, so it must be easy to understand, to learn and to put in practice. Verbose languages with a complicated metamodel are hard to read and understand and often become blockers for adoption. The constructs of this language are presented in chapter 7, the properties of the language are studied in section 9.2.6.

Implementation of a runtime for this language

runtime framework enabling powerful reasoning

To illustrate the practical feasibility of our proposal, we have created a runtime library. This runtime framework gathers and interprets both functionality descriptions exposed by simple services and the capabilities exposed by service factories. As presented in chapter 8, descriptions in our language are compiled and merged to be executed on a rule based system. Transformed into rules, these descriptions make it possible to answer functionality discovery queries issued by client applications. Our framework includes reasoning about factories and particularly about composing factories which makes automatic service composition possible.

9.2.2 Transitional Technologies: Why it Matters

Our work represents the convergence of three domains:

- Intelligent environments and pervasive computing as it is our domain of application and involved specialists make our target audience.
- Software engineering methods favoring sharing, reuse and dynamic interoperability.
- Automatic service composition of semantically described services.

helping transition to new methods

One aspect of our contributions spans across the three domains above. We studied, improved and adapted existing technologies to make them more adapted to the various specialists we target and to their domains. We name these contributions "transitional technologies" as explained hereafter. Transitional technologies are bridges from actual practices and habits to state of the art technologies and methods.

XHTML goal

The term of "transition technologies" we propose is originally inspired by transitional XHTML. We can describe XHTML as an evolution over HTML: it transforms HTML to make it conform to the XML syntax. Parsing HTML is complicated and contains many particular cases. On the contrary XHTML being an XML dialect, it can be parsed using any existing XML tool and library. Two versions of XHTML were created:

- XHTML 1.0 Strict: a "pure" version where document structured is restricted (e.g. no presentation elements are allowed);
- XHTML 1.0 Transitional: an easier to adopt version with less restrictions but still using an XML syntax.

transitional XHTML

When creating XHTML, the W3C (World Wide Web Consortium) could have created only the "strict" profile and dropped the non-perfect "transitional" profile. Without the transitional profile, switching to XML and switching to a stricter document model would have been tied together, preventing easy migration to XML.

measuring adoption of transitional technologies

The importance of transitional technologies have already been measured for the case of XHTML, in a non-academic publication (http://nikitathespider.com/articles/ByTheNumbers/fall-2008.html). The web page [Url-r] gives various statistics about HTML formats in use. The article highlights the importance of transitional technologies as shown in the following quote ("doctype" being the XML abbreviation of "document type"):

> *Interestingly, transitional doctypes (both XHTML and HTML) dominate their respective fields.*

evaluating impact of transitional technologies...

In these statistics, transitional XHTML represents 48% of the pages and strict XHTML only 32%. From an adoption point of view, transitional XHTML is one and half times more successful than strict XHTML. The important question is what scenario would we have in the absence of transitional XHTML:

- there would be 80% of the pages in (strict) XHTML: this extreme possibility would blame transitional technologies as slowing down the adoption of the "perfect" technology,
- there would be less than 32% of the pages in XHTML: this possibility would mean that transitional technologies help even the adoption of the core technology,
- there would be between the 32% and 80% of the pages in XHTML: this most varied and most probable possibility places a cursor between the two.

. . . would require parallel worlds

The hypothetic scenario of the absence of transitional XHTML is impossible to simulate which makes it impossible to choose between the 3 scenarios. Our guess is that, in the absence of transitional XHTML, XHTML use would be close to the current use of strict XHTML.

example: programming languages. . .

The transitional requirement explains why C++ is so close to C, why Java and C# syntaxes are close to C++, etc. Whatever their intrinsic qualities and advantages, technologies that are too different from current practice have only a tiny chance of being widely adopted. We can use the example of C++ to illustrate a common trap when creating transitional technologies. The C++ language is an example of transitional technology as it is build upon a C-like syntax and introduces new concepts of object orientation. From a transitional point of view, the defect of C++ is to be almost a strict superset of its predecessor: the "worst" C++ program will not be more object oriented than a C program. C++ fails in the sense that it introduces new object oriented concepts but does not ensure or favor their use from a developer in transition from C.

. . . with partial success

objective: adoption and best practices

Our work places itself at a transitional level: we extract most important aspect from existing technologies and make them acceptable by our target audience composed of specialists in various domains. In the design of our method, tools and language, we made the effort to be both an acceptable step for the specialists and sufficiently constrained to ensure good practices.

9.2.3 Difficulties of Evaluations in Pervasive Computing Engineering

absence of standard benchmarks

The field of intelligent environments is by itself at the confluence of multiple domains. Each individual domain may be mature and may have standardized evaluation methods but there is no common method for the evaluation of contributions around intelligent environments. Advances in intelligent environments are exposed in the form of theories and methods with some practical illustration of effectiveness and applicability: no benchmark or quality measure is used to compare systems for intelligent environments. Applications are often complicated and rarely resemble each other. In such intelligent environments, major tasks still needs to be found and defined.

what can be easily evaluated

In the domain of software engineering, certain aspects of an approach can easily be evaluated but others are very difficult to evaluate. This duality is the same concerning one practical aspect of software engineering: frameworks and middleware. Operating performances of the systems are the part that can be easily measured: building systems and benchmark that measures performances (execution time, memory, bandwidth, etc.) is relatively easy. The more complicated aspect of evaluation is how to assess the usability of a software engineering method. We have studied this problem in the particular context of software engineering for pervasive computing.

evaluation of architectures

In our work, we propose a design method, service oriented middleware and tools, description language, reasoning and composition runtime to favor the collaborative implementation of good software architecture by developers with highly different backgrounds. We aim at obtaining good sharing and reuse of the production from individual specialists and also at favoring dynamic runtime integration and interoperability of elements from different systems. State of the art methods for the evaluation of system architecture concentrate on the evaluation of architectural decisions. As an architecture and an architectural decision cannot be separated from the context of this decision, these methods are based on scenarios written down by the stakeholders of the considered project. These methods aim at measuring, given the context formed by these scenarios, different software qualities provided by the chosen architecture: modifiability, extensibility, security, performance, time-to-market, etc.

possible evaluation protocol. . .

We have proposed a design method for non-experts in software architecture, by giving guidelines about their day-to-day implementation tasks. Inspired by evaluation techniques for software architectures, we imagined an evaluation method in 3 steps:

- identify representative use cases involving the design of an application for intelligent environments (such as building an extensible automatic meeting recorder)
- have it implemented by experimentation subjects using either our method, an alternative method or no method (user's free choice)
- evaluate, using existing architecture evaluation methods, architectural choices made by the experimentation subjects

... with usability studies

With such experimental settings, we could evaluate a design method. This approach is the one used for most usability studies: knowing the quality of a tool or method involves testing with a statistically meaningful and representative set of users. Methods can be compared by evaluating the quality of the results produced by target users with different methods. This evaluation process is ideal for most usability studies and is heavily used to evaluate contributions on Human Computer Interaction. In our context, user studies are unfortunately hard to organize as we will detail below.

users are rare...

User studies require to have a sample of users that is sufficiently big to be statistically meaningful. The target audience of our method is highly specific and requires skills in at least one of the domains involved in intelligent environments conception. This rareness of eligible candidates for user studies is a considerable obstacle to put the evaluation method in practice.

... and experiments are long...

The reservoir of available candidates is even more restricted due to the next particularity of our domain. The kind of tasks we want to evaluate involve modeling and design activities: they are long, require important concentration and let an important space for creativity. These specificities of the design task makes it impossible to do an experiment in a reasonable user time such as 30 minutes or an hour per person. This is manageable, but still difficult, to find people that will give 30 minutes of their time to do "cool" interactive experiments. In our context, how many would give a few days for software design experiments?

... for complex tasks

An interesting usability study, with some problems close to ours, has been conducted by García-Barriocanal in [García-barriocanal 2005]. They have evaluated ontology editors for knowledge authoring. To make the evaluation possible they had to restrict the complexity of the task to the simplest possible cases. It is only with this restriction that they were able to keep experimental sessions under one hour and to find enough people for experimentations. The restriction they made is acceptable: they evaluated the simple use case but that is common to any other use case. Because of their restriction, they cannot draw conclusion about usability of the software after the first few minutes or hours of use.

restricting tasks is difficult

We imagined restricting our experiments to a simple case. As we want to evaluate the properties and quality of architectures created using our design method, any restriction would be problematic. We cannot honestly evaluate toy architectures that would be produced in a few minutes by persons with almost no contextual informations. A software learning curve can be cut in the middle to be evaluated in parts but it is far more difficult to stop a software architecture in its early creation and evaluate it.

solution: "student studies"...

We found only one direction for such user studies but we could not put it into practice due to the involvement and the particular conditions it requires. We can call this family of evaluation method "student studies". Our solution is to transform students (or persons in formation) into "volunteers" for the experiments. This solution is a mutually rewarding as students would be trained in state of the art concepts (service oriented architectures, semantic web, etc.) while their application project would serve as data for statistical comparisons of a design methods. Setting up such a course and project would obviously require a considerable effort to be both efficient on the experimentation aspect and on the pedagogical aspect. It would be unacceptable to waste a teaching module of an entire promotion of students for scientific purpose. With such a setup, we can gather sufficient data that would be representative of what we could expect from a real situation.

... conducted with care

Even with a setup like student studies (or an equivalent non-affordable solution where all participant would be paid), some tricky aspects remains. We are dealing with methods that requires some training of the participant in the experiment (in the form of lectures for example). From a pedagogical point of view, such training would involve many examples as most software engineering training does: sensibility to software engineering requires experience which can at best be "emulated" by giving examples. The comparison of the results obtained by students using different methods will be heavily influenced by the examples provided in the lectures. The more the examples will be close to the logic behind a given method, the best the mental model of the students will be ready for the application of the method. If no attention is given to this aspect, the evaluation will inevitably be biased. The bias would most probably favor the new method that we want to evaluate. A kind of double-blind approach would be ideal: the person setting up the experiment tells the person giving the lectures that he or she must present two methods, without telling what is the experimental motivation of it.

9.2.4 Evaluations at OMiSCID Level

In addition to the properties of the architectures issued from our method, we can evaluate how our middleware and its new API are fulfilling their objectives. We will give quantitative measurements of fundamental properties of our middleware and some use statistics.

Measurement of Induced Latency

performance is an argument...

One major claim of the initial implementation of OMiSCID was to have good performance and low overhead over a non-service architecture. This argument has been a support for our design method when we recommend to split the system in many reusable parts. We measured this overhead in some controlled conditions.

... e.g. to separate video acquisition from processing

In the systems we presented, we split the video acquisition from the image processing. This is the most critical use case and the most arguable design decision as it requires an additional copy, through our middleware, from the video service to the video processor service. We used this setting with simulated services to improve control over the parameters of the evaluation. To reason about the worst case, we did all benchmarks with the Java implementation of OMiSCID that happen to be the one written with the least concern for performance.

latency measurement: experimental setup

Figure 9.1 depicts the experimental setup with an UML sequence diagram. We want to measure the overhead of our middleware independently from the network so all elements are running on the same computer. This replicates exactly the situation with the video service and an image processor.

- Messages are emitted at a fixed rate, we show results with 25 Hz (the rate of the camera). Comparable results have been obtained at both 100 Hz and 10 Hz.
- Messages have a size that we will change during the experiment. We vary this parameter from 0 bytes to more than 3MB. A typical camera image weights around 350kB (384x288 with 3 channels).
- The thread that creates the message sends it simultaneously with two methods. Firstly, it posts the message into a queue (a standard Java ArrayBlockingQueue) that is consumed by another thread running in the same process. Secondly, the message is sent using OMiSCID to a service running in another process. We measure the time difference between the reception by the local thread and the OMiSCID client.
- Each latency measurement is repeated 100 times. Average value and standard deviation are show on the graphs presented hereafter.

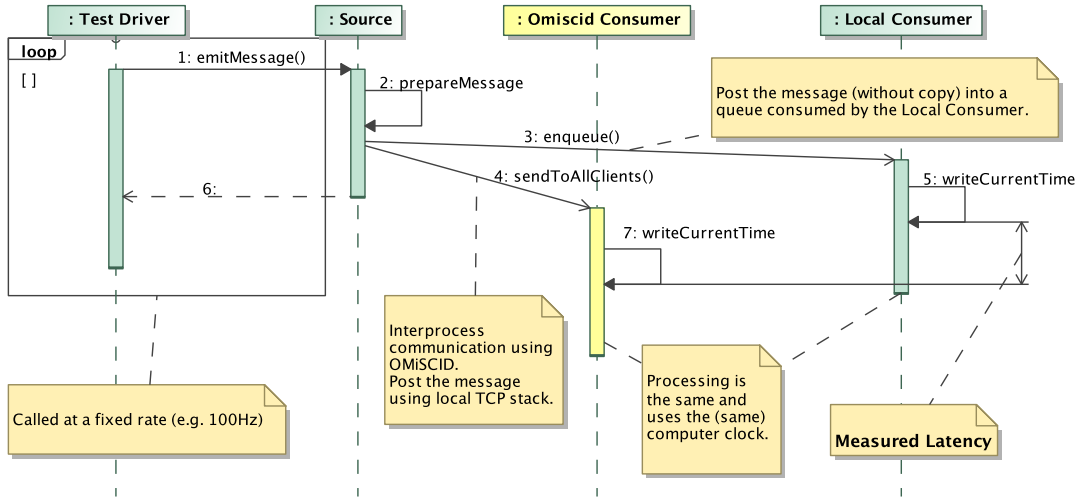


Figure 9.1: *UML Sequence Diagram of the Experiment on the Latency Induced by OMiSCID. An OMiSCID process (in blue) sends messages both to another thread in the same process and to another process (in yellow) on the same machine using OMiSCID. The objective is to measure the latency induced by the use of OMiSCID for local video transfer.*

results with image size packets...

Figure 9.2 show results from the measurement of the latency involved by OMiSCID usage. In this figure, the range of message sizes is limited to a 400 kB, a little more than typical image size we use (around 350kB). Average value and error range (standard deviation) are shown for each measurement.

... latency < 4ms

These results show that the additional latency induced by the use of OMiSCID to transfer images is of 2 milliseconds in average. We can consider that the induced latency stays mostly under 4 milliseconds. Normal distribution tells us that less than 16% of the values are above the threshold of the average plus one standard deviation. This threshold is rarely above 4 ms in our measurements.

with huge packets...

Figure 9.3 shows a wider range of message sizes. The graph exhibits some strange "discontinuities" in the latency. These discontinuities are repeatable and machine dependent. With the most recent test machine of the two, we only note a step when messages grow from 1.5MB to 2MB. This effect is probably due to some optimizations in memory management of the Java virtual machine or in the TCP Stack.

... perturbations

Still visible in figure 9.3, the behavior of our slowest machine is surprising, and also repeatable. If message size is increased over 900kB, latency jumps from a few milliseconds to 50 ms. A second jump is visible at 1075kB where the latency reach 200 ms. Latency jumps back down to the expected value for message above 2.1MB. These observation are difficult to explain as they probably have complex and interfering causes: we suppose a phenomenon of TCP saturation is responsible for the increase for the increased latency and an full computer saturation causes the return reasonable latency (all processes, including the message producers are equally slowed down).

acceptable in normal operation

Our measurements tell us that we can safely send images at video rate with a reasonable additional latency around 2ms. The results also warns us for the case of bigger messages: important latencies can make some applications totally unusable. The most important lesson learned from the surprising measurements is that we should profile a computer if we plan to send big messages.

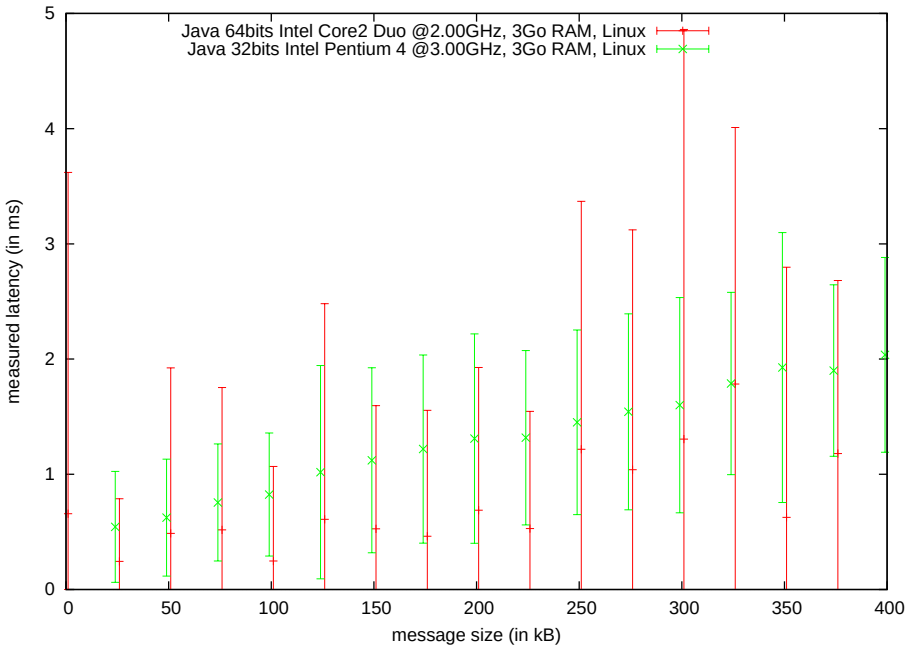


Figure 9.2: *Latency measurements results: messages of normal size.*

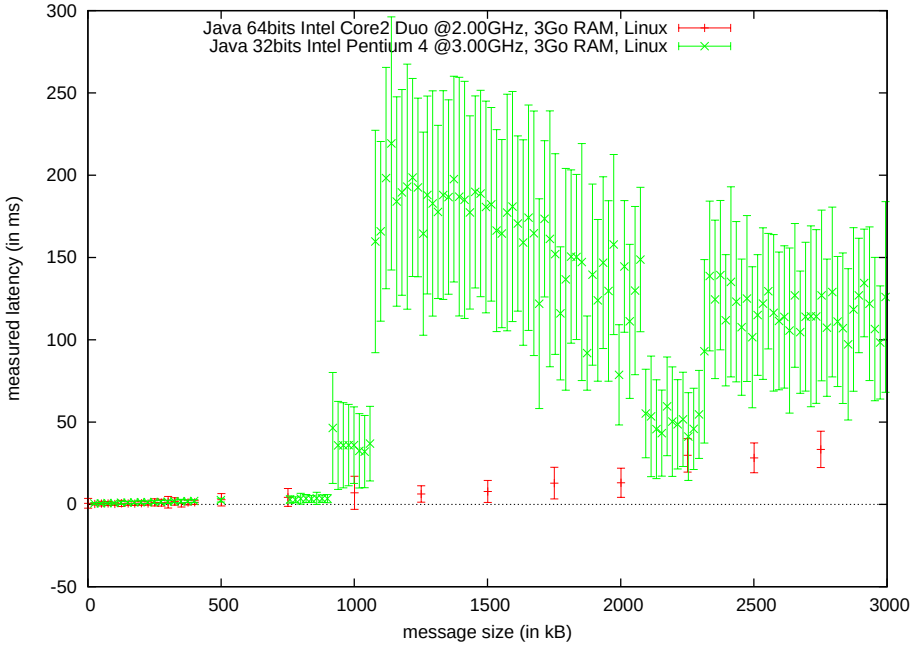


Figure 9.3: *Latency measurements results: big messages.*

Clarity and Verbosity of the User-Oriented API for OMiSCID

user-oriented API...

In chapter 5, we gave an insight of the new OMiSCID API and gave some numbers to compare it to the old BIP API. To cover most aspects of OMiSCID functionalities, we gave some facts about the effort needed to declare a simple service: the declared service should have one variable and one connector, it looks for another service and connects to it. We can recall from chapter 5 what this use case involves:

- Using BIP
  - 6 BIP classes with their full implementation details, most of them requiring to be instantiated and managed by the developer,
  - 25 different methods names involving all the nifty-gritty details of BIP implementation
  - $\Longrightarrow$ 45 lines of codes purely induced by the use of BIP.
- Using new OMiSCID API
  - 2 classes and 4 "concepts", the total number is the same as for BIP but the complexity is greatly reduced and the abstraction higher,
  - 10 method names,
  - $\Longrightarrow$ 15 lines of easily maintainable code purely induced by the use of OMiSCID.

... favors good practice

The objectives of the new API are to maximize usability and favor good design practice. To meet these objectives, typical code must be easy to write and to understand. In addition, it must be hard to do things wrong: ideally it must be almost impossible to do thing wrong. The least requirement is that it must be simpler to do things right than wrong.

detailed code analysis...

One fundamental aspect for an API is to produce code that is both concise and easy to understand. Figures 9.4 and 9.5 contain some code written against BIP and the corresponding OMiSCID version. Elements enclosed in green solid lines are all information that the designer need to express. Elements enclosed in orange dashed lines are meta-information that we consider necessary to provide with the raw design information. All the code that is not in a box can be considered pure "useless" technical code induced by the API.

exhibit short but readable code...

The code produced using the new API is clearly shorter. Comparing both versions side by side puts a spot light on the improvements of the new API: the code is shorter but it is also far easier to understand. In our case, the code reduction implies an increased understandability. This relation is not systematic. Using 2-letter abbreviations for all API methods would also had reduced the code size but understandability would have been cut down.

... with explicit semantic

The code supports our argument about understandability. Going from figure 9.4 to figure 9.5, we notice no major diminution of orange dashed boxes (green boxes are inputs that *must* be specified). These oranges boxes constitute the glue that gives a semantic to the green boxes. The new API ensures that overall code is smaller but that no concession is made on this glue code that provides understandability.

important OMiSCID adoption

Almost all members of the team are now using OMiSCID to communicate between their modules. We measure an increase in the number of available services. This cause more and more contributions to be used and reused by other team members. Figure 9.6 depicts the evolution of the number of Java and C++ source files using OMiSCID in the code repository of the team. The number of such files increases regularly. This increase illustrate the fact that more and more services are designed and made available to the team. The relative amount of source files using OMiSCID remain almost constant between 20% and 25%: this stability reflects the continuous sharing of new projects as services.

important GUI adoption...

As far as the graphical user interface is concerned, it is systematically used by all service

```
class MovieMaker {
  ControlServer ctrlServer;
  TcpServer commandServer;
public:
  MovieMaker() : ctrServer("MovieMakeService") {}

  bool StartExample() {

    ctrlServer.SetStatus(ControlServer::STATUS_BEGIN);

    commandServer.SetServiceId(ctrlServer.GetServiceId());
    commandServer.Create(0);
    commandServer.SetCallBackOnRecv(MovieMaker::Callback_Receive, this);

    InOutputAttribut* ioattr;
    ioattr =ctrlServer.AddInOutput("command", commandServer.Cast(),
                                   InOutputAttribut::IN_OUTPUT);
    ioattr->SetDescription("Use to send commands to MovieMaker\n");

    VariableAttribut* va =ctrlServer.AddVariable("current source");
    va->SetValueStr("...");
    va->SetType("string");
    va->SetDefaultValue("");
    va->SetAccessRead();

    if (ctrlServer.StartServer()) {
      ctrlServer.StartThreadProcessMsg();
      ctrlServer.SetStatus(ControlServer::STATUS_INIT);
      return true;
    } else return false;
  }

  TCPClient tcpClient;
  char* hostname;
  int port;
  void SetAddr(const char* addr, int e_port, int port_control) {
    port = e_port;
    //portControl = port_control;
    if (!addr) {
      hostname = NULL;
    } else {
        if (hostname) delete[] hostname;
        hostname = new char[strlen(addr)+1];
        sprintf(hostname, "%s", addr);
    }
  }

  boolean FindAndConnectExample() {
    tcpClient.SetAddr(NULL, 0);
    tcpClient.SetServiceId(serviceId);

    CWaitForServices* waitForServices = new CWaitForServices();
    int index = waitForServices->NeedService("AudioRouter"
                                   "_bip._tcp", AudioRouterClient::IsValidForMe, (void*)&tcpClient);

    if (index != -1){
      waitForServices->WaitAll();
      if (!(*waitForServices)[index].IsAvailable()) {
        fprintf(stderr, "Error in wait: not resolved\n");
        return false;
      } else {
        tcpClient.ConnectToServer(addr, port);
        tcpClient.SetCallbackReceive(MovieMaker::CallBackReceive, this);
        return true;
      }
    }

  static bool IsValidForMe(const char * fullname, const char *hosttarget, uint16_t port,
                           uint16_t txtLen, const char *txtRecord, void * UserData) {

    MovieMaker* mm = (MovieMaker*)UserData;
    ControlClient ctrl_client(mm->tcpClient.GetServiceId());

    if (ctrl_client.ConnectToCtrlServer(hosttarget, port))
      if (ctrl_client.QueryGlobalDescription())
          for (ctrl_client.GetOutputNameList().First();
               ctrl_client.GetOutputNameList().NotAtEnd();
               ctrl_client.GetOutputNameList().Next())
            if (ctrl_client.GetOutputNameList().GetCurrent() == "AudioForMovies") {
              InOutputAttribut* ioa = ctrl_client.QueryOutputDescription("AudioForMovies");
              if (ioa) {
                mm->SetAddr(hosttarget, ioa->GetTcpPort(), port);
                return true;
              }
            }
    return false;
  }

  void CallBackReceive(MsgSocketCallBackData* data)
  {...}

};
```

Figure 9.4: *API Comparison: example code using before OMiSCID. Green boxes: information that the designer need to express. Orange dashed boxes: glue code that gives a semantic to green boxes.*

```
using namespace Omiscid;

class MovieMaker : public ConnectorListener {

  Service *service;

public:

  bool StartExample() {

    service = Factory.Create("MovieMaker");

    if (service == (Service*)NULL) {
      throw SimpleException( "Unable to create service" );
    }

    service->AddConnector("Command", "Use to send commands to MovieMaker", AnInput);
    service->AddConnector("AudioInput", "Input of all audio", AnInput);

    service->AddVariable("current source", "string", "Sources listes", ReadAccess);
    service->SetVariableValue("current source", "...");

    service->Start();
  }

  boolean FindAndConnectOnceExample() {
    ServiceProxy remote = service->FindService
                                          And(
                                              NameIs("AudioRouter"),
                                              HasConnector("AudioForMovies", AnOutput)
                                              ));
    service->ConnectTo("AudioInput", remote, "AudioForMovies");
    service->AddConnectorListener("AudioInput", this);
  }

  virtual void MessageReceived(Service &TheService,
                               const SimpleString LocalConnectorName,
                               const Message &Msg)
  {...}

};
```

Figure 9.5: *API Comparison: same code as in 9.4 but using the OMiSCID API.*

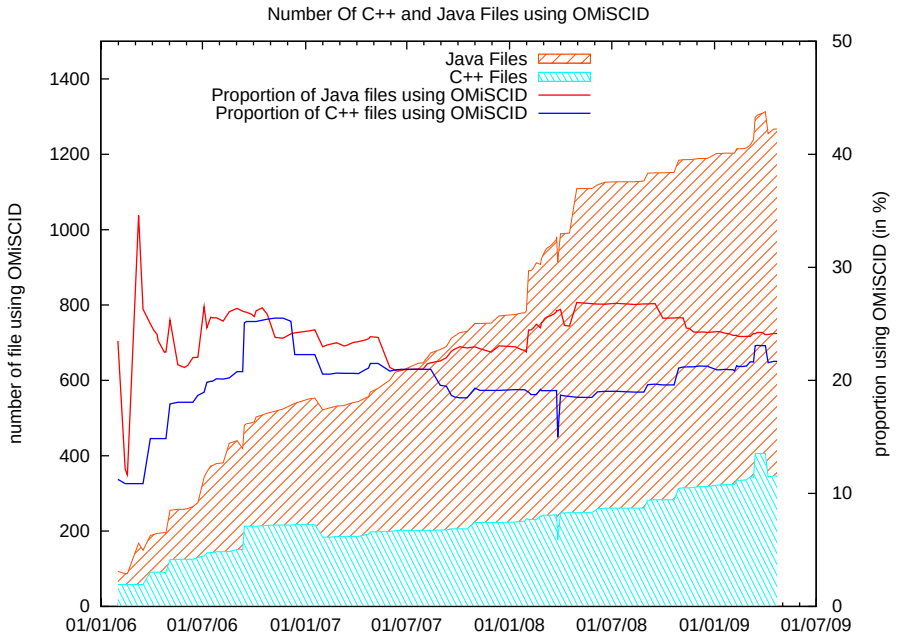


Figure 9.6: *OMiSCID Adoption. Absolute and relative number of source files using OMiSCID. In blue: C++ source files. In red: Java source files.*

designers and consumers. Most of the service designers also develop views for their services as plugins for the GUI. An interesting phenomenon is that the automatic update system provided by the GUI is a driver for integration. Developers get used to installing and updating plugins in an automatic manner. The desire to take advantage of this easy way of sharing executables has dragged some applications to be developed as plugins for the GUI.

... and re-contributions

To measure this emulation, we made some statistics on the code base of all GUI extensions that is stored under the source code management tool "Subversion" (see [Pilato 2004] for a complete introduction). We measure the size of commits "diffs", i.e. the number of changed lines between two revisions. The graph in figure 9.7 plots the cumulated diff size. Two values are plotted: total cumulated size and cumulated size of commits made by other people than the creator of OMiSCID Gui (and author of this manuscript).

detailed analysis of contributions

On figure 9.7, we observe a relative increase of the proportion of contributions by other team members. Not show on the graph, we also observe an increase in the number of contributors. The upper part, made of contribution from the author, includes maintenance on files checked in by other persons. A script is run on the projects to make them public and available as automatic updated in the GUI. This script automatically generates a release number and updates some headers and configuration files in the project. Due to access restriction on the publishing server, this script is mostly run by one person, contributing artificial increase in the cumulated commit size.

overall success of OMiSCID

We made OMiSCID as useable as possible. Our objective is to have it adopted first by all persons working in team and then by other persons. We can consider that adoption in the team is a real success for both its API and the GUI. From a wider view angle, the partners in some of our research projects have also used OMiSCID but no real spontaneous dissemination has taken place. Working on communication could improve this dissemination, for example by releasing real demo applications to market the middleware and the GUI.

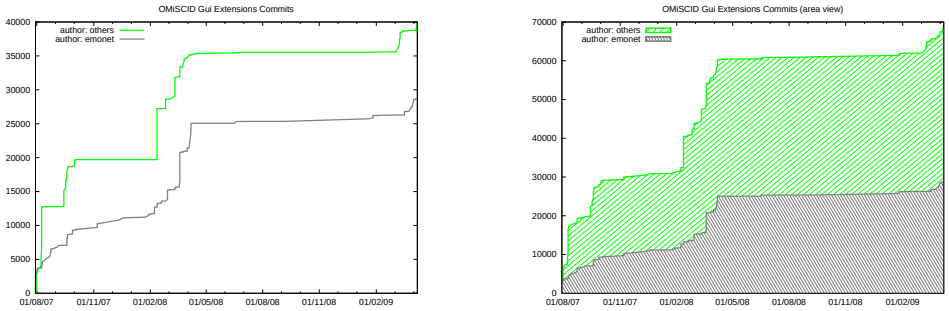


Figure 9.7: *OMiSCID Gui Adoption: commit size of extensions. Cumulated size of the "svn commits" done on the code base of OMiSCID Gui extensions. Commit of the Gui author and of the rest of the team. On the left: raw values; on the right: stacked values.*

9.2.5 Evaluation of our Design Method

In previous section, we evaluated OMiSCID and its adoption. In this section we analyze the properties that our design method implies on systems.

analyzing redesigned systems: tracker and cameraman

Having examined how we could evaluate a design method if were possible to put it in practice, we still evaluate our work as possible. A classical evaluation of middleware and design method is to show how we can build systems using the proposed method. This is the case for various works on middleware such as [Escoffier 2007b] and also with service composition as mentioned in [Brønsted 2007] We already gave some examples of systems designed with our method and tools: when we introduced the method in chapter 6 and the language in chapter 7. The two systems that we reimplemented using our method and language are a 3D tracking system and an automatic meeting recording system. In this evaluation section, we will study and illustrate the properties that the systems gained (and lost) in the reimplementation process. Apart from this analysis, we will do present some heuristic measurement about the usability of the tools we propose including the new service oriented middleware API, its extensible graphical user interface and the proposed language for semantic descriptions.

fragmentation improves dynamicity and sharing

Our design method makes it possible to describe and implement a 3D visual tracking system as presented in chapter 6. Figure 6.6 (page 99) depicted the new architecture issued from our method. This new architecture heavily uses semantic service functionality descriptions and service factories. We can start looking at the properties of this new architecture. The new architecture is more fragmented as we separated the 2D image processing from the 3D to 2D coordinates projection from the 3D tracking component. This separation, together with the abstraction of functionalities provided and required by different components, makes it possible to add and replace different elements without modifying the others and possibly during the execution of the system. This ability to handle dynamic environmental conditions is one of the most important requirements to bring pervasive intelligent computing to life.

dynamic extensibility using functionalities

We can illustrate the improved dynamicity and adaptability by demonstrating an integration of a new 3D detector based on a priori geometric information about the environment and statistical room usage obtained from machine learning. With the previous monolithic architecture, such information would probably have been added by modifying the 3D tracker component to cope with this knowledge. Using the new architecture, this information can be provided as a 3D detector that can transparently be used by the 3D tracker component without modifications. This added dynamicity comes from the plain use of services and from the proper split of our system. No functionality descriptions are needed as we add a new detector designed after the system itself. Figure 9.8 illustrates this dynamic detector addition together with a second one that makes use of semantic service descriptions. In the figure, the added "Room Occupancy Statistics" service directly implements the functionality required by the 3D tracker.

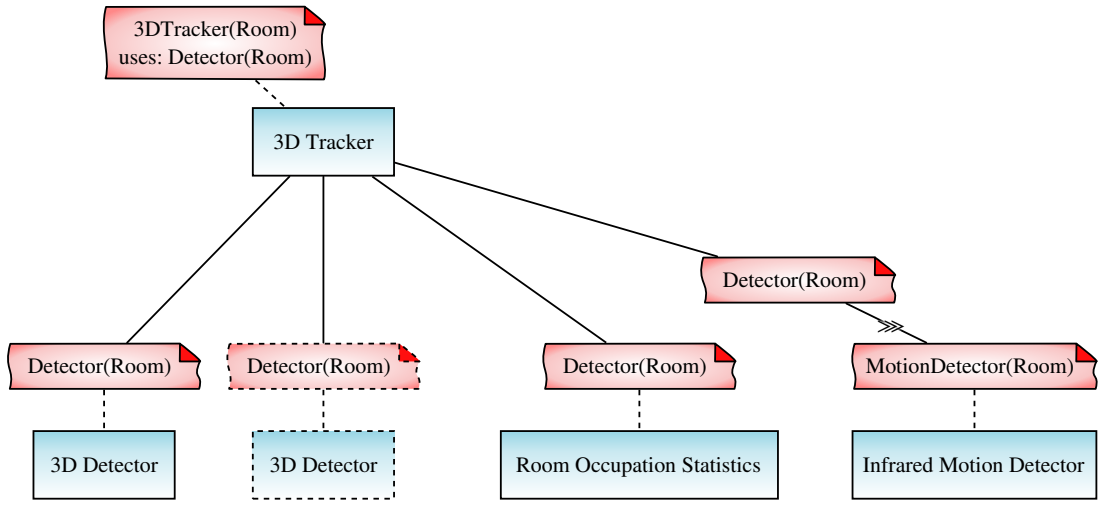


Figure 9.8: *Dynamic Addition of 3D Detectors to the Tracking System. Extending the system with a newly built detector (Room Occupancy). Making the system integrate an existing service (Infrared Motion Detector).*

dynamic integration using correspondences

The second integration scenario, illustrated by figure 9.8, uses functionality descriptions and correspondences. We integrate an existing "Infrared Motion Detector" service into the tracking system by expressing a correspondence between its "MotionDetector" functionality and the "Detector" one used in the 3D tracking system. As the infrared motion detector already exists, it is virtually impossible to modify its behavior as other services and applications might still be using it. By using functionality correspondence here, we keep backward compatibility with existing services. We also do not need to modify the motion detector service for which we might even have only binary code. The functionality correspondence makes it possible for the 3D tracker to know that it can use the infrared motion detector. To actually use it, a protocol adapter is required. This protocol adapter can be provided at the same time the functionality correspondence is, in the case the correspondence is used directly as in the example.

apparently complex system...

The system fragmentation improves its awareness to dynamic changes but can make it more complex to deploy and understand. At first sight, as illustrated in figure 9.9, the system after our redesign is more complex. This system complexity, as perceived by a component developer, is a real drawback of our method as it may frighten people from adhering to the method and adopting it. This perceived complexity, however, eventually helps system understanding for the one who gets over first appearance.

...due to explicit configuration (blue)...

Most of the services and descriptions added in the architectures are configuration elements that have been materialized and made dynamic. This is the case of camera calibration and 2D image-based detector configuration. Factories have also been used at this configuration level to produce derived configurations and interpret this new form of configuration. All the configuration elements and their derived products are circled in blue dotted-line in figure 9.9. These elements were already present in the original system either in a configuration file or embedded in raw source code.

... and factory products (red)

Intermediate elements that factories generate from exposed descriptions are circled in red dashed line in figure 9.9. These elements are direct consequences of the other descriptions and thus add no comprehension overhead once the rest is understood. They also have no impact on system deployment as their instantiation is automatic and transparent for the developer.

remains two factories...

Eventually, the remaining "architectural overhead" introduced by our method is only composed of the service factories and the separation of the image processing from the video acquisition. The "Composite Transformation Factory" is a simple extraction of the knowledge of how

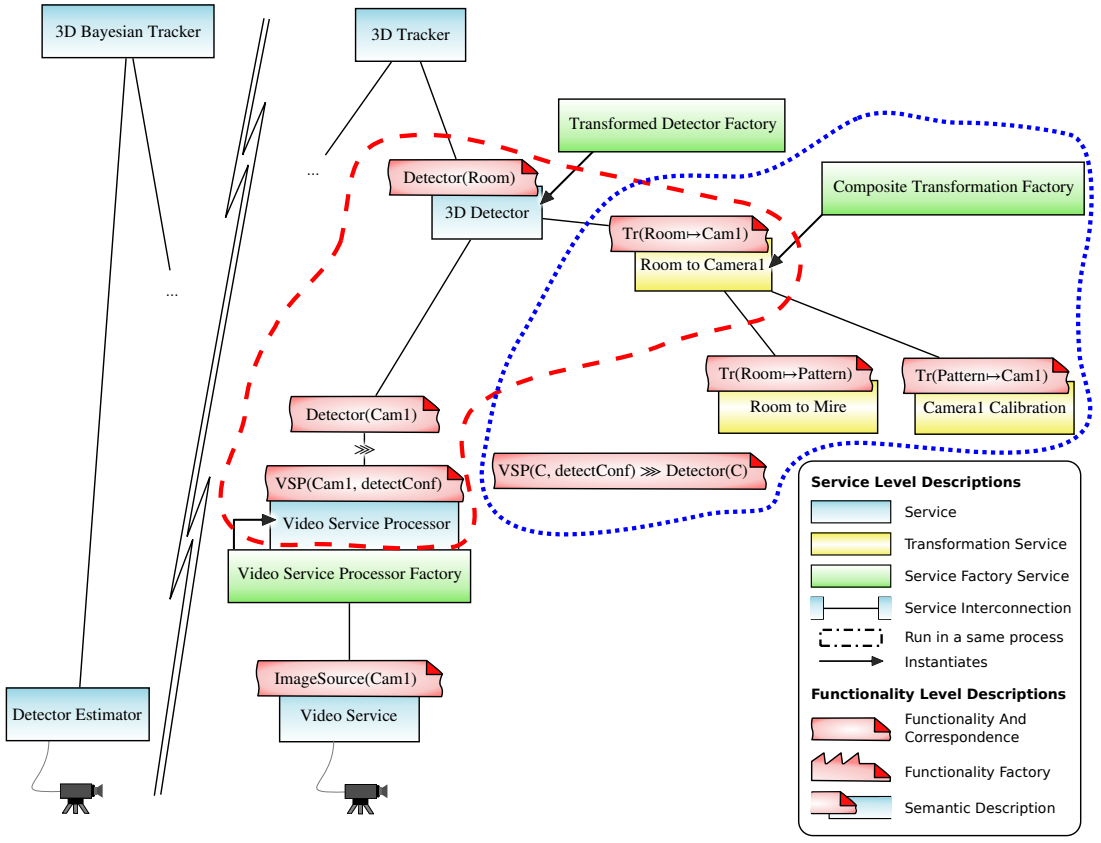


Figure 9.9: *Perceived Architectural Overhead: old and new tracker, side by side. Blue dotted line area: explicit configuration elements. Red dashed line area: product of services factories.*

to compute calibration: this knowledge was previously encapsulated in the calibration software, and was only used at configuration time, not shared for runtime reuse. In the same manner, the "Transformed Detector Factory" is an extraction from the 3D tracker of the knowledge of how to pass from 3D space to 2D space. This factory is the result of the complete separation between the 3D tracking and the detection at 2D level. This separation was necessary to make possible the dynamic addition of new and foreign detectors as presented previously in this section. The "Video Service Processor Factory" materialize the possibility to share a camera among several image processors.

... reflecting capabilities of the environment

Far from adding clutter to the architecture, factories simplify dynamic reuse of code and even stimulates this reuse and the interaction in the research group. When no 3D tracker is running, what does a developer see?

- With the old architecture: nothing, as the camera must be released in case someone wants to use it.
- With our architecture: the video services representing the cameras, the factories expressing their capability to instantiate image processing services and all of the necessary information about camera calibration (if calibration has been performed).

These latent services are reflecting the real functionalities, hardware and software, provided by the environment. Having this view of what other developers provide automatically favors the reuse and interactions among team members.

need for a visualization of the classification

With proper identification of the role of each element added in the design, the initial perceived complexity can be easily understood. At the time of this evaluation, no particular mechanism has been implemented to automatically visualize the equivalent of the dashed and circled areas in figure 9.9. We can state that, without this automatic visualization, understanding a running architecture requires significant efforts. To be useable, our method must not be overly complex. From this point of view, additional improvements are required for first contact with an existing system.

same gain for the cameraman

The automatic cameraman, rearchitectured using our method, also inherits from the properties that our method added to the 3D tracking system:

- Improved dynamic behavior: splitting the system incites the developer to handle dynamic appearance and disappearance of used services. This makes the development, testing and deployment life cycles less constrained, different elements being started and stopped independently.
- Improved adaptability: as important parameters and configuration of the system has been materialized, they can be provided at runtime, even when the system is running. Adapting the cameraman to new environmental conditions can be done without any modification by describing what perception service plays what role.
- Improved extensibility: split of the system and abstraction of each part in term of provided functionality makes it possible to implement this functionality in a novel way. With functionality level abstraction, this adaptability can reach a level often considered as extensibility. The automatic cameraman can, for example, use an alternative way to detect slide changes, for example by using some software to monitor the state of the presentation player or by using a manual backup solution where someone clicks on a button when slides are changed.
- Possible dynamic integration: integrating the automatic cameraman with another already implemented system is simplified by the functionality layer. A plausible use case is the installation of the automatic cameraman in a room already equipped with a localization system developed by someone else for another purpose. With functionality correspondences and protocol adaptation, the integration of the two systems does not require any of them to be modified.

- Improved service reuse: both the split of the system and the use of factories improves potential service reuse. Splitting the system improves the self containment and cohesion of individual services. Service factories expose at any time the entire family of services we can instantiate. Compared to a situation where only complex in-use services are exposed, seeing the range of available functionalities favors reuse.
- Increased complexity: our method makes the architecture more complex to understand at first contact. This increased complexity is currently an important drawback of our method even if some communication actions can improve the situation.

9.2.6 Evaluation of UFCL, the Language

In previous section, we evaluated the impact of our design method on the redesigned systems. We will now give some evaluation about UFCL, our language for semantic functionality description.

simplicity vs expressiveness

Designing a description language is always a compromise between expressive power and tractability. We designed the language metamodel so as to make reasoning possible while making it possible for the designers to express both functionality correspondences and services factories. This makes it possible to do integration between foreign designs and to reason about possible compositions and possibly instantiable services.

language evaluation criteria

Criteria to evaluate description languages are numerous. In the context of programming languages, these evaluation criteria are usually surveyed in computer science courses such as in [Url-s] and [Url-t]. We base our language analysis on these criteria and see how they fit our particular context. Programming language evaluations used a set of criteria, most commonly:

1. *Readability* measures how easy it is for a programmer to read and understand programs. This criterion is of capital importance as it has a direct impact on all the others. Readability is not limited to syntax considerations as we will see.
2. *Writability* is the equivalent of readability but for writing programs. An overly verbose language would be highly readable but hardly writable by hand. On the opposite, languages using many abbreviations and implicit syntax are easy to write, harder to read (e.g. Perl).
3. *Reliability* measures what properties are ensured by the use of the language and how the contribute to the reliability of the implied program. The more compilation check the language will make possible, the more reliable the language will be considered. As reliability depends also on the ability to avoid writing bugs it is highly influenced by writability and readability, through code reviews.
4. The *cost* induced by using the language is another important criterion. This criterion is the main refusal criteria invoked and is what diminishes popularity and adoption of languages. The cost criterion depends on previous criteria but also on other aspects. Language cost is made of cost of various sub costs: training programmers, writing programs, buying and using tools such as editors and compilers, hardware and time for running tools and program, doing corrective maintenance and evolutions.

different facets of UFCL

Different aspects of language influence the presented quality criteria of a language. We will enumerate these aspects. For each aspect, we evaluate the criteria to which it contributes and we position our language with regards to this aspect. We use numbering to reference the 3 first criteria presented above. Cost is a more synthetic criterion, so we treat it in a synthesis paragraph below. First item of the following list is an syntactic example:

- *e.g. Numbering (1, 2): we use numbers to reference criteria, so this example aspect contributes to criteria 1 (readability) and 2 (writability) as numbers are simpler to write and also read (remembering the number-criteria association is simple in this case).*

- *Simplicity* (1, 2, 3): our language is dedicated to a particular task and the number of features are highly limited. Our language is simple as it only gives a single way to express constructs: functionality facet, functionality correspondence, service factory.
- *Level of Abstraction* (1, 2, 3): our language is used as a high level of abstraction. Both functionality correspondences and service factories descriptions constructs are highly declarative and describe high level mechanisms. Given its relative simplicity, our language has a well chosen high level of abstraction. On the contrary to other languages such as BPEL, we do not try to build a new programming language approaching Turing completeness. This level of abstraction is in adequacy with the task targeted by our language.
- *Orthogonality* (1, 2): different building blocks of our language can be easily combined or used in different context. This is the case at multiple levels including the conceptual level with the separation of simple descriptions from factories and correspondence that are all expressed independently but interacting. Our language also exhibits some orthogonality properties through the low coupling between functionality description and grounding expressions.
- *Syntax Simplicity* (1, 2): we made the choice of a textual syntax as we can find in the Structured Query Language (SQL). UFCL syntax uses natural language words to make expressions sound like sentences. This choice should increase the simplicity of the language even if user's background have an important impact as noted for SQL in [Bell 1992]. Compared to existing languages for comparable tasks, such as OWL-S, our language is less verbose.
- *Expressiveness and Syntax Conciseness* (2): our language is SQL-like so some added keywords could be removed to make the language more concise. The overall level of abstraction of our language makes it expressive in spite of these syntactic additions. Our language would be probably considered inadequate by people that does not like SQL.
- *Support for Custom Abstraction* (2, 3): this property designs the ability given by the language to the programmer to build higher level abstraction using the language. On this aspect, our language is not good. The only custom abstraction can come from the implicit semantic of the terms employed by the description writer to represent its functionalities and their properties. Our language provide no way to build higher level abstraction that get additional checking by the compiler.
- *Compile Time Validation* (3): our language is open in the sense that it accepts any terms to be used for functionality and property names. This openness makes compiler checks more difficult when not impossible. Our language makes it easy for the compiler to check simple properties such as the coherent usage of wildcards in a factory expression (we check that only declared wildcards are used). We currently do not support more complex checks and, for example, typographical errors on functionalities names are not automatically detected. This aspect should be improved to ease development and debugging.

UFCL: a good DSL (currently) lacking debuggability

As a summary of the evaluation of these different aspects, we can notice that our language is a good Domain Specific Language (DSL) but lacks debuggability. Our language is good on readability and writability but get poor evaluations concerning reliability. Mapping these evaluations to the cost criterion (which is a major blocker for adoption), our language has an acceptable cost that should not prevent its adoption:

- training programmers should be relatively short as the language is simple and tailored for the task,
- in the same way, writing semantic descriptions should be relatively easy and fast using UFCL,
- writing descriptions requires only a text editor and no dedicated tools,
- compiling and using semantic descriptions uses only open source freely available software but need a normal desktop machine and will not run on constrained devices,
- debugging however will require a significant amount of time, particularly if descriptions are written without care and contains conceptual errors.

9.3 Overall Analysis of the Work

In previous section, we evaluated different aspects of our works in isolation. This section is devoted to the lessons we learnt from our investigation.

From the focused analysis presented in previous chapter and the evaluation in this chapter, we take a step back and analyze our work in its entirety. This analysis is an occasion of doing an overall evaluation of this research work and of reflecting on future directions opened by our conclusions.

9.3.1 Development Methods and Tools: the importance of a complete solution

OMiSCID and its Graphical User Interface in Symbiosis

When we originally developed the graphical user interface (GUI) for OMiSCID services, we thought it would be only a platform used to release modules managing stable and widely used services. Our initial idea was to make it a platform that capitalizes complete software such as shared infrastructure and demonstrations. This initial objective was already a good one.

a GUI designed for demonstrations...

After the release of the GUI and its presentation to the team, colleagues began to develop small extension modules. Contrary to our expectations, most contributions were made of new modules developed to provide manipulation for new services. People favored developing small modules for their work in progress rather than porting their existing graphical interfaces into the GUI. This portage came but it came later.

... used for daily works...

From this point of view, the GUI exceeded our expectation. We expected only a few number of application-like modules to be created but we rather got more small modules. This situation where modules are small and providing elementary functionalities perfectly fits with the service oriented vision.

... against expectations

The GUI and its adoption has followed and supported the advent of OMiSCID and of service orientation in the team. Without the GUI, using OMiSCID would have a lesser perceived advantage and its adoption would be lower.

Convergence of Service Semantic Functionality and the Concept of Tasks in the GUI

Interestingly, we introduced many similar concepts in the service functionality descriptions presented in this thesis and in the GUI extension mechanism (see page 78 for details). These two aspects have been developed in parallel with similar objective but different priorities.

decoupled sharing and reuse

In both the GUI and semantic description we aim at sharing and reusing functionalities without modifying existing services. Functionality descriptions have been designed to allow automatic reasoning for runtime integration of services and require compilation of descriptions. The two main constraints in the GUI were: clean integration in Netbeans and the Java programming language and, extremely fast understanding by the new developer. In the GUI we use selectors and tasks although we use the concept of functionality and correspondences in semantic descriptions.

tasks in the GUI

Figure 9.10 shows the similarities of functionalities and GUI tasks. Each concrete service, in the GUI, might be transformed as any number of "tasks" that each can in turn be processed by any number of actions. The concept of task plays the role of a shared representation: multiple selectors instantiate a particular type of task from various type of concrete services. To a given type of task might also be associated multiple actions. Both new selectors and actions can be added without modifying neither the task nor already existing selectors and actions.

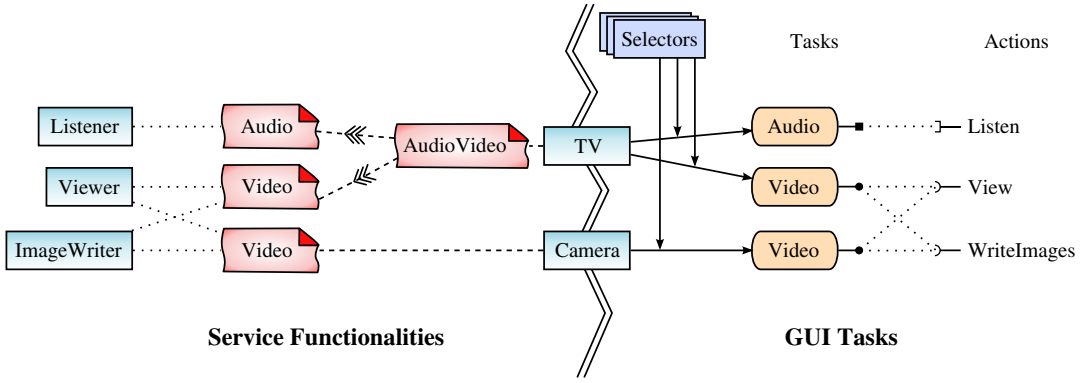


Figure 9.10: *Similarities Between Functionalities and GUI Tasks. Example with audio video services and clients. In the Graphical User Interface, "tasks" are the abstraction: selectors map services to tasks that are required by actions. With functionalities, "functionalities" are the abstraction: inference is done on different functionalities to map from provided to required functionalities.*

functionality correspondences

In the case of semantic service functionality descriptions, we did not want to have a shared centralizing element (such as the "tasks"). Functionalities correspondences and protocol adapters play the role of the selectors in the GUI: they convert concrete services to the required functionality. Having a central representation would not be an option as it limits the openness of the system: it makes impossible to integrate two systems without modifying one of them.

take the best of the two approaches

One lesson we can draw here is that the GUI concept of task was adopted by every contributor while the concept of functionality is harder to understand and to put in practice. From this observation, we can draw some guidelines for further improvement of our work as discussed in section 9.3.2. Basically, "tasks" are a code level materialization of service functionalities and reasoning in term of tasks and task adapters is probably a promising direction. Care must still be taken not to lose the dynamic openness of the systems.

9.3.2 Requirement for Broader Protocol Adaptation

separating protocol from functionality...

In the design of system, for functionality level interoperability, we separated semantic functionality description from protocol description. Functionalities are abstracted from the protocol used to access them, this makes it possible to express functionalities and reason about it more easily. As we mentioned in chapter 6, functionality correspondence is relatively easy to describe but hard to detect automatically and, protocol adaptation description can be tedious and verbose but can be done automatically.

... to allow reasoning on factories...

The separation between functionalities and protocol is supported by the fact that protocol adaptation is already a research field by itself. An interesting discussion about component protocol adaptation can be found in [Reussner 2003]. Isolating functionality descriptions from protocol is what simplified reasoning and made it possible to introduce factories in this reasoning.

... and automatic adapter generation

We initially envisioned to use the separation between functionality and protocol to enable automatic generation of protocol adapter. We imagined basing this generation on multiple approaches: using service unit tests to infer protocols, using machine learning to test and evolve different adapters and using the more classical solution where protocols are highly formalized often in the form of finite state machines. Our investigation took us in a different direction. Interestingly, the recently started ICT-CONNECT research project (Emergent Connectors for Eternal Software Intensive Networked Systems) target the automatic synthesis of connectors between devices (details on their research statement on their web site [Url-u]). They mention modeling and reasoning about functionalities and automatic synthesis of connectors (protocol adapters) in highly heterogeneous systems.

OMiSCID as an abstract API

In our work, we only made a simple protocol adaptation layer based on OMiSCID. One interesting direction to take would be to consider the OMiSCID API as an abstract API hiding protocol but also middleware heterogeneity. With such an approach, OMiSCID user could transparently use the API to access various services and devices, not only OMiSCID services. For example, one can imagine discovering and interacting with bluetooth services or zigbee sensors. Such approach, were a common API is used to access multiple service discovery protocols and multiple communication protocols, is explored in the Amigo Interoperable Service Discovery & Interaction Middleware.

expose protocol adaptation as manipulable information

Current simple protocol adaptation can be improved by taking example from other investigations. We insist on the fact protocol adaptation should remain an information that can be dynamically added into a running system to improve integration at runtime. A consequence of our separation between functionality and protocol is that improvements in protocol adaptation is totally orthogonal to our framework for functionality description and reasoning. A conclusion is that it is profitable to improve protocol adaptation even if there is a risk that the functionalities framework will be modified.

9.3.3 Interleaving Services, Functionalities and Context Awareness

The most important lesson that we have learned from our investigation and that does not appear in previous pages of this manuscript concerns the relation between open service oriented architecture and context aware computing. In one sentence, service declarations, semantic description of service functionalities and context are all interrelated and should be expressed in an interoperable form.

using "locator" functionality...

In a conference article, we applied our method for the design of an automatic recording system from scratch. This design differs from the one presented in this manuscript and it puts an emphasis on reasoning about locations in a building. Interested reader can read [Emonet 2008] for more details. To enable reasoning about location, we had to add a virtual functionality

named "locator" that locates a person in a space. It is then possible, using service factories and functionality correspondences, to express inclusion of a space in one another, etc.

... to represent location context

The information about user location, that we materialized as virtual functionalities, is the base of most context aware applications. Context is made of any information that characterize the interaction of the users and the application (location, time, schedule, activity, mood, etc.). Our analysis is motivated by the example of "locators" and discussions we had with people working on context aware applications. Service discovery needs to be context aware and context includes information about services present in the environment.

context + service discovery

We have a strong link between context awareness and service functionalities. In addition, service functionalities should be the real interface between services: services provides an abstraction over the low level implementation details but this abstraction must be pushed up to the level of functionalities. To enable real dynamic service architectures, services must interact based on their functionalities, in total abstraction of the communication protocol they use. This implies that every service discovery should be based on semantic functionality rather than exact protocol.

semantic + simple service discovery

With such semantic requirement for discovery, low level service description are relegated to simple grounding information and need not to be separated. Our reasoning leads us to the conclusion that context information, semantic service functionality descriptions and low level service descriptions form an indissociable whole.

all need to be integrated

This fundamental question and the lack of early maturity of our runtime has prevented us from pushing it in the team. Our work explores interesting paths for simplifying semantic functionality descriptions and mixing it with service factories. However, contrary to OMiSCID and its GUI, we did not see a direct benefit of inciting team members to invest in semantic descriptions before solving this major issue.

promising directions

Research on context aware middleware is fertile but most of the contributions have no interest in using service oriented methods. State of the art in context awareness uses tuple spaces to share and enrich contextual informations. They rarely take interest in describing software and hardware capabilities, nor in adapting information to realize integration. Our opinion is that one major investigation that goes in the right direction is the work on PersonisAD presented in [Assad 2007] in 2007. This investigation begins to get citation from notable peers such as A. Dey in [Dey 2009]. Our thinking about our problematic has much convergence with the PersonisAD proposal and feeds our "future direction" section below.

9.3.4 Retrospective Conclusions

we promoted OMiSCID

We actively promoted some aspects of our investigation for use in our research group. These aspects are the one that are most mature and the closest to the users (developers). They mainly concern the service oriented middleware and its graphical user interface. As presented in previous sections, we were reluctant to enforce the use of more advanced facets of our investigation as they could have lead the team in a wrong direction. Nevertheless, many of these aspects of our investigation have an important potential and deserve to be put in practice.

we tried factories with success...

We introduced the concept of service factory to an engineer in our team, in the context of 3D tracking. Factories were only introduced at the service level not at the functionality level: a service factory is a service that can instantiate precise concrete services but no UFCL description is attached to it. This introduction of service factories is a success and provides the advantages mentioned previously in this manuscript: ease of deployment, improved visibility of what is available in the environment, possible reuse of services with different configurations, etc.

... but without UFCL

Service factories have proven to be a valuable element of dynamic and reusable architectures.

The always running "video service processor" factories, visible in the GUI by everyone, had their impact on team members. Using factories has became an option for service designers: they spontaneously imagine implementing service factories, for example for audio processing services. Plain service factories have the advantages to be easily put in practice with no concern about functionality descriptions: we clearly separated the factory concept from its semantic level description from the time we introduced service factories.

we should push our design method...

The observation that factories can be first introduced without the semantic description aspect, can be generalized to most of what we propose in our design method in chapter 6. Ignoring functionality is a good incremental and transitional approach that we should have applied before. This application was prevented by our missing of a clear vision on the problem of interrelation between service discovery and context awareness. In the short term, we recommend inciting people to apply the step of our design method without caring about service functionality.

With semantic functionality descriptions set aside, our design method still involves:

- splitting applications and systems in independent services;
- introducing plain service factories when possible;
- abstracting each service not in term of service functionality but at least studying how the service can be made more generic;
- sharing suitable applications as services or even as service factories.

Even if integration and reuse will involve more manual coding than with semantic descriptions, these simple guidelines are still good to be distilled.

... and favor mutual understanding

There is a last major underlying pragmatic observation in our investigation. It transpires when we prone using UFCL as a hub between designers in section 7.1.2 and when we recommend materializing configurations at runtime in chapter 6 and in section 9.2.5 of current chapter. There is a major need for better communication and mutual understanding between people designing system parts for pervasive computing.

... by improving intelligibility...

Cooperation should not be limited to exchanging black box pieces of software. To make the vision of dynamic and spontaneous interoperability possible, all contributors must share their knowledge and make it available at runtime. Exposing configurations as knowledge present at runtime makes system more intelligible for other contributors. Improving intelligibility and mutual understanding among developers is a good way of favoring interactions between different domains of pervasive computing.

... for everyone's benefit

Already mentioned in this manuscript, the interaction between low abstraction level perceptive works (computer vision, etc.) and higher level activity modeling needs to be activated. Both tiers would benefit from the interaction: lower abstraction level services and systems are no simpler than higher level ones. For instance, solving the problem of vision requires to solve the Problem of artificial intelligence (and also activity modeling by the way).

9.4 Thinking About the Future

9.4.1 Shared Infrastructure with Domain Specific Descriptions

user orientation is the key...

We could think of many small improvements to different aspects of our work. These small improvements can be trivially derived from the analysis of our investigation. More interesting are the directions that our overall research experience tells us to explore.

... to effective evolution of intelligent environments

We think that creating an infrastructure for runtime sharing of knowledge and descriptions is fundamental. This infrastructure must allow different kinds of information to fit in it and interact with each others. Inspiration must be taken from the three of:

- software engineering methods,
- knowledge representation methods, and
- context aware computing.

From our approach must be kept the usability aspect that in fact covers many sociological and psychological facets.

Many existing research investigations are positioned at the confluence of two of the three domains above. The domain of semantic web services already explores the relations between service oriented architecture and the domain of knowledge representation. In the same way, the interesting investigation done in PersonisAD presented in [Assad 2007] already interrelates context aware frameworks with device description and discovery. The IST Amigo project has also many interesting contributions in the domain of device heterogeneity handling and semantic service descriptions. Advances made in the ICT-CONNECT project should also be monitored as we can expect interesting contributions around protocol adaptations (most information are on the project web site [Url-v]).

Our recommendation for an future infrastructure supporting development of ubiquitous computing is to insist on two aspects:

- Having a knowledge base that is distributed, open, enriched at runtime and shared between different domains and applications.
- Querying the knowledge base and contributing to it using different, target specific tools.

Having a knowledge base that is distributed, open, enriched at runtime and shared between different domains and applications

The runtime sharing of descriptions between application parts is a recurring need. The need for such a knowledge base is clearly identified for context aware computing (were tuple spaces are mostly used) and implicitly present in service oriented architectures (with service repositories). The same need is appearing in the domain of real-time higher-level computer vision: integration between methods is becoming the more promising direction for major improvements. Replicating the implementation effort of each of these knowledge bases is a waste of time. More importantly, isolating these knowledge bases from each others is an error: we have identified intrinsic relations between context and services, and putting an artificial barrier between the two is a real constraint.

Querying the knowledge base and contributing to it using different, target specific tools

On top of the shared knowledge base, multiple dedicated languages and APIs must be designed for each target domain and developers. These interfaces to the knowledge base must be tailored for their target users. OMiSCID API can be such an interface: service declarations contribute to the knowledge base while service discovery is a way to query this base. A domain specific language such as UFCL can be used to fill the knowledge base with factory descriptions and functionality level descriptions. A specific API and a language could be used to contribute and query contextual information. The knowledge base interface for context aware computing could for example provide an abstraction to simplify the reasoning about uncertainty and probabilities.

9.4.2 Concluding Remark: User-Orientation is Key

The user is the key! This affirmation is well established in human computer interaction and is more and more present in pervasive computing as the field matures. However, the user does not provide the only technology acceptance test. Surveys, usability tests, focus groups and field studies generate much information that is not limited to acceptability.

Observing and studying users provide knowledge about their habits, their requirements and their expectations that is almost impossible to obtain through other means. Observing social interactions between users gives a good insight of what real problems are. Observations can also stimulate innovation from the observer. Conceptions methods with users involved in "converge" faster than simple test and retry methods.

Success of bringing intelligence in pervasive computing is conditioned by the existence of proper methods that allows various specialists to cooperate. The creation of conception methods should be as much user centered as any other conception methods. In this case the developer becomes the user. All intelligent environments stakeholders must be studied in their habits, psychology and sociological aspects. Designing software engineering methods must involve a global reasoning and much observation of designers, developers, end-users and also the interactions between these people.

As a concluding remark, we can say that the success of intelligent pervasive computing is tied up with its openness to social and psychological aspects influencing both end-users and system designers.

> *We must learn to live together as brothers or perish together as fools.*
> *— Martin Luther King Jr.*

Chapter 10

Résumé Étendu

10.1 Avant-Propos

Le manuscrit original de ces travaux de thèse est écrit en anglais. Ce document est un résumé étendu du manuscrit original.

10.2 Des Ordinateurs aux Appareils Communicants

L'évolution des ordinateurs et du matériel informatique a été considérable depuis les années 1960. La tendance générale a été l'augmentation de la puissance de calcul et de la mémoire des ordinateurs. Les ordinateurs réservés aux scientifiques pour des applications de calcul et de simulations ont progressivement évolués pour être utilisés dans les entreprises pour la gestion de leurs systèmes d'information. Les possibilités alors offertes par les ordinateurs ont conduit à l'ère de l'ordinateur personnel où de nombreuses applications ludiques et multimédia sont apparues. La vision de l'ordinateur personnel est d'avoir un ordinateur pour chaque utilisateur.

Suite à cette augmentation de la puissance des ordinateurs, la nouvelle tendance est maintenant à l'augmentation du nombre d'appareil à la disposition et utilisé par chaque utilisateur. Ainsi, avec le réseau internet, les nombreux téléphones mobiles et les ordinateurs portables, un utilisateur moyen utilise aujourd'hui environ une dizaine d'appareil différent, de manière quotidienne. Cette tendance s'accentue et l'ère qui y correspond s'appelle l'informatique ambiante qui imagine un environnement physique ou l'accès à des ressources informatiques peut se faire à tout endroit, à tout instant.

Avec l'avènement de l'informatique ambiante, les attentes des utilisateurs augmentent fortement. Étant donné que tous les appareils présents dans l'environnement des utilisateurs sont à la fois programmables et dotés de capacité de communication, les utilisateurs attendent un comportement « intelligent » de la part de ce réseau d'appareils. L'ensemble des appareils doit fournir un service plus abouti que la somme des services fournis par chaque appareil. Ces attentes correspondent à ce qui est appelé « intelligence ambiante ».

10.3 De la Programmation au Génie Logiciel

Initialement, le programme exécuté se rentrait à l'aide d'interrupteurs actionnés manuellement par un opérateur. L'utilisation des cartes perforées à matérialisé le programme et a rendu possible la plus importante méthode de métaprogrammation : la compilation. La complexité des programmes à alors permis de créer des applications et de toucher de nombreux domaines d'applications nouveaux. Le génie logiciel s'intéresse au processus de création des applications :

ce processus est aujourd'hui compliqué et implique de nombreuses personnes de différents horizons.

L'intelligence ambiante regroupe de nombreux domaines. Encore à l'état de recherche, les spécialistes de ces domaines n'ont pas le temps de se maintenir à jour des dernières recherches en génie logiciel. On constate donc des problèmes d'intégration et de réutilisation importants dans le domaine de l'intelligence ambiante. Les architectures à services ont des propriétés très intéressantes aux vues des requis de l'intelligence ambiante.

10.4 Problème et Approche

L'intelligence ambiante est un domaine intrinsèquement interdisciplinaire : les spécialistes de nombreux domaines doivent collaborer pour que le domaine progresse efficacement. L'intelligence ambiante n'est pas seulement la juxtaposition des différentes disciplines mais plutôt une interconnexion et un enrichissement mutuel de ces disciplines. Une double veille technologique serait nécessaire pour que ces différents spécialistes soient à la fois à jour dans leur spécialité et dans le domaine de l'ingénierie logicielle. L'intégration des travaux des différents domaines nécessiterait cette veille dans le domaine de l'ingénierie logicielle. Malheureusement, cette double veille technologique est trop coûteuse en temps pour être menée à bien par l'ensemble des spécialistes. Il en découle que l'intégration et la réutilisation des travaux des différents spécialistes est très difficile et qu'en pratique beaucoup de travaux restent isolés.

Notre approche est d'étudier et d'adapter les solutions d'ingénierie existantes pour qu'elles soient utilisables et profitables pour l'ensemble des spécialistes pouvant contribuer à l'intelligence ambiante. Dans notre approche, nous nous proposons de rendre accessible des méthodes comme les architectures à services (SOA) et la description sémantique des fonctionnalités des services. Nous proposons aussi de nouveaux outils conceptuels comme les usines à services pour résoudre les problèmes spécifiques liés à l'architecture logicielle pour l'intelligence ambiante.

10.5 Architectures à Services pour l'Intelligence Ambiante

Les solutions actuelles pour mettre en œuvre des architectures à service limitent leur acceptabilité auprès du large public des spécialistes pouvant contribuer à l'intelligence ambiante. Les principaux obstacles à l'adoption des architectures à services sont variés et dépendent de l'implémentation considérée : retour sur investissement peu clair, complexité supposée ou limitation à un langage de programmation. Pour être acceptable par l'ensemble des acteurs de l'intelligence ambiante, une solution à services doit respecter de nombreux critères : facilité d'apprentissage, facilité d'installation, accessibilité dans de nombreux langages de programmation, performance (débit, latence, etc.), robustesse, etc.

10.6 Services and Représentation de Connaissances

10.6.1 Du Découplage d'Implémentation au Découplage de Conception

L'utilisation d'interfaces claires ou de services dans la conception d'une application permettent de s'abstraire de l'implémentation d'un composant et de se concentrer sur son interface d'accès constituée des messages qu'il peut émettre ou recevoir. Les interfaces permettent de découpler les implémentations des éléments logiciels : ils ne dépendent plus que de l'interface des autres services.

Cependant, pour utiliser un élément logiciel, il est nécessaire de connaître son interface. En utilisant des services, cette connaissance doit nécessairement être donnée à la conception du

logiciel qui utilise un service particulier via son interface. Une standardisation ou un accord sur tous les types de services existants serait nécessaire à permettre une intégration de tous les éléments potentiellement existant dans un environnement intelligent. Nous pensons cette standardisation impossible et proposons dans la suite une solution permettant l'intégration voulue.

10.6.2 La Conception Comme Connaissance

Nous proposons d'ajouter une couche d'adaptation qui transforme un type de service donné en un autre. Ces transformations seraient réalisées par des adaptateurs. L'adaptation peut être réalisée de manière manuelle ou automatique. Cette idée d'adaptation de services particuliers en d'autres services correspond exactement à une problématique du domaine des représentations de connaissances : différentes personnes produisent des connaissances qu'il faut ensuite fusionner et mettre en correspondance sans pour autant pouvoir modifier les connaissances initiales. Notre problématique et celle de la représentation de connaissance partagent la propriété d'être des exercices de modélisation : toute modélisation pose la question de l'interaction avec d'autres modélisations du même objet.

10.6.3 Représentation de Connaissance et le Web Sémantique

Le web sémantique part du constat que le web actuel contient une quantité importante de données mais qu'elles sont malheureusement exprimées qu'en langue naturelle. Le web sémantique propose de structurer ces données pour améliorer la précision et la qualité des recherches. Grâce à une représentation structurée des données, il devient possible d'exprimer des requêtes très précises ne se limitant pas à la coprésence de plusieurs mots dans une page web. Le concept de page tend à disparaître avec le web sémantique : une page est en fait remplacée par une vue sur les données.

Pour la représentation des données, le web sémantique utilise des langages de représentation de connaissances comme RDF ou OWL. Un défit clair du web sémantique est de mettre en correspondance des connaissances venant de plusieurs sources pour transformer des îlots isolés d'informations en un réseau unique de données.

10.6.4 Description Sémantique de Services

La description sémantique des fonctionnalités offertes par les services se propose de décrire les services de manière plus abstraite que leur interface. La réelle fonctionnalité offerte par le service est exprimée sous une forme compréhensible par les machines de façon à permettre un raisonnement automatique sur les fonctionnalités.

Deux implémentations principales de cette approche de description sémantique de services existent. Ces implémentations reposent toutes deux sur les services web. La première, OWL-S (Web Ontology Language for Services) se base sur OWL pour décrire les fonctionnalités des différents services. La seconde implémentation est WSMO (Web Service Modelling Ontology). Les deux approches ont leurs avantages et leurs inconvénients.

Les web services sémantiques (OWL-S et WSMO) se basent sur les web services classiques et ajoutent une description sémantique des fonctionnalités des services. La description sémantique utilise les principes (et langages) du domaine de la représentation des connaissances. Ces approches de descriptions restent cependant complexes pour le développeur moyen, susceptible de contribuer à l'intelligence ambiante. D'autre part, les services web sémantiques ne travaillent qu'avec des services web et donc sur un modèle ignorant les connections (et déconnections) entre services. Ces éléments limitent l'utilisabilité directe de ces technologies dans le contexte de l'intelligence ambiante.

10.7 Approches pour la Composition et la Description de Services

Même composés manuellement, les services apportent l'avantage d'abstraire les détails d'implémentation derrière l'interface claire et bien définie du service. Au delà de la composition manuelle de services, des méthodes plus automatisées existent. On peut distinguer les méthodes à base de workflow et les méthodes inspirées de l'intelligence artificielle ou dites à base de planification.

Les méthodes de composition manuelle peuvent être assistées par différents outils de présélection ou de filtrage des services existants au moment de la création du service composite. Suite à la composition fournie par un opérateur humain, l'orchestration ou l'exécution de cette composition peut être mise en œuvre de manière automatique grâce à l'abstraction que propose l'interface des services.

Les méthodes à base de workflow proposent au concepteur de définir un plan abstrait de composition, contenant potentiellement des tests et des structures de contrôle de flot. Les différents éléments abstraits de ce plan de composition sont recrutés automatiquement à l'exécution : des services concrets sont découverts et utilisés pour remplir les services abstraits utilisés dans le plan.

Les méthodes à base de planifications considèrent les invocations aux services comme des transformations, soit des données soit du monde (l'environnement du système). Ces méthodes essaient de trouver automatiquement une chaîne d'invocation de services permettant de passer de l'état courant du monde (ou des données) à l'état désiré par l'utilisateur du système. Un exemple simple de méthode à base de planification est la génération automatique de pipeline de conversion de fichiers vidéo : en partant d'un fichier encodé dans un format donné, le système est capable d'assembler différents décodeurs et convertisseurs pour obtenir, en fin de chaîne, un format que le module d'affichage est capable de comprendre. Il faut par exemple assembler un décodeur d'images JPG puis un convertisseur d'espace de couleur et enfin un module de redimensionnement de l'image pour l'affichage final. En utilisant l'entrée donnée, les sorties acceptables et la description de chaque modules, il est possible de construire la chaîne de conversion de manière automatique.

10.8 Intelligence Ambiante, Intégration et Problèmes Associés

Par l'analyse de différents systèmes d'intelligence ambiante, nous orientons le reste des contributions de notre travail. Pour mettre en évidence les différents problèmes récurrent dans le développement de l'intelligence ambiante, nous présentons en particulier le cas d'un système de suivi utilisant des caméras et développé au sein du projet de recherche PRIMA. La section courante correspond au chapitre 4 du manuscrit original.

L'intelligence ambiante étant hautement interdisciplinaire, le partage de logiciel entre les différents contributeurs à l'intelligence ambiante est capital. Les différentes disciplines sont très variés et incluent la perception de bas niveau (vision par ordinateur, perception acoustique, robotique, etc.), des domaines plus intégrés (perception multimodale, modélisation de l'activité humaine, etc.). Ces différentes disciplines ont des objectifs souvent différents et leur intégration est souvent limitée de part le temps qu'elle requiert.

L'exemple du système de suivi développé au sein du projet PRIMA est symptomatique : les problématiques de visions sont trop « simples » pour les chercheurs en vision pure, et, paradoxalement, les résultats produit par le système de suivi sont de trop mauvaise qualité pour les personnes susceptibles de l'utiliser pour la reconnaissance d'activité par exemple. Ce phénomène vient de la difficulté d'intégrer les travaux de bas niveau d'abstraction et d'en tirer profit efficacement dans le système de suivi. Cette intégration est actuellement possible mais nécessite un effort et un temps important pour être menée à bien.

Le système de suivi visuel de PRIMA a évolué depuis un suivi en 2 dimensions dans les images d'une caméra à un suivi dans l'espace, en 3 dimensions, en utilisant plusieurs caméras. L'acquisition et le traitement des images de plusieurs caméras a nécessité de distribuer le système de suivi sur plusieurs ordinateurs. À l'époque, essentiellement pour simplifier le déploiement du système, la distribution a été faite avec un intergiciel à services développé pour l'occasion au sein de PRIMA et nommé BIP. Les différents services composant le système de suivi se recrutent entre eux en utilisant une découverte de service basée sur le protocole Zeroconf. Le besoin de distribution est l'occasion idéale d'introduire de nouvelle pratique de développement dans une population de développeurs. Pour distribuer une application, le développeur doit utiliser de nouveaux outils ; nous profitons de ce besoin pour pousser une approche à service qui facilite la réutilisation et l'intégration de travaux.

Un des objectifs que nous tirons des problèmes d'intégration et du besoin de distribution est de promouvoir l'adoption des architectures à services. Au delà des architectures à services, nous identifions aussi des problèmes dus à un déploiement répétitifs de certains éléments des systèmes ambiants. Nous identifions aussi un besoin de s'abstraire de l'interface exacte d'un service lors de la conception. Avec une simple approche à service, il est nécessaire de connaître, dès la phase de conception, les interfaces exactes des services que le système sera capable d'utiliser. Nous voulons permettre une intégration a posteriori des différents services sans avoir à les modifier et à les relancer pour qu'ils puissent interagir. Les chapitres suivant du manuscrit visent à apporter des éléments de réponse à ces problématiques.

10.9 OMiSCID : un middleware à services

L'intelligence ambiante souffre de son interdisciplinarité : la réutilisation et l'intégration des travaux est difficile et coûteuse. Une intégration continue serait cependant bénéfique pour tous les acteurs concernés. Le besoin de distribution de certains travaux, en particulier des travaux de perception, est l'occasion d'introduire des méthodes de développement plus adaptés à l'intégration nécessaire à l'intelligence ambiante. Nous proposons donc d'introduire un intergiciel à services comme moyen de distribution pour l'intelligence ambiante. La solution proposée doit être aussi facile et rapide à utiliser pour les développeurs ciblés que n'importe quel autre moyen de distribution qu'ils pourraient trouver. La section courante correspond au chapitre 5 du manuscrit original.

L'intergiciel BIP est un intergiciel à services à l'origine conçu pour la distribution des composants de vision. Le problème majeur de BIP, que l'on retrouve dans certaines autres solutions d'architectures à services, est son manque d'utilisabilité. Beaucoup de solutions à services ne sont pas utilisées car les utilisateurs potentiels y voient des problèmes potentiels : limitation à un langage de programmation, problèmes de performance, difficulté d'apprentissage, etc. De notre analyse nous listons des requis pour un intergiciel à services utilisable par les contributeurs à l'intelligence ambiante. Ces requis incluent la facilité d'apprentissage et d'installation, l'accessibilité dans de nombreux langages, l'intégration au processus de développement usuel des contributeurs (pas de nouvel environnement dédié) et les performances.

Partant des requis que nous avons exhibé, nous créons une nouvelle API (Application Programming Interface) pour la création et la recherche de services appelée OMiSCID. L'implémentation de cette API se base sur l'ancien intergiciel BIP. OMiSCID est une simple bibliothèque utilisable en C++, Python et Java (et donc Matlab et tous les langages de scripts de la jvm). L'API OMiSCID est commune entre les 3 langages et permet ainsi un transfert direct des compétences lors du changement de langage de programmation. En comparaison avec BIP, OMiSCID limite les mauvaises utilisations de l'intergiciel et propose une API beaucoup plus concise et simple à appréhender pour un développeur souhaitant ou utiliser un service. La découverte de service est par exemple largement simplifiée par l'introduction d'une API déclarative : l'utilisateur exprime un filtre de services en composant des filtres existant avec des conditions logiques (et/ou).

L'implémentation OMiSCID a été suivie de phases d'évolution, de maintenance et de communication. Ainsi, OMiSCID est disponible en open-source, possède un site publique dédié et a été décrit dans une publication scientifique. L'intergiciel a aussi été présenté chez différents partenaires du projet PRIMA.

Nous avons étudié les besoins en outils autour d'un intergiciel à services. Il en est ressorti un besoin majeur : une interface graphique pour la mise au point, l'observation et l'interaction avec les services. Nous voulons aussi transformer une telle interface graphique en lieu d'échange entre les développeurs des différents services. L'interface graphique doit donc vérifier les propriétés suivantes : lister les services démarrés, permettre d'interagir de manière générique avec ces services, être facilement extensible par les différents créateurs de services, permettre le partage et la mise à jour des différentes extensions proposées par les créateurs de services.

Nous avons implémenté OMiSCID Gui qui rempli les besoin exposés précédemment. Basé sur la plateforme Netbeans, l'interface graphique d'OMiSCID est très facile à mettre à jour et à étendre avec de nouveaux modules. Le développement de nouveaux modules ne nécessite seulement la connaissance du langage Java et de bases sur la bibliothèque d'interface graphique Swing (par défaut dans Java). OMiSCID Gui, dans sa version de base, permet de lister les différents services OMiSCID et de surveiller leurs interconnections et d'interagir de manière générique avec leurs variables et connecteurs. De nombreux extensions (ou modules, ou plugins) ont été développés par les créateurs de services : visualisation du flux d'images d'une caméra, visualisation de la sortie d'un télémètre laser, contrôle et pilotage de caméra, etc. Toutes ces extensions peuvent être installées en quelques clicks par n'importe quel utilisateur d'OMiSCID Gui.

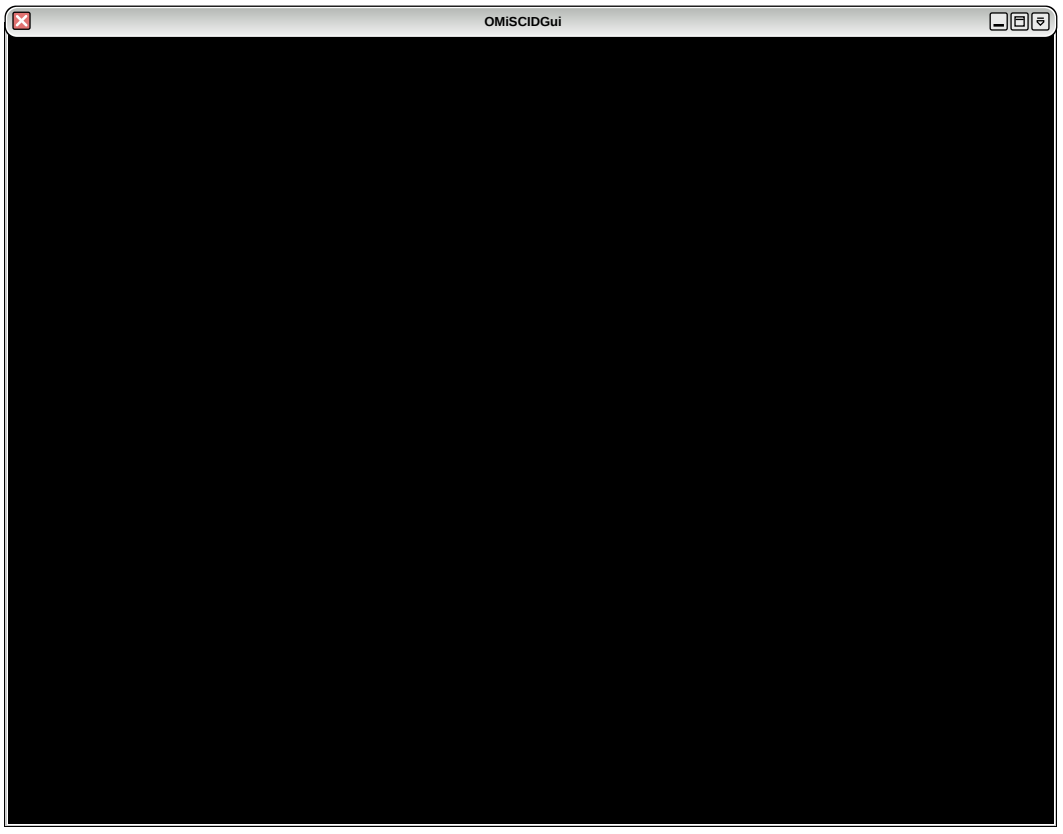


Figure 10.1: *Utilisation d'OMiSCID Gui pour le contrôle et la surveillance d'un assemblage de services utilisés pour le suivi 3D au niveau d'un bâtiment. Cinq extensions (modules/plugins) sont contribuent à la vue présentée dans cette capture d'écran.*

10.10 Concepts et Méthodes pour la Conception de Systèmes Dynamiques et Ouverts

Le chapitre 4 du manuscrit exhibe un certains nombres de problèmes qui ne sont résolus que partiellement par l'utilisation d'un intergiciel à service comme celui présenté dans le chapitre 5. En effet, les niveaux d'abstraction supérieurs comme le domaine de la modélisation d'activité nécessitent des abstractions permettant de construire de manière durable sur les éléments de plus bas niveaux d'abstraction. En construisant sur l'utilisation d'un intergiciel à service par l'ensemble des acteurs de l'intelligence intelligente, nous proposons une méthode de conception permettant une meilleure gestion de la dynamicité des systèmes d'intelligence ambiante. Cette méthode se base en partie sur les principes utilisés dans le domaine des web services sémantiques. Nous introduisons aussi le nouveau concept « d'usine à services » qui ressort de notre analyse comme nécessaire. La section courante correspond au chapitre 6 du manuscrit original.

Nous avons introduit le concept « d'usine à services » : une usine à service est un service dont le rôle est de créer de nouveaux services sur demande. Les usines à services résolvent deux problèmes majeurs constatés entre autres dans le système de suivi 3D :

- lors des expérimentations et de la mise au point du système, il est fastidieux d'arrêter et de redémarrer l'ensemble des services composant le système. Cette difficulté incite d'ailleurs les concepteurs de services à faire des services tout-en-un donc moins extensibles et moins réutilisables. De plus, le redémarrage manuel augmente le risque d'erreur lors de la configuration de chaque service à son lancement.

- ce qu'expose l'environnement à services est très limité par rapport à ce qui est réellement possible. Dans l'exemple du système de suivi 3D, des services de traitement d'image sont utilisés. Ces services sont démarrés et configurés manuellement et l'environnement expose donc sa capacité à réaliser ces traitements d'images particuliers. L'environnement n'expose que les services démarrés alors qu'il pourrait exposer sa capacité à créer une variété de services de traitements d'images (dépendant de la configuration du service de traitement d'image).

Les usines à services permettent de créer sur demande un service parmi une famille de services donnée. La création est paramétrée par les éléments de configuration nécessaires.

Ainsi, dans l'exemple du système de suivi 3D, c'est le système de suivi lui même qui a connaissance des configurations des services de traitement d'image. Le système de suivi demande aux usines à services de créer les services de traitement d'image avec la configuration qu'il désire. Tous les éléments de configuration sont donc regroupés dans le service principal du système de suivi et non plus distribués sur l'ensemble des lignes de commandes des services lancés manuellement.

Les web services sémantiques apportent des solutions au problème d'intégration a posteriori de services conçus indépendamment. Nous réutilisons ces solutions sous la forme de fonctionnalité de services. Un service abstrait les détails de son implémentation derrière son interface de services, c'est à dire les messages qu'un service peut recevoir et émettre. Le concept de fonctionnalité vise à s'abstraire encore plus de l'implémentation et des messages échangés pour permettre un raisonnement automatique de plus haut niveau. Une fonctionnalité de service abstrait l'interface d'un service derrière un concept fonctionnel. Par exemple, un service représentant une caméra et émettant ses images avec un format précis peut être abstrait derrière la simple fonctionnalité de « source d'images ».

Fondée sur les architectures à services, sur les usines à services et sur les principes des services web sémantiques, nous proposons une méthode de conception destinée aux contributeurs de l'intelligence ambiante et visant à améliorer la réutilisabilité et l'extensibilité de leur services. Cette méthode s'articule en 6 étapes qui peuvent être soit prises dans l'ordre, soit

considérée comme des lignes directives à suivre pour les développeurs. Ces 6 étapes sont détaillées dans le manuscrit et sont illustrées sur deux exemples : le système de suivi et un système d'enregistrement automatique de séminaire. Les 6 étapes de la méthode sont les suivantes et s'appliquent aussi bien à la reconception qu'à la conception initiale d'un système :

- identifier les responsabilités dans le système ;
- séparer le système en services ;
- introduire des usines à services si nécessaire ;
- abstraire chaque service derrière une fonctionnalité ;
- exposer sa contribution sous forme de service, de fonctionnalité et si pertinent d'usine à service ;
- intégrer ses fonctionnalités avec celle existantes par ajout de correspondance de fonctionnalités et d'adaptateurs.

En suivant cette méthode, nous avons amélioré la réutilisabilité des différents éléments constituant le système de suivi 3D et le système d'enregistrement automatique de séminaires. En particulier, avec la nouvelle architecture obtenue, les caméras peuvent être partagées et plusieurs services de traitement d'images peuvent facilement être démarrés pour remplir les différents besoins simultanés qui peuvent apparaître (plusieurs systèmes de suivi, couplage système de suivi/reconnaissance de visage, etc.). La figure 10.2, issue du manuscrit, donne un aperçu des services en jeu et des descriptions de leur fonctionnalités. Cette figure illustre aussi les descriptions et services que les développeurs doivent apporter et les choses qui en sont automatiquement dérivées par l'environnement d'exécution.

10.11 UFCL, un Langage de Description Sémantique de Services

Le chapitre 6 du manuscrit propose une méthode de conception pour les développeurs impliqués dans l'intelligence ambiante. Cette méthode vise à améliorer la réutilisabilité et l'intégration dynamique des différentes contributions des différents développeurs. Pour « donner vie » à cette méthode et supporter son application, nous proposons un langage dédié, simple à comprendre et à écrire. La section courante correspond au chapitre 7 du manuscrit original.

Nous avons créé le langage UFCL (User-oriented Functionnality Composition Language) qui s'adresse au développeur de services logiciels pour l'intelligence ambiante. UFCL vise à simplifier les principes existants dans les web services sémantiques tout en incluant le concept d'usines à services que nous avons introduit. Au niveau langage, UFCL propose un nombre de constructions limité de façon à facilité son apprentissage. Le niveau d'abstraction d'UFCL en fait un langage de description puissant.

En utilisant le principe de fonctionnalité comme dans les web services sémantiques, UFCL vise à permettre une intégration a posteriori et déclarative des services développés en isolation. Ainsi, plutôt que d'avoir une intégration où un intégrateur doit comprendre et modifier les différentes contributions pour les intégrer (figure 10.3), l'intégrateur va ajouter des adaptateurs entre les différentes fonctionnalités contribuées par différents développeurs. Ces adaptateurs peuvent impliquer beaucoup de code mais par l'utilisation des fonctionnalités de services, les adaptateurs peuvent se limiter à quelques lignes d'UFCL, le reste étant géré automatiquement par l'environnement d'exécution.

UFCL propose trois constructions syntaxiques pour l'expression des informations au sujet des services :

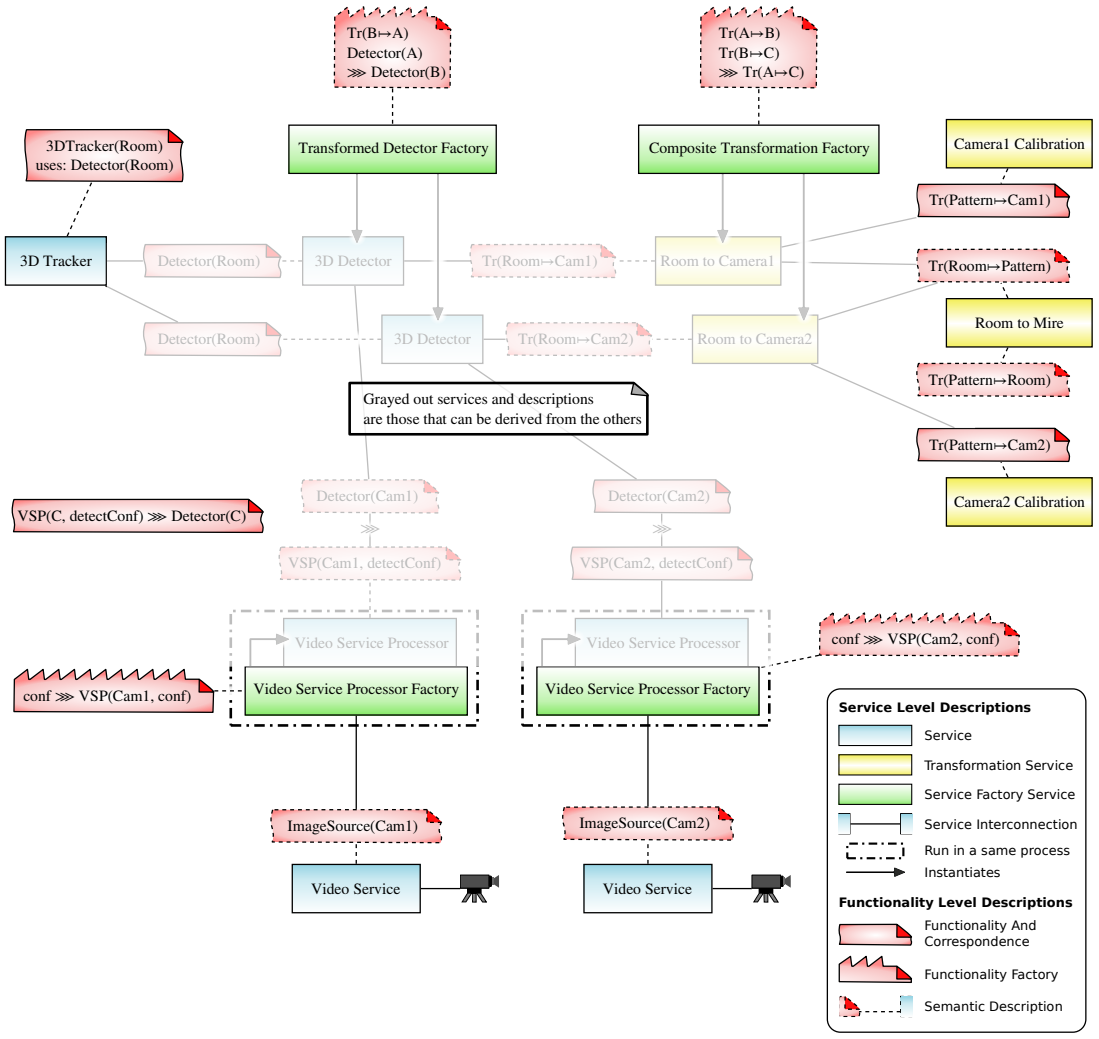


Figure 10.2: *Nouvelle architecture modulaire du système de suivi 3D. Cette architecture inclue services, fonctionnalités et usines à services. Les parties plus claires sont celles qui sont automatiquement dérivées du reste par l'environnement d'exécution. Les parties pleines sont la connaissance amenée par les différents concepteurs.*

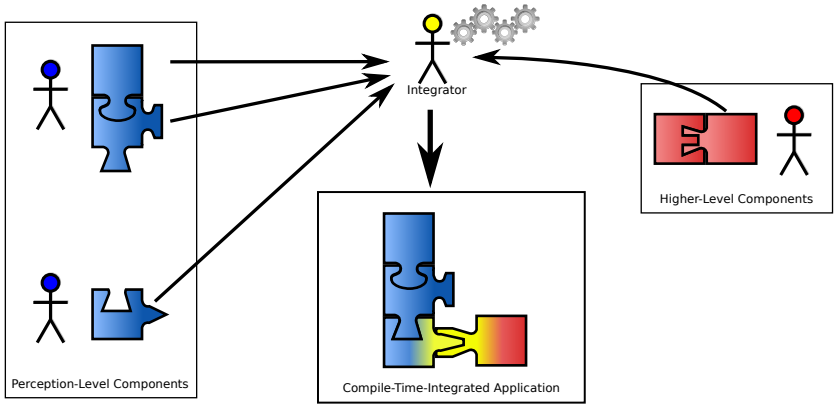


Figure 10.3: *Intégration « classique ». L'intégrateur doit comprendre et modifier la plupart des composants qu'il doit intégrer. Les composants sont modifiés pendant la phase de développement pour produire l'application.*

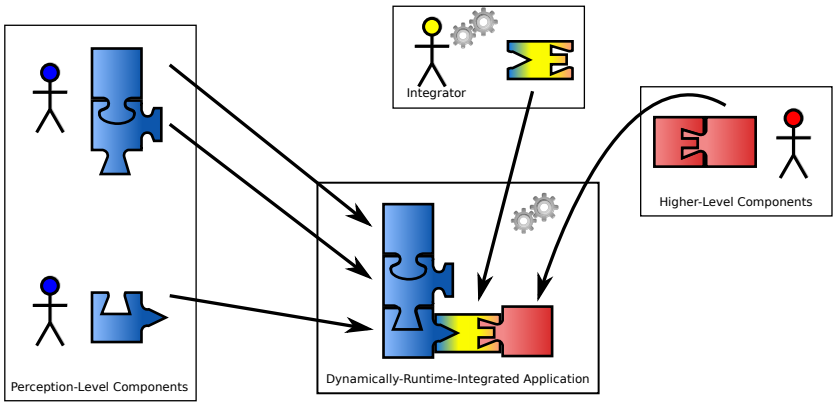


Figure 10.4: *Intégration avec UFCL. Grâce à l'encapsulation en services et fonctionnalités, l'intégrateur ne doit comprendre que les fonctionnalités exposées et requises par les services. Les correspondances entre fonctionnalités et les adaptateurs sont introduits à l'exécution, sans modifier aucun composant existant.*

- description des fonctionnalités exposées par un service ;
- description des correspondances entre fonctionnalités ;
- description des usines à services.

La première construction UFCL permet de décrire la ou les fonctionnalités proposées par un service donné. Ainsi, un service OMiSCID peut par exemple exposer la description UFCL suivante :

```
1 | namespace is http://emonet09#
2 | this isa Timer
3 |   with freq = 1
4 |   with grounding = "C(tick)"
```

Cette description signifie que le service exposant la description UFCL (mot clé « this ») a une fonctionnalité de « Timer » avec une fréquence de 1. Un « Timer » est ici un élément logiciel émettant des événements à une fréquence donnée. La propriété « grounding » doit être donnée par l'auteur de la description UFCL mais n'est utilisées qu'automatiquement par l'environnement d'exécution. Cette propriété référence l'implémentation concrète de la fonctionnalité « Timer ». Ainsi la fonctionnalité peut être accédée sur le connecteur OMiSCID (« C(...) ») nommé « tick ».

La seconde construction syntaxique d'UFCL permet d'exprimer des correspondances entre fonctionnalités. Pour l'intégration, il est possible d'exprimer qu'un type de fonctionnalité avec un ensemble de propriétés peut être considéré comme une autre fonctionnalité. Un exemple d'une telle expression est donné ici :

```
1 | namespace is http://emonet09#
2 | this isa Timer
3 |   with freq = 2
4 |   with grounding = "C(tickTwice)"1 | namespace is http://emonet09#
2 | this isa Timer
3 |   with freq = 1
4 |   with grounding = "C(tick)"
5 | this isa Timer
6 |   with freq = 1
7 |   with grounding = "C(tick)"
5 | a Métronome
6 |     having bpm = ?f
```

```
7 |    isa Timer
8 |      with freq = ?f / 60
```

Cet exemple UFCL exprime le fait qu'une fonctionnalité de « Métronome » ayant une valeur quelconque (nommée ?f) pour sa propriété « bpm » (beats per minute, battements par minutes) peut être considéré comme une fonctionnalité « Timer » avec une fréquence 60 fois plus petite que ?f. Formulé autrement, tout service exposant une fonctionnalité « Métronome » exposera automatiquement une fonctionnalité « Timer » de part la présence de cette description UFCL.

La dernière construction UFCL permet de décrire les usines à services. Les usines à services sont très variées mais l'on peut en distinguer deux types principaux : les usines « ouvertes » et les usines « qui composent ».

Un exemple d'usine « ouverte » est l'exemple de l'usine à service de traitement d'image : cette usine peut créer n'importe quel service de traitement d'image étant donnée une configuration. Un autre exemple plus simple d'usine « ouverte » est l'usine à service « Timer » qui crée, étant donnée une fréquence voulue, un service « Timer » avec cette fréquence. La « TimerFactory » peut être décrite en UFCL comme suit :

```
 9 |  composing
10 |   grounding "C(create)"
11 |   format    "create timer ?f"
12 |  gives a Timer
13 |   with freq = ?f
```

Cette description exprime simplement que le service exposant la description UFCL est une usine à services et qu'il peut créer n'importe quelle fonctionnalité « Timer » avec une fréquence arbitraire ?f. La variable ?f est ici non contrainte et n'importe quelle valeur est acceptée. Les propriétés particulières « grounding » et « format » sont utilisées par l'environnement d'exécution pour déterminer comment effectivement invoquer l'usine à service. Ainsi, pour créer un « Timer » ayant une fréquence de 42, l'environnement d'exécution enverra à l'usine le message « create timer 42 » sur le connecteur OMiSCID « create » de l'usine.

Les usines à services peuvent aussi servir à décrire des patrons de composition abstraits. Il est aussi possible de mélanger les usines à paramètre (comme la « TimerFactory » et des usines de composition). Un exemple d'usine de composition pure est celui de la composition de services de traduction qui peut être exprimé comme suit en UFCL :

```
14 | composing
15 |    a Translator ?t1
16 |    a Translator ?t2
17 |    grounding "C(start)"
18 |    format    "compose '?t1#' and '?t2#'"
19 |  having
20 |    ?t1.to = ?t2.from
21 |  gives a Translator
22 |    with from = ?t1.from
23 |    with to   = ?t2.to
```

Cette description UFCL exprime le fait que le service qui l'expose est une usine de composition. Cette usine est capable de composer deux fonctionnalités « Translator », ?t1 et ?t2, du moment que la langue source de ?t2 correspond à la langue destination de ?t1 (« ?t1.to = ?t2.from »). Le résultat de la composition est alors une fonctionnalité « Translator » traduisant la langue source de ?t1 en la langue destination de ?t2. Les propriétés de grounding et de format sont ici aussi utilisée automatiquement par l'environnement d'exécution pour l'appel à l'usine à service. Ces propriétés doivent être fournies par le fournisseur de service mais sont transparentes pour celui qui ne fait qu'utiliser le service (l'usine).

Les 3 constructions d'UFCL sont relativement simples mais permettent d'exprimer de nombreux cas d'utilisation. Ces constructions sont simples et expressives mais peuvent aussi être utilisées de manière automatique par l'environnement d'exécution pour répondre à la requête d'un consommateur de fonctionnalité.

10.12 Compilation d'UFCL et Raisonnement Automatique

UFCL permet d'exprimer les descriptions sémantiques des services présents dans un environnement intelligent. Pour donner vie à ces descriptions sémantiques, il est nécessaire d'exploiter de manière automatique ces descriptions pour aider les développeurs d'applications. Étant donné une requête exprimée sous forme d'une fonctionnalité et des propriétés associées à ces fonctionnalités, le développeur peut s'attendre à ce que l'interprétation des descriptions UFCL et les éventuellement nécessaires appels à des usines à services soient réalisés de manière totalement automatique. Automatiser l'exploitation des descriptions UFCL à l'aide de raisonnement automatique est la fonction de l'environnement d'exécution que nous avons proposé. La section courante correspond au chapitre 8 du manuscrit original.

L'objectif de l'environnement d'exécution est de trouver automatiquement, à partir de la requête fonctionnelle d'une application et de l'ensemble des descriptions UFCL, un plan de composition de la fonctionnalité requise si un tel plan existe. Ce plan peut nécessiter l'appel à des usines à services et la création d'adaptateur. L'environnement d'exécution est aussi responsable de l'application automatique du plan trouvé (appels aux usines, etc.).

Pour l'implémentation de notre environnement d'exécution, nous avons utilisé un système à base de règles. Étant donné la présence d'usines à services pouvant produire un nombre arbitraire de fonctionnalités, l'hypothèse d'utiliser un chaînage avant mène automatiquement à des boucles infinies. Nous avons donc utilisé, dans notre implémentation, un système de chaînage arrière. Le principe est de réécrire le besoin fonctionnel initial en de nouveaux besoins. Cette réécriture continue jusqu'à trouver des fonctionnalités remplissant suffisamment de besoins pour composer la fonctionnalité initiale. Nous avons décidé d'utiliser le système de règle Jena pour sa spécialisation dans le domaine du web sémantique : il utilise RDF pour la représentation de ses connaissances internes et permet donc une intégration directe avec une partie des ontologies existantes. Malheureusement, Jena supporte le chaînage arrière mais, à notre mauvaise surprise, uniquement de manière limitée. Nous avons dû ainsi implémenter un système de chaînage arrière basé sur le chaînage avant présent dans Jena. Cette implémentation est difficile et coûteuse en temps mais permet aussi de porter aisément notre système vers un système de règle à chaînage avant (plus courant que le chaînage arrière).

Un point important dans la génération de règle est que ces règles ne sont pas écrites par les développeurs. Les développeurs peuvent ignorer la forme exacte des règles voire même leur existence. Certaines règles étant compliquées, il aurait été inenvisageable de laisser le développeur écrire ces règles ou même écrire des règles qui interagirait avec les règles générées. Dans notre cas, un piège classique aurait été de se dire que l'utilisateur serait plus libre si on lui laissait écrire des règles. Avec cette liberté, vient une complexité et une surcharge mentale qui mettrait en grand danger la stabilité et l'utilisabilité de la méthode proposée avec UFCL.

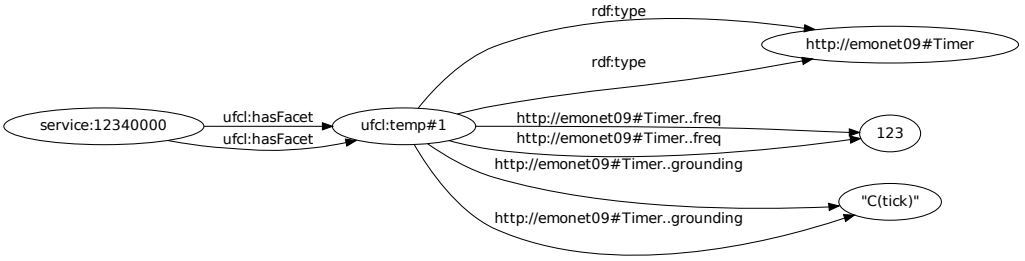
Jena propose une notation pour la représentation des faits dans sa base de connaissance. Ces faits ne peuvent être que des triplets « sujet prédicat objet ». Jena propose aussi une notation pour les règles, notation que nous utilisons dans le manuscrit.

La compilation des descriptions de fonctionnalités exprimées en UFCL sont compilées uniquement sous formes de faits et de manière très directe. Par exemple, considérons la description fonctionnelle suivante :

```
1 | namespace is http://emonet09#
2 | this isa Timer
3 |   with freq = 123
4 |   with grounding = "C(tick)"
```

Cette description signifie que le service exposant la description UFCL a une fonctionnalité de « Timer ». Notons que ce service pourrait exposer d'autres fonctionnalités. Supposons que ce service OMiSCID aie pour identifiant « 12340000 » dans l'intergiciel. La compilation de cette description produira les faits suivants (représentés textuellement et graphiquement) :

```
1 | (service:12340000 ufcl:hasFacet ufcl:temp#1)
2 | (ufcl:temp#1 rdf:type http://emonet09#Timer)
3 | (ufcl:temp#1 http://emonet09#Timer..freq 123)
4 | (ufcl:temp#1 http://emonet09#Timer..grounding "C(tick)")
```



Les autres constructions UFCL (usines et correspondance de fonctionnalités) sont compilées sous formes de règles. Ces règles de réécrivent un besoin en un nouveau besoin et, si le nouveau besoin est rempli, elles remplissent le besoin d'origine en chaîne. Comme détaillé dans le manuscrit, les correspondances de fonctionnalités ont été ramenées à un cas particulier des usines à services.

Les règles générées sont longues et des exemples sont présentés dans le manuscrit complet. Pour illustrer le mécanisme, considérons un exemple :

- un service expose une fonctionnalité A
- deux correspondances de fonctionnalités sont exprimées : « a A isa B » et « a B isa C »
- l'application cliente de l'environnement d'exécution demande une fonctionnalité C

La figure 10.5 illustre les différentes étapes déclenchées par les différentes règles.

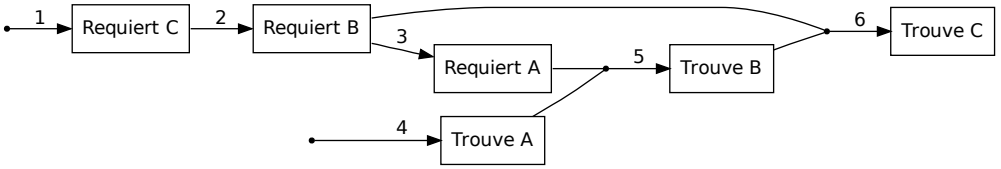


Figure 10.5: *Exemple de réécriture du besoin (chaînage arrière). Le besoin initial en C est réécrit en besoin en B puis en A. Le besoin en A étant rempli, le besoin en B le devient puis le besoin initial en C.*

Le même principe de réécriture du besoin est appliqué pour les usines à services. Ainsi, dans le cas de l'usine qui compose deux traducteurs (description UFCL rappelée ci dessous), tout

recherche d'un « Translator » va donner lieux à la recherche d'un « Translator » ?t1 compatible. Si un tel ?t1 est trouvé, il donnera alors lieu à la recherche d'un ?t2 compatible avec ?t1 et la requête initiale. Si un ?t2 correspondant est trouvé, le besoin initial en fonctionnalité sera rempli.

```
composing
   a Translator ?t1
   a Translator ?t2
   grounding "C(start)"
   format    "compose '?t1#' and '?t2#'"
 having
   ?t1.to = ?t2.from
 gives a Translator
   with from = ?t1.from
   with to   = ?t2.to
```

10.13 Évaluation Critique et Perspectives

L'évaluation de nos travaux est faite à plusieurs niveaux : chaque aspect de notre proposition est évalué de manière puis nous évaluons la globalité de notre travail et comment les différents éléments s'entre-complètent. La section courante correspond au chapitre 9 du manuscrit original.

Pour améliorer la situation face aux problèmes d'intégration et de réutilisation des logiciels pour l'intelligence ambiante, nous avons proposé quatre éléments de solution :

- un intergiciel à service
- une méthode de conception
- un langage pour supporter cette méthode
- un environnement d'exécution pour automatiser l'utilisation de ce langage

Dans toutes ces propositions, nous avons promu une approche que nous nommons « transitionnelle » dans le sens où nous voulons créer des méthodes et outils très simple à utiliser par les développeurs visés. Le terme de « technologie transitionnelle » s'inspire du format « transitionnal (X)HTML » qui vise à aider les développeurs à faire la transition d'un format HTML sous contraint à un format HTML plus adapté à l'interprétation automatique comme le HTML stricte. Notre objectif est de proposer des méthodes et technologies aidant les développeurs à faire la transition du développement d'applications complètement statiques au développement d'application extensible, dynamique et sensible à l'apparition et la disparition des éléments logiciels.

L'évaluation des méthodes de génie logiciel et des frameworks est difficile. Il est facile de mesurer des performances brutes mais il est très difficile d'évaluer l'acceptation d'une de la méthode par les développeurs. Les modes d'évaluation possible reposent sur des études utilisateurs où l'utilisateur est le développeur d'application. Un des problèmes se posant lors d'étude d'utilisabilité pour les méthodes de développement logiciel est que le nombre potentiel d'utilisateurs est très limité, par exemple par rapport au cas d'une évaluation d'une nouvelle modalité d'interaction. Les scénarios d'évaluation sont de plus relativement longs : une demi-heure ne suffit pas pour apprendre et utiliser une méthode de développement logiciel.

Nous avons proposé mais pas mis en œuvre une méthode d'évaluation utilisant des étudiants en informatique comme sujets d'évaluation. L'idée serait de séparer une promotion d'étudiant en 2 groupes et de comparer leur progression en leur faisant utiliser deux méthodes de développement différentes pour remplir un même objectif. Cette piste est intéressante et permettrait d'aider à l'évaluation dans certains domaines ayant trait au génie logiciel. Ces évaluations à

base d'étudiants sont à mettre en œuvre avec beaucoup de précautions : il n'est pas acceptable de « sacrifier » une promotion d'étudiants. De plus, toute erreur doit attendre l'année (ou le semestre) suivant pour être répercutée sur le protocole expérimental.

Nous avons évalué OMiSCID sur plusieurs aspects en commençant par la latence induite par l'intergiciel en fonction du débit visée. Cette évaluation a montré que l'utilisation d'OMiSCID est acceptable pour des applications nécessitant une faible latence, et elle a validé le découpage architectural fait dans les chapitres précédents. Nous avons aussi évalué le degré de simplification apporté par la nouvelle API d'OMiSCID qui réduit considérablement la complexité et le volume du code nécessaire. Ces propriétés rendent l'écriture de programmes utilisant OMiSCID moins propices aux erreurs de programmation. Nous avons aussi évalué l'adoption d'OMiSCID et de son interface graphique au travers de statistiques faites sur la base de code de l'équipe de recherche PRIMA. Le nombre de logiciels utilisant OMiSCID suit l'évolution du volume de code (donc OMiSCID tend à être utilisé de manière régulière dans tout l'ensemble de la base de code). L'interface graphique d'OMiSCID tend aussi à être étendue par les différents développeurs de services.

L'évaluation de notre méthode de conception a été réalisée par l'analyse des propriétés des systèmes que nous avons reconçus en utilisant la méthode de conception. Ces systèmes sont plus dynamiques et plus adaptables. Ils permettent aussi une meilleure extensibilité et rendent possible une intégration dynamique a posteriori avec d'autres systèmes. La fragmentation en services permet de refléter fidèlement les capacités de l'environnement informatique et améliore effectivement la réutilisation des services existants.

Nous avons évalué notre langage UFCL sur des critères classiques concernant les langages de programmation : lisibilité, facilité d'écriture, fiabilité et coût global. De notre analyse ressort le fait qu'UFCL est un très bon langage dédié à la tâche considéré mais qu'il manque légèrement de possibilités de débogage.

Globalement, notre approche à base de technologies transitionnelles a fonctionné : OMiSCID est utilisé largement au sein de l'équipe PRIMA et de ses collaborateurs dans des projets de recherche. OMiSCID Gui, l'interface graphique d'OMiSCID, a eu une meilleure adoption que nous l'espérions. Avec pour modeste objectif initial de servir de plate-forme d'intégration pour les démonstrations, l'interface graphique est en fait utilisée quotidiennement par les concepteurs de services pour leur développement. Le concept d'usine à service a aussi été adopté rapidement dans le contexte de l'équipe PRIMA. L'utilisation d'UFCL est plus limitée que l'utilisation d'OMiSCID de part son accessibilité uniquement en Java. Une implémentation en C++ d'UFCL permettrait de l'utiliser plus largement. Cette seconde implémentation serait l'occasion de prendre en compte les différentes leçons que nous tirons de l'implémentation en Java.

En conclusion, nous pouvons noter que l'aspect transitionnel d'une technologie est capital pour son futur, en particulier dans l'intelligence ambiante où la variété des développeurs et spécialistes est importante. Le succès de l'intelligence ambiante repose sur la capacité des experts en génie logiciel à créer ces méthodes de conceptions qui soient utilisables par tous les acteurs impliqués dans le domaine. D'une manière plus large, les aspects sociologiques et psychologiques affectant à la fois l'utilisateur final et les concepteurs de logiciel sont à prendre en compte pour espérer que la vision de l'intelligence ambiante prenne vie.

Bibliography

[Url-a] "Ambient Intelligence" on Wikipedia
http://en.wikipedia.org/wiki/Ambient_intelligence.

[Url-b] The DBpedia website: a community effort to extract structured information from Wikipedia.
http://wiki.dbpedia.org/About.

[Url-c] "RDFa" on Wikipedia
http://en.wikipedia.org/wiki/RDFa.

[Url-d] The Semantic Extension to MediaWiki.
http://semantic-mediawiki.org/.

[Url-e] Protégé Platform and Editor Website.
http://protege.stanford.edu/.

[Url-f] W3C Submission for "OWL-S: Semantic Markup for Web Services"
http://www.w3.org/Submission/OWL-S.

[Url-g] The ESSI WSMO working group website: research and development efforts in the areas of Semantic Web Services.
http://www.wsmo.org/.

[Url-h] Video recording of a presentation on WSMO by John Domingue, chair of the WSMO working group.
http://videolectures.net/iswc07_domingue_wsmo/.

[Url-i] Selected presentation about Semantic Web Services technologies, OWL-S and WSMO.
http://www.wsmo.org/papers/presentations/SWS.ppt.

[Url-j] "Blocks World" on Wikipedia
http://en.wikipedia.org/wiki/Blocks_world.

[Url-k] The CASPER project website (in french).
http://www-prima.inrialpes.fr/casper/.

[Url-l] Example camera using a 1600x1200 sensor and delivering images at 30 Hz
http://www.prosilica.com/products/ge1650.html.

[Url-m] OMiSCID website including download, documentation and screencasts.
http://omiscid.gforge.inria.fr/.

[Url-n] W3C recommendation for Namespaces in XML
http://www.w3.org/TR/xml-names/.

[Url-o] Jena Semantic Web Framework.
http://jena.sourceforge.net/.

[Url-p] Some Publications Around Jena.
http://www.hpl.hp.com/semweb/publications.htm.

[Url-q] "Comparison of computer algebra systems" on Wikipedia
`http://en.wikipedia.org/wiki/Comparison_of_computer%_algebra_systems`.

[Url-r] "Nikita the Spider" web pages statistics including "doctypes" (HTML, transitional, etc.).
`http://nikitathespider.com/articles/ByTheNumbers/fall2008.html`.

[Url-s] Slides about programming language concepts and evaluation criteria.
`http://www.csee.umbc.edu/331/fall00/notes/finin1.pdf`.

[Url-t] Slides, in french, about programming language characteristics and evaluation criteria.
`http://www.site.uottawa.ca/~nat/Courses/PL-Course/PL-2.ppt`.

[Url-u] Research axes of the ICT-CONNECT project (Emergent Connectors for Eternal Software Intensive Networked Systems).
`http://www-rocq.inria.fr/arles/connect/connectresearch.html`.

[Url-v] Website of the ICT-CONNECT project (Emergent Connectors for Eternal Software Intensive Networked Systems) started in 2009.
`http://connect-forever.eu/`.

[Assad 2007] Mark Assad, David Carmichael, Judy Kay and Bob Kummerfeld. *PersonisAD: Distributed, Active, Scrutable Model Framework for Context-Aware Services*. pages 55–72. 2007.

[Auer 2008] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak and Z. Ives. *DBpedia: A Nucleus for a Web of Open Data*. In Proceedings of 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference (ISWC+ASWC 2007), pages 722–735. November 2008.

[Baldauf 2007] Matthias Baldauf, Schahram Dustdar and Florian Rosenberg. *A survey on context-aware systems*. International Journal of Ad Hoc and Ubiquitous Computing, pages 263–277, June 2007.

[Bell 1992] J.E. Bell and L.A. Rowe. *An exploratory study of ad hoc query languages to databases*. pages 606–613, Feb 1992.

[Benatallah 2003] B. Benatallah, Q. Z. Sheng and M. Dumas. *The Self-Serv environment for Web services composition*. Internet Computing, IEEE, vol. 7, no. 1, pages 40–48, 2003.

[Berardi 2005] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo and Massimo Mecella. *Automatic Composition of Process-based Web Services: a Challenge*. In Proc. of the WWW'05 Workshop on Web Service Semantics: Towards Dynamic Business Integration (WSS 2005), 2005.

[Bouchenak 2006] S. Bouchenak, N. De Palma, D. Hagimont and C. Taton. *Autonomic Management of Clustered Applications*. pages 1–11, Sept. 2006.

[Boudreau 2007] Tim Boudreau, Jaroslav Tulach and Geertjan Wielenga. Rich client programming: plugging into the netbeans platform. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[Brown 1996] A.W. Brown and K.C. Wallnan. *Engineering of component-based systems*. pages 414–422, Oct 1996.

[Brumitt 2000] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern and Steven A. Shafer. *EasyLiving: Technologies for Intelligent Environments*. In HUC '00: Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing, pages 12–29, London, UK, 2000. Springer-Verlag.

[Brønsted 2007] Jeppe Brønsted, Klaus Marius Hansen and Mads Ingstrup. *A Survey of Service Composition Mechanisms in Ubiquitous Computing*. Workshop on Requirements and Solutions for Pervasive Software Infrastructures, 2007.

[Chen 2004] H. Chen, T. Finin and A. Joshi. *Semantic Web in the Context Broker Architecture*, 2004.

[Choi 2006] Namyoun Choi, Il-Yeol Song and Hyoil Han. *A survey on ontology mapping*. SIGMOD Rec., vol. 35, no. 3, pages 34–41, 2006.

[Christensen 2002] Henrik Baerbak Christensen and Jakob Bardram. *Supporting Human Activities - Exploring Activity-Centered Computing*. In UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing, pages 107–116, London, UK, 2002. Springer-Verlag.

[Coen 1998] Michael H. Coen. *Design principles for intelligent environments*. In AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence, pages 547–554, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[Cook 2006] D.J. Cook. *Health Monitoring and Assistance to Support Aging in Place*. Journal of Universal Computer Science, vol. 12, no. 1, pages 15–29, 2006.

[Coutaz 2005] Joëlle Coutaz, James L. Crowley, Simon Dobson and David Garlan. *Context is key*. Commun. ACM, vol. 48, no. 3, pages 49–53, 2005.

[Crowley 2007] James L. Crowley, Daniela Hall and Rémi Emonet. *Autonomic Computer Vision Systems*. In 2007 International Conference on Computer Vision Systems, ICVS'07. Springer Verlag, Mar 2007.

[Dey 2001] A. Dey, D. Salber and G. Abowd. *A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications*, 2001.

[Dey 2009] Anind K. Dey. *Modeling and intelligibility in ambient environments*. Journal of Ambient Intelligence and Smart Environments, vol. Volume 1, 2009.

[Doan 2003] Anhai Doan, Jayant Madhavan, Pedro Domingos and Alon Halevy. *Ontology Matching: A Machine Learning Approach*. In Handbook on Ontologies in Information Systems, pages 397–416. Springer, 2003.

[Ducatel 2001] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten and J. C. Burgelman. *Scenarios for Ambient Intelligence in 2010*. Rapport technique, IST Advisory Group, February 2001.

[Emonet 2006] Rémi Emonet, Dominique Vaufreydaz, Patrick Reignier and Julien Letessier. *O3MiSCID: an Object Oriented Opensource Middleware for Service Connection, Introspection and Discovery*. In 1st IEEE International Workshop on Services Integration in Pervasive Environments, June 2006.

[Emonet 2008] Rémi Emonet and Dominique Vaufreydaz. *Usable developer-oriented Functionality Composition Language (UFCL): a Proposal for Semantic Description and Dynamic Composition of Services and Service Factories*. In 4th IET International Conference on Intelligent Environments, 2008.

[Erber 1975] Norman P. Erber. *Auditory-Visual Perception of Speech*. J Speech Hear Disord, vol. 40, no. 4, pages 481–492, 1975.

[Escoffier 2007a] C. Escoffier, R.S. Hall and P. Lalanda. *iPOJO: an Extensible Service-Oriented Component Framework*. pages 474–481, July 2007.

[Escoffier 2007b] Clement Escoffier and Richard S. Hall. *Dynamically Adaptable Applications with iPOJO Service Components*. 2007.

[Essa 2000] I.A. Essa. *Ubiquitous sensing for smart and aware environments*. Personal Communications, IEEE, vol. 7, no. 5, pages 47–49, Oct 2000.

[Euzenat 2007] Jérôme Euzenat and Pavel Shvaiko. Ontology matching. Springer-Verlag, Heidelberg (DE), 2007.

[Euzenat 2008] Jérôme Euzenat, Jérôme Pierson and Fano Ramparany. *Dynamic context management for pervasive applications*. Knowledge Eng. Review, vol. 23, no. 1, pages 21–49, 2008.

[Flissi 2008] Areski Flissi, Jérémy Dubus, Nicolas Dolet and Philippe Merle. *Deploying on the Grid with DeployWare*. In Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGRID'08), pages 177–184, Lyon, France, may 2008. IEEE. Rank (CORE) : A.

[Fowler 2004] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. `http://www.martinfowler.com/articles/injection.html`, 2004.

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design patterns: Elements of reusable object-oriented software. Addisson-Wesley, Toronto, Ontario. Canada, 1995.

[García-barriocanal 2005] E. García-barriocanal, M. A. Sicilia and S. Sánchez-alonso. *Usability evaluation of ontology editors*, 2005.

[Garlan 1995] David Garlan, Robert Allen and John Ockerbloom. *Architectural Mismatch, or, Why it's hard to build systems out of existing parts*. In Proceedings of the 17th International Conference on Software Engineering, pages 179–185, Seattle, Washington, April 1995.

[Garlan 2007] David Garlan and Bradley Schmerl. *The RADAR Architecture for Personal Cognitive Assistance*. International Journal of Software Engineering and Knowledge Engineering, vol. 17, no. 2, April 2007. A shorter version of this paper appeared in the 2006 Conference on Software Engineering and Knowledge Engineering (SEKE 2006).

[Gennari 2002] John H. Gennari, Mark A. Musen, Ray W. Fergerson, William E. Grosso, Monica Crubézy, Henrik Eriksson, Natalya F. Noy and Samson W. Tu. *The Evolution of Protégé: An Environment for Knowledge-Based Systems Development*. International Journal of Human-Computer Studies, vol. 58, pages 89–123, 2002.

[Giard 1999] M. H. Giard and F. Peronnet. *Auditory-Visual Integration during Multimodal Object Recognition in Humans: A Behavioral and Electrophysiological Study*. J. Cognitive Neuroscience, vol. 11, no. 5, pages 473–490, 1999.

[Gruber 1993] Thomas R. Gruber. *A translation approach to portable ontology specifications*. Knowl. Acquis., vol. 5, no. 2, pages 199–220, June 1993.

[Heineman 2001] George T. Heineman and William T. Councill. Component-based software engineering: Putting the pieces together (acm press). Addison-Wesley Professional, June 2001.

[Hofer 2003] Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger, Josef Altmann and Werner Retschitzegger. *Context-Awareness on Mobile Devices - the Hydrogen Approach*. In HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9, page 292.1, Washington, DC, USA, 2003. IEEE Computer Society.

[Hotho 2003] Andreas Hotho, Steffen Staab and Gerd Stumme. *Wordnet improves Text Document Clustering*. In In Proc. of the SIGIR 2003 Semantic Web Workshop, pages 541–544, 2003.

[Hunt 1999] Andrew Hunt and David Thomas. The pragmatic programmer: From journeyman to master. Addison-Wesley Professional, October 1999.

[Iftene 2008] Adrian Iftene and Alexandra Balahur-Dobrescu. *Named Entity Relation Mining using Wikipedia*. In Proceedings of the Sixth International Language Resources and Evaluation (LREC'08), Marrakech, Morocco, may 2008. European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2008/.

[Jovanovic 2007] M. Jovanovic and B. Rinner. *Middleware for Dynamic Reconfiguration in Distributed Camera Systems*. pages 139–150, June 2007.

[Kavantzas 2005] Nickolas Kavantzas, David Burdett, Greg Ritzinger, Tony Fletcher, Yves Lafon and Charlton Barreto. *Web Services Choreography Description Language Version 1.0*. World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109, November 2005.

[Khalaf 2003] Rania Khalaf, Nirmal Mukhi and Sanjiva Weerawarana. *Service-Oriented Composition in BPEL4WS.* In WWW (Alternate Paper Tracks), 2003.

[Kopecky 2007] J. Kopecky, T. Vitvar, C. Bournez and J. Farrell. *SAWSDL: Semantic Annotations for WSDL and XML Schema*. Internet Computing, IEEE, vol. 11, no. 6, pages 60–67, Nov.-Dec. 2007.

[Lam 1997] Ioi K. Lam and Brian Smith. *Jacl: a Tcl implementation in java*. In TCLTK'97: Proceedings of the 5th conference on Annual Tcl/Tk Workshop 1997, pages 4–4, Berkeley, CA, USA, 1997. USENIX Association.

[Lara 2004] Rubén Lara, Dumitru Roman, Axel Polleres and Dieter Fensel. *A Conceptual Comparison of WSMO and OWL-S*. In ECOWS 2004, volume 3250 of *LNCS*, pages 254–269. Springer, 2004.

[Layaida 2005] Oussama Layaida and Daniel Hagimont. *Adaptive Video Streaming for Embedded Devices*. IEE Proceedings - Software, vol. 152, no. 5, pages 238–244, octobre 2005.

[Lömker 2006] Frank Lömker, Sebastian Wrede, Marc Hanheide and Jannik Fritsch. *Building Modular Vision Systems with a Graphical Plugin Environment.* In ICVS, page 2. IEEE Computer Society, 2006.

[Marples 2001] D. Marples and P. Kriens. *The Open Services Gateway Initiative: An Introductory Overview*. IEEE Communications Magazin, vol. 39, pages 110–114, December 2001.

[Metze 2005] Florian Metze, Petra Gieselmann, Hartwig Holzapfel, Thomas Kluge, Ivica Rogina, Alex Waibel, Matthias Wölfel, James Laurence Crowley, Patrick Reignier, Dominique Vaufreydaz, François Bérard, Bérangère Cohen, Joëlle Coutaz, Sylvie Rouillard, V. Arranz, M. Bertran and H. Rodriguez. *The "FAME" Interactive Space*. In 2nd Joint Workshop on Multimodal Interaction and Related Machine Learning Algorithms, page 4 pages, Edinburgh United Kingdom, 2005.

[Miller 1995] George A. Miller. *WordNet: a lexical database for English*. Commun. ACM, vol. 38, no. 11, pages 39–41, 1995.

[Mozer 1999] M.C. Mozer. *An Intelligent Environment Must Be Adaptive*. Intelligent Systems and their Applications, IEEE, vol. 14, no. 2, pages 11–13, Mar/Apr 1999.

[Noy 2000] N. Noy, W. Grosso and M. Musen. *Knowledge-acquisition interfaces for domain experts: An empirical evaluation of protege*, 2000.

[Pautasso 2005a] Cesare Pautasso. *JOpera: An Agile Environment for Web Service Composition with Visual Unit Testing and Refactoring.* In VL/HCC, pages 311–313. IEEE Computer Society, 2005.

[Pautasso 2005b] Cesare Pautasso and Gustavo Alonso. *The JOpera visual composition language.* J. Vis. Lang. Comput., vol. 16, no. 1-2, pages 119–152, 2005.

[Pilato 2004] Michael Pilato. Version control with subversion. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[Rao 2004] Jinghai Rao and Xiaomeng Su. *A Survey of Automated Web Service Composition Methods.* In Jorge Cardoso and Amit P. Sheth, editeurs, SWSWPC, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2004.

[Reussner 2003] Ralf H. Reussner. *Automatic component protocol adaptation with the CoConut/J tool suite.* Future Gener. Comput. Syst., vol. 19, no. 5, pages 627–639, 2003.

[Roman 2005] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler and Dieter Fensel. *Web Service Modeling Ontology.* Appl. Ontol., vol. 1, no. 1, pages 77–106, 2005.

[Roman 2006] Dumitru Roman, Jos de Bruijn, Adrian Mocan, Holger Lausen, John Domingue, Christoph Bussler and Dieter Fensel. *WWW: WSMO, WSML, and WSMX in a Nutshell.* pages 516–522. 2006.

[Román 2002] Manuel Román, Christopher Hess, Renato Cerqueira, Roy H. Campbell and Klara Nahrstedt. *Gaia: A Middleware Infrastructure to Enable Active Spaces.* IEEE Pervasive Computing, vol. 1, pages 74–83, 2002.

[Schilit 1994] Bill Schilit, Norman Adams and Roy Want. *Context-Aware Computing Applications.* In In Proceedings of the Workshop on Mobile Computing Systems and Applications, pages 85–90. IEEE Computer Society, 1994.

[Shafiq 2007] Omair Shafiq, Matthew Moran, Emilia Cimpian, Adrian Mocan, Michal Zaremba and Dieter Fensel. *Investigating Semantic Web Service Execution Environments: A Comparison between WSMX and OWL-S Tools.* Internet and Web Applications and Services, International Conference on, vol. 0, page 31, 2007.

[Shvaiko 2008] Pavel Shvaiko and Jérôme Euzenat. *Ten Challenges for Ontology Matching.* pages 1164–1182. 2008.

[Srinivasan 2006] Naveen Srinivasan, Massimo Paolucci and Katia Sycara. *Semantic Web Service Discovery in the OWL-S IDE.* In HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences, page 109.2, Washington, DC, USA, 2006. IEEE Computer Society.

[Tecnologica 2004] Scientifica E Tecnologica, P. Traverso, M. Pistore, Istituto Trentino Di Cultura, P. Traverso and M. Pistore. *Automated Composition of Semantic Web Services into Executable Processes*, 2004.

[Weiser 1991] Mark Weiser. *The computer for the 21st century.* Scientific American, vol. 265, no. 3, pages 66–75, September 1991.

[Weiser 1996] Mark Weiser and J. Seely Brown. *Designing Calm Technology.* PowerGrid Journal, vol. 1.01, July 1996.

[Wrede 2006] Sebastian Wrede, Marc Hanheide, Sven Wachsmuth and Gerhard Sagerer. *Integration and Coordination in a Cognitive Vision System.* In ICVS, page 1. IEEE Computer Society, 2006.

[Zaidenberg 2008] Sofia Zaidenberg, Patrick Reignier and James L. Crowley. Reinforcement learning of context models for a ubiquitous personal assistant, volume Volume 51/2009 of *Advances in Soft Computing*, pages 254–264. Springer Berlin / Heidelberg, september 2008.

[Zaragoza 2007] H. Zaragoza, J. Atserias, M. Ciaramita and G. Attardi. *Semantically Annotated Snapshot of the English Wikipedia v.1 (SW1).* `http://www.yr-bcn.es/semanticWikipedia`, 2007.