User-centric Service Composition - Towards Personalised Service Composition and Delivery

Eduardo Manuel Gonçalves da Silva

UNIVERSITY OF TWENTE.

Enschede, The Netherlands, 2011

CTIT Ph.D.-Thesis Series, No. 11-191

Cover Design:	Nayeli Arias López
Cover Illustration:	"Lines of choice", by Nayeli Arias López
Cover Effects:	Nayeli Arias López
Book Design:	Lidwien van de Wijngaert and Henri ter Hofte
Printing:	Ipskamp, Enschede, the Netherlands

Graduation commitee:

Chairman, secretary:	prof.dr. P. J. J. M. van Loon (University of Twente)
Promotor:	prof.dr.ir. M. Akşit (University of Twente)
Assistant Promotors:	dr. L. Ferreira Pires (University of Twente)
	dr.ir. M. J. van Sinderen (University of Twente)
Members:	prof.dr.ir. M. Aiello (University of Groningen)
	prof.dr. S. Dustdar (Vienna University of Technology)
	prof. C. Pautasso (University of Lugano)
	prof.dr.ir. L.J.M. Nieuwenhuis (University of Twente)
	dr.ir. A. Pras (University of Twente)
	dr. A. Wombacher (University of Twente)

CTIT Ph.D.-Thesis Series, No. 11-191
ISSN 1381-3617
ISBN 978-90-365-3182-5
Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

USER-CENTRIC SERVICE COMPOSITION - TOWARDS PERSONALISED SERVICE COMPOSITION AND DELIVERY

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 11 mei 2011 om 14.45 uur

door
Eduardo Manuel Gonçalves da Silva
geboren op 20 mei 1981
te Vila Real, Portugal

Dit proefschrift is goedgekeurd door:
prof.dr.ir. M. Akşit (promotor) , dr. L. Ferreira Pires (assistent-promotor),
en dr.ir. M. J. van Sinderen (assistent-promotor)

Abstract

With computing devices and the Internet becoming ubiquitous, users can make use of network-based software applications in different places and situations. Mobile devices with Internet connectivity are a good example of this trend. Network-based software applications are being exposed to users as web services, which makes them accessible on demand, when and where the users require them in their daily life.

Often users have to make use of several services to fulfil their requirements. The required services can be combined into a service composition, to deliver a value-added service that fulfils all the different requirements of a given user at a given moment. Creating such service compositions beforehand is difficult, or even impossible, as service developers would need to define all the possible service compositions that end-users may require. However, and given that services can be exposed through the Internet, service compositions can possibly be created on demand, driven by the users, whenever users require a given functionality that cannot be delivered by a single existing service.

In this thesis we address the problem of personalised service delivery through on demand composition of existing services. To achieve such personalised service delivery, we claim that *user-centric service composition* supporting approaches are required. User-centric service composition aims at the creation of a new service composition to fulfil the requirements of a specific end-user. Furthermore, we also consider that service composition supporting process needs to be personalised to the user creating the service composition, which is not necessarily the service end-user. Proper attention should be given to the fact that users are heterogeneous, i.e., they have different knowledge, technical skills and may use services in different situations. Furthermore, this process will mainly take place at runtime, which imposes real-time behaviour constraints to the service composition supporting system.

We assume that in many application domains users have lim-

ited technical knowledge, which implies that the supporting system for service composition must shield the users from the details of the service composition process. To achieve this, users must be provided with some degree of automation on the service composition process. To provide such automation we have developed the *DynamiCoS* framework, which automates the discovery and composition of services. Such automation is achieved by using semantic-based technologies: semantic service descriptions, ontologies and semantic reasoners. DynamiCoS also supports the process of semantic services publication. Published services are represented in a language-neutral formalism, which allows the publication and composition of services described in different languages.

To evaluate DynamiCoS we have developed a prototype implementation. We observed that there is a lack of evaluation methodologies, and a shortage of real world semantic services collections, to help in the evaluation of semantic service composition approaches. Consequently, we have developed a framework for the evaluation of semantic service composition approaches. Following the defined evaluation framework, we have showed that the DynamiCoS approach is capable of automatically discovering relevant service compositions for a given service request. Furthermore, the service composition process can be delivered in real-time, although the processing time can rapidly increase if many semantic services are handled. However, DynamiCoS assumes that users can specify declaratively all the properties of the required service (composition) in one interaction. Such an assumption limits the set of users that can be supported, as most of the users lack application domain knowledge to specify all the details of the required service in one interaction. Generally, users require multiple interactions with the supporting system to acquire information about the application domain in order to drive the service composition process.

To overcome this limitation, we have extended DynamiCoS to support a flexible and multi-step interaction between the user and the supporting system. This support is adaptable and allows, for example, to assist users with limited application domain knowledge, by allowing them to acquire information about the domain, and its services, so that they can specify and decide on the services to be used in the service composition. This extension of the DynamiCoS framework resulted in the A-DynamiCoS framework (*Adaptable-DynamiCoS*). The A-DynamiCoS framework consists of two parts: a domain specific *front-end user support*

and a generic *back-end supporting system*. The front-end communicates with the back-end by using predefined commands, or *primitive commands*, which can embed the user intentions at each step of the composition process. *Primitive commands* provide a given functionality, which is implemented by the basic components of the composition framework, which are on the back-end supporting system. Primitive commands can be used to define a strategy in terms of *command flow*. Such strategies are domain specific, and depend on the target user population to be supported in a given usage scenario that makes use of service composition to support personalised service delivery. Commands can be exposed to the users with intuitive interfaces that can collect the user intentions and requirements. New primitive commands can be introduced, if necessary, without having to change the previously defined commands. Furthermore, the same primitive commands can be used to design multiple front-end user support command flows, to support different types of users in different application domains. A-DynamiCoS also addresses the problem of incremental composition and execution of services. This problem arises from the fact that users many times decide, after executing a service composition, that they require further services. To tackle this the process of service composition and execution can interleave in A-DynamiCoS.

To evaluate A-DynamiCoS, we have developed a prototype implementation. The prototype reuses the components of the DynamiCoS framework implementation, namely the basic components of the composition framework, which are used to implement the different primitive commands. Furthermore, we have developed other components to provide the aimed adaptability of our approach. To validate the A-DynamiCoS applicability in different application domains and to support different types of users, we have used A-DynamiCoS to support two different use cases, one in the *e-government* domain and another in the *entertainment* domain. Users of these application domains have different characteristics and require different types of support regarding their service composition process. The user interfaces were developed as normal web pages, defining dialogues with the users. The A-DynamiCoS generic back-end supporting system was able to provide the required support to both use cases with the same set of primitive commands, combined in different command flows. Furthermore, we have also evaluated the performance of A-DynamiCoS back-end supporting system, by measuring the time taken to process the different commands issued from the front-

end user support. We observed that the responses were delivered in real-time, i.e., the users did not experience long waiting times for the issued commands. Based on the performed evaluations we can conclude that the A-DynamiCoS framework is adaptable and applicable to different application domains and users, and furthermore it delivers real-time response to users. These results provide a good motivation for using this type of techniques to provide users with more personalised service delivery.

Acknowledgements

O valor das coisas não está no tempo que elas duram, mas na intensidade com que acontecem. Por isso existem momentos inesquecíveis, coisas inexplicáveis e pessoas incomparáveis.
–Fernando Pessoa

Life is a composition of events, actions and moments, but mainly it is about people we get to know, to live with, to work with, to share moments with. In these last four years I have been very fortunate to meet incredible people. I want to thank all of them at this very important moment of my life.

First I want to thank the people that so warmly received me in the ASNA group when I arrived in the cold February 2007. I have wonderful memories of the moments we spent together. We also faced not so good moments, but we all managed to move on and show that we all could do great and relevant work, and this is what remains at the end. Still in 2007 I moved to a new group, TRESE, where I felt very welcome and immediately integrated. I want to thank all the people that made that transition smooth and also thank for the nice environment you always provided me in the course of my PhD. Specially I want to thank our secretary, Jeanette, who helped me so much in the last agitated moments of the PhD, dealing with all the bureaucratic issues that needed attention.

A PhD is a very intense process, where many times one feels lost or needs to take critical decisions. On those moments it feels great to have the support and words of wisdom from people that can very quickly grasp the problems we are facing. I want to thank my promoter Mehmet Aksit for always providing me very constructive comments and suggestions. It was a pleasure to work with you, and I certainly learned a lot with you. Furthermore, daily I had the supervision of two amazing people and researchers, Luís Ferreira Pires (with whom I discussed research and life in Por-

tuguese) and Marten van Sinderen (from whom I got a constant inspiration to work hard and be persistent). I hope you enjoyed our collaboration as much as I did. You have taught me so many things and always kept me motivated and inspired during this entire journey. Thank you for everything, I hope we will keep working together, and I am sure we will continue creating new and interesting things together.

Throughout this journey I have had the pleasure of working and sharing many moments with two other very special people. My office mate and my "neighbour", and my good friends, and paranymphs in my defense, Laura Maria and Luiz Olavo. I want to tell you that you made my PhD life nicer and more enjoyable. When I was disappointed with something, you were there to tell the right words, and that meant a lot to me. I know we will keep in close contact with each other, since you are both very special people to me.

I want to thank my students: Jorge, Joni and Edwin, who have contributed immensely to the ideas, concepts, prototypes and validation of this thesis. Thank you for everything and for the very nice moments we spent together.

In my defense I feel very honoured to have Prof.dr. S. Dustdar, Prof.dr. M. Aiello, Dr. C. Pautasso, Prof.dr. B. Nieuwenhuis, Dr. A. Pras and Dr. A. Wombacher as members of my graduation committee. I want to thank you for accepting the invitation and for your constructive comments on my thesis.

I want to send a special word to my colleagues and friends at the University of Twente: Tiago, Rodrigo, Ricardo(s), Tom, Ramon, Rita, Eduardo, Rafael, Idilio, all the UTKring futsal players, and all the others. It was very nice to share this time with all of you. I also want to send a special word to my latin friends, with whom I enjoyed so many nice moments outside the university, namely: Julian, Oscar, Daniela, David, Juan, Diruji, Maite, Jorge, Mario, Jealemy, Christian and all the others. You made the grayish Dutch days look brighter.

I also want to share this special moment with my older friends. You kept showing me that even away from each other, we are still those good friends, which can so quickly see through each other, no matter how long we don't see each other.

To my family: Zulmira, Cândido, João, Filipe (e Sara). Vocês foram, são e serão parte fundamental da minha vida, e do que mais bonito e profundo eu tenho. Quero agradecer-lhes por estarem sempre presentes, apesar desta distância fisíca que temos vivido. O culminar do meu doutoramento não é uma conquista apenas

minha, e eu quero que vocês a sintam também, pois vocês sempre me fizeram acreditar e sonhar que posso realizar os meus objectivos.

Lastly, I want to share this moment also with Nayeli, my girlfriend, and also her family (Maria Reyna, Miguel Angel, Itzel, Mattias, Frijol). Nayeli obrigado por estares ao meu lado nestes quatro anos tão importantes na minha e na nossa vida. Esta conquista não é apenas minha, é nossa!... uma de muitas que teremos juntos! *Te amo.*

Eduardo M. Gonçalves da Silva
Enschede, April 2011

Contents

Chapter 1

Introduction

With computing devices and Internet becoming ubiquitous, users can make use of network-based software applications in different places and situations, possibly exposed to users as services. Services are abstractions that deliver value from a provider to a user. In many situations the requirements of a specific user cannot be fulfilled by any single existing service. In such situations the user can be delivered with several existing services, which composed can fulfil the user requirements. In this thesis we investigate the problem of *user-centric service composition*, which aims at supporting the process of creating service compositions to personalise service delivery to the specific requirements of an end-user. To achieve such personalisation, we claim that users have to play a central role in the composition process, driving the process of service composition. However, users are heterogeneous, i.e., they have different knowledge, technical skills and may use services in different situations. Therefore, the system that supports the service composition process needs not only to capture the end-users' requirements but also to capture the users' characteristics in order to support them accordingly.

This chapter is organised as follows: Section 1.1 presents some background to position the topic of this thesis; Section 1.2 presents our motivation; Section 1.3 states the problems and research questions; Section 1.4 defines the thesis' objectives; Section 1.5 presents the approach followed in the research; Section 1.6 presents the scope of the thesis; and finally Section 1.7 presents the structure of the thesis.

1.1 Background

The first computers were a very selective technology affordable by few institutions and individuals [1]. However, in the last decades

of the 20^{th} century, with the advances in electronics and its miniaturisation, computers became a commodity [2]. At the beginning of the 21^{th} century we are observing a continuous increase of daily use of computing devices, namely mobile devices and embedded computing devices, which are making the idea of *ubiquitous computing systems* [3] a reality. We have observed a similar evolution with communicating systems, from selective use to ubiquity nowadays. The Internet is representative for this evolution of communication systems. In the last decade of the 20^{th} century we observed the appearance of the *world wide web* (WWW) [4] [5], which made the Internet a central pillar of our society. The *Web* allowed initially to create a global network of documents, which could be interconnected and accessible from "anywhere". The Internet also fostered other types of applications, the so called network-based applications, which enables different computers to collaborate in a distributed fashion and deliver a given functionality, not achievable by only one entity.

Distributed computing [6] allows the creation of new applications as combination of different parts, possibly residing in different computers in different networks. This allows to distribute the computing workload and functionality required for a given computation over different computers. For example one computer may be responsible for computing the application presentation to its users, while all the other computations are delegated to other computers. This organisation of computing systems offer many possibilities to the system designers, namely to organise and reuse applications for different purposes. Another major advantage is that different companies, specialised in different activities, can use each other's services. This allows each company to focus on a different problem, while reusing third party services for supporting auxiliary activities. For example, many companies nowadays outsource their customer relationship management (CRM) system, or invoice system, to third party service providers [7] that are specialised in such activities. This improves the efficiency of the company, since they do not need to *re-create* yet another system, or have a dedicated department, to handle these auxiliary activities. To achieve the reuse of services and define such collaborations designers are following *service-oriented* principles, which define how providers can deliver services that other parties can make use of.

1.1.1 Services Orientation

Service-orientation can be used to design distributed systems and to structure the collaboration of distributed applications. The basic construct in this methodology is the concept of *service*.

There are many definitions of service, for example WordNet[1] provides the following definition: "the work done by one person or group that benefits another". Although this is a general definition describing the services provided by a person, it clearly presents the basic entities that participate in a service, namely one entity performing an action to deliver value and another entity that receives the value produced.

Figure 1-1 Service-orientation

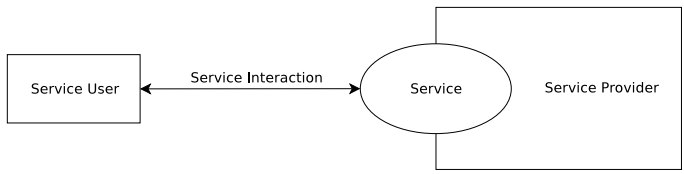


Figure 1-1 presents the basic architecture of a *service-oriented system*. There are two entities in the architecture: *service provider* and *service user*. A service provider is an organisation or individual capable of delivering some functionality. Service providers "externalise" their functionality, or their system, as services, which interested users can benefit from. A service user interacts with the service provider, and this *service interaction* defines the service delivery, i.e., the delivery of the service value from the provider to the user.

Service-oriented systems are common in our society. For example: a computer repair shop can fix computers from people that have broken computers. In this case, the service provider is the computer repair shop, and service users are people with broken computers. The computer shop may have many different services, for example: repair computer at home, install new operating system, repair or change hardware, etc. All the repair services are supported by a computer repair shop system that consists of technicians, computer pieces, software tools, etc. The computer repair shop system internal organisation may not be visible to the people that need their computers repaired. It only needs to expose the services delivered, which represent the "external perspective" of the computer repair shop, i.e., the services offered to people that need their computers fixed.

When service-orientation principles are applied to computing

[1] http://wordnet.princeton.edu/

systems, one normally use the term *service-oriented computing* (SOC) [8] [9]. The main objective of SOC is to make computing system applications available as network-based services. Application owners become service providers, as they expose their applications as network-based services. The beneficiaries of the services, i.e., the service users, consume the services.

Providing applications as services brings many advantages to the design of distributed systems. For example, services are normally message-oriented, i.e., they can be used by exchanging defined messages following a given protocol. This message-oriented approach allows to expose applications in a technology-independent fashion. Different applications can thus be implemented in different programming languages and running in different operating systems and still collaborate with each other via the defined service messages. Furthermore, services are normally described following a given standard specification language, for example, Web Service Description Language (WSDL) [10] in case of Web services [11] [12]. Such a service specification provides interested users with the necessary information to contact the service provider and use a service, possibly without further prior knowledge on the service provider and its applications. These service characteristics enable one of the major advantages of service-oriented systems, the possibility of composing different services to deliver a value-added functionality.

1.1.2 Service Composition

Because many network-based applications are exposed as services, one can combine different services in order to define value-added services. We refer to the process of combining multiple services into a new *composite service* as *service composition* [13].

Figure 1-2 shows an example of a service composition. The service composition is accomplished by a workflow that a computer repair shop can define to support its customers on requesting a service to repair computers. We consider that four basic component services are used: *ProblemDescription*; *AtHomeRepair*; *AtStoreRepair* and *EmailService*. The flow of activities is as follows:

1. The user arrives at the store's web site and is presented with the interface to interact with the *ProblemDescription* service; using this service he describes what is the problem and where he wants the computer to be repaired (at home or at the store). The information provided by the user is passed to the store technicians to arrange the repairing of the computer;
2. Based on the user selected service (at home or at the store re-

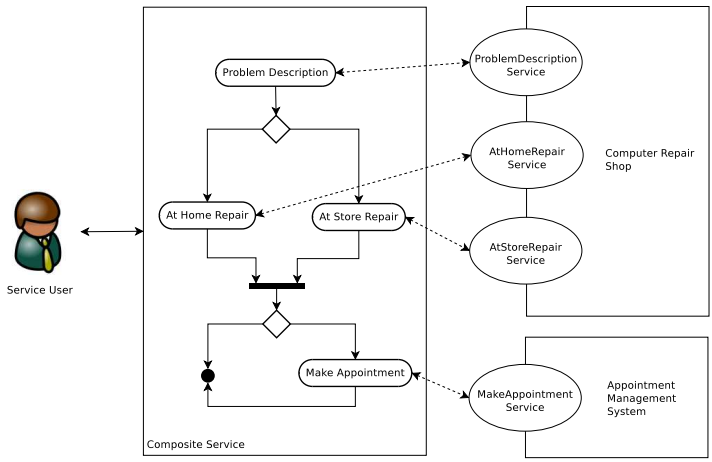


Figure 1-2 Example of service composition

pair), or given the nature of the problem, he is automatically forwarded to either the *AtHomeRepair* or the *AtStoreRepair* service. In the selected service the user enters his data and gets an estimate of the costs of the repair;

3. If the user agrees with the proposal, he proceeds with setting the arrangements for the repair; the final arrangements are confirmed by sending an email to the user and communicated to a technician. If the user does not agree with the proposal, he does not proceed and the process stops.

From this simple example we can see that the service composition is a powerful mechanism, as one can reuse existing services and define a process as a flow of services that realise multiple requirements that need to be supported. The same system could have been developed as a dedicated application service, only performing these tasks. However, such solution would be inflexible and expensive, as its functionality, which implement the basic components functionality, would not be reusable in other contexts.

The creation of service compositions can take place at two distinct times: *design-time* and *runtime*.

Design-time service composition is being widely studied and used already in industry. Many different approaches exist [14] [15] for design-time service composition. There are already industry accepted standards, such as the Business Process Execution Language (BPEL) [16], which is a language to define service compositions, as orchestration of basic component services. These approaches assume that there is a set of requirements at design-time for defining a service composition. The requirements are identified based on the needs of a given set of users or organisa-

tion. A service developer defines a composite service, by composing existing services in such a way that the set of requirements are satisfied. Once the service composition is developed, it can be deployed so that users can make use of it. In these approaches, most of the times, there is a professional service developer that defines a service composition for service end-users.

In the case of *runtime* service composition the process is different. The idea is that the composite service is created at runtime, when the service is required. This means that service creation and service execution are not separated in time, both take place at runtime. Therefore, the role of the professional service developer needs to be automated, otherwise the "on demand" requirements of such runtime process cannot be met. The main objective of this type of approaches is to make the service composition more personalised to its final user, or service end-user. In the design-time approaches, the service composition is designed for several possible end-users, who may be different and have different requirements. In the runtime approaches, such user heterogeneity can be captured and taken into account during the composition process, personalising service compositions to their specific end-users. For this, flexible approaches are required, to adapt and take the specific requirements of an end-user into consideration. This type of service composition support is the central research topic of this thesis. We refer to this process as *user-centric service composition*.

1.2 User-centric Service Delivery

Runtime service composition enables the personalisation of service delivery by combining a set of services into a composite service that a given user requires at a given moment. Such personalised service delivery is what we call *user-centric service delivery*. This process contrasts with other service delivery approaches, where one service (composition) is defined without knowing the specific end-user that is going to make use of the service.

To motivate user-centric service composition, we present a simple example, Figure 1-3, where two users have a set of specific requirements that need to be fulfilled. User 1 requires a service that allows him to meet R1, then R2, and if he manages to get a positive result from R2 he has still R4. User 2 requires a services that allows him to meet R4 and then R2. We can observe that in the available set of services there are services that allow the

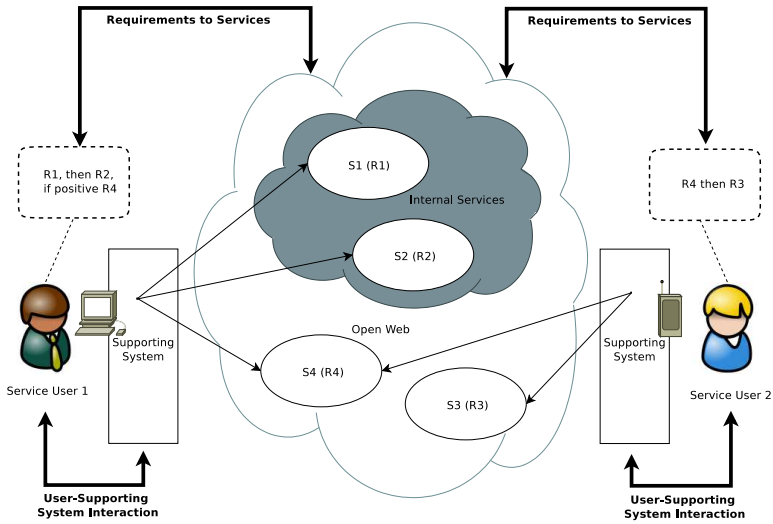


Figure 1-3 User-centric service composition example

users to meet each of the requirements they have. Furthermore, we consider that users are in two different situations, for example: User 1 is at work, using a desktop computer and part of his requirements deal with his company's internal services (e.g.: check availability of a colleague, or booking a meeting room); while User 2 is a mobile user requiring only services from the "open web".

In general, service users are heterogeneous. For example, they have different knowledge of the application domains where they are seeking services, they have different knowledge on the services that exist in the domain, they have different technical skills to interact with the service composition *supporting system*, they use different devices, they are in different situations or have different context, etc. If we consider the example presented above, we can observe that User 1 is distinct from User 2, although they could actually be the same person, in different situations (e.g.: at work and at home). This is a consequence of the current service requirements the user has at each moment, and the domains of the services that deliver the required functionality. For example, when designing support for service delivery, it is reasonable to assume that User 1, which is at work, has knowledge of the domain where he is seeking services, on the contrary one can not assume that User 2 has a complete knowledge on the general services from the "open web". Furthermore, one cannot assume that these two users have the same technical skills to define a service composition that fulfil their needs through service delivery.

In this example we can see that the user can play multiple

roles in the service composition process. User can be the service *end-user* or the service *composer*, or service composition developer. We normally refer to the term *user*, however it may have different meanings in the context of user-centric service composition processes. *End-user* role is played by the user that provides the requirements for the service composition to be composed, and is the user that executes the resulting service composition. However, during the process of creating a service composition, to meet the requirements of a specific end-user, the user plays the role of *composer*, by interacting and commanding the composition environment in order to compose the service. The same user can play both roles, however there may be situations that such does not happen.

To cope with user (*end-user* and *composer*) heterogeneity, the supporting composition environment has to be designed taking in consideration the characteristics and requirements of the user that is to be supported. To support user-centric delivery the supporting composition environment has to bridge two types of requirements from the user: *service requirements (or service goals)* (for end-users) and *user intervention in the process of service composition* (for composers). *Service requirements* need to be bound to concrete services that can deliver the functionality needed by the end-user. *User intervention in the process of service composition* has to do with how the user interacts with the composition environment during the service composition process. These interactions are very much dependent on the characteristics of the user driving the composition process. In this thesis we focus on these two dimensions in order to define mechanisms to support *user-centric service composition*.

1.3 Problem Statement

Service composition can be used to personalise service delivery in order to fulfil the specific requirements of service end-users. We define this process as *user-centric service composition*. To achieve such personalisation, the service composition process has to be performed on demand, based on requirements of a specific service end-user. In this thesis we assume that users playing the role of service composition developers (composers) are mainly non-professional developers, i.e, they have limited technical knowledge on the service composition process. In order to support these users in the service composition process, we further assume that

the whole process is mediated by a computer agent, which is capable of interpreting the user requirements, find meaningful services and compose them to fulfil the end-user requirements. Based on these assumptions, the main research question in this thesis is:

How to support user-centric service composition to achieve personalised service composition and delivery?

This question can be sub-divided into the following research questions:

- **RQ1**: What mechanisms are required to automate the service composition process so that non-professional users can drive the service composition process, possibly at runtime?
- **RQ2**: How to support different types of users, i.e., users with different requirements and characteristics, such as, for example, application domain knowledge, services knowledge and technical skills?

1.4 Objectives

The main objective of this thesis is to advance the area of user-centric service composition by developing methodologies, architectures and infrastructural support. The process of supporting user-centric service composition should be adaptable to the characteristics of the user who is creating a service, as a composition of existing services, to deliver services personalised to the requirements of a end-user.

Users have different knowledge and technical skills. To cope with this user heterogeneity the supporting system has to be adaptable to the user, i.e., the system has to be capable of reacting to the particular characteristics and *intentions* of users. To provide adaptable support, the system has to be designed as a configurable architecture, where configuration can be done based on the requirements and characteristics of the user that is being supported.

To address these issues, and the problems stated in Section 1.3, we define the following objectives in this thesis:

1. *Automation*: define automated service composition support, to enable non-professional users to drive the service composition process, whenever they need it, possibly at runtime;
2. *User-centricity*: support users according to their requirements (end-user) and characteristics (composer);

3. *Adaptability*: define adaptable and flexible support that allows different users to be supported in different ways, according to their knowledge and technical skills.

1.5 Approach

To address the objectives defined for our research, we have used the following approach, also shown in Figure 1-4:

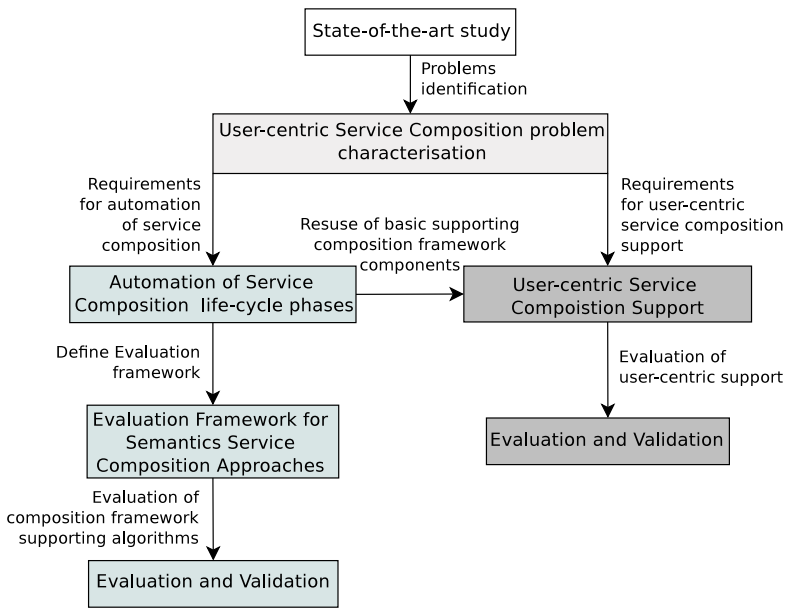


Figure 1-4 Approach to thesis research

– We have performed a state-of-the-art study on the topic of user-centric service composition techniques. From this study we observed that the existing trends on user-centric service composition focus on the automation of the composition process and also on the creation of intuitive graphical approaches to define service compositions, mainly at design-time. However, most of the approaches neglect the characteristics of the users interacting with the system, specially how do the users communicate requirements for the service composition process;
– To be able to characterise the users and to define their participation in a user-centric service composition process, we have studied the user-centric service composition process aiming at identifying requirements for systems that are designed to support such processes. In user-centric service composition users

can play two basic roles: the *end-user*, or consumer, of the service composition that is being created; and the *composer*, or service developer, of the service composition being created for the end-user. These two roles are not always performed by the same actor. Although in general we refer as user centric service composition to the process of creating a service composition based on the requirements of a specific end-user, the user playing the role of *composer* also needs user-centric support, so that he, with his specific characteristics, can succeed on driving the service composition process. Based on this study we have defined a set of design issues to be taken into account when designing user-centric service composition supporting systems;

– To support on-demand service composition, and to shield users from the technicalities of the service composition process, we have developed automated support for the different phases of the service composition life-cycle. These developments resulted on a framework for *dynamic service composition*, the DynamiCoS framework. We have used semantic services and ontologies to enable automatic reasoning on the different phases of the service composition process;
– We observed that evaluation methodologies for semantic service composition approaches are lacking. Different authors use different evaluation methodologies, which makes it difficult to have fair and objective comparison of different approaches. To contribute in this area we have developed an evaluation framework for semantic service composition approaches. This evaluation framework was used to evaluate the implementation of our dynamic service composition approach since it implements a semantic service composition approach;
– The proposed framework for dynamic service composition only tackled the automation of the service composition process. The characteristics of the user driving the service composition process were not considered. To overcome this, we extended the basic dynamic service composition framework with an adaptable coordinator, resulting on the A-DynamiCoS framework. The aim of this extension is to adapt the service composition support according to the characteristics of the user driving the service composition process. The adaptable coordinator, allows to implement different commands, which realise different behaviours required to support the service composition process, such as service discovery, service selection, service composition, service execution, etc. These commands

can be used in different orders, according to the requirements of the user driving the service composition process. Furthermore, these commands can be embedded in different types of user interfaces, allowing to shield users from the actual technical details associated with the commands and the service composition process. This provides a flexible mechanism, which is adaptable to characteristics of different types of users, since different users can use different commands, in different orders, to drive the composition process, which makes the process user-centric;

– We have developed a prototype of the proposed user-centric service composition approach. To validate the approach, we have applied it in two different use cases in two different domains, with different requirements and different types of users. For each use case a specific front-end user support was designed. In this validation we observed that the proposed approach can be applied in different situations, and support different users, which demonstrated its adaptability and applicability to different situations.

1.6 Scope

In this thesis we focus on the development of mechanisms necessary to support user-centric service composition. We concentrate specially on the process of supporting the composition of existing services to satisfy specific end-user requirements, possibly at runtime. We assume that users driving the service compositions creation, by making use of a supporting system, are heterogeneous, which means that they may have different requirements from a supporting system.

The proposed supporting approach aims at providing the necessary mechanisms to fill the gap between users' requirements and the existing services that composed can better fulfil the users' requirements. We focus on the user characterisation and on the development of an adaptable support that can fit the user being supported, which may not be the service end-user.

In this thesis we do not address the following issues:

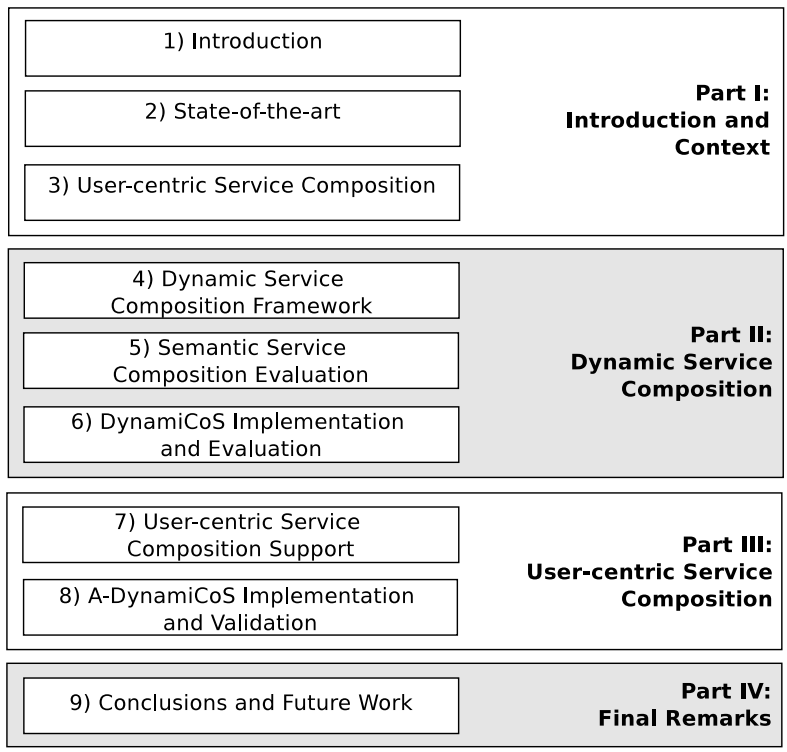
– *User privacy, security and trust*: this is an important issue, but it requires a separate research effort;
– *Ontology engineering*: although we use ontology-based techniques to automate parts of the service composition process, we do not investigate how to design or reason on ontologies;

– *User context and preference models*: user context and preferences information are considered to facilitate the service composition and execution processes. However, we do not investigate neither develop models to optimise the representation and interpretation this type of information.

1.7 Thesis Structure

This thesis consists of four parts: 1) Introduction and Context; 2) Dynamic Service Composition; 3) User-centric Service Composition; 4) Final Remarks. Figure 1-5 presents the thesis structure, indicating how the chapters of the thesis relate to these parts. In the following we introduce each of the chapters of the thesis.

Figure 1-5 Thesis structure



Part I: Introduction and Context

– *Chapter 1 - Introduction*: provides an introduction of the thesis, including some background information, the problem addressed in the thesis, its objectives, the approach we have

taken, and its scope;
- *Chapter 2 - State-of-the-art*: provides an overview of the basic concepts necessary to understand the problem of user-centric service composition. Presents an overview on the typical service composition life-cycle phases and the stakeholders that participate on the service composition process. Based on the service composition life-cycle we define a classification scheme, which we use to present and compare relevant approaches to user-centric service composition;
- *Chapter 3 - User-centric Service Composition*: introduces the user-centric service composition problem. Discusses the central role of the user in the composition process and present some example scenarios for user-centric service composition. It discusses user heterogeneity, which should be considered when designing supporting environments for user-centric service composition. We also analyse how the properties of the application domain influence the user-centric service composition process. This chapter concludes with the formulation of design issues to be considered on the development of user-centric service composition approaches.

Part II: Dynamic Service Composition

- *Chapter 4 - Dynamic Service Composition Framework*: presents our approach towards the automation of the activities of the service composition life-cycle, which resulted in the DynamiCoS framework;
- *Chapter 5 - Semantic Service Composition Evaluation*: proposes an evaluation framework for semantic service composition approaches. This framework focuses on assessing the quality of proposed service compositions and the scalability of the supporting discovery and composition algorithms;
- *Chapter 6 - DynamiCoS Implementation and Evaluation*: presents the prototype implementation of the DynamiCoS framework. Based on the developed prototype implementation, we evaluate our approach for dynamic service composition, using the evaluation framework presented in Chapter 5, as in DynamiCoS we develop a semantic service composition approach.

Part III: User-centric Service Composition

- *Chapter 7 - User-centric Service Composition Support*: discusses the design of the framework to support user-centric service composition. The objective is to extend the Dynam-

iCoS framework in order to consider the user participation in the service composition process. The extended framework, A-DynamiCoS, aims allowing the development of supporting environments for users with different characteristics. This is achieved by defining user commands, which are realised, on demand, by the back-end supporting system, providing in this way the required adaptation to the user being supported in the service composition process;

– *Chapter 8 - A-DynamiCoS Implementation and Validation*: presents the prototype implementation the A-DynamiCoS framework. To validate A-DynamiCoS applicability to support different users and situations, we have used the developed prototype to support two different use case scenarios, in two different application domains, where different types of users are supported in the task of service composition;

Part IV: Final Remarks

– *Chapter 9 - Conclusions and Future Work*: reflects on the work presented in this thesis. We discuss the most important contributions and the main problems encountered in the course of the thesis. Furthermore, we present some challenges and problems in the area of user-centric service composition and delivery.

Chapter 2

State-of-the-Art

In this chapter we provide an overview on the state-of-the-art in the area of user-centric service composition. In our study we distinguish between two groups of approaches. The first group, *dynamic service composition*, focuses on approaches that mainly addresses the service automation of the composition life-cycle phases. The second group, *user-centric service composition*, focuses on approaches that aim at involving users in the process of composing their services.

This chapter is organised as follows: Section 2.1 presents basic terminology and concepts in the area of user-centric service composition, which are used in the remaining of this thesis; Section 2.2 presents the classification scheme for characterising and positioning the different approaches to service composition analysed in our state-of-the-art study; Section 2.3 presents a literature review on approaches for *dynamic service composition* and *user-centric service composition*; and finally Section 2.4 presents a discussion on the state-of-the-art study performed.

2.1 Terminology

2.1.1 Service-orientated Computing

Service-Oriented Computing (SOC) [17] is a paradigm used for designing distributed system based on the concept of network-based services. In the following we present some fundamental terminology and concepts in the area of service-oriented computing.

Service-Oriented Architecture

The *Service-Oriented Architecture* (SOA) [18] [19] has been proposed to define design principles for creating service-oriented computing systems.

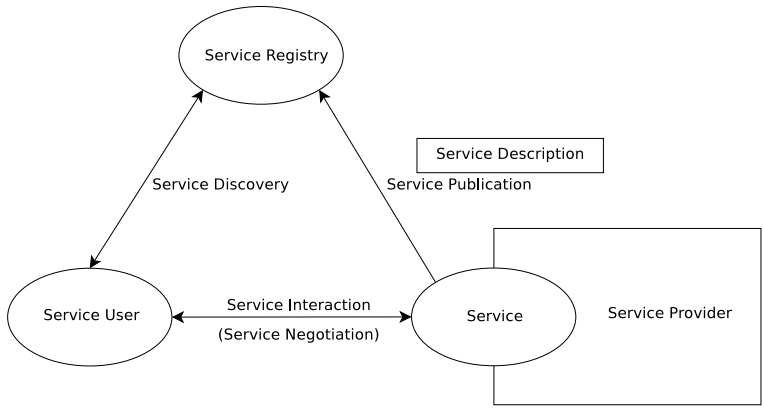


Figure 2-1 Service-Orientated Architecture

Figure 2-1 shows a diagram describing the different entities and their relations in a service-oriented architecture. There are three basic entities in a service-oriented architecture: *service provider*, *service user* and *service registry*. *Service provider* offers services. *Service user* makes use of services, by interacting with service providers. *Service registry* stores *service descriptions*, also known as *service specifications*. Service descriptions are created by the service providers to specify and advertise what their services do and where they can be reached. Given this, they can be seen as a means to find the service provider and to govern how clients can consume a given service. The service registry offers publication and discovery functions. The publication function allows service providers to publish their services, by using service description documents. The discovery function allows interested service users to locate services that can fulfil certain requirements. Once a service user has discovered a service, a negotiation phase may take place between the service user and the service provider to establish the conditions of the service consumption. The negotiation usually is concerned with *service level agreements* (SLAs) the provider has to deliver to the service user.

Although SOA principles are defined for computing systems that expose their applications as services, similar principles are applied in other fields. For instance, in the example presented in Section 1.1.1 of the "computer repair shop", a similar approach can be adopted to govern the whole system. The computer repair shop (service provider) can publish, or advertise, its services in the yellow pages. A person with a broken computer (service user) discovers the computer repair shop services in the yellow pages. Then the person contacts the computer shop, discussing the problem and negotiating the conditions to repair his computer. Once

they reach an agreement, the computer shop delivers the repairing service to the user.

SOA defines general design principles to define service-oriented collaborations. SOA does not define a specific implementation. Many different implementations can be derived from the SOA design principles. The most common implementation of SOA are based on Web services [11] [12].

Service Coordination

Considering that multiple services exist and can perform different tasks, one can define coordinations of different services in order to accomplish a complex task that cannot be delivered by only one existing service. The collaboration between different services, or multi-party collaboration, is normally defined as service *choreography* [20], while the centralised coordination of different services to accomplish a given task is normally defined as service *orchestration* [20].

Figure 2-2 shows an example of a *choreography* of services. Services participating in the choreography may belong to different parties. The aim is that the participating services collaborate to implement a given process. In Figure 2-2 the process consists of three different services. The service user triggers the process by invoking service A with a request. Service A processes the user request, and then invokes service B, which performs an operation and then invokes service C. Service C processes the request from service B and sends the result to the service user. This choreography implies that all the services participating in the process are aware of the services they have to collaborate with. In practice, this means that the execution of this process is multi-party, each participating service has to implement its logic to comply with the overall choreography. Web Services Choreography Description Language (WS-CDL) [21] is a language approach defined for the specification of web services choreography. Nevertheless, WS-CDL specifications are not executable and need to be transformed into different executable parts, to be executed in the different participants on the choreography.

Figure 2-3 shows an example of an *orchestration* of services. Although, as in the choreography case, services participating in the orchestration may belong to different providers, in an orchestration they are coordinated from a central entity, the *orchestrator*. The orchestrator invokes each service according to a given strategy. For example, in Figure 2-3, the process consists of the same services as in Figure 2-2, but in this case services are coordinated

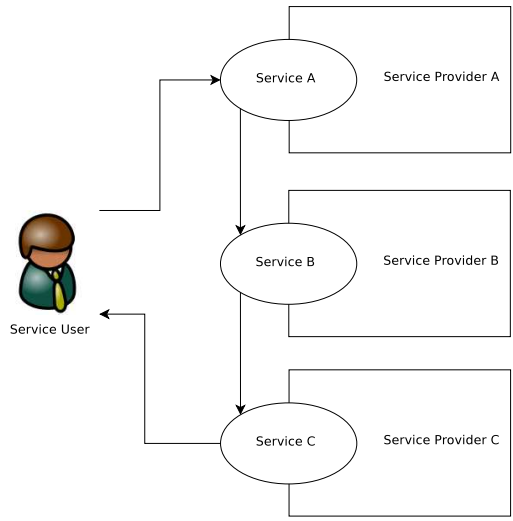


Figure 2-2 Choreography of services

by another service, the *composite service*, which defines the composition of the services participating in the process. The service user triggers the process by invoking the orchestrator, or composite service. Once the orchestrator receives the user request, the first action it takes is to invoke service A. There can be different interaction patterns with services that participate in an orchestration, we refer to [22] for an overview on service interaction patterns. The next activity in the process, after invoking service A, is invoking service B. Service B processes the request and responds with a message. Based on this response message, service C is invoked. Finally, when the orchestrator receives the response message from service C, it replies with a message to the service user. There are several approaches to service orchestration processes [14]. The most commonly used is Business Process Execution Language (BPEL) [16], which is an executable language that allows the definition of service orchestrations. BPEL processes can be deployed in a BPEL engine, which makes the orchestration available as a new service, a *composite service*, ready to be executed. From the perspective of the service user the orchestration, or composite service, is like any other service, with a given interface and protocol to be used. Users do not have to be aware of the internal implementation of the (composite) services, which is another major advantage of service-oriented computing.

In this thesis, we focus on service composition, which basically can be defined as an orchestration process, where multiple services are composed to deliver a value added functionality.

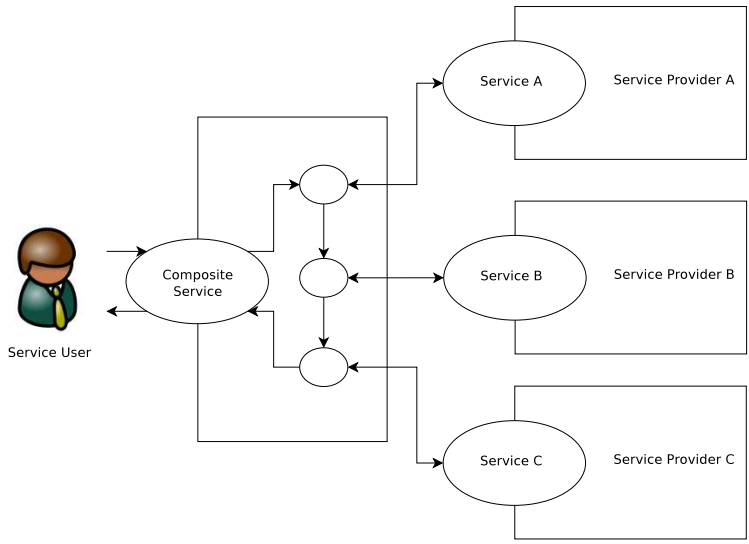


Figure 2-3 Orchestration of services

2.1.2 Service Composition Life-cycle

Figure 2-4 presents a service composition life-cycle [23], depicting the phases and stakeholders associated with the service composition process, and their relations. This life-cycle is similar to other proposed service composition life-cycles, such as [24]. In [24], the authors divide the service composition into five phases: 1) *planning phase*; 2) *definition phase*; 3) *scheduling phase*; 4) *construction phase*; and 5) *execution phase*. In the service composition life-cycle used in this thesis we consider two other auxiliary phases, which address the creation and publication of new services. These phases are of interest in our work, as they focus on the creation and publication of the basic components that later can be used in service compositions.

Stakeholders

We consider two stakeholders in this life-cycle: *Service developer* and *End-user*.

The *Service developer* is the stakeholder that creates services and publishes them in the service registry. These services are used as the basic components in the composition process. Service developer can create new services from scratch or as composition of existing services. Service developers create services or compose services at design-time.

The *End-user*, in our life-cycle is a stakeholder that drives the service composition process to create a service for himself. End-

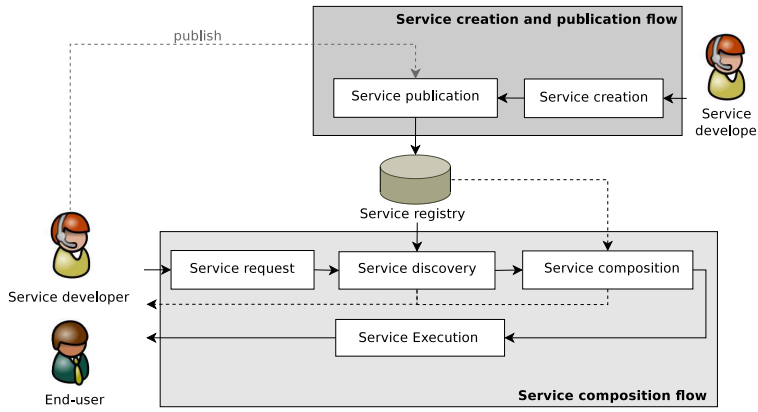


Figure 2-4 Service composition life-cycle

users normally create service composition at runtime, whenever they require a given service. This stakeholder normally does not play a direct role in all the activities of the service composition process, it only provides requirements to drive the service composition process.

Service Creation and Publication

The *service creation* phase is performed by service developers, who create new services by programming new applications and making them available as services, or build new service compositions from existing services, making the resulting compositions available as a new services. The service creation phase also encompasses the definition of service description documents by the service developer.

The service description documents are used in the *service publication* phase to publish the functional and non-functional information of the services in a service registry. It allows to advertise services, so that they can be discovered whenever they are required in a service composition process.

Service Composition

The first phase of the service composition flow is the specification of a *service request*, where the user indicates requirements and preferences for the composite service to be created.

Once the service request is defined, the *service discovery* phase takes place. In this phase candidate services for the service composition are discovered in the service registry. In case no services are discovered, the requirements for the service may need to be

reformulated and refined.

The following phase is the *service composition* phase, in which the discovered services are composed to meet the requirements specified in the *service request* phase. In the service composition phase further interactions with the service registry may take place, in case other services are necessary to complement the already discovered services.

Once the specified service requirements can be fulfilled by the created service composition, the resulting service can be executed. In the *service execution* phase the end-user makes use of the resulting service. Alternatively, in the case that a service developer is driving the service composition process, the resulting service composition may be published in the service registry so that it can be used by other end-users or service developers in the future.

2.1.3 Semantic Services

Services can be described at different levels, namely *syntactical* and *semantical*. Syntactical service descriptions are defined in a *human readable* formalism, such as WSDL [10], which define the service operations interfaces and protocols that need to be followed to invoke the services. To compose different services, for example in an orchestration, developers have to agree on the semantics of the operations and data structures handled by each service, i.e., what do they represent and what they do. In syntactical service descriptions this knowledge is not specified in the service description document, it is managed in an ad-hoc fashion by the service developers and interested service users. The management of this information in this way becomes difficult and cumbersome if many services are handled. Furthermore, it makes it difficult to introduce any kind of automation in the composition process, as human interaction is always needed to make the selection of services or to validate that the correct services are being composed.

The problem of managing large sets of information and resources is in fact becoming critical in many areas, namely in the world wide web (WWW), and the Internet in general, where daily large amounts of new information and resources are being made available [25] [26]. This increase of available resources makes the task of discovering the most suitable resource difficult for users. To tackle this problem new techniques are being developed, for example the *semantic web* [27]. The semantic web aims at providing web resources with semantic information, which define and specify the meaning of the resources. The semantic information is

defined in machine readable structures, that refer to concepts of a given domain and how they relate to each other. This domain conceptualisation is normally defined as *ontology* [28]. Ontologies allow to describe the concepts of a given domain, i.e., what they mean and how they are related to each other. Figure 2-5 provides an extract of a wine ontology [29], which is used to describe types of wines and how can these be combined with certain food dishes. Based on this domain ontology, one can annotate a given web resource as belonging to a given class of the ontology. For example, wine producers can define which wines they produce, e.g.: "PinotBlanc" or "Burgundy", and whenever someone is looking for a given type of wine, this information can be used to filter out the wine producers that produce the wine the user is interested in.

Figure 2-5 Excerpt of wine ontology

Semantic web techniques are being applied to web services, defining the so called *semantic web services* [30]. Semantic web services consist of adding semantic information to the existing syntactical description, e.g.: the WSDL file description. The semantic description annotates the different operations and parameters of the services with semantic information, referring to a domain on-

tology. Figure 2-6 presents an example of a semantic web service description, referring to the wine ontology, introduced in Figure 2-5. In this simple example one can see that the service interface, which describes an operation supported by the service, is described at a *syntactical* level and at a *semantical* level. At the syntactical level the service interface has one input of type "string" and an output of type "string". Such syntactical description is required to grant that the correct data type is used when invoking this service or using the output of the service, for example to compose with another service input. However, such description is not meaningful, i.e., one can not automatically reason on what the input or the output represents. However, if one considers the semantical description, such automatic reasoning is possible. In this case the input parameter is of type "Wine" and the output of type "Country". Based on this information, which is a reference to the wine ontology, which has these two concepts and describes their relation and properties, automatic reasoning can be performed. For example, and assuming that this service allows to find from which country a given wine is possible, one can discover this service based on the semantical description of the service interface parameters.

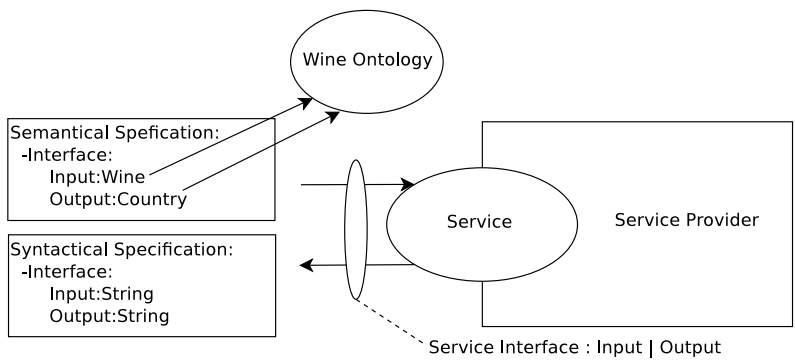


Figure 2-6 Semantic service

The use of semantic information on a service description enables several automatic computations, not possible if only a syntactical description is available. For example, services can be discovered based on the semantic annotations of the service operations. Furthermore, semantic information also enables automatic composition of services, by composing a given service output parameter with another services input parameters, if they are of the same semantic type or semantically related.

Semantic services are used in this thesis to enable the automation of the different activities in the service composition process.

2.1.4 User-centric Service Composition

User-centric service composition refers to the process of composing services to fulfil the requirements of a specific service end-user, possibly on demand, whenever the user requires such service (composition). This process allows to personalise the service delivery to the specific needs of the service end-user. To support such personalisation the service composition process may take place at different moments, namely at runtime.

Figure 2-7 presents our reference architecture of a user-centric service composition supporting system.

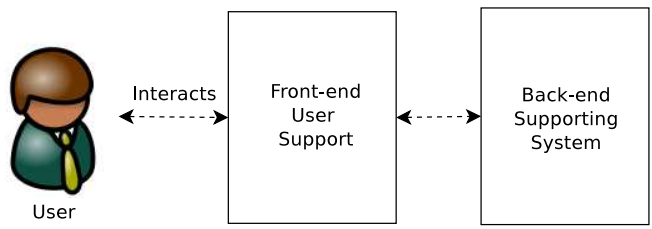


Figure 2-7 User-centric service composition reference architecture

We distinguish the following components in the user-centric service composition architecture:

- *User*: the user driving the service composition process to create a service composition that fulfils the requirements of a specific end-user. The user driving the service composition process can also be the service end-user;
- *Front-end User Support*: provides the interface to the user, and governs the way the user interacts with the service composition back-end supporting system. This component does not have to be aware of the internal details of the service composition process, it is mainly concerned with mediating the interactions between the user and the back-end supporting system;
- *Back-end Supporting System*: the system that manages the service composition process.

This thesis mainly focus on the back-end supporting system, and the management of the user-centric service composition processes.

2.2 Classification Scheme

To perform our state-of-the-art study and compare the different service composition approaches, in a common setting we define a classification scheme.

The proposed classification scheme is based on the phases of

the service composition life-cycle and the level of automation that the service composition approaches deliver on each of the activities that support these phases. In the following we present the classification scheme, considering the service composition life-cycle phases presented in Section 2.1.2, namely: *service request phase*; *service discovery phase*, *service composition phase* and *service execution phase*.

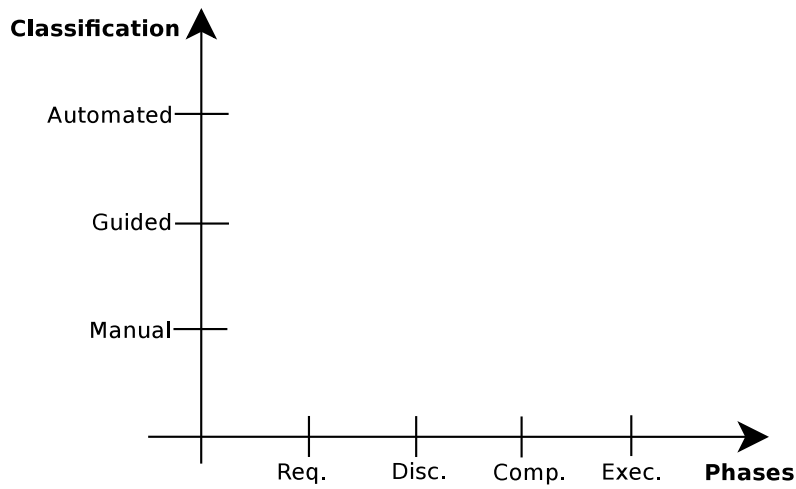


Figure 2-8 Service composition support

Figure 2-8 presents the service composition life-cycle phases considered in our study and the level of automation that can be provided to support these phases. We consider three levels of automation: *Manual*, *Guided*, and *Automated*. *Manual* support means that the user is required to interact at each step of the activities that support the phase, namely handling and validating each activity being performed to support a given life-cycle phase. *Guided* support means that the system guides, or assists, the user by automating several steps of the activities performed in the phase. However, in this level of automation the user is still required to interact in some steps of the activities performed in the phase. *Automated* support means that the system handles automatically the activities performed in the service composition life-cycle phase, without requiring intermediary interactions in the course of performing the activities required to support a given life-cycle phase.

Furthermore, in our classification scheme we also consider the role of the user in the service composition process. We consider that users can have two roles in a user-centric service composition process: compose services (*composer* role), and/or executing

services (*end-user* role). *Composer* is the role of the user that creates a service composition, by driving the service composition process based on a set of requirements from a specific *end-user*. The *end-user* is the role of the user that provides the requirements for the service composition to be created, and the user that then executes the resulting service composition. Although users can play both roles, there will be situations where different users play these two roles. We consider that the situations where user plays both roles are the most interesting for user-centric service composition processes, as it corresponds to the situation where the service end-user, which provides the requirements for the service to be composed, also guides the service composition process to create the service composition that delivers the different requirements. This corresponds to the situation where a given service composition is created on demand to personalised to the specific requirements the user has.

To present and compare the classification of the service composition approaches analysed in our state-of-the-art study we use the Table 2-1. This table has five columns. The first four columns present the level of automation that the service composition approach delivers in the four service composition life-cycle phases we presented above. The level of automation can be *manual*(•), *Guided* (••), and *automated* (•••). The fifth column presents the user roles that are considered in the service composition approaches, namely *Composer* (C) and/or *End-user* (E).

Table 2-1 Classification table

	Req.	Disc.	Comp.	Exec.	User
Classification					

(•) Manual | (••) Guided | (•••) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

2.3 Literature Review

In the following we present a literature review of approaches related with user-centric service composition. We distinguish two groups of approaches: *dynamic service composition* and *user-centric service composition* approaches. We report on the different service composition approaches using the classification scheme presented in the previous section.

2.3.1 Dynamic Service Composition

Dynamic service composition aims at supporting "on demand" composition of services, to satisfy a given set of requirements a user has at a given moment. To cope with such "on demand" requirement, some of the service composition life-cycle phases to provide an *automated* or *guided* support.

We have divided the dynamic service composition approaches in two groups: *automatic service composition algorithms* and *dynamic service composition frameworks*. The *automatic service composition algorithms* specially focus on automating the service discovery and composition phases of the service composition life-cycle. The *dynamic service composition frameworks* address the service discovery and composition phases, but furthermore they also address other issues and phases required to enable the service composition process, namely the description and publication of services.

Dynamic service composition has received a lot of attention in recent years. We refer to [15] [31] [32] [33] for an overview of existing approaches for dynamic service composition. In [34] we have also studied some relevant frameworks to support dynamic service composition.

Automatic Service Composition Algorithms

Zhang et al. [35] propose an algorithm for automatic composition of semantic services. They aim at automating the whole service composition process. The service composition is represented as a directed graph, where nodes, representing services, are linked by edges that represent semantic matching compatibility ($Exact$, $Subsume$, $PlugIn$, $Disjoint$) [36] between the output parameter of a service and the input parameters of another service. All the possible compositions between the services published in the registry are pre-computed, and re-computed every time new services are added to the service registry. Based on the services compositions graph and the requirements for the service composition, namely starting conditions or inputs and goals or desired outputs, the composition algorithm finds the shortest sequence of web services from the starting conditions or inputs to the goals or desired outputs. This approach computes the best composition according to the semantic similarity of output and input parameters of web services composed in the path from requested inputs to requested outputs. However, it does not grant that the compositions found deliver the behaviour the user expects, as it does not specify what

requirements have to be delivered by the different services of the service composition, only the desired service (composition) inputs and outputs. Table 2-2 presents our classification of this approach.

Table 2-2 Classification of Zhang et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	•	• • •	• • •	×	C

(•) Manual | (••) Guided | (• • •) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Rao et al. [37] propose an approach for automatic semantic web services composition using linear logic (LL) theorem proving techniques. They use the DAML-S [38] *ServiceProfile* semantic description language to describe services, to be published on their service registry. Furthermore, they also use a DAML-S representation to describe the user service request. Both, service description and user requests are translated to an internal representation that consists of extralogical axioms and proofs in linear logic. To represent the created composite services, they use a process calculus. The composite service is generated by performing a theorem proving technique, LL theorem prover. The LL theorem prover computes whether the user's request for a service can be achieved by a composition of the available services. If this is the case, the process model for the composite service is automatically extracted from the proof. The resulting composite service is transformed into DAML-S *ServiceModel* or to BPEL [16], which allows the user to execute the created service composition. Semantic reasoning is used in the course of the composition process, namely to detect subtyping between the services semantic descriptions. Table 2-3 presents our classification of this approach.

Table 2-3 Classification of Rao et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	•	• • •	• • •	×	C

(•) Manual | (••) Guided | (• • •) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Lécué et al. [39] [40] propose an automatic service composition approach based on the notion of *semantic links*. In this approach services are discovered based on a set of requirements that the user specifies. All the discovered services are stored in a matrix, the *causal link matrix* (CLM), which contains all the possible semantic compositions between input and output parameters of the discovered services. Based on the CLM the authors propose in [40] an AI planning-based service composition, which finds a composition of services, in the CLM, that satisfy the user service request. The

service request specifies two types of parameters: 1) the desired inputs, or the user *knowledge base*, which represent the inputs the user know and can provide; 2) the desired outputs, or the *goal* states, which represent the results the user wants to obtain when he executes the service (composition). Both types of parameters are defined as semantic concepts, which are references to the ontologies considered in the domains of the services considered. The service request parameters are used to find the most suitable composition of services in the previously constructed CLM. Table 2-4 presents our classification of this approach.

Table 2-4 Classification of Lécué et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	•	•••	•••	×	C

(•) Manual | (••) Guided | (•••) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Dynamic Service Composition Frameworks

METEOR-S [41] is one of the most comprehensive frameworks for semantic-based service composition. The approach provides mechanisms for semantic services annotation [42], service discovery [43], and service composition [44]. However, METEOR-S focuses mainly on design-time creation of service compositions. It supports two types of service composition processes, *static* composition and *template-based* composition. Service composition is based on the semantic service descriptions. In case of static composition, the services used in the service composition are selected at design time. In the case of *template-based* service composition [45], at design time the composition is defined as a template that consists of types of services and their execution flow, while at runtime there is a dynamic binding with concrete services. This dynamic binding is performed based on the user preferences and QoS parameters. Service composition execution is based on an orchestration engine. Table 2-5 presents our classification of the METEOR-S approach.

Table 2-5 Classification of METEOR-S

	Req.	Disc.	Comp.	Exec.	User
Classification	•	••	••	×	C

(•) Manual | (••) Guided | (•••) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

The SODIUM (Service-Oriented Development in a Unified Framework) [46] [47] project proposes a unified methodology for the integration of heterogeneous services, namely web services, p2p

services and grid services, which employ different forms of supporting the different phases of the service creation, description, discovery and composition. To approach such heterogeneity they propose a unified representation (*Generic Service Model* - GeSMO) for the different types of services using a UML-based representation. Given this common representation they propose a methodology for graphical service composition, for which they developed a language (*Visual Service Composition Language* - VSCL). To support service discovery they propose a unified language (*Unified Service Query Language* - USQL), which allows to describe the different types of services. The SODIUM service composition methodology consists on four phases: 1) user describes a workflow of tasks in a high-level abstract representation (using VSCL); 2) the abstract representation of the service composition, the tasks, is used to generate queries for services discovery; 3) the discovered services are used to substitute each of the abstract tasks defined in phase 1, hence transforming the abstract representation into a concrete service composition; 4) the concrete service composition representation is transformed into an executable composite service. This approach is mainly focused on supporting design-time service composition, aiming at facilitating the discovery and integration of different types of services. This work uses several mechanisms developed on [48], namely the visual mechanisms developed to support service composition. Table 2-6 presents our classification of the SODIUM approach.

Table 2-6 Classification of SODIUM

	Req.	Disc.	Comp.	Exec.	User
Classification	•	••	••	×	C

(•) Manual | (••) Guided | (• • •) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

The MOSCOE approach [49] proposes a framework for service composition and execution. Their main distinguish factor from other approaches is the support of reformulation of the service request [50]. The service composition is performed in three steps: *abstraction*, *composition* and *refinement*. *Abstraction* allows to specify service descriptions and service requests in a high-level formalism, namely state machines representation. Service providers describe their services by using the combination of the semantic service description language OWL-S [51] and the web services description language (WSDL) [10]. The user request is expressed in a state machines representation, where basically the user specifies functional and non-functional properties for the service to be composed. The composition process is performed using

Symbolic State Systems (STS), i.e., both the service request and service representations have to be translated from the state machines representation to the STS formalism. The service composition process is carried automatically, based on the user service request and the available services. If there is a set of services that can satisfy the service request, the composite service is returned, otherwise the user is presented with a message stating where the composition process could not proceed. Based on this information the user can refine the service request, and request a new iteration in the composition process. This refinement process stops whenever the user decides or a composite service that matches the user service request is found. Whenever a composite service is successfully created it can be translated to BPEL [16], which allows then the deployment and execution of the created service composition. They propose the use of non-functional properties for the composite service, which can be monitored once this is deployed for execution and in case they do not meet the requested levels they can be reported to the user. Table 2-7 presents our classification of the MOSCOE approach.

Table 2-7 Classification of MOSCOE approach

	Req.	Disc.	Comp.	Exec.	User
Classification	••	•••	•••	×	C

(•) Manual | (••) Guided | (•••) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Fujii and Suda [52] [53] propose a dynamic service composition approach based on the use of semantic services. They propose an architecture with three components to deal with the whole service composition process. For service representation and modelling they introduce a model called Component Service Model with Semantics (CoSMoS), which integrates the semantic information of a component and the functional information of a component into a single semantic graph. A unified interface named Component Runtime Environment (CoRE) to convert different component implementations onto the CoSMoS representation. Given the semantic support provided by CoSMoS a semantic-based service composition mechanism called SeGSeC was developed, which generates a service composition to match the service request. In the CoSMoS model, service operations may be semantically described using predicates, meaning that the operation is associated with a predicate to represent its semantics, which the authors claim is more precise that the definition of operations in terms of IOPEs. CoRE allows to convert different services, described in different technologies, to the CoSMoS formalism. This allows

then the SeGSeC to make use of these services for the actual service composition. CoRE has two interfaces: Discovery Interface and Access Interface. SeGSeC offers a natural language service request interface and generates an execution path, specifying which operations or properties in which components should be accessed and in what order. The SeGSeC architecture consists of the following processes, implemented as agents: user request analysis (RequestAnalyser), execution path generation (ServiceComposer), semantic matching (Reasoner), and service execution (ServicePerformer). The RequestAnalyser parses the user request string into a CoSMoS semantic graph representation. The service composition process consist on the following steps:

1. User request analysis: given the user natural language service request, it is broken into individual words, next for each word the RequestAnalyser discover corresponding CoSMoS nodes and links through the CoRE discovery interface. They assume the service request contains a single predicate (e.g.: print), a noun which is the object of the predicate (e.g.: direction) and auxiliar nouns with prepositions (e.g.: from home, to restaurant), the following is the syntax definition for the user service request in SeGSeC: *UserRequest := PredicateObject*, where *Object := Noun(Concept Noun)*;
2. Operation Discovery: given the user service request in the SeGSeC format (semantic graph format), the ServiceComposer first searches for an operation that performs the predicate in the request, using the CoRE Discovery interface. If no operations are found, the ServiceComposer discover other operations by finding synonymous of the predicate or by generalizing the predicate using ontologies;
3. Input Complement: after discovering the target operation, the ServiceComposer finds the necessary components that can be supplied as inputs to the operation. These components can be: 1) specified by the user in the service request, 2) outputs of other components operations, and 3) properties of other components. A component can be provided as input to an operation if the data types and concepts (semantic) are compatible with the operation input;
4. Semantic Matching: given the possible execution path through the input complement process the ServiceComposer checks whether the semantics of the discovered path match the user request or not, through the usage of the Reasoner component;
5. Service Execution: if the service composition semantic match

the user request, it is passed to the ServicePerformer component, which is responsible for the service execution. In case the user disagrees with the proposed service composition, it is passed to the ServiceComposer to complement the found service composition inputs. Execution also makes use of the user context information and history in order to facilitate the service execution process.

In this approach the user can play both roles: *composer* and *executor*. Although we did not find many details with respect to the execution phase, where the user executes the found service composition. Furthermore, and although natural language service requests are interesting and intuitive approaches, the authors did not show detailed evaluation on the application of this approach for service request. Table 2-8 present our classification of this approach.

Table 2-8 Classification of Fuji and Suda

	Req.	Disc.	Comp.	Exec.	User
Classification	●●	●●●	●●●	●● / ×	C-E

(●) Manual | (●●) Guided | (●●●) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Kona et al. [54] propose an approach for the automatic composition of semantic web services. They propose an approach that uses a graph-based service composition algorithm. The composition process is performed using a *multi-step narrowing algorithm*. The user specifies a service request, or a *query service*, specifying the *IOPE* parameters for the desired service. The composition problem is then addressed as a discovery problem, starting by discovering the request inputs and preconditions, and iteratively resolving the *open* outputs and post-conditions (or effects) until the requested outputs and post-conditions are resolved. They assume that service providers describe their services in USDL [55]. Since all the service discovery and composition processes are performed in Prolog with Constraint Logic programming, services are pre-processed from USDL and transformed to Prolog terms. Pre-processing tends to be time consuming, which is sensitive in the case of runtime service composition support. Each time a service is added to the registry, the complete registry has to be pre-processed and updated with the new structure. Table 2-9 presents our classification of this approach.

Table 2-9 Classification of Kona et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	●●	●●	●●	×	C

(●) Manual | (●●) Guided | (●●●) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

2.3.2 User-centric Service Composition

User-centric service composition aims at supporting the creation of service compositions to fulfil the requirements of a specific end-user. To accomplish this, these approaches consider a higher level of involvement of users in the service composition process, when compared with the dynamic service composition approaches presented in the previous section. Furthermore, users driving the service composition process can have different characteristics and can have limited technical knowledge. These approaches have to consider the user characteristics to provide a suitable support in the service composition process.

From our study we could observe that the majority of research efforts in user-centric service composition are concentrated on the creation of visual and intuitive representations of the service composition process. The most common type of such visual approaches is based on the concept of *mashup* [56]. A mashup is a methodology that can be used to combine and aggregate data, presentation or functionality, from two or more sources to create a new service. Mashups are being extensively applied in web pages, allowing to create new web pages as a combination of different sources of data and presentation information. The most popular type of mashup is the display of specific locations in Google Maps. Mashups are also being applied in the service composition process by defining intuitive interfaces that allow to combine different resources (services). Services are represented normally with visual abstractions that shield users from the technicalities associated to the services representation and also the service composition process. These characteristics allow to involve users without advanced technical skills in the service composition process. Apart from these visual approaches, we have also identified other approaches that apply service composition in a user-centric manner. In this class there are approaches designed for very specific application domains, such as, for example pervasive computing and context aware applications. These approaches define mechanisms that take the user context, preferences and actions to drive the service composition process without the user being explicitly informed of such.

Mashup-based approaches

Liu et al. developed an approach [57] [58] [59] to support users, mainly non-professional users, in composing services. In their approach services are assumed to be published and *tagged* with information, which characterise and give semantics to the service descriptions. Based on these service descriptions they proposed an approach to *mine* the existing services' tags and based on these present the users with a "cloud tag", which can be used by the users to select services. Upon the selection of a service the user is prompted with services, of the same class, and furthermore with other services that can be composed onto the service outputs. User is also presented with quality of service (QoS) information on the different services, which allow to support the user on the selection of a service. The user drives the composition process and decides when a service composition fulfils his requirements. The support tooling has a mashup-like graphical interface, where the services and recommendations are presented. The architecture is divided in three levels:

- Service Layer: where a crawler is used to create the service repository and a service analyser to mine the existing services and service tags, and present matchings given the user decisions;
- Knowledge Layer: where the cloud tag is created, the service advisor presents suggestions to the user, and the composition graph is managed;
- User Layer: where the user interacts with the service composition mashup, receives suggestions from the service advisor, and manages and executes the composition.

They assume two types of creation of cloud tags:

- Top-down: by creating ontologies used to describe services semantically. They claim that this approach may be too complex, and difficult to maintain;
- Bottom-up: by employing techniques used in social networking websites, which provide user-based lightweight semantic annotations (by using tags, folksonomies and simple taxonomies). They claim that this approach will allow the creation of "service communities", which promote self-organisation and possibly automatic semantic annotation of services.

The authors specially focus on the process of "just-in-time", or runtime, service composition, where users create a service composition for a specific set of requirements. However, we did not find enough information about the actual execution of the generated service composition, namely if the user creating the service

composition is the same that will execute it. From the discussions we assume that the objective of the authors is to assist users with limited technical skills in creating their own service as composition of existing services. However, and even considering the abstractions proposed by the authors, it may still be overwhelming for some types of users to be presented with such interfaces to drive the composition process. Table 2-10 presents our classification for this approach.

Table 2-10 Classification of Liu et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	●●	●●	●●	●/×	C-E/×

(●) Manual | (●●) Guided | (● ● ●) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Han et al. [60] propose the notion of "Business Service" to shield end-users from implementation and technical details of services, allowing users to grasp what the services do, not having to know how. Services are described at the semantic level, which enables automatic reasoning. The proposed approach is created based on the VINCA approach [61], which defines a visual approach for business-level composition of services. Furthermore, their approach defines rules to control and optimise the service composition process. They propose two types of rules:

- Service dependency rules: define what and how a service depends on other services. For example: Service A *precedes* Service 2; Service A *leads to* Service B, etc. These dependency rules allow to guide users in the composition process by suggesting a given action to be taken in case a service is selected. Furthermore, such rules allow to check and enforce the correctness of the composed service. These rules are defined by the domain experts.
- Personalisation rules: define "roles" a user can assume in the composition process. Based on these roles different services will be available for a user, with a given role, to use in the composition process. For example: role= {"traveller | services = {Book flight", "Reserve taxi", "Rent car"}}. This means that whenever a user assumes the role "traveller" he will be able to "book a flight", "reserve a taxi" and/or "rent a car". These roles are also defined by domain experts.

Based on these two types of roles the authors propose the notion of "Active Service Spaces" and "Personalised Active Service Spaces". The first is obtained by adding service dependency rules into service spaces (registry of services) and the second by additionally also take into account the personalisation rules.

The proposed approach seems very applicable in closed domains, where experts can define the different rules, however it may be difficult to apply and maintain in any domain. We can observe that the authors aim at defining an approach for user-centric service composition that aims at supporting a specific user, with a specific role, on creating and executing the service composition. We could not find much information about the actual execution of the created services, this is not the focus on the presented approach, although it is clear that the approach is aimed at supporting runtime service composition, whenever a user is in a given "active space". Table 2-11 presents our classification of this approach.

Table 2-11 Classification of Han et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	●●	●●	●●	●/×	C-E

(●) Manual | (●●) Guided | (●●●) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

The European project OPUCE (Open Platform for User-centric service Creation and Execution) [62] defines an approach for user service composition. The idea is to deliver easy to use tools that allow non-technical users to create services (compositions). They define a supporting platform that has "base services", which can be used by users to create more complex and personalised services, according to a set of requirements. The user can play two roles in the process: 1) *service creator* (active user), in which the user is the end-user of the new service that is being composed. Furthermore, the service can also later be made available for other end-users; 2) *service consumer* (passive user), in which the user only makes use of a service (composition), created by another OPUCE user, or maybe by himself in a previous moment. *User-centricity* is achieved by using the user context, preferences, community evaluations/opinions. Services take the user specific properties into account, e.g., user telephone number, user preferred communication mechanisms, etc., which are stored in the "user sphere". Furthermore, the service compositions take the user context into consideration, specifying in which context conditions the services of the composition are used, and using it to decide the activities flow in the service execution process. Two service composition environments were proposed:

– Full featured editor: supports a rich graphical interface (web based) that has three different areas: services pallet, design pallet and control pallet. The user specifies order, conditional execution, and has intuitive graphics representing the existing

services, which allows to better understand the service functionality. Furthermore it allows define event-action control mechanisms over services. After the creation the environment can also be used to publish the resulting composition;

– Simplified editor: simple interface that allows the users to define an ordered list of services as a service composition. This list can be automatically resolved, by automatically composing services O→I. If problems arise this graphical representation guides the user in order to solve the encountered problems.

Service representation is the same for both types of editors (full or simplified). This is essential since the services are the same, only the way of creating compositions vary according to the user device, and user expertise. Service discovery is implemented in a "push" and "pull" versions. "Push" is more advertising/sharing oriented, i.e., users can share service compositions with colleagues/friends. "Pull" is more traditional way of service discovery, in this case based on the UDDI standard. The execution of the created service compositions is performed in a service execution environment, which consists of a BPEL engine that orchestrates the created composite service.

Although the proposed service creation environment abstracts the user from many details of the composition process, it may be still complex for some types of users, mainly the "full featured editor". Nevertheless, it may be an option for some application domains, for example enterprise domain, where users can be instructed on how to use the service creation environment. The execution phase is not explored in depth in the project, although an execution environment is proposed. We could observe that OPUCE distinguishes between "service creator" (or as we defined *composer*) and the "service consumer", which in our opinion is very important to design appropriate user-centric service composition approaches. They assume that there are users that can play both types of user roles. Table 2-12 presents our classification for the OPUCE approach.

Table 2-12 Classification of OPUCE approach

	Req.	Disc.	Comp.	Exec.	User
Classification	•	••	••	•/×	C-E

(•) Manual | (••) Guided | (• • •) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Volker et al. [63] [64] discuss the problem and complexity of enterprise's IT departments, and the advantages of using mashups on enterprises, which they describe as *enterprise mashups*. The

idea is that IT departments need to shift their roles, mainly to maintain the IT ecosystem, while *non-IT* users, or business experts, are presented with mechanisms for service selection, discovery and composition in order to define/specify enterprise processes. So, the IT department starts to work as an intermediary and control check entity. They propose that the enterprise non-IT users can use mashup-like environments to create services on demand for a concrete need, being able to use the result immediately or make it available to a group of other users. The service composition environment is based on a high-level representation of resources, where the user can combine different resources and create a service composition. They define the service composition environment as an *enterprise mashup* environment. There are three basic components in *enterprise mashups*:

- Resources: which are the contents, services, applications, etc.;
- Widgets: which are graphical representations that represent existing resources;
- Mashup: which is the final combination of resources that are represented as widgets.

They follow the St. Gallen Media Reference Model (MRM) [65] for defining Enterprise Mashup environments. This model specifies that the environment needs to encompass the following components:

- Layers:
  - Community viewpoint: describes the participating agents and organisational structure;
  - Interaction viewpoint: procedural description of the interaction events;
  - Service viewpoint: provides the necessary services for carrying out the described process steps in the interaction viewpoint;
  - Infrastructure viewpoint: communication protocols and standards that comprise the groundwork for the implementation of services;
- Phases: knowledge; intention; contract (design); settlement.

In the interaction viewpoint, the IT department plays its role as the inter-mediator between producers and consumers of services (mashups). They also monitor the service compositions. The consumers are the ones consuming the basic services provided by the producers. Producers compose these resources as mashup. Semantic information is used to describe services (formally, in a top-down way), and to gather the user evaluations and opinions, in a bottom-up way.

The authors specify a distinction between mashup creator and consumer. They specially focus on the support to the creator, although they also present some mechanisms for the consumers to evaluate the proposed mashups, the user evaluations that contribute with a bottom-up annotation of the service compositions. No details are provided about the actual execution of the services. Table 2-13 presents our classification of this approach.

Table 2-13 Classification of Volker et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	•	••	••	•/×	C-E/×

(•) Manual | (••) Guided | (•••) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Nestler et al. [66] [67] propose an approach that extends the ideas of *mashups* to support service composition by non-technical users. Many projects assume that users have some IT knowledge, namely that they understand the graphical interface to support the process of composition of services. However, users without technical skills require abstraction from the technical details associated with the process of composing different services to create a new valued added service. The authors propose different mechanisms to provide such an abstraction. They use service annotations to represent service information, so that the users can define compositions at the "presentation layer", without having to understand completely all the technicalities of the service. The approach defines a mashup-like graphical interface to support the service composition process. The UI is dynamically derived from the services' annotations. The following phases are considered in the composition process:

1. Service developers, IT experts, define services and annotate them;
2. Services are imported in the system, and its operations are made available to the users so they can make use of them in a high-level manner to define their service compositions;
3. Users define the service composition in an intuitive environment.

The authors specially focus on the definition of a simplified visual approach for service composition, to support users without advanced technical skills. Their focus is on the support of the composition phase. Although they refer that the support is given to end-users of the service, they do not discuss the execution support. Table 2-14 presents our classification of this approach.

Table 2-14 Classification of Nestler et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	•	••	•	×	C

(•) Manual | (••) Guided | (• • •) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Ro et al. [68] present an approach that aim at ease the way users participate in the creation of new services, as composition of existing services. They use "visual metaphors" to address the composition process, namely by using the idea of a "story board" to define the flow and activities the user wants to be delivered by the service composition. This approach is implemented as a visual mashup. They have three different constructs on their approach:

- Stones: represent services, or service operations, i.e., the executable pieces of services;
- Story board: playground where the user can place stones to define a story, i.e., a service composition;
- Story: represents the service composition, i.e., how the composition of services, as combinations of "stones" in the "story board" can be specified.

The composition process is intuitive, and performed in a web-based interface. All possible services are exposed to the users, as stones. Users place the stones iteratively in the story board. Each time the user puts a stone in the story board, the next stones that can be selected are highlighted. These are limited by the outputs of the previous stone, i.e., only stones that have inputs that can be composed with the previous stone (service) outputs can be used. Once a story is created, the user can store it for future use. The created stories can also be made available to other users. Usability tests, and surveys showed that the end-users and developers find this type of support useful.

The proposed approach shows a very interesting visual metaphor that facilitates the involvement of users with limited technical skills in the service composition process. The approach assumes that the user creating a "story", is also aiming at executing it, i.e., it provides the necessary inputs of the services. They assume a pure "forward" composition, i.e., the next services to be added are composed with the current service outputs, however if the user is not able to provide a given input the approach is not able to support the user on finding services that may deliver an output that can be composed with the missing input. Table 2-15 presents our classification for this approach.

Table 2-15 Classification of Rao et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	●●	●●	●●	●	C-E

(●) Manual | (●●) Guided | (● ● ●) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

User-driven Approaches

Sirin et al. [69] propose a semi-automatic approach to support the composition of web services. This approach supports user in the discovery, selection and composition of web services during each step in the composition process. Services are described using semantic descriptions in DAML-S [38] (ServiceProfile, PocessModel and Grounding). DAML-S semantic service descriptions refer to ontologies, which in this case were OWL [70] ontologies. The discovery process consists of finding matching services, which consist of web services that provide outputs that semantically match inputs of services that the user has previously selected for the composition. This is a *backwards chaining service composition* process, as the user starts with the service that wants, and then is guided on a process of discovering services to fulfil the preconditions of this service. After the discovery, selection and composition of services that meet the user requirements, the user validates the service composition, and it can be translated into an executable format, namely by generating a DAML-S CompositeProcess model and the WSDL files of the constituent web services. The composition is executed by calling each service separately, and passing the results between services according to the flow specifications. This approach defines a guided service composition, i.e., the user is involved in each step of the composition process, which makes the generated service composition more consistent with the user specific requirements, as the user validates the insertion of each of the constituent services on the composition. Furthermore, the role of the user can be simply *composer*, defining a service for other users, but also can be the situation where the user is the end-user of the service being created. Table 2-16 presents the classification for this approach.

Table 2-16 Classification of Sirin et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	●	●●	●●	●	C-E

(●) Manual | (●●) Guided | (● ● ●) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Mokhtar et al. [71] [72] explore the use of semantic services as basic constructs in service delivery in pervasive computing envi-

ronments. The different devices and functionality in the environment are exposed as services, which can be composed as required by the user, to deliver functionality the user needs at a given moment. This approach differs from the approaches discussed before in the sense that it is limited to service delivery in specific pervasive computing environments, i.e., a very specific and closed domain. This allows the approach to identify the different types of users of the environment and shape the supporting environment accordingly. They define a conversation-based approach to support the users in achieving theirs objectives, as composition of services available in their environment, whenever they need them. This approach also considers the service execution phase. Service compositions are created on demand by a user for him to use. In a pervasive computing environment there are another variables that play an important role, namely the user context and profile, which facilitates not only the composition but also the execution of the resulting composition. Table 2-17 presents our classification of this approach.

Table 2-17 Classification of Mokhtar et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	●●	●●	●●	●●	C-E

(●) Manual | (●●) Guided | (● ● ●) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

Sheng et al. [73] present a multi-agent based architecture to provide distributed, adaptive and context aware personalised services in wireless environments. They use web services, agents and a publish/subscribe system. The end-user does not create a service composition. Service compositions are created beforehand and made available as template compositions that at runtime can be grounded/binded to concrete services as the end-user selects them for execution. This binding is done based on the user preferences and context. The system is divided in four layers:

– User layer: where the user requests a service, which already exists, he simply selects it, its template, which then is shaped according to the user preferences and context;
– Context layer: manages user context, device context and service context;
– Orchestration layer: manages and executes a service composition to deliver a service to the user;
– Service layer: where services are deployed, and offered from possible third party providers.

User agent acts on behalf of the user, by using the user personal preferences, mobile device characteristics, and user context. Or-

chestration tuples: define ECA (Event-Control-Action) rules for service compositions. Service compositions are represented as process templates, which have the following properties:

– They are pieces of functionality the user can select from his mobile device;
– They are defined by "template providers";
– They can also be edited by the end-user;

This approach aims at being applied in a specific application domain. The assumptions made are reasonable because the domain is closed, i.e., the type of services and activities are known, as the possible types of users. Nevertheless, some assumptions, such as the possibility of users changing/editing process templates, may be difficult to realise. In this case, the user has only the role of end-user executing a service, although it requests and defines the instances to be used in the process template. Table 2-18 presents our classification of this approach.

Table 2-18 Classification of Shenq et al.

	Req.	Disc.	Comp.	Exec.	User
Classification	•	••	×	••	E

(•) Manual | (••) Guided | (• • •) Automated
(C) Composer | (E) End-user | (×) not addressed/unknown

2.4 Discussion

Table 2-19 presents an overview on the four groups of service composition approaches we have presented in our state-of-the-art study. Two groups concern with the *dynamic service composition* approaches; while the two other groups concern *user-centric service composition* approaches. Figure 2-9 presents a graphical representation of the classification of all the studied approaches. In our classification we also considered the user roles in each approach, which could be *composer* and/or *executor*. Table 2-19 also presents the user roles in each of the studied approaches.

Automatic service composition algorithms focus on the automation of the discovery and composition phases of the service composition life-cycle. The requests are specified by the users, assuming that they know what services they want. Furthermore, they do not focus on the support of service execution. This means that the users in this type of approaches are mainly playing the role of *composers*. Although they can also be the end-users of the service compositions that are being created. This is not emphasised in the presented approaches, as they mainly concentrate on

Table 2-19 State-of-the-art approaches

Group	Sub-Group	Approach	C	E
Dynamic	Automatic Service Composition Algorithm	1: Zhan et. al. [35]	×	
		2: Rao et al. [37]	×	
		3: Lécué et al. [39]	×	
	Dynamic Service Composition Frameworks	4: METEOR-S [41]	×	
		5: SODIUM [46]	×	
		6: MOSCOE [49]	×	
		7: Fujii and Suda [52]	×	×
		8: Kona et al. [54]	×	
User-centric	Mashup-based	9: Liu et al. [59]	×	
		10: Han et al. [60]	×	×
		11: OPUCE [62]	×	×
		12: Volker et al. [64]	×	
		13: Nestler et al. [67]	×	
		14: Ro et al. [68]	×	×
	User-driven	15: Sirin et al. [69]	×	×
		16: Mokhtar et al. [72]	×	×
		17: Sheng et al. [73]		×

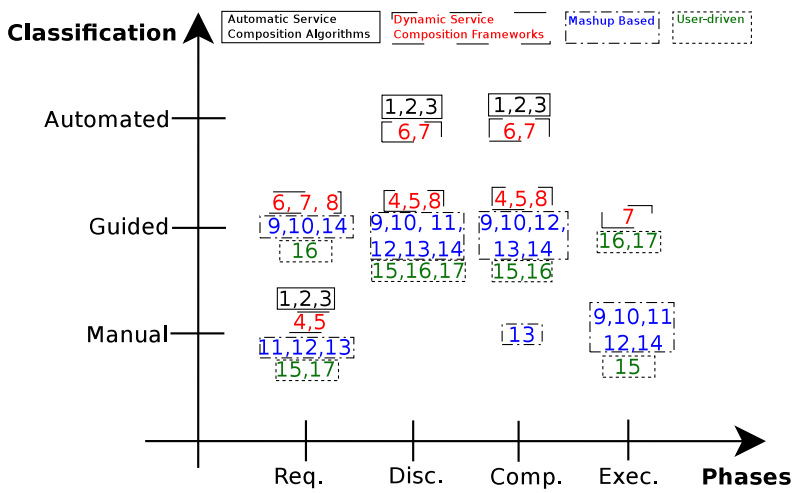


Figure 2-9 State-of-the-art approaches classification

the definition of mechanisms to automate the service composition problem.

Dynamic service composition frameworks mainly target guided, or semi-automatic, support to discovery and composition of services, although some approaches (6 and 7 in Table 2-19) claim to perform these activities in an automated way. Approach 7 also focuses on the execution phase, claiming the use user context to optimise the service execution. In our classification we mainly focused on analysing the service composition and execution supporting phases, but these approaches also tackle other activities related with the service composition process, namely the descrip-

tion and the publication of the services to be used in the composition process. We could observe that most of these approaches do not cover the execution phase, or do not aim at supporting users on the *executor* role. The main focus of these approaches is to support all the activities that allow the *dynamic composition of services*, i.e., facilitate a on demand creation of a service composition.

From the study of *automatic service composition algorithms* and *dynamic service composition frameworks*, we could observe that several approaches have been developed to automate the different activities required to support the service composition process. However, we could also observe that most of these approaches do not cover extensively the actual execution of the generated services, and most importantly they do not emphasise the user participation in the definition of the requirements for the service composition. These approaches assume that the user specifies the requirements without support from the system, i.e., manual specification. Only approach 6 assumes that there is further interaction with the user for the refinement of the service requirements, in case no service composition is found that delivers the specified requirements.

The *Mashup-based* approaches considered in our literature study focus on involving non-technical users in the service composition process. As it can be observed in Figure 2-9, these approaches address service discovery and composition mainly as a guided support, which allows users to interact and define the service composition as required to fulfil their specific requirements. The service request phase is address by defining manually the service request (approaches 11,12,13), although in some approaches (9, 10 and 14) also provide a guided support for service request definition, where the user is assisted regarding the definition of the service request. The mashup-based approaches presented in our study mainly aim at the definition of simplified graphical interfaces to support the composition of services. Nevertheless, most of these approaches require still a lot of technical skills from the user to understand the different parts of the user interface, e.g.: services, service connections/compositions, etc. In these approaches users mainly play the role of composers of the service composition. In some cases (approaches 9, 10, 11, 12 and 14) users also address the service execution phase. Some approaches refer the fact that mashup-based service composition approaches may be the key to support "end-user service composition", enabling the so called "just in time" service composition, i.e., the service (composition) that

a user requires at a given moment is defined by the end-user himself, as a composition of existing services. Nevertheless, in our study we observed that the connection between composition and actual execution is somehow neglected, which may bring many limitations to the practical application of these approaches.

In the second group of user-centric service composition approaches, the *User-driven* approaches, we have presented approaches that focus mainly on providing guided support to the users in the service composition process. Two of the three approaches presented in this group are focused on closed, or dedicated, application domains, i.e., they are defined to support a given set of users with specific characteristics in a specific environment. Provided with such properties, the composition approaches can deliver a more suitable support, where the user is normally queried for a given decision driving the service composition process in a conversational manner, without explicitly having to know that services are being composed on the background. In some situations, e.g.: 16 and 17, the approaches use the user context and preferences to decrease the number of interactions required with the user in the composition process. The user context and preferences are also used in the service execution phase to simplify the execution of services in the service composition.

Chapter 3

User-centric Service Composition

Users have to play a central role in the user-centric service composition process. However, users are heterogeneous, i.e., they have different requirements and characteristics, which implies that they also require different types of support in the service composition process. Given this, user requirements and characteristics must be taken into account when designing systems to support user-centric service composition. This chapter discusses the user-centric service composition process, providing a context and a discussion on the design issues to be taken into account when designing systems to support this process.

This chapter is organised as follows: Section 3.1 discusses the central role users have in the service composition process; Section 3.2 presents some motivating example scenarios for user-centric service composition; Section 3.3 discusses user heterogeneity, which needs to be taken into account when designing supporting systems for user-centric service composition; Section 3.4 details the influence that application domains have on the user-centric service composition process. Finally Section 3.5 identifies further issues to be considered when designing user-centric service composition supporting systems.

3.1 Users Driving Composition Process

The creation of a service as a composition of existing services can take place at design-time or runtime. The set of requirements considered when creating a service composition is identified for a specific user or for a target user population. In design-time service composition approaches, these requirements are normally collected by a professional service developer, who writes a require-

ments document that can be used in the process of developing a service composition that delivers the requirements previously identified.

When we consider user-centric service composition, we assume that a specific end-user provides the requirements for the service to be composed. The requirements of a given end-user may be different from the requirements of another end-user for the same usage scenario. Given this, each (end-)user needs to be provided with a different service composition, according to his needs. To develop a service composition that is capable of fulfilling the specific requirements of a (end-)user, when the user requires the service, the service composition development process has to take place on demand, probably at runtime. Classical service composition techniques, for example workflow-based techniques, may not be suitable to support such on demand composition. Such approaches are too technical to be handled by the service end-users to create their own service compositions, given that they may have limited technical knowledge to handle the tool support for such approaches. Professional service developers are required to handle such approaches. Given this, these approaches do not scale, given that professional service developers cannot develop, on demand, a service composition personalised for each possible end-user.

In order to deliver personalised service compositions on demand for each end-user, i.e., provide user-centric service delivery, the development process needs to be automated. To accomplish this, the (end-)user must be provided with automated intermediation and support in the service composition process, which will perform the task of the human professional service developer of interpreting the user requirements and developing a service composition that fulfils such requirements. Figure 3-1 presents the players of the user-centric service composition process, namely: *user*, *composition environment* and the *services*.

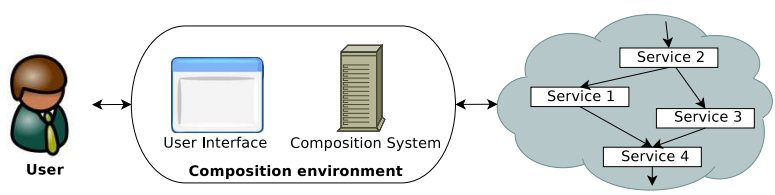


Figure 3-1 User-centric service composition

User-centric and user-driven service composition [74] is identified as a major topic of work in order to achieve personalised service delivery. The user-centric service composition process allows to take specific requirements of a user into consideration to find

a combination of services to fulfil his needs. This process brings social and economic benefits to users and service providers. Users can access more personalised content, instead of making use of generic services (compositions) that do not completely fit their requirements. Service providers can focus on creating basic services and then delegate the personalisation of their combination to the users, assisted by an appropriate supporting system for user-centric service composition.

3.2 Example Scenarios

In this section we discuss two different scenarios, in two different application domains: *e-government* and *entertainment*, where user-centric service composition can be of advantage. Many other domains are particularly suitable for user-centric service composition, such as assisted living, telecom services, tourism, travelling and e-commerce [75]. The scenarios described in this chapter are used in Chapter 8, when we present the validation of the solution proposed in this thesis to support user-centric service composition processes.

3.2.1 E-Govenment Scenario

Governments world-wide are regarded as bureaucratic institutions that are responsible for the public order and for attending the basic needs of their citizens. Governments define public services to support and interact with their citizens. Electronic service are of extreme importance to improve the way governmental organisations support their citizens [76]. For instance, web-based services have been adopted by many governments as a means to facilitate the provision of public services to citizens. Based on this, governments have realized that these technologies might potentially increase the effectiveness, efficiency and transparency of the public sector. This marks the beginning of the electronic government, also called *e-government*. We refer to [77] for the definition of e-government: *e-government refers to the use of electronic information and communications technology to integrate the citizen into the activities of government and the public service.*

In practice, there are still several gaps between the promises of e-government and the actual provided services. Among all e-government initiatives, the creation of web portals is an important step to reduce the distance between the government and the citizens. Governments have been developing portals to assist citizens

since the second half of the 90's. However, many of the governmental portals still reflect the complex organisational structure of government organisations, causing two important drawbacks [78]:

1. Information and services are available in a fragmented way, obliging the users (citizens) to figure out which services satisfy their needs and where to find them. This happens because services websites simply mimic in an electronic way the structure and logic of regular non-electronic public services;
2. E-government websites offer public services without taking the perspective of the citizen into account. Although some websites layouts are well organised, usually they are more attached to the governmental logic than events that take place in the society and are relevant to citizens.

An approach to tackle both issues is called "Life events", which suggests a more citizen-centric organisation and presentation of electronic public services [79]. According to this approach, a given web portal can better attend its users if it assumes the perspective of the users, in this case citizens, while shielding them from the complexity of the public service structure. This organisation model is based on the main events that take place in a citizen's life. Examples of life events are birth registration, school registration, change of address and marriage. Multiple services may be related to these events. Another issue is that many public services require interactions amongst different departments, which should be supported at the technological level. This implies that multiple services that are related to a given event must be realised transparently to the user, although possibly being supported by different departments of the government. Citizens should not need to know how the government deals with the information within its different department in order to use these services. This strategy potentially enables interactions with third-party organisations unrelated to the public administration services, but that offer services that may be relevant for a certain event. Life event-oriented portals can be implemented using simple models based on a well defined hierarchy of life-event related topics allowing citizens to browse the content of these portals in an intuitive manner.

Although life-events may provide already directions for the automation and the resolution of the preconditions required to execute each of the services the user needs it imposes new requirements that can only be fulfilled by making use of other government services. For example, if a user wants to renew his driver license, he accesses the public service that provide such a functionality. However, to renew the driver's license, the user needs to have

a birth certificate, city hall registration, car registration, among other preconditions. If the user does not have such documents, he needs to go to other government websites and request such documents. Then, after gathering all the required preconditions, or documents, he may use the service that allows him to renew his drivers license. This situation can be facilitated by allowing the user to compose the different services he requires to fulfil his requirements, instead of having to use the different services independently of each other. In other words, when the user accesses the service, he could provide the documents he already has and simply make use of the services required to retrieve the missing documents, or preconditions. This user-centric service composition process is very interesting to manage the use of public services. It allows to address the fact that different citizens may have different needs of assistance, for example one may have already a birth certificate while another may need to issue it. Given that public services can be centrally managed, this facilitates the definition of inter-operation between the different services of the different public departments, while still keeping the different departments independent of each other.

3.2.2 Entertainment

Increasingly more people use computer systems for entertainment and leisure activities, for example, by using multimedia, gaming, mobile applications and social networks to entertain themselves [80]. Entertainment services can assist users on finding information about leisure activities and places relevant for them. Internet-based entertainment services to find and plan activities is an area receiving a lot of attention in recent years. This is empowered by the increasing usage of mobile communication devices that allow users to access such services, providing them with assistance on finding and planning leisure activities, wherever they are. With such ecosystems of Internet services and computing devices that can access services from wherever the users are, users are becoming more active players on defining, finding and selecting entertainment activities.

Service providers normally focus on delivering their customers with one service, which provides a given functionality. For example, in the domain of leisure activities, one can have services that assist users on finding restaurants, finding hotels, booking cinema tickets, etc. However, most of the times users require multiple services to satisfy their requirements, and plan a given set of activities, for example: find restaurant, then find the route

to go to the restaurant, or book a cinema ticket and then find a restaurant nearby the cinema. Recently we have observed the appearance of services that combine different services, in the so called "mashups"[56]. Mashups allow to combine the content of different services, for example show the hotels on a map, or show reviews, from different sources, of the movies available in a given movie theatre website. However, these service compositions are "hardcoded", and exposed to the users as another service. If the users want to get a slightly different service, the mashup will not be able to deliver it, and the user may need to use other services to get such functionality. Given these limitations, and considering the increasingly larger ecosystem of services in the entertainment domain, we claim that user-centric service composition approaches will benefit users by allowing them to get more personalised service delivery based on the composition of existing services, on demand, according to their specific needs at a given moment. On the other hand, service providers will also benefit from such techniques, as their services are used in more situations, when composed with services from other providers, and furthermore their users get added value because the service composition fits better the user requirements.

3.3 Users Heterogeneity

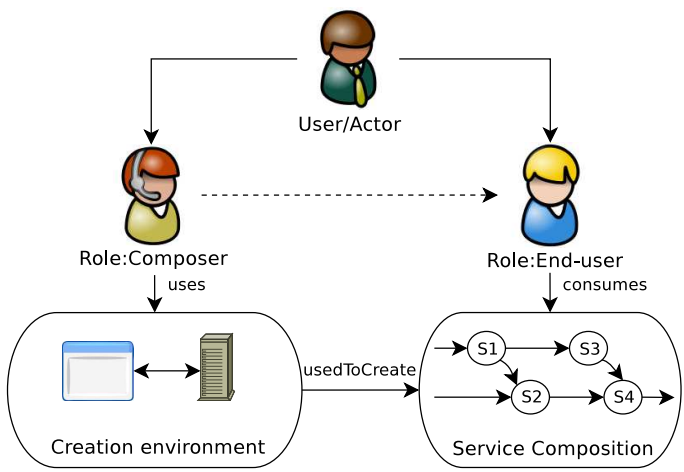
Different users have different characteristics and requirements. Some users have knowledge of the application domain in which they are looking for services, while others may not have this knowledge. Some users have technical skills, and are able to understand and manipulate advanced interfaces for service composition, while others do not have such technical skills. The user's situation and preferences are other examples of properties that are specific of a user, which may be considered in the service composition process. Because of these different characteristics, we conclude that users are heterogeneous. This heterogeneity makes the process of user-centric service composition complex, namely because different users will require different support to accomplish the creation of new services as a composition of existing services. Below we present some characteristics of users, in order to identify requirements and issues to be used in the design of supporting systems for user-centric service composition. We characterise users according to their role in the service composition process, their knowledge on the domain(s) of the services used in the service composition

and their technical skills concerning the service composition supporting system.

3.3.1 User Roles

In the user-centric service composition process, the user is an actor that can play two different roles: 1) *composer*, whenever the user plays an active role in the creation of the service composition; 2) *end-user*, whenever the user provides the requirements for the service composition being created and is the user that will consume the created service (composition). Figure 3-2 depicts the possible roles of a user in an user-centric service composition system.

Figure 3-2
User roles



Although in a user-centric process a service is created to fulfil the requirements of a specific *end-user*, the *end-user* may not be the one creating the service composition, i.e., the user that plays the *composer* role. Given this, the user-centric paradigm is twofold in our definition of user-centric service composition. The service being created is *end-user-centric*, but the supporting composition environment has also to be *composer-centric*, providing the composer with suitable support to successfully create service compositions. *Suitability* of the composition environment can only be achieved if the users are characterised and their needs identified. Such characterisation of the composer users can then be used to design the supporting system accordingly to their needs. However, different people may play the role of composer in a given usage scenario in an application domain. To address such heterogeneity the supporting system needs to provide some degree of adaptability, according to the possible heterogeneity of the users

playing the role of composer in the usage scenario.

3.3.2 User Knowledge

Users normally have different levels of knowledge and understanding of the things that surround them. This knowledge allows users to comprehend things and take decisions on their daily lives. To accomplish user-centric service composition, users require knowledge on different aspects of the service composition process. In our work we consider the following dimensions of user knowledge:

- *Domain knowledge*: knowledge on the characteristics of a given application domain;
- *Services knowledge*: knowledge on the services of a given application domain;
- *Technical knowledge*: knowledge on the environment that supports the service composition process.

Users have to have knowledge in these three dimensions to be able to drive the service composition process in a user-centric fashion.

In this section we characterise in detail these three dimensions of user knowledge. However, to better understand and discuss these dimensions of user knowledge, we first introduce the theory of *knowledge hierarchy*, which aims at describing the possible levels of content organisation in the human mind.

Data-Information-Knowledge-Wisdom

The *Data-Information-Knowledge-Wisdom* (DIKW) hierarchy presents a theory to describe the possible levels of organisation of content in the human mind. This theory has been investigated by several researchers, including Ackhof [81] and Zeleny [82].

Ackhof [81] provides the following description of the DIKW elements:

- *Data* is defined as symbols that represent properties of objects, events and their environment. They are the products of observation, but are of no use until they are in a usable (i.e. relevant) form;
- *Information* is contained in descriptions, answers to questions that begin with such words as who, what, when and how many. Information systems generate, store, retrieve and process data. Information is inferred from data;
- *Knowledge* is know-how, and is what makes possible the transformation of information into instructions. Knowledge can be obtained either by transmission from another who has it, by instruction, or by extracting it from experience;

– *Wisdom* is the ability to increase effectiveness. Wisdom adds value, which requires the mental function that we call judgement. The ethical and aesthetic values that this implies are inherent to the actor and are unique and personal.

These elements are inter-related and built on top of each other, *from data to wisdom*. Figure 3-3 represents the DIKW pyramid.

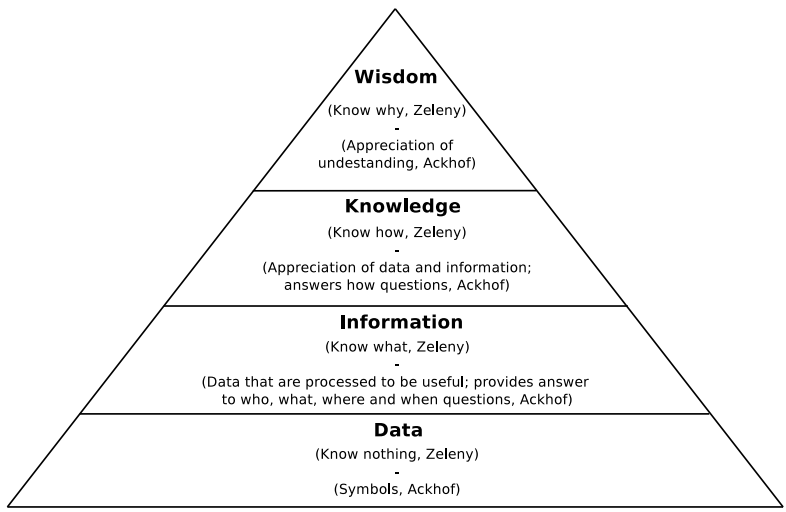


Figure 3-3 DIKW pyramid

The DIKW knowledge hierarchy provides a model to describe how users organise content in their minds. This is of high importance when designing a supporting system for user-centric service composition, given that users have to be equipped with the necessary knowledge to deal with the different activities of the service composition process. To accomplish this, the users may require mechanisms to assist them when they lack knowledge in a given aspect of the composition process, which may be addressed by, for example, providing the user with further information on the issue encountered, to assist the user on deciding what actions shall be taken next.

Domain Knowledge

Users require some familiarity with the domains in which they are seeking services. Users may have different levels of familiarity with these application domains. We refer to this familiarity with the domain as *domain knowledge*. Domain knowledge has to do with the knowledge of a user regarding the concepts and tasks that exist in the domain, Figure 3-4. The user can *gather* this knowledge by experience, learning, consuming advertisement, etc.,

i.e., by collecting and processing data and information about the domain.

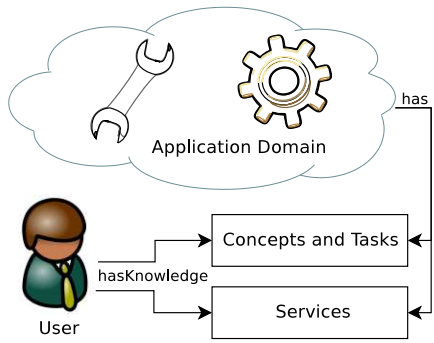


Figure 3-4 User domain knowledge

Domain knowledge is required for users to define the requirements they have at a given moment and to use the services that can fulfil such requirements. For example, if a user wants to buy a new mobile phone, he has to know what is a mobile phone. The user also needs to know where he can buy a new mobile phone, for example in an online store that was advertising mobile phones on sale. To buy the mobile phone, the user accesses the online store website and starts the process of purchasing a new mobile phone. An interface is presented to the user, where he is asked for the desired features of the mobile phone. These features are, for example, WiFi connection, 3G connection, GPS, size of the screen, etc. To be able to proceed in the purchase process the user has to be able to decide which features the mobile phone should have, but he may not know some of them. Given this, the user has to "learn" about the mobile phone features. This knowledge can be acquired by consulting other websites, or services, where data and information about the different mobile phone features exist. The online store web site can provide the user with links to these websites. Furthermore, the online store can also provide means for the user to contact a human assistant who can explain the different features of the mobile phone to the user. Based on the data and information gathered from any combination of sources of information the user can define the characteristics of the mobile phone he desires, which allows the system to present him with the mobile phones that match the specified characteristics.

From this simple example we can observe that in the process of using a service often the user start with limited knowledge of the application domain, and requires some interactions with components of the domain, possibly executing some other "auxiliary" services, in order to gather knowledge, and information, to the

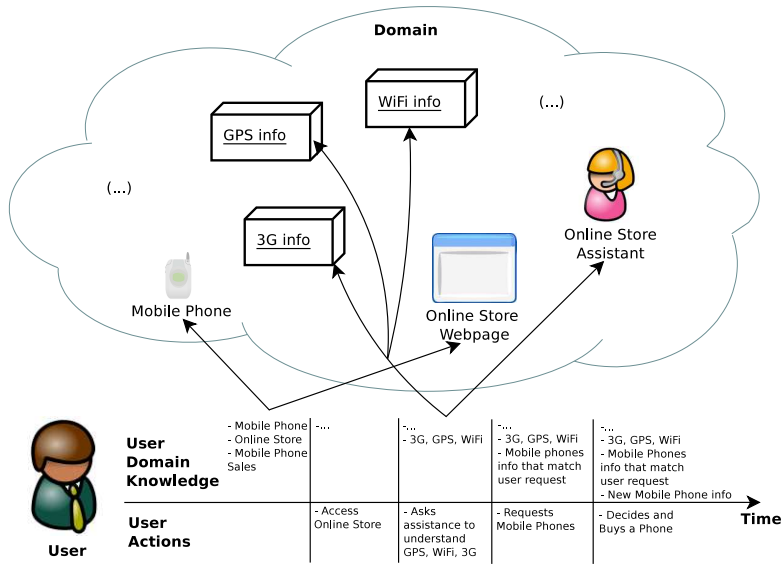


Figure 3-5 User domain knowledge example

point of being able to take decisions. During this process the user gathers to data and information about the domain, which constitute the base for the user knowledge on the domain. Figure 3-5 presents a possible flow of actions the user takes in the process of buying a mobile phone, in the example presented above. Furthermore, the figure also presents how the user knowledge evolves in the process. The user starts with limited knowledge on the application domain. As he advances in the process of purchasing the mobile phone, the user interacts with the domain and acquires data and information about the things he does not know, for example, the mobile phone features (3G, GPS and WiFi). Based on this data and information, the user learns about the application domain, which allow the user to select the mobile phone he wants to buy.

Based on this example we can conclude that a user-centric service composition environment needs to support the user not only on the composition process itself, but also on acquiring further knowledge about the domain, if necessary. This aspect has been ignored in most of the approaches for user-centric service composition found in the literature [14] [31] [33] [34], which mainly focus on the functional aspects of supporting the service composition process.

Service Knowledge

The *domain knowledge* is common to different aspects and activities the user can perform in a domain. On the other hand, the services offered in an application domain require specific knowledge that concerns the services, their functionality and properties, their providers, how they can be used, etc., which is specific to the services and not general to the rest of the domain.

This knowledge is acquired by interacting with service providers, service registries, etc. With service knowledge the user is able to determine which services he wants to use.

In the example presented in the previous section, where the user is buying a mobile phone on an online store, the service knowledge has to do with the online store web service and the user understanding of this service. The user becomes aware of this service through advertisements, i.e., he learns who delivers the service, where he can access it, etc. Once the user accesses the service, through its website, he is exposed to the necessary information to make use of the service, namely the mobile phone features the user wants for his mobile phone. We have assumed that the user does not understand some of the features that the mobile phone can have. Given this, the user needs to learn about them. One possibility of learning about the mobile phone features is to use other websites, i.e., other services, that may deliver such information. To accomplish this, the user can explicitly look for this information in other websites. This can be achieved if the user knows such services, i.e., if he has such *service knowledge* on services that can provide information on the features the user does not know.

Service knowledge and domain knowledge are inter-related and one can lead to the other. For example, every time a user executes a given service he may learn more about the domain. In the service composition process the understanding and knowledge about the services to be used in a service composition is essential to be able to define the composition of services that deliver the functionality the users requires.

Technical Knowledge

Service composition is being used in different application domains nowadays. Service compositions are mainly created by professional users or developers, which can handle complex tooling and understand the composition process. For example, many companies use Web services technology, in which WSDL (Web Ser-

vice Description Language) [10] is used to describe the services available in the company, and BPEL (Business Process Execution Language) [16] is used to compose and coordinate multiple services. However, not all users, specially non-professional users, are expected to know these technical artifacts.

We consider two levels of technical knowledge in the service composition process: *composition environment interface* and *composition process*, Figure 3-6.

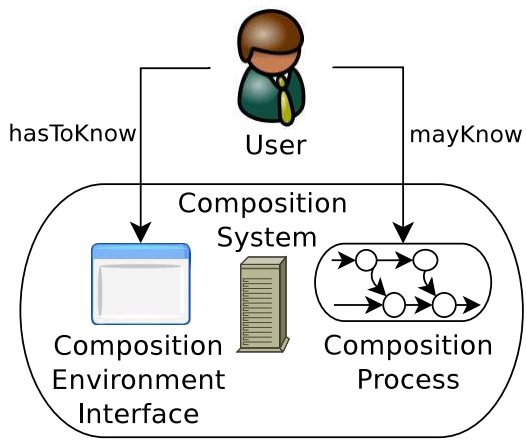


Figure 3-6 User technical knowledge

The user is required to understand the service creation supporting environment *interface* to handle the composition process. This means the user has to understand what are the different parts of the interface, the concepts and elements of the interface, how to interact with them and possibly what are the consequences of using them, i.e., what do they trigger on the back-end supporting system. There should be mechanisms to explain the composition environment interface to the users, possibly "on the fly", given that users do not always have the time to stop and learn about the composition environment.

The other level of technical knowledge, *composition process* knowledge, is related with the technicalities of the service composition process, namely the notion of service, service interface, service composition, orchestration of services, workflow technologies, etc. Furthermore, this level of knowledge also encompasses the details associated with the activities of the service composition life-cycle (service discovery, selection, composition, execution, etc.). The user may not need to know these details, if the supporting composition system is developed to automatically mediate these activities on behalf of the user. However, if the user has such technical knowledge, the composition environment can

offer a more advanced and detailed interface for the user to control all the aspects of the composition process. Such advanced users are typically service developers or software engineers. In this thesis we specially focus on users with limited technical knowledge, since supporting mechanisms for users with advanced technical skills are already becoming mature [14] [15].

3.3.3 Types of Users

Table 3-1 shows a simple classification of user types, based on the user knowledge dimensions identified in the previous section.

Table 3-1 Types of users

	User Knowledge		
Type of User	*Domain*	*Services*	*Technical*
Layman	+−	−−	+−
Domain Expert	++	++	+−
Technical Expert	+−	−−	++
Advanced	++	++	++

(++) has knowledge | (+−) has limited knowledge | (−−) has no knowledge

A *Layman* is a user who has limited knowledge on the application domain and its concepts and tasks, and has no knowledge on the services of the domain. Furthermore, this user has also limited technical knowledge on the tooling offered to support the composition process. A *Domain Expert* is a user who knows the application domain concepts and tasks and its services but has limited technical knowledge on the environment supporting the composition process. A *Technical Expert* is a user who has technical knowledge on the service composition environment, but may not know the application domain in depth. An *Advanced* user is a user who has technical knowledge on the supporting composition environment and on the composition process, and furthermore knows the application domain and the services offered in the domain [83].

We expect other user types to be identified, possibly in between the types we have identified. The purpose of our classification is to emphasise the heterogeneity of users, which motivates the requirement for different types of supporting strategies in a user-centric service composition system.

3.4 Application Domain Characteristics

In a user-centric service composition process the user plays the central role, delivering requirements for the design of suitable supporting systems. Nevertheless, the supporting system is also dependent on the application domain where it is to be deployed to support usage scenarios where users make use of service compositions.

Application domains may have different services and may define different service composition systems with different objectives and possibly aiming at supporting different types of users. For example, in the *assisted living* application domain, there can be usage scenarios where caregivers specify procedures (or sequences of activities) to supervise a particular patient remotely, e.g., 1) measure blood pressure and 2) send a message to the patient's doctor, if the blood pressure is above a threshold. In this situation the caregiver plays the role of *composer*, who composes a service for a given patient. However, the patient is the service composition *end-user*. The characteristics and roles of the users in this process are enforced by the application domain characteristics. This follows from the fact that the patient does not have the domain knowledge required to define such types of service compositions, and such knowledge cannot be easily transmitted to the user, as it requires extensive training and it is mission critical. Nevertheless, if one considers other application domains, for example the situation where the user wants to buy a mobile phone, the user can play both roles, *composer* and *end-user*, as one can assume that the user can learn about the domain on the fly and the composition process is not of high complexity.

Based on this reasoning, we can conclude that the supporting system for user-centric service composition is not only influenced by the users to be supported but also the application domains where it is to be deployed.

3.5 Supporting System Design Issues

In this section we present a set of design issues identified from the discussions performed in the previous sections on the problem of user-centric service composition. These issues have to be taken into account when designing a supporting system for user-centric service composition. In the following we enlist the design issues identified:

- **Identify Target User Population**: several users can make use of services in an application domain. One has to identify the target user population that will make use of the composition environment, characterising their knowledge and technical skills, so that a suitable support can be defined;
- **Adaptation to Composer**: different users may use the supporting system for user-centric service composition. The supporting system needs to be flexible and adaptable to the characteristics of the user playing the role of composer;
- **User Requirements**: the final goal of a user-centric service composition is to support the delivery of specific services to a specific end-user. To accomplish this, one has to collect the end-user requirements to be taken into account in the service composition process;
- **User Abstraction**: the user playing the role of composer may not have enough technical knowledge of the service composition process. To support such user in the process of service composition he must be provided with abstractions that allow him to drive the service composition process;
- **Runtime Composition**: although user-centric service composition can be performed at design-time, its most interesting application is when it can be applied to support runtime service composition, to support users to compose new services whenever they actually need them. To accomplish this the supporting system has to deliver real-time responses to the user;
- **Iterative Composition and Execution**: users may have limited knowledge on the application domain and its services when they start the service composition process. To succeed on creating a service composition that delivers the functionality the user requests, the users may need to interact with the supporting system to acquire data, information and knowledge that allows them to make use of the services they want. This interactive process may need several services to be composed, and executed, in order to be able to deliver the service the user needs. To accomplish this the supporting system needs not only to manage the services being composed but also the services being executed during the composition process;
- **Application Domain Properties**: different application domains have different properties. These properties have to be taken into account to design the most suitable supporting system for user-centric service composition.

3.6 Discussion

In this chapter we discuss in detail the problem of user-centric service composition in order to identify requirements and issues to be addressed when designing a system to support such a user-centric service composition processes.

User-centric service composition aims at supporting users during the creation of service (compositions) personalised to specific end-users requirements. However, users with different characteristics may use the supporting system to create service compositions. To address this heterogeneity and design a supporting system that fits the users' requirements and characteristics we have proposed a characterisation of users according to their knowledge. Three different dimensions of user knowledge were presented, namely application domain knowledge, services knowledge and technical knowledge on the service composition supporting system. Based on these three dimensions of user knowledge, we identify different types of users and derive the required support that allows users to accomplish the process of service composition. We have also discussed the possible roles the user can play in the service composition process, namely, *composer* and *end-user*. Users can play one or both of the roles. This is influenced by the application domain in which the supporting system for user-centric service composition will be deployed.

Chapter 4

Dynamic Service Composition Framework

This chapter presents the first part of our approach to address the problem of supporting user-centric service composition processes. In this part we mainly address the automation of the phases of the service composition life-cycle, by developing a framework to support *dynamic service composition* (*DynamiCoS*). To achieve automation, we use semantic services and ontologies in the DynamiCoS framework. Automation on the service composition process enables runtime service composition, and furthermore it also allows to shield users from some parts of the service composition process. Such characteristics enable users with limited technical skills to participate and benefit from service composition processes.

The chapter is organised as follows: Section 4.1 presents an overview on the requirements for systems that support dynamic service composition; Section 4.2 presents the design process followed to develop the DynamiCoS framework; Section 4.3 presents the architecture and general properties of the *DynamiCoS* framework; Section 4.4 presents the different components of the DynamiCoS framework.

4.1 Dynamic Service Composition

In [52] the authors define *dynamic service composition* as the process of composing a new service, on demand, as a composition of existing services, to fulfil the needs of a particular user in a given situation. Other authors [84] [85] propose a different description of dynamic service composition, describing it as the process of adapting an existing service composition, at runtime, to a specific context situation or user preferences. In our research [23] [75] we

mainly concentrate in the first definition.

We consider that dynamic service composition approaches can be used to address some of the issues identified in Section 3.5 in order to design approaches to support user-centric service composition. Namely, *runtime service composition* and *user abstraction*.

Runtime service composition is a process where services are composed at runtime, i.e., when the services are needed and are to be used. Runtime service composition is very sensitive with respect to the time taken for the creation of a service composition. With design-time service composition the time taken for the creation of service compositions is not so critical. To comply with such time constraints, runtime service composition approaches need to be provided with mechanisms to automate the support to some activities of the service composition life-cycle, namely the discovery of services and their composition, in order to speed up the composition process. If such automation is not provided, the process of service composition may not comply with such time constraints.

Shielding users from the technical details of the service composition process is an important feature of dynamic service composition approaches as they are to be used for the creation of new services, whenever users require them. The user driving the service composition process may actually be the service end-user. We assume that end-users do not have the technical knowledge to handle complex service composition processes, i.e., automation and shielding strategies are required to enrol end-users in the process of service composition.

4.2 Framework Design

To develop a framework to support dynamic service composition, we consider the service composition life-cycle presented in Section 2.1.2. In the following we present the design decisions made to address the requirements of the different phases of the service composition life-cycle.

4.2.1 Service Creation and Publication

Although the service creation phase and the service publication phase are not directly part of the service composition process, we consider that two requirements need to be addressed in these phases in order to enable dynamic service composition:

– Basic services published in the service registry need to be de-

scribed in a formalism that enables automatic service discovery and composition;
- Services can be created in different technologies and described in different formalisms, however they have to describe the same properties and parameters to allow their discovery and composition;

The first requirement has to be addressed in the service creation phase, where the basic services are created and described in some service description formalism. On the other hand, the second requirement is mainly addressed in the service publication phase, which has to allow the publication of services described in multiple service description formalisms, while granting that they comply with the service representation considered in the service registry and in the framework components.

4.2.2 Service Composition

The composition flow of the service composition life-cycle is divided into four phases: *service request*, *service discovery*, *service composition* and *service execution*.

Service Request Phase

Because users need to be shielded from the details of the service composition process, they will only be active during the service request phase. Hence, the user must be provided with an interface where he can specify the different requirements for the service to be composed. Such requirements are then used during the other phases of the service composition life-cycle to actually find and compose services. To address this the following requirement has to be addressed:
- The service request phase has to allow the collection of the necessary information from the (end-)users to automatically drive all the other phases of the service composition process;

Service Discovery Phase

The service discovery phase interprets the service request of the user and finds services that fulfil the requirements specified by the user in the service request phase. The following requirements need to be addressed in the service discovery phase:
- The service request has to be interpreted, collecting the requirements the user specified for the service;
- The set of services that allow to satisfy the different requirements specified by the user have to be discovered automati-

cally, without user intervention;

Service Composition Phase

In the service composition phase, a service composition is created to fulfil the requirements specified by the user in the service request phase. In the service composition phase the following requirement needs to be addressed:

– Automatically compose the discovered services, in order to create a service (composition) that meets all the requirements specified by the user;

Service Execution Phase

Multiple service compositions may be created that meet the user specified requirements, so a selection may have to take place. The user requesting a service can play two roles, namely *service developer* or *end-user*. The process of service selection may be defined according to the role the user is playing:

– *End-user*: a single service (composition) is expected to be returned, i.e., the system has to select one service, possibly based on the user service request, preferences and context;
– *Service developer*: a ranked list of services that match the service request may be returned.

In the case that the user is playing the role of *end-user*, the selected service composition has to be deployed, so that the user can make use of the service. Normally this process is done in two stages:

1. Translate the service composition representation into an executable representation;
2. Deploy the executable representation of the service composition into an engine that can execute the service composition.

4.3 DynamiCoS Framework

In the sequel we present the details of our framework for dynamic service composition, *DynamiCoS*.

4.3.1 Architectural Overview

Figure 4-1 shows the DynamiCoS[1] framework [86] [87] [88] architecture.

The architecture of the DynamiCoS framework follows the ser-

[1] http://dynamicos.sourceforge.net

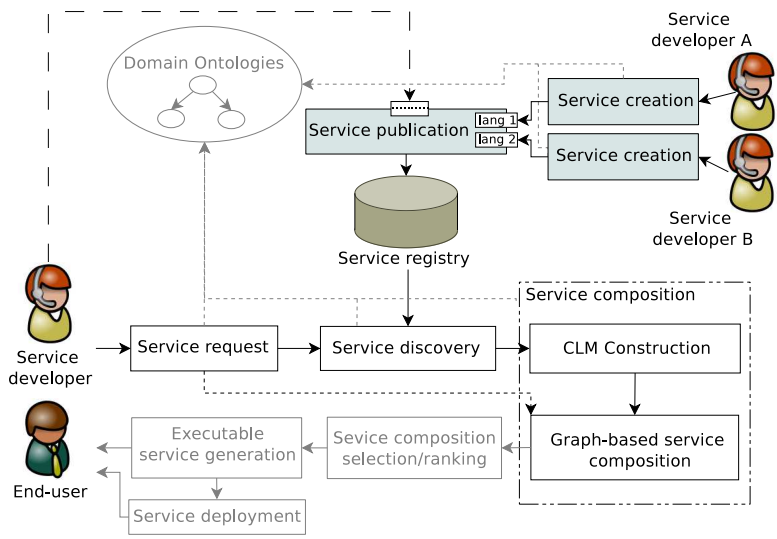


Figure 4-1 Dynami-CoS framework

vice composition life-cycle, as presented in Section 2.1.2. To support the different activities of each of the life-cycle phases, and comply with the design issues identified in the previous section, we have developed components to handle the activities required to support each of the life-cycle phases. Before we discuss the details of the components of the DynamiCoS framework, we will first present some general properties of the framework.

4.3.2 General Properties

To support automation in the service composition process, the information handled in the DynamiCoS framework is represented in a *machine readable* format, which enables *automatic reasoning*. To accomplish this, the service description, publication, discovery and composition are performed using semantic service descriptions [30]. DynamiCoS applies ontologies [28] [27] that define the conceptualisations of the application domains being supported. These ontologies are used by all the framework components. Furthermore, the service developers creating new component services to be published in the service registry, have to comply with these ontologies when describing their services. These ontologies may be defined by different stakeholders, for the different application domains where the DynamiCoS framework is to be applied to support service composition processes.

Given the complexity of the dynamic service composition life-cycle and its stakeholders' heterogeneity, the core components of

the framework had been developed to be technology-independent, namely agnostic with respect to specific service description and service composition languages. To achieve language-neutrality, inside the framework a service and a service composition are represented as a tuple and a graph, respectively. These language-neutral representations allow us to create mappings between the specific languages for the representation of services and the internal representation used by the framework, and also the service composition representation and possible executable languages to represent service compositions. Language-neutrality allows different technologies to be used to describe services, as long as the necessary mappings are defined between the service description languages and the internal representation of services. Interpreters for the service description languages allow to collect the necessary information from the service description document to be published in the framework service registry.

In the framework registry a service is represented as a nine-tuple $s = < ID,\ Desc,\ EndPoint,\ I,\ O,\ P,\ E,\ G,\ NF >$, where ID is the service identifier, $Desc$ is a natural language description of the service, $EndPoint$ is the location of the service endpoint; I is the set of service inputs, O is the set of service outputs, P is the set of service preconditions, E is the set of service effects, G is the set of goals the service realises, NF is the set of service non-functional properties and constraint values.

A service composition is represented as a directed graph $G = (N, E)$. Graph nodes, N, represent services, i.e., each node $n^i \in N$ represents a discovered service s^i. A node can have multiple ingoing and outgoing edges. Each graph edge in E represents the coupling between the i^{th} output/effect of a service and j^{th} input/precondition of another service, i.e., $e_{i \to j} = n^i_{O/E} \to n^j_{I/P}$, where $i \neq j$, since we do not allow a service instance to be coupled with itself. Nevertheless, different instances of the same service may be composed, if they do not introduce deadlock situations. The modelling choice of using a graph to represent service compositions was based on the fact that the topology of service compositions can easily be mapped into a graph structure. Services are represented as nodes, and the composition of services as edges between graph nodes. Graph theory is a mature area of research, which means that there are many techniques to handle and verify properties in graphs, which in our context was extremely relevant. For example, each time we add a new service to a service composition, or node to the graph, we could verify if such an operation does not lead to deadlock situations. Furthermore, there are many

existing programming libraries and tools to handle graphs, which facilitates the development of prototypes to validate our approach for dynamic service composition.

4.4 DynamiCoS Components

Next we present each of the DynamiCoS framework components.

4.4.1 Service Creation

We assume that whenever a service developer creates a new basic service this takes place outside the DynamiCoS framework in some particular service implementation environment. However, to comply with the capabilities of the DynamiCoS framework the services have to be semantically described, in terms of inputs, outputs, preconditions, effects ($IOPE$), goals (G) and non-functional properties (NF), using the framework domain ontologies' semantic concepts.

In the Appendix B.1 we show some examples of service description documents using the SPATEL language [89], which allows to describe services according to the format defined in the DynamiCoS framework. Nevertheless, other semantic services description language could have been used.

4.4.2 Service Publication

To support the publication of services described in different languages or formalisms, the DynamiCoS framework service publication is divided in two phases. There should be an interpreter for each supported service description language, so that first the interpreter reads the service description document and extracts the necessary information for publication (*Desc, EndPoint, I, O, P, E, G, NF*). This makes the service representation in the framework *language-neutral*. Second, the extracted service information is published in the service registry using the DynamiCoS generic service publication mechanism. The service registry allows one to publish, store and discover semantic services.

4.4.3 Service Request

A service request consists of a set of parameters (I, O, P, E, G, NF) that describe *declaratively* the properties of the desired service. These parameters are semantic annotations, like the ones used for representing services within the framework, and refer to

concepts of the ontologies available in the framework. The interface provided to the user for service request specification may vary according to the type of users to be supported, as long as the service request provides the information required by the framework to perform automated discovery and composition. For example, in the IST-SPICE project [90] we have investigated the use of *Natural Language* service requests [91], which gives users with limited technical knowledge a simple and intuitive way to request services. We do not prescribe any specific approach to define the interface for users to specify their service request, as long as the service parameters are gathered, and comply with the framework ontologies.

Running Example. To illustrate the service request process, in the following we present an example of a service request specification, Code 1.

Code 1 Service request

```
<ServiceRequest>
    <input>IOTypes.owl#CellNumber</input>
    <output>Core.owl#MedicalPlaces</output>
    <goal>Goals.owl#FindLocation</goal>
    <goal>Goals.owl#FindMedicalLocation</goal>
</ServiceRequest>
```

This service request specifies requirements for a service to find the medical places nearby the user location. In this running example we consider ontologies and services defined for the *e-health* domain, presented in Appendix A. The service request parameters specified are references to concepts of the ontologies used in the *e-health* domain. This example will be used to demonstrate how DynamiCoS addresses the service discovery and composition phases too.

4.4.4 Service Discovery

DynamiCoS discovers candidate component services before starting the actual service composition process. We argue that most of the services required for the service composition process can be discovered beforehand based on the service request parameters. This happens because in the defined formalism to express the service request the user specifies *declaratively* which activities he wants the service composition to fulfil (goals), which information he is able to provide (inputs/preconditions) and which information he wants to get as result (outputs/effects). The service discovery

process consists of querying the service registry for all the services that *semantically* match the service request parameters. Other approaches for service discovery can also be supported, such as a pure goal-based service discovery, i.e., to discover only the services that realise the goals that the user has defined in the service request. Since DynamiCoS uses semantic-based service discovery and composition, it discovers not only exact semantic matches with the service request $IOPE$ and G semantic concepts, but also partial semantic matches, namely with the concepts that are semantically subsumed by the service request parameter concepts, i.e., $RequestedConcept \sqsupseteq DiscoveredConcept$. This is one of the benefits of using semantic information and ontologies in the process makes it possible to reason on the service descriptions, so that when two concepts are not equal, but are related to each other, this relation can be inferred.

Running Example. Considering the service request from our running example ("find the nearest medical location"), and assuming a service discovery based only on the service request goal parameters, Table 4-1 presents the list of discovered services.

Table 4-1 Discovered services

ID	Input	Output
$S1$	IOTypes.owl#CellNumber	IOTypes.owl#Coordinates
$S2$	IOTypes.owl#Coordinates	Core.owl#Hospital
$S3$	IOTypes.owl#Coordinates	Core.owl#Medical_Center

ID	Service Name	Service Goal
$S1$	locateUser	Goals.owl#FindLocation
$S2$	findHospital	Goals.owl#FindHospital
$S3$	findMedicalCenter	Goals.owl#FindMedicalCenter

4.4.5 Service Composition

In the DynamiCoS framework the service composition process is divided in two steps:

1. Process and organise the discovered services. This process is performed by computing the *Causal Link Matrix* (CLM) [39] [86];
2. Compute service compositions that fulfil the issued service request. This process is performed by means of an automatic service composition algorithm [92];

Causal Link Matrix (CLM)

The *Causal Link Matrix* (CLM) [39] [86] [93] stores all possible semantic connections, or *causal links*, between the discovered services' output and the discovered services' input concepts and re-

quested service output concepts. CLM rows (Equation 4.1) represent the discovered services input concepts (DS_I). CLM columns (Equation 4.2) represent all the possible semantic concepts with which discovered services outputs can be composed with. These concepts correspond to the union of the discovered service inputs (DS_I) and the service request output parameters (SR_o) semantic concepts.

$$CLM_{rows} = DS_I \tag{4.1}$$
$$CLM_{cols} = DS_I \cup SR_O \tag{4.2}$$

We place a service s in the row i and column j position if the service has an input of the semantic type of the row i semantic concept of the CLM and an output that can be semantically composed with the semantic concept on the column j semantic concept. The semantic connections/links/compositions between a service output and a semantic concept of column j is stored in the matrix, represented as the value of *Semantic Similarity*, at the position (i,j). We consider that four types of semantic similarity are possible, as in [36]:

- *Exact* ($\equiv$): if the output parameter Out_s_y of s_y and the semantic concept of column j, $SemCol_j$, are equivalent concepts; formally, $\mathcal{T} \models Out\_s_y \equiv SemCol_j$.
- *PlugIn* ($\sqsubseteq$): if Out_s_y is sub-concept of $SemCol_j$; formally, $\mathcal{T} \models Out\_s_y \sqsubseteq SemCol_j$.
- *Subsume* ($\sqsupseteq$): if Out_s_y is super-concept of $SemCol_j$; formally, $\mathcal{T} \models SemCol_j \sqsubseteq Out\_s_y$.
- *Disjoint* ($\bot$): if Out_s_y and $SemCol_j$ are incompatible; formally, $\mathcal{T} \models Out\_s_y \sqcap SemCol_j \sqsubseteq \bot$.

The use of the CLM allows us to optimise different aspects of the discovery and composition phases. It reduces the number of interactions with the service registry for discovery, since it is not necessary to inquire the registry while performing the composition algorithm. It allows us to verify whether all the parameters of the service request are offered by the discovered services before starting the composition process. In case some of the requested IOPE parameters are not present in the matrix, the user can be requested to refine the service request. The CLM also facilitates the execution of the service composition algorithm, since it provides all the possible semantic connections between the IOPE parameters of the discovered services. This simplifies the semantic reasoning on the service composition algorithm. Furthermore, services can easily be added or removed from the CLM is required.

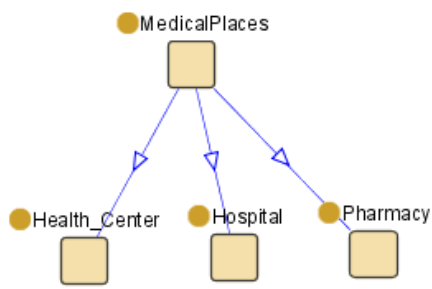
Running Example. Considering the set of discovered services presented in the previous section, the CLM is presented in Table 4-2 is created. As it can be observed the discovered services are part of the matrix. For example the element (2,3), which specifies that the service *S2: findHospital* has an input of type *IOPE 1: IOTypes.owl#CellNumber* and an output that has plugin semantic match ($\sqsubseteq$) with the semantic concept *IOPE3: Core.owl#MedicalPlaces*. This is not an exact semantic match, i.e., it is not the output of the service, however, there is a semantic relation between them. Figure 4-2 presents an excerpt of the *Core.owl* ontology, where the semantic concepts *Core.owl#MedicalPlaces* and *Core.owl#Hospital*, the output of the services *S2: findHospital*, and their relation is shown. We can observe that the concept *Core.owl-#Hospital* is a subclass of the *Core.owl#MedicalPlaces*, which explains the plugin semantic link between the output *Core.owl#Hospital* of service *S2: findHospital* and the concept *IOPE3: Core.owl#MedicalPlaces*, which in this case represents one of the outputs the user specified in the service request.

Table 4-2 Causal Link Matrix (CLM)

	IOPE1	IOPE2	IOPE3
IOPE1	0	$S1,\equiv$	0
IOPE2	0	0	$S2,\sqsubseteq$ $S3,\sqsubseteq$

ID	Service Name
*IOPE*1	IOTypes.owl#CellNumber
*IOPE*2	IOTypes.owl#Coordinates
*IOPE*3	Core.owl#MedicalPlaces

Figure 4-2 Excerpt of Core.owl ontology



Automatic Service Composition Algorithm

The composition algorithm tries to find a composition of services that fulfil the service request by using the CLM. Algorithm

1 shows our graph-based composition algorithm in a simplified pseudo code formalism.

The process starts by analysing the CLM matrix to check if it contains all the specified service request parameters (IOPE/G). The CLM is then inspected for services that provide the service request outputs, or semantically related concepts, which are used to create the initial graph nodes. The algorithm proceeds with a backwards composition strategy towards the service requested inputs. An *open* (or not yet composed) input of the graph is resolved at each algorithm iteration. The algorithm matches the *open* inputs of the services in the graph with the concepts from the CLM columns. If multiple services exist that match a given graph service input, a new composition graph is created, representing an alternative service composition behaviour. During each iteration the aggregated non-functional properties in the composition graph are checked, to verify whether they match the requested non-functional properties. If a composition graph does not match the requested non-functional properties, it is discarded from the set of valid service compositions. The algorithm finishes when all the requested inputs, preconditions and goals in all the alternative service compositions are resolved.

Running Example. Based on the CLM, presented in the previous section, the service composition algorithm performs the following iterations to find service compositions that deliver the requirements specified in the service request:

1. The CLM is queried for services that can deliver the requested outputs, in this case *IOPE3: Core.owl#MedicalPlaces*. This corresponds to check which services are listed in the CLM columns that semantically related with the *IOPE3: Core.owl-#MedicalPlaces* semantic concept column. The *S2: findHospital* and *S3:findMedicalCenter* services are returned. These two services lead to the instantiation of two different graphs, representing two alternative service compositions that deliver *IOPE3: Core.owl#MedicalPlaces*. In each of the graphs, these services fulfil one requested goal, namely *Goals.owl-#FindMedicalLocation*. Given that the graphs do not yet satisfy all the parameters of the service request, they are set as *open* and added to the set of open graphs (*openG*));
2. One of the graphs is set as active to be resolved, for example *S2: findHospital*. This service has one input, *IOTypes.owl#Coordinates*. Given this, the CLM is queried for services that can be composed with the *IOPE2: IOTypes.owl-#Coordinates* semantic concept. The *S1: locateUser* service

Algorithm 1: Graph composition algorithm

Input: CLM, $ServReq$
Result: $ValidComps$

```
   // Variables
1  activeG;  // Graph that is active in the algorithm
      iteration
2  activeN;  // Node that is active in the algorithm iteration
3  openG;  // Set of open graphs
4  validG;  // Set of completed and valid graphs
   // Initialisation
5  if CLM_{rows∪colu} ⊇ ServReq_{I,O} then
       // Create new graph
6      activeG ← createNewGraph();
7      createInitialNodes();
8      openG ← activeG;
9  else
       // Discovered services cannot fulfil the service
          request
10     Stop;
   // Graph construction cycle
11 while |openG| > 0 do
       // Close graph if it matches ServReq_{I,G}
12     if activeG_{I,G} ⊇ ServReq_{I,G} then
13         validG ← activeG;
14         openG ← openG \ activeG;
15         activeG ← openG^0;
16         activeN ← activeG_{openN^0};
17         break;  // Goes to next openG
       // Checks CLM for services that match open inputs
18     foreach semCon ∈ activeN_I do
19         if CLM_{colu} ⊇ semCon then
20             activeN ← CLMmatchingNode;
21         else
22             openG ← openG \ activeG;
23             activeG ← openG^0;
24             activeN ← activeG_{openN^0};
25             break;  // Not possible, goes to next open graph
       // Check if graph NF props comply with requested NF
          props
26     if activeG_{NF} ∩ ServReq_{NF} = ∅ then
27         openG ← openG\activeG;
28         break;  // If Not, composition is not possible
       // prepare next cycle
29     openN ← openN \ activeN;
```

is returned. This service is composed with the *S2: findHospital* service. This service resolves the *Goals.owl#FindLocation* requested goal. Furthermore, it delivers the input the user specified in the service request (*IOTypes.owl#CellNumber*). At this point this service composition fulfils all the requirements specified in the service request, which means that this service composition is set as completed;

3. The set of open graphs is inspected, and still there is one open graph, which has the *S3: findMedicalCenter*. This service has one input of type *IOPE2: IOTypes.owl#Coordinates*. Given this, the CLM is queried for services that can be composed with *IOPE2: IOTypes.owl#Coordinates*. Again, the service *S1: locateUser* is returned, which is composed with the *S3: findMedicalCenter* service. This service composition is also set as completed, as it resolves all the requirements of the service request. This graph is removed from the set of open graphs;
4. The set of open graphs is inspected, and no more graphs are set as open, which means that the composition process is completed, and the resulting graphs can be returned.

4.5 Discussion

In this chapter we present our framework to support dynamic service composition, the DynamiCoS framework. The main objective of this framework is to automate the service composition process, so that runtime service composition can be achieved. Furthermore, such automation allows to shield users from the technical details associated with the service composition process, which enables users with limited technical skills to participate and benefit from service composition processes. This is of special importance if we consider runtime service composition usage scenarios. In runtime service composition usage scenarios, services are composed on demand based on a set of requirements. This enforces the necessity of supporting mechanisms to automate the service composition process and to shield the users from the details of discovering and composing services.

The DynamiCoS framework was designed by defining several components, which support all the activities required to support automatically the different phases of the service composition lifecycle. We have not addressed the service execution phase, it is addressed in later chapters.

Chapter 5

Semantic Service Composition Evaluation

Semantic service composition approaches have been widely proposed as a solution to provide higher quality results and better matchmaking between user requirements and service composition results. Furthermore, semantic service composition approaches also enable the automation of the activities that support the service composition life-cycle phases. Although many approaches are being proposed for semantic service composition, common evaluation methodologies, artefacts and metrics are still lacking. In this chapter we define a framework for the evaluation of semantic service composition approaches. This framework addresses the definition of evaluation scenarios based on a common semantic services collection and the definition of common evaluation metrics. Provided with the same evaluation artefacts, a fair and systematic evaluation and comparison of different semantic service composition approaches can be made.

This chapter is organised as follows: Section 5.1 discusses the problem of defining a framework for the evaluation of semantic service composition approaches; Section 5.2 presents approaches for the definition of semantic service collections; Section 5.3 discusses metrics for the evaluation of semantic service composition approaches; Section 5.4 presents the procedure followed to evaluate semantic service composition approaches; and Section 5.5 gives an example of the application of the proposed evaluation framework.

5.1 Problem Definition

Semantic service composition approaches [15] [31] [94] have been widely researched in recent years. However, evaluation frame-

works to compare such approaches are still lacking. The evaluation of semantic service composition approaches is not trivial and requires the creation of several artefacts and metrics to enable a comprehensive evaluation.

Several works have been proposed to address the problem of evaluating semantic service discovery and matchmaking approaches [95], however these approaches are not directly suitable to support the evaluation of semantic service composition approaches. In order to define a framework to evaluate semantic service composition approaches and to compare them in a fair and repeatable manner we need:

- A semantic services collection which is used as the common services search space for all evaluated service composition approaches;
- A set of evaluation scenarios, created based on the semantic services collection, which are common to all the evaluated service composition approaches;
- A set of evaluation metrics independent of the evaluation environments, namely hardware, operating system, etc.

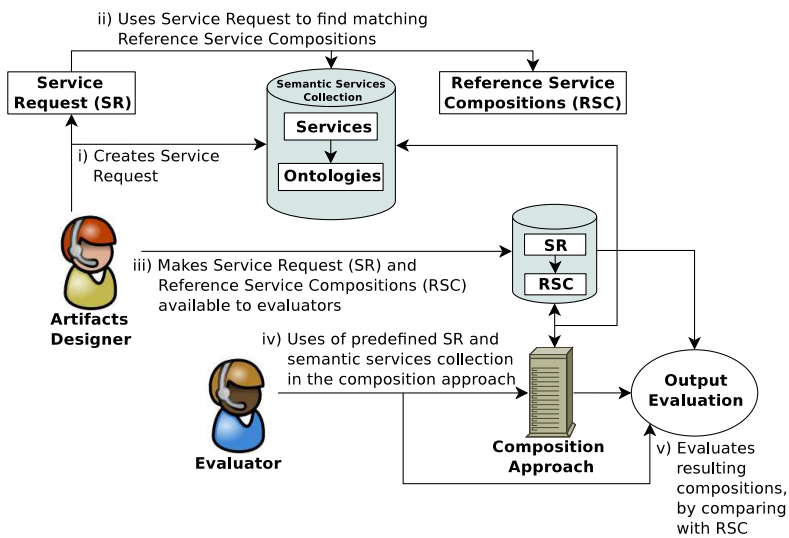


Figure 5-1 General system architecture

Figure 5-1 presents the basic stakeholders and system components that we consider in our evaluation process to address these requirements.

We identify two main stakeholders in the evaluation process: the artefacts *designer* and the service composition approach *eval-*

uator. The *designer* focuses on the development of artefacts to define common evaluation scenarios. The *evaluator* performs the actual evaluation of service composition approaches. The evaluation performed by the *evaluator* is carried out using the artefacts produced by the *designer* and using the evaluation metrics defined in the evaluation framework.

Figure 5-1 also depicts the flow of activities performed to generate the necessary artefacts for the evaluation process (steps i, ii and iii). The evaluation process flow is also depicted in the figure (steps iv and v).

An evaluation scenario consists of a set of service requests, each representing requirements for service composition to be created in the evaluation experiment, based on the semantic services collection. Semantic services collection may be defined by the artefacts *designer* or may be imported from existing semantic services collections. Based on the set of *service requests* (SR) the *designer* defines or finds a set of correct and meaningful *reference service compositions* (RSC) that match and fulfil each of the service requests (SR). These artefacts and the evaluation metrics enable *evaluators* to assess the quality of the approaches being evaluated, and to compare their performance results.

5.2 Service Collections

To enable meaningful and relevant evaluations, the set of services considered in the evaluation process should be large and realistic, or used in practice. However, the creation of a large collection of semantic services with such properties is not a trivial task. Furthermore, services from the collection must yield service compositions. In the sequel we discuss the problems of defining a semantic services collection for the evaluation of semantic service composition approaches, and present our approach to address these problems.

5.2.1 Existing Semantic Services Collections

There are several collections of real world semantic services, which are manually defined by humans. The Semantic Web Services Challenge (SWS-Challenge)[1] [96], Semantic Service Selection (S3)[2] Contest, and the Semantic Web Services Test Collection (SWS-

[1] `http://sws-challenge.org`

[2] `http://www-ags.dfki.uni-sb.de/~klusch/s3/`

TC)[3] [97] are some of the most representative collections of semantic services. The SWS-Challenge offers a set of semantic services that can be used to define a collaboration scenario. The focus of this services collection is on the evaluation of the suitability of approaches to solve mediation problems. These services collection is small, containing around a dozen services. In the SWS-Challenge, services are specified in detailed natural language descriptions, and have to be translated by the participants of the challenge to the formalisms they use. The S3 contest offers a larger collection of services, containing more than one thousand services. This collection was created from existing (web) services' description documents (WSDL [10] descriptions) by extending them with semantic annotations, as references to a set of different ontologies. This collection has services from different domains with very different properties. However, this collection has been criticised [95] because most of the services are not realistic and are poorly specified. The SWS-TC approach provides a set of 241 services described in OWL-S [98], semantically annotated with semantic concepts from a single ontology. This collection is not as large as the S3 collection (OWLS-TC), but services are clearly specified.

An approach similar to the ones discussed above has been reported in [99]. This approach aims at the creation of a large collection of real world services from different application domains with diverse properties. To achieve this a community portal (OPOSSum)[4] has been launched, where people can publish their semantic services. As the collection grows, it can be used as a testing collection in the evaluation process. The SWS-Challenge, S3 contest and SWS-TC semantic services collections have already been imported to this collection.

Suitability for Service Composition Evaluation

To the best of our knowledge, the Semantic Service Selection (S3) Contest provides the largest collection of semantic services, consisting of more than a thousand services. However, after a detailed inspection, we observed that the services in this collection are not suitable for the evaluation of service composition approaches. This collection does not have services that can be composed with each other, i.e., service outputs of one service cannot be composed with service inputs of other services. The absence of this property in the S3-contest services collection may be explained by the fact

[3] `http://projects.semwebcentral.org/projects/sws-tc/`
[4] `http://fusion.cs.uni-jena.de/opossum/`

that the initial purpose of this collection was the evaluation of semantic services discovery and matchmaking.

An alternative is to define evaluation scenarios based on the SWS-TC services collection [96]. This collection's services yields service compositions. Nevertheless, this is a small collection of services, which may not be suitable to measure the scalability of the approaches being evaluated.

Although the ideal situation would be to use real world services, i.e., services that are used in practice (currently defined manually), we observed that the existing collections of such services are not ready for being used on the evaluation of semantic service composition approaches. In the following we discuss some possible directions to overcome the shortcomings of these service collections.

5.2.2 Generation of Evaluation Scenarios

A service composition evaluation scenario consists of a common set of *service requests* and *semantic services collection* containing candidate component services, which yield service compositions. Each *service request* has at least one *reference service composition* that can be created with the services in the semantic services collection. We have identified two distinct approaches to generate evaluation scenarios:

1. *Top-down* approach: introduce a set of services in an existing services collection that yield service compositions capable of fulfilling a service request;
2. *Bottom-up* approach: identify service requests in an existing semantic service collection, which yields to at least one service composition per specified service requests.

The *top-down* approach introduces services that lead to possible service compositions in a service collection. This approach has some practical disadvantages, since it is not trivial to produce large semantic service collections manually. However, additional semantic services can be automatically generated to complement the manually defined semantic services that yield service compositions.

The *bottom-up* approach assumes the existence of a semantic services collection, from which one identifies service requests that yield to service compositions, i.e., reference service compositions for service requests. This approach also has some problems, namely the identification of the service requests and service compositions that can fulfil the requirements of the service request, in case the semantic services collection is large.

Independently of how the semantic services collections are defined, the services, ontologies, service requests (SR) and reference service compositions (RSC) have to remain consistent and preserve their properties, even if they are translated to other description languages or formalisms employed by the different evaluated approaches. This is a precondition to enable a fair comparison of evaluations between different approaches.

5.2.3 Automatic Generation of Semantic Services

A possible direction to overcome the lack of real world semantic services collections is to automatically generate semantic services collections. However, when using automatic generation it is difficult to create collections of services with useful semantics, i.e., generation of services with practical application. Nevertheless, the automatic generation of collections of services may be a suitable direction to overcome the problems and limitations of existing semantic service collections.

Existing Approaches for Automatic Generation

In the literature there are some approaches [100] [101] which focus on the automatic generation of web service descriptions, without semantic annotations. These approaches aim at the generation of large collections of services for the evaluation of web service composition approaches. To generate service collections they allow to specify some characteristics and parameters for the services to be generated, e.g., number of input/output parameters, number of operations, etc., which are then used to drive the generation of the services.

These approaches do not allow the generation of semantic services' descriptions. The only approach we found in the literature that performs automatic generation of semantic service descriptions is presented in the context of the Web Services Challenge[5]. However, the evaluations performed in this contest are mainly focused on performance evaluation, aiming at the creation of a service compositions within the shortest time possible. Services in this collection are not defined with useful semantics.

Automatic Generation

The automatic generation of meaningful and useful semantic services is a difficult undertaking. The main problem is to automati-

[5] `http://ws-challenge.georgetown.edu/`

cally annotate services, and their parameters, in a meaningful way with concepts from the ontologies.

To overcome this problem and generate large collections of semantic services, which allow valid service compositions, using real world semantic services, we propose a *top-down* approach. This approach consists of two parts:

1. The manual creation of a set of real world services, by humans, annotated with the same set of ontologies, which allow several service compositions;
2. The automatic generation of a large set of semantic services, which not necessarily yield to service compositions, annotated with the same ontologies used for the semantic annotation of the manually defined services, which allows these services to be discovered and possibly considered in the service composition process.

5.3 Evaluation Metrics

To evaluate semantic service composition approaches we consider three different groups of metrics:

1. Metrics based on *confusion matrix* values;
2. Time-based metrics;
3. Qualitative metrics.

Some approaches to evaluate semantic services discovery and matchmaking use similar metrics [99]. Service composition can be seen as a special case of service discovery, where the discovered service to satisfy a given set of requirements is a composition of several services.

5.3.1 Confusion Matrix-based Metrics

A *confusion matrix* [102] , or table of confusion, is widely used as a visualisation aid in the field of predictive analytics for the evaluation of classification systems. A confusion matrix contains information about actual values known for a given query and values proposed by a "classification system". A confusion matrix is a two-by-two matrix, as shown in Table 5-1. Confusion matrix columns represent the actual known positive (P) classifications, and negative (N) classifications, for a given query. Confusion matrix rows represent the values classified positively (P') and negatively (N') by a classification system.

The cells of the matrix contain four different classifications:

– *True positives* are values evaluated as positive when they are

Table 5-1 Confusion matrix

		Actual Values	
		P	N
Classified Values	P'	True Positives (TP)	False Positives (FP)
	N'	False Negatives (FN)	True Negatives (TN)

actually positive;
- *False Positives* are values evaluated as positive when they are actually negative;
- *False Negatives* are values evaluated as negative when they are actually positive;
- *True Negatives* are values evaluated as negative when they are actually negative.

Service composition approaches only perform *positive classifications* (P'), which correspond to the compositions proposed, or found, for a given service request. *Negative classifications* (N'), namely *false negatives*, are valid compositions that are not found by the service composition approaches. Figure 5-2 shows a Venn diagram for all the possible classifications, positive and negative, of a given service composition approach. Furthermore, Table 5-2 presents the confusion matrix for the service composition evaluation process.

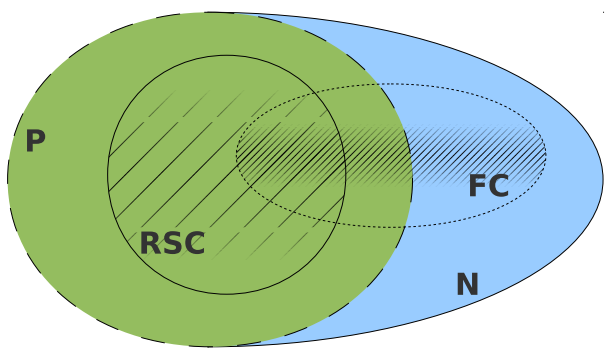


Figure 5-2 Venn diagram for classifications

P represents the whole set of valid compositions for a given service request, while N represents all the other possible compositions, which may be possible with the services of the collection but are not valid for the issued service request. VC (*Valid Compositions*) is the sub-set of FC (*Found Compositions*), which provide valid service compositions for a given service request. The RSC (*reference service compositions*) are the only known valid compositions when some approach is evaluated. However, since this set may not be complete there may be valid compositions returned that are actually correct but are not included in the RSC. This

Table 5-2 Composition approaches confusion matrix

	P	N
P'	$FC^{P}_{Judge} \cup (FC \cap RSC)$	$FC \setminus (FC^{P}_{Judge} \cup (FC \cap RSC))$
N'	$\simeq RSC \setminus (FC \cap RSC)$	$Unknown$

set of valid compositions that are not in the RSC corresponds in Figure 5-2 to $P \cap FC \setminus RSC$. This set is defined by the evaluators when the found compositions are assessed. We call this set the *valid compositions found by judgement* - FC^{P}_{Judge} . Given this, the set of valid service compositions returned by a service composition approach, or *true positives* (TP) corresponds to the set $FC^{P}_{Judge} \cup (FC \cap RSC)$. The remaining valid compositions that were not found by the service composition approach, nor are part of the RSC, are unknown and are not considered in the evaluation process. They correspond to the set $P \setminus (RSC \cup FC^{P}_{Judge})$, however, and since we do not know these service compositions, we consider only the RSC when computing the *false negatives* (FN).

Based on the confusion matrix shown in Table 5-2, we define the following evaluation metrics:

$$Precision\ (PPV) = \frac{|TP|}{|TP| + |FP|} \tag{5.1}$$

$$Recall\ (TPR) = \frac{|TP|}{|TP| + |FN|} \tag{5.2}$$

$$FDR = \frac{|FP|}{|FP| + |TP|} \tag{5.3}$$

$$Acc^{TP} = \frac{|TP|}{|TP| + |FN| + |FP|} \tag{5.4}$$

$$SNR^{TP}_{FP} = \frac{PPV}{FDR} = \frac{|TP|}{|FP|} \tag{5.5}$$

Precision and *recall* metrics are commonly used to evaluate information retrieval systems [103]. *Precision or Positive Classified Values (PPV)* (Equation 5.1) defines the percentage of compositions found by the evaluated approach that are correct. *Recall or True Positives Rate (TPR)* (Equation 5.2) defines the percentage of correct compositions that are found by the evaluated approach. *False Discovery Rate (FDR)* (Equation 5.3) defines the percentage of compositions found by the evaluated approach that are incorrect. *Positive Accuracy* (Equation 5.4) defines the accuracy of the composition approach, i.e., the percentage of the correct compositions found over correct, incorrect and other known (but not found) correct compositions. Finally, *Signal-to-noise-Ratio*

(Equation 5.5) is a metric based on *signal theory*, which represents the ratio between found compositions that are correct and the found compositions that are incorrect. This metric provides an overview on how *robust* the service composition is against retrieving incorrect service compositions.

Some of these metrics can be visualised as shown in Figure 5-3. The graph represents the metrics *precision* and *recall* against the *false discovery rate* metric. If precision and recall values of an approach lie below the dashed line, the compositions found by a the approach being evaluated are predominantly *false positives*. In Figure 5-3 this is the case for Approach 1. In contrast, if the precision and recall values of an approach lie above the dashed line, the compositions found by the composition approach are predominantly *true positives*. In Figure 5-3 this is the case for Approach 2.

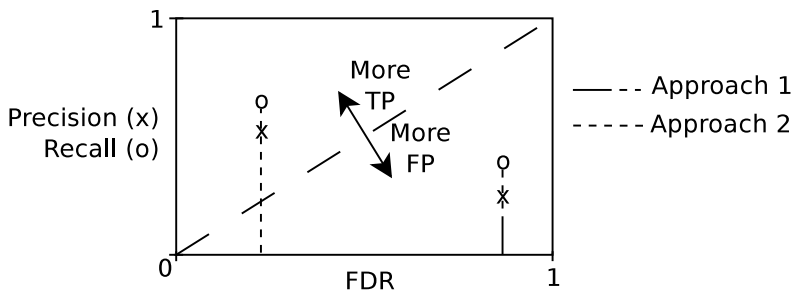


Figure 5-3 Metrics graphical representation

The confusion matrix metrics can also be applied to the service discovery phase of the service composition process. In this case, the confusion matrix is created based on the discovered services and known valid services for each service request, in the evaluation scenarios.

5.3.2 Time-based Metrics

Time-based performance metrics allow one to measure the processing time of service composition approaches. These metrics are especially relevant for verifying the suitability of service composition approaches that target the support of automatic service composition at runtime. Semantic service composition enable automation on the activities that support the service composition process, which make them suitable for the support of runtime service composition. Nevertheless, automation comes with the cost of longer processing times, taken to perform semantic reasoning.

Time-based metrics, such as the processing time of the service

discovery and composition processes depend on the *execution environment* (hardware, operating system, machine load, etc.) in which the experiments are being performed. Given this, these time metrics are not suitable for *absolute value comparison* because different evaluators may perform their evaluations in different *execution environments*. Therefore, either other performance metrics should be defined to factor out the influence of the *execution environment*, or the evaluators have to use these metrics simply as an indication of the performance of their service composition approaches, but not for comparison. The first alternative is difficult to realise; typically it requires to count the number of instructions executed during the composition process, which is similar to measuring the complexity of the composition algorithm.

In order to factor out the influence of the *execution environment* and allow the objective comparison of different approaches, one can also consider the variations in the processing time, resulting from variations in the number of services considered in the service composition process, or services available in the search space or service registry. This metric is often called *scalability* and determines how the system scales when the number of services varies. This allows one to compare the performance of different service composition approaches, even if they are evaluated in different execution environments, given that it quantifies variations and not absolute times performance. Nevertheless, time measures, or indications concerning the time taken to perform the different activities of the service composition process, must also be reported to assess the performance of the approaches, and their suitability to support *real time* service composition processes.

We propose two time-based evaluation metrics: *composition processing time* and *scalability*.

The *composition processing time* ($compPT$) metric captures the time taken between a service request and a response in which one or more service compositions that match the service request is returned. This metric can be characterised as the sum of the following times:

- *service request processing time* ($servReqProcT$): time required to process a user request for a service (composition), i.e., receive the request and extract the different parameters of the request to be used in the service discovery and composition phases;
- *service discovery time* ($servDiscT$): time taken to discover all the services that deliver the requirements specified in the service request;

– *composition time* (*servCompT*): time taken to find compositions from the discovered services that fulfil the requirements specified in the service request.

The *scalability* metric captures the variation of composition processing time (*compPT*) when the number of component services (*#servs*) is varied. Approaches that have a lower variation on the composition processing time, when the services in the registry are varied, have better scalability than the ones that have higher variations.

Scalability (Equation 5.6) is inversely proportional to the variation of composition time as function of the variation of the number of services of the semantic services collection considered in the composition process. We consider that the semantic services collection consists of N groups, which are added incrementally to the service registry.

$$Scalability = \left(\frac{\partial compPT}{\partial \#servs} \right)^{-1} \simeq \left(\frac{1}{N-1} \sum_{i=2}^{N} \frac{compPT(i) - compPT(i-1)}{\#servs(i) - \#servs(i-1)} \right)^{-1} \quad (5.6)$$

5.3.3 Qualitative Metrics

In order to compare approaches in a fair way, only approaches that address the service composition process in the same way can be compared. For example, an approach that supports automatic service composition can only be compared with approaches that support automatic service composition and will not be compared with an approach that is semi-automated.

We propose a classification that distinguishes between levels of automation provided by the semantic service composition approaches. We have distinguish two main classes of semantic service composition approaches, although other more refined classes could be defined:

– *Automatic Service Composition*: these approaches perform discovery and composition of services automatically, based on a set of requirements from the users;
– *Guided Service Composition*: these approaches assist the user in the different activities of the service composition process. Users perform several interactions with the supporting system in the course of the service composition process.

The approaches belonging to each of these classes have to be evaluated differently.

Automatic Service Composition approaches can be evaluated in terms of scalability based on the service composition processing time (*compPT*). Furthermore, evaluation of the quality of the composition results is possible using the confusion-matrix based metrics.

Guided Service Composition approaches can be evaluated with respect to scalability of the service discovery process, based on the service discovery processing time (*servDiscT*). The confusion matrix-based metrics can be applied also to the service discovery process, measuring the quality of the discovered services that are to be used in the service composition process.

5.4 Evaluation Procedure

Figure 5-4 depicts the complete procedure for the evaluation of semantic service composition approaches.

Before the evaluation takes place, the *Evaluator* has to make sure that the framework's semantic services collection, reference service compositions and service requests are compatible with the languages/formalisms used in his composition approach. If not, the *Evaluator* has to translate them to the formalisms of the approach being evaluated, preserving the original services collection, service request and reference compositions properties. An extra effort may be necessary for these translations. However, it would be unrealistic for the framework artefacts designer to cover all the possible service and ontology description languages/formalisms. Similar transformations are used by some of the state-of-the-art works in the area of evaluation of semantic-based mechanisms, for example, in the SWS-Challenge [96].

Once the necessary artefacts are compatible with the approach to be evaluated, the user can make use of the service requests predefined by the artefacts designer, to start the service composition process evaluation. A service request is passed to the service composition approach, which then returns composition results for the given request. The composition time of the different activities of the composition process have to be monitored, so that the time-based performance metrics can be computed. For the scalability metrics, the number of services in the registry has to be varied in a defined number of steps, and the composition time has to be measured. At the end of the service composition process, the resulting service compositions are used in conjunction with the reference service compositions to evaluate the composition approach

according to the proposed evaluation metrics. To compute the quality of the composition approach according to the *confusion matrix*-based metrics, the evaluator has to inspect the resulting compositions, so that *true positives* not defined in the *reference service compositions*, the so called *validate compositions found by judgement*, can also be identified.

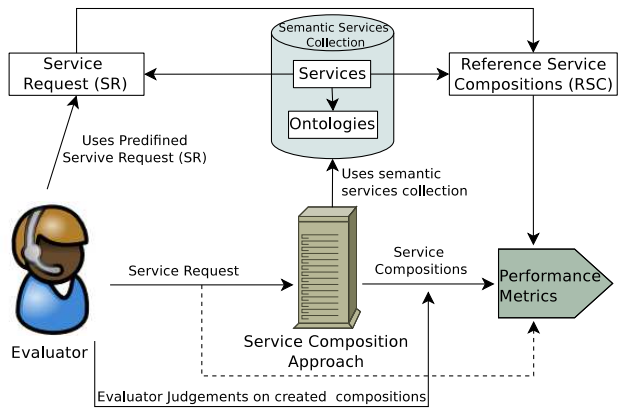


Figure 5-4 Evaluation procedure

5.5 Example

This section presents an example of the application of the proposed framework. We first present the semantic services collection used. Then, we present evaluation scenarios that are created based on the semantic services collection used. Finally, we use one of the defined evaluation scenarios to demonstrate the evaluation procedure of two service composition approaches, evaluated according to the framework metrics.

5.5.1 Semantic Services Collection

We consider services in the domain of *e-health*. These services aim at supporting people with health conditions to carry out different activities in their daily life. The services in this collection are divided in two groups: *manually created* and *automatically generated*, defined in a *top-down* fashion.

The services *manually created* are defined and semantically annotated by humans and have clear semantics, they are presented in Appendix B. This collection consists of 13 services. The services support different activities of the user, such as: find medical loca-

tions (hospitals, pharmacies, etc.), make medical appointments, request medical transportation, find user locations and routes, find doctors in hospitals, etc.

The collection of manually defined services allows to test some aspects of the service composition approaches, namely the quality of the proposed service compositions. However, and due to the collection size, it does not allow to test other aspects, such as the scalability of the approaches. To overcome this, we have complemented the manually generated collection with a large number of automatically generated services. The automatically generated services collection, presented in Appendix B.1.2, consists of 500 semantic services. These services are annotated with the same domain ontologies (Appendix A) that were used to describe the manually defined services.

5.5.2 Evaluation of Composition Approaches

In this section we assume the existence of two semantic service composition approaches: *Approach 1* and *Approach 2*. These approaches are evaluated using *evaluation scenario 2* presented in Appendix B.2. This scenario consists of the following service request (SR): "Find the nearest medical location". The approaches are evaluated according to the evaluation framework metrics presented in Section 5.3.

We assume that *Approach 1* and *Approach 2* belong to the class of *Automatic Service Composition* approaches, according to the *Qualitative* metric, which allows them to be compared with each other.

General Results

We assume that *Approach 1* discovers one of the two *reference service compositions* (RSC) and it does not propose any incorrect composition (*false positives*). Table 5-3 presents the confusion matrix for this composition approach.

Table 5-3 Confusion matrix - Approach 1

		Actual Values	
		P	N
Classified Values	P'	1	0
	N'	1	-

We assume that *Approach 2* discovers two of the two *reference service compositions* (RSC). Additionally, it proposes two incorrect compositions (*false positives*). Table 5-4 presents the confusion matrix for this composition approach.

Table 5-4 Confusion matrix - Approach 2

		Actual Values	
		P	N
Classified Values	P'	2	2
	N'	0	-

Confusion Matrix-based Metrics

Table 5-5 shows how the two composition approaches perform in terms of the confusion matrix-based metrics.

Table 5-5 Metrics comparison

Metrics	*Approach 1*	*Approach 2*
Precision or PPV	$1/1 = 1$	$2/4 = 0.5$
Recall or TPR	$1/2 = 0.5$	$2/2 = 1$
FDR	$0/1 = 0$	$2/4 = 0.5$
Acc^{TP}	$1/2 = 0.5$	$2/4 = 0.5$
SNR^{TP}_{FP}	$1/0 = \infty$	$2/2 = 1$

Comparison

In this evaluation we considered only the confusion-matrix metrics. We did not consider time-based metrics in this evaluation.

Figure 5-5 gives the graphical representation of *Precision* and *Recall* against the *False Discovery Rate* (FDR) of the two approaches evaluated. *Approach 1* has a has a higher precision than *Approach 2*, and a lower FDR. This means that *Approach 1* retrieves proportionally more correct compositions than *Approach 2*, and also proportionally less incorrect compositions. In contrast, *Approach 2* is able to retrieve all the valid compositions (Recall value equal to 1), however it also retrieves a higher percentage of incorrect compositions.

From these results we can draw some conclusions. For example, *Approach 1* may be more suitable to automatically retrieve services for end-users, since it retrieves only correct results. However, if the user driving the service composition process wants to know all the possible service compositions that satisfy his requirements, *Approach 2* would be the best candidate, although it does not have a good precision.

5.6 Discussion

In this chapter we discuss the problem of designing a framework for the evaluation of semantic service composition approaches. The evaluation framework should enable a systematic evaluation of semantic service composition approaches, and allow a fair com-

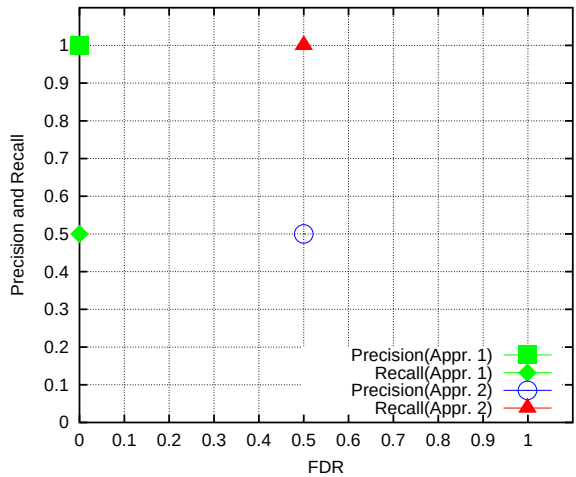


Figure 5-5 Reference service composition set

parison of the performance of different approaches.

We have identified several issues for the design of such an evaluation framework. A major issue we identified is the lack of large and suitable collections of real world semantic services collections. Existing large service collections, e.g., S3-Contest services collection, have not been devised to allow composition of the contained services. To overcome this problem we have proposed two alternatives: insert a set of composable services into an existing collection of services; or create a new collection consisting of a small set of composable services, defined manually and annotated with a set of ontologies, and then add a large set of services generated automatically, which allows to test the scalability of the service composition approaches.

In our work we used the second alternative, i.e., manually defined services complemented with a large set of services automatically generated. We define evaluation scenarios based on the manually defined services. These scenarios are used to evaluate the performance of our approach using the confusion matrix-based metrics. Then, we incrementally add the services automatically generated to the service registry and measure the variation on the processing times. This allows us to evaluate our approach using the scalability metric. In Appendix B we provide a detailed description of the designed evaluation scenarios.

Chapter 6

DynamiCoS Implementation and Evaluation

To evaluate the DynamiCoS framework we have developed a prototype that implements the different components of this framework. We evaluate the DynamiCoS framework by using the evaluation framework proposed in Chapter 5. The evaluation focus on two different points, the quality of the service compositions proposed by the service composition approach and the scalability of the approach with respect to the number of services considered in the composition process.

The chapter is organised as follows: Section 6.1 discuss the implementation of the DynamiCoS framework; Section 6.2 presents the performed evaluation and analyses the results; and Section 6.3 presents some final remarks on the implementation of DynamiCoS and its evaluation.

6.1 Implementation

Figure 6-1 presents the DynamiCoS framework, its different components and the technologies used to implement each component. Furthermore, it discusses the workflow of activities defined to support the service composition process, i.e., the order in which the different components are used to support the dynamic service composition process.

We have defined two main flows of activities for the DynamiCoS framework: 1) creation and publication of basic component services that can be used in the service composition process; and 2) support to the dynamic service composition process, focusing on the automation of the different activities required to support service composition, possibly at runtime. Each of the flows is discussed in the sequel.

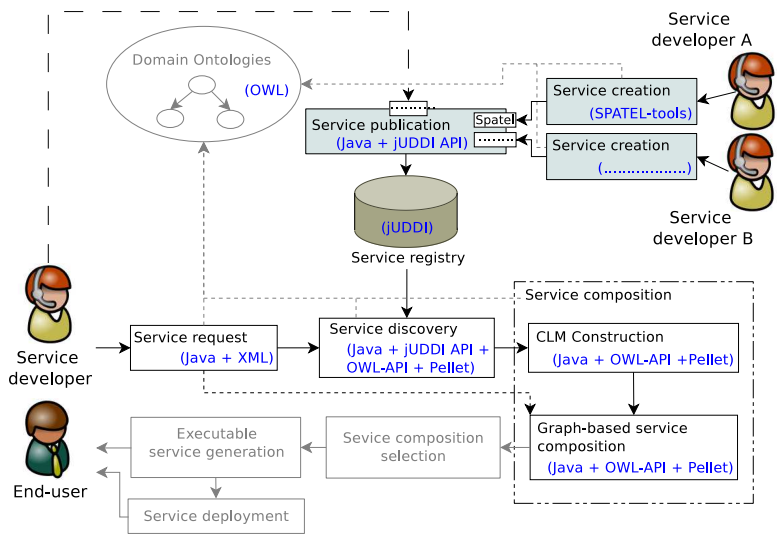


Figure 6-1 DynamiCoS framework implementation

6.1.1 Service Creation Components

The service creation flow has two supporting components: *Service Creation* and *Service Publication*.

Service creation

In our prototype we have used a language called SPATEL [89] to describe semantic services. SPATEL is a language developed in the context of the European IST-SPICE[1] project [90], where the development of the DynamiCoS framework was started. SPATEL allows one to semantically annotate service operation *inputs*, *outputs*, *preconditions* and *effects* parameters, and *service goals* and *non-functional properties*. The ontologies used in the framework are described in OWL [70]. In our experiments we use the following ontologies:

- *Goals.owl*: which contains the descriptions of goals that can be realised in the supported domains. These goals can be used to describe the functionality that can be delivered by the services, and also to collect the goals the user wants to achieve by making use of services;
- *NonFunctional.owl*: which defines non-functional properties to be used in service descriptions and user service requests;
- *Core.owl* and *IOTypes.owl*: which are used to describe services and service request *IOPE* (Inputs, Outputs, Preconditions

[1] `http://www.ist-spice.org/`

and Effects) parameters.

These ontologies have been defined in the context of the IST-SPICE, but the framework is general enough to support the use of other ontologies, according to the application domains to be supported.

DynamiCoS is independent of the techniques and tooling used to build the component services. Different technologies can be used to implement and describe services. The only requirement to comply with the framework is that the services are described according to the parameters defined in DynamiCoS to describe a service ($IOPE$,G,NF), and the service description language is supported by the DynamiCoS publication component, i.e., there is an interpreter for the language in the DynamiCoS publication component.

Service publication

Since in our prototype we have used the SPATEL language for service description, we have developed a SPATEL interpreter in order to import (publish) SPATEL service description documents in our framework's service registry. Other semantic service description languages can be supported as long as the interpreters for those languages are available. The SPATEL interpreter was implemented based on a Java API generated from the SPATEL Ecore model using the Eclipse Modelling Framework (EMF) [104]. Each service is then published in a UDDI-based service registry that we have extended to support the publication of semantic services. We use jUDDI [105] as service registry implementation, which is a Java-based implementation of the UDDI specification [106]. jUDDI offers an API for publication and discovery of services. We have extended the basic jUDDI implementation with a set of UDDI models (*tModels*) to store the set (*categoryBag*) of semantic annotations (I, O, P, E, G, NF) used to describe the different service parameters in our framework.

The published service information is language-neutral, i.e., even if other service description languages are used, the representation of the service in the framework service registry is the same nine tuple $< ID, Desc, EndPoint, I, O, P, E, G, NF >$. This implementation allows the service composition modules to use the same formalism for services representation, independently of the original services' description languages. Furthermore, the developers of the component services do not have to know the internal representation of the services, since the service descriptions' interpreters handle this process on their behalf.

6.1.2 Service Composition Components

The service composition flow of the DynamiCoS framework has three basic supporting elements: *Service Request*, *Service Discovery* and *Service Composition*. Other components may also be presente in the framework, such as, *Service Composition Selection*, *Executable Service Generation* and *Service Deployment*. However, we have not implemented these components in our prototype. In our research we have mainly focused on the automating the service composition process, i.e., discovery and composition activities.

Running Example

In order to explain the dynamics of the different components of the framework we use a running example. This running example is based on the service collections defined in the *e-health* domain, Appendix B. This running example consists of the service request: *Make a medical appointment for a given medical speciality at the nearest hospital*. This running example is also used later to report on the evaluation of the DynamiCoS framework.

Service request

We have implemented two interfaces for the specification of service requests in DynamiCoS: a simple Java-based graphical interface (Figure 6-2) and a web-based interface (Figure 6-3). The aim has been to facilitate the access to the prototype for experimentation and testing. The information entered by the user in either interfaces is transformed to an XML-based document that represents the *desired service*.

Running Example: Considering our running example a service request can be created using the interfaces for service request specification. To specify the service request, the users have to know the requirements for their services (compositions), i.e., they want a service that given a medical speciality, finds a hospital nearby the location of the user, finds a doctor in the hospital, and makes an appointment with the doctor. This service request is shown in Code 2.

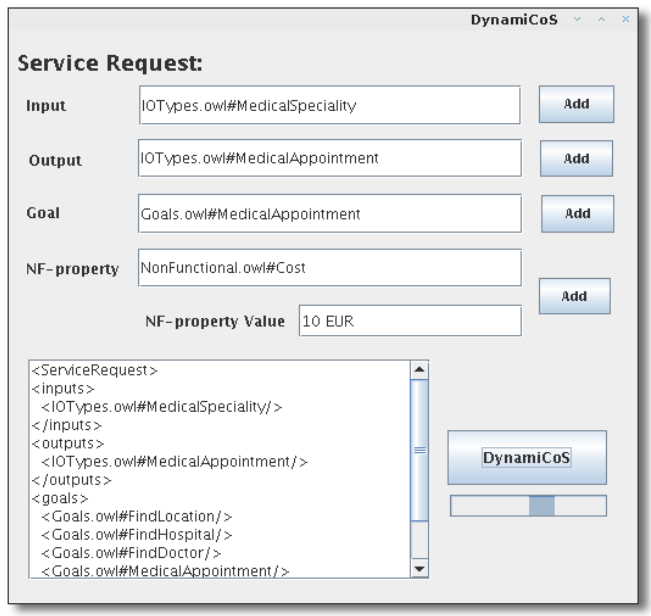


Figure 6-2 Service request GUI

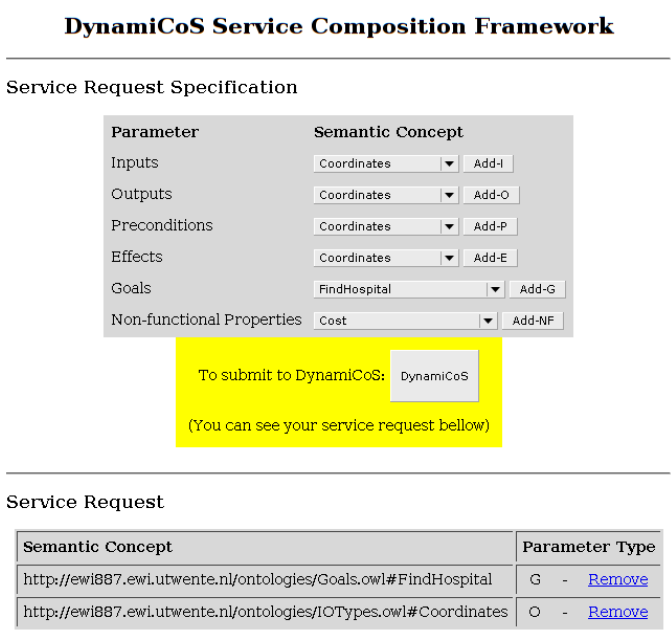


Figure 6-3 Service request Web interface

Code 2 Service request

```
<ServiceRequest>
    <input>IOTypes.owl#MedicalSpeciality</input>
    <output>IOTypes.owl#MedicalAppointment</output>
    <goal>Goals.owl#FindLocation</goal>
    <goal>Goals.owl#FindHospital</goal>
    <goal>Goals.owl#FindDoctor</goal>
    <goal>Goals.owl#MedicalAppointment</goal>
</ServiceRequest>
```

Service discovery

The service request XML document is analysed, and the $IOPE$, G and NF parameters' semantic concepts are extracted from this document. The service registry is queried using the service request parameters through the jUDDI API Inquiry function for services with $IOPE$, G and NF that are semantically related to the service request $IOPE$, G and NF parameters. The service discovery module allows to perform different types of service discovery, for example, pure goal-based service discovery, or discovery only based on a given output parameter. To enable a semantic service discovery, we use the OWL-API [107]. OWL-API allows to handle programmatically OWL ontologies, and use semnatic resoners, in our case Pellet [108], which enables semantic reasoning, namely to compute the subsumption relation between semantic concepts on an ontology.

Running Example: Table 6-1 shows the services that have been retrieved in our running example when we perform a pure goal-based service discovery.

Table 6-1 Discovered Services

ID	Input	Output
$S1$	IOTypes.owl#CellNumber	IOTypes.owl#Coordinates
$S2$	IOTypes.owl#Coordinates	Core.owl#Hospital
$S3$	IOTypes.owl#MedicalSpeciality Core.owl#MedicalPlaces	Core.owl#Physician
$S4$	Core.owl#Physician Core.owl#Patient	IOTypes.owl#MedicalAppointment
$S5$	Core.owl#Physician Core.owl#Patient	IOTypes.owl#MedicalAppointment

ID	Service Name	Service Goal
$S1$	locateUser	Goals#FindLocation
$S2$	findHospital	Goals#FindHospital
$S3$	findDoctor	Goals#FindDoctor
$S4$	makeMedicalAppointment	Goals#MedicalAppointment
$S5$	makeMedicalAppointmentAtHome	Goals#MedicalAppointment

Service composition

In our framework, service composition is performed in two steps:

1. Organise all the discovered services in the *Causal Link Matrix* (CLM), by computing all the possible semantic connections, or causal links, between the discovered services $IOPE$;
2. Automatically discover service compositions that fulfil the service request, based on the discovered services stored in the CLM.

The CLM is constructed by using the OWL-API [107], which allows one to handle and perform semantic inference in OWL ontologies through a semantic reasoner, in our case Pellet [108]. The service composition algorithm (Algorithm 1 in Section 4.4), has been implemented in Java, making use of the jGraphT [109] graph management library to manage the created service compositions. We use jGraph [110] graph visualisation library to print out the resulting service compositions (graphs).

Running Example: Table 6-2 shows the set of discovered services for our running example in a CLM. Figure 6-4 shows the service compositions generated by our composition algorithm based on the CLM of Table 6-2.

Table 6-2 Causal Link Matrix (CLM)

	IOPE1	IOPE2	IOPE3	IOPE4	IOPE5	IOPE6	IOPE7
IOPE1	0	$S1$,≡	0	0	0	0	0
IOPE2	0	0	0	$S2$,⊑	0	0	0
IOPE3	0	0	0	0	0	$S3$, ≡	0
IOPE4	0	0	0	0	0	$S3$, ≡	0
IOPE5	0	0	0	0	0	0	$S4$,≡ $S5$,≡
IOPE6	0	0	0	0	0	0	$S4$,≡ $S5$,≡

ID	IOPE Semantic Type
$IOPE1$	IOTypes.owl#CellNumber
$IOPE2$	IOTypes.owl#Coordinates
$IOPE3$	IOPTypes.owl#MedicalSpeciality
$IOPE4$	Core.owl#MedicalPlaces
$IOPE5$	Core.owl#Patient
$IOPE6$	Core.owl#Physician
$IOPE7$	IOTypes.owl#MedicalAppointment

6.2 DynamiCoS Evaluation

In this section we discuss the evaluation of the DynamiCoS framework. We evaluate DynamiCoS according to the evaluation framework discussed in Chapter 5. In the following we present our evaluation environment, then we present the results of our evaluation in terms of the evaluation framework metrics, and analyse the obtained results.

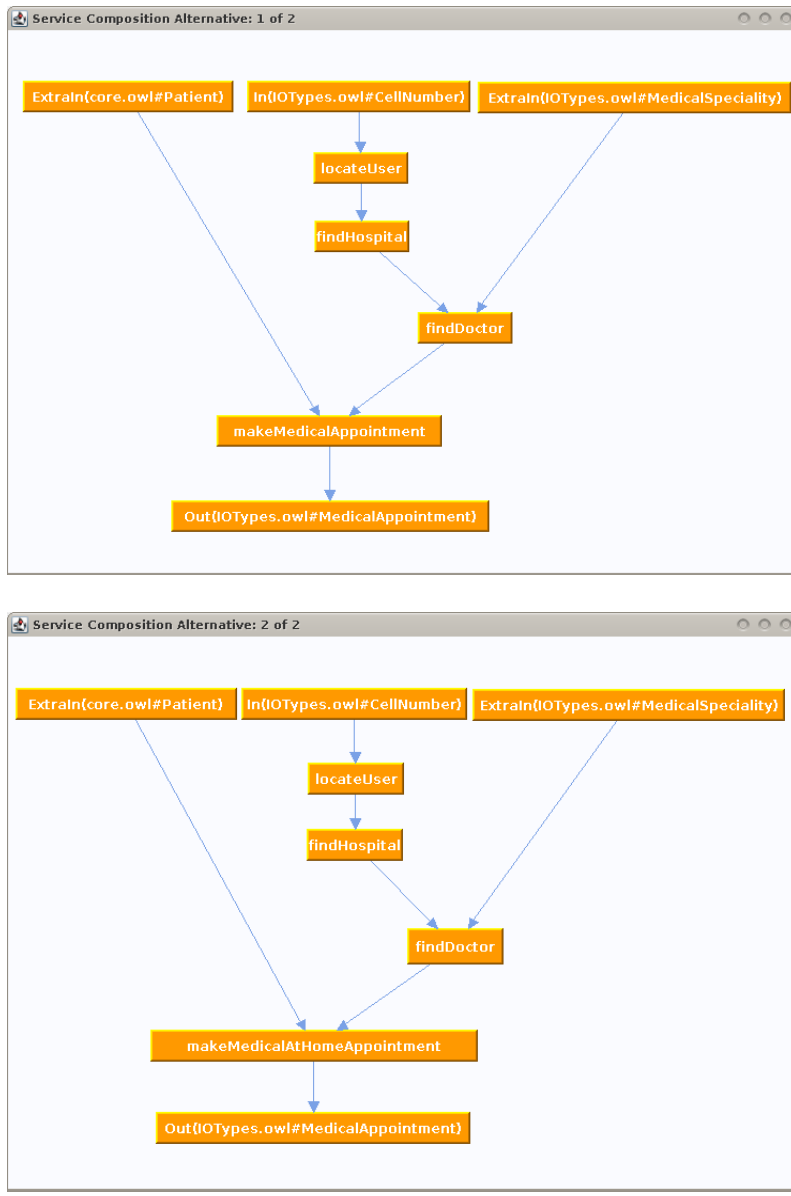


Figure 6-4 Generated service compositions

6.2.1 Evaluation Strategy

For the evaluation of DynamiCoS we have used a service collection defined in the domain of *e-health*, Appendix B.1. This collections contains two groups of semantic services, *manually defined services* and *automatically generated services*.

Manually defined services consist of services created and semantically annotated by humans, i.e., the services are defined with clear semantics. We have defined a set of 13 services of this type. These services are semantically annotated using the ontologies presented in Appendix A. These semantic services allow to perform different service requests, yielding to multiple service compositions that satisfy the service requests;

Automatically generated services consist of services automatically generated by a tool we have implemented (*RandServGen*). This tool creates SPATEL [89] semantic service descriptions, which can then be published in our framework registry. This set of semantic services contains 500 services.

Based on the *manually defined services*, we have defined two evaluation scenarios, Appendix B.2,which consist of a *service request* (SR) with a set of *reference service compositions* (RSC) that fulfil the goals specified in the service requests. Further evaluation scenarios can be created based on this services collection. In the evaluation of DynamiCoS we use *evaluation scenario 1*: "Make a medical appointment for a given medical speciality in the nearest hospital".

Evaluation Environment

Figure 6-5 shows our evaluation environment, which consists of three machines: the *DynamiCoS Client* (Intel Core 2 Duo 1.66GHz, 2GB memory), which supports the client's user interface; the *DynamiCoS Server* (Intel Pentium D 3.4GHz, 2GB memory), which performs all the service discovery, composition and delivery actions and the *DynamiCoS Registry* (Intel Pentium 4 2.4GHz, 512 MB memory), which stores the services published by service developers.

The evaluation was performed in our department, i.e., all the machines used (DynamiCoS client, server and registry) in the evaluation were in the same network.

6.2.2 Measurements

For the computation of the evaluation metrics values we have performed the following measurements:

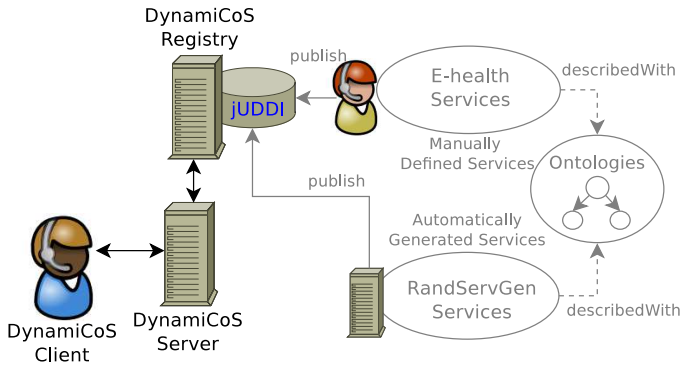


Figure 6-5 Evaluation environment

- *Number of discovered services (#discServs)*: number of services discovered or candidate component services;
- *Number of relevant services (#releDiscServs)*: number of discovered services (*#discServs*) used in service compositions that are correct (*releServComps*);
- *Number of IOPE (#IOPEs)*: total number of inputs, outputs, preconditions and effects operation parameters handled in the service composition process;
- *Number of discovered compositions (#discServComps)*: number of created service compositions for a service request;
- *Number of relevant compositions (#releServComps)*: number of created service compositions that are correct, i.e., meet the requirements specified in the service request (*SR*);
- *Service composition processing time (compPT)*: sum of the time required to process a user service request (*servReqProcT*), to perform the discovery of services (*servDiscT*) and to perform composition of services (*servCompT*).

6.2.3 Services and Composition Results

Table 6-3 presents the measured variables for service discovery, service compositions and *IOPE* parameters in the case when only manually defined services were taken into consideration in the service composition process.

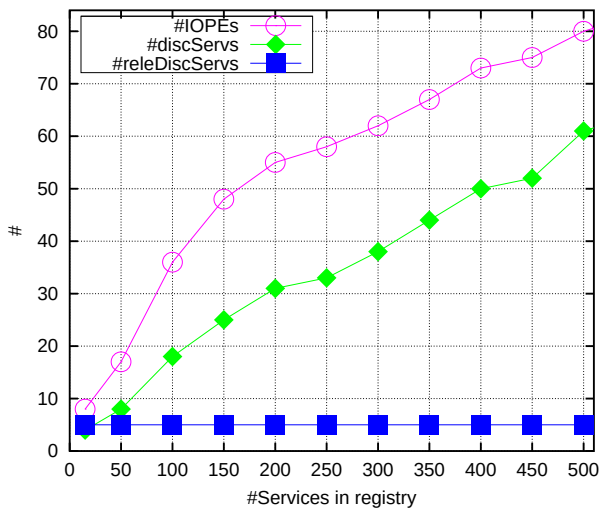
Figure 6-6 shows that the number of discovered services (*#discServs*), the number of services used in correct service compositions (*#releDiscServs*) and the total number of IOPE *#IOPEs* parameters of the services handled in the service composition process, when the number of services in the registry is gradually increased. The increase of the number of discovered services and service parameters (*#IOPE*) handled is almost *linear*. However, the num-

Table 6-3 Variables in the manually defined services scenario

Variable	Measure
#discServs	5
#releDiscServs	4
#IOPEs	8
#discServComps	2
#releServComps	1

ber of relevant services (*#releDiscServs*) is constant during the different experiments.

Figure 6-6 *IOPE* and discovered services



The increase of services handled in the service composition process when the number of services in the registry increases, was not reflected in an increase on the number of service compositions. Two service compositions have been obtained in all the experiments. Although the number of services increased, the newly added services were generated automatically, which did not introduce any additional correct service composition. Many services are discovered, since they were randomly annotated with goals that match the goals list of the service request, however their IOPE parameters are not coherent with the service goals, which makes them inappropriate to be component services in valid service compositions. In a more realistic situation, with real world services, services with common goals would be annotated with concepts that are semantically close to each other, so that the number of services discovered and *IOPE* semantic concepts han-

dled would probably be smaller.

Composition Time

When only manually defined services were used, in the first iteration (13 services in the registry) in Figure 6-7, the composition time was in the order of few seconds. This result showed that our approach for dynamic service composition could deliver a response time suitable for the support of runtime service composition situations, where real-time responses are required.

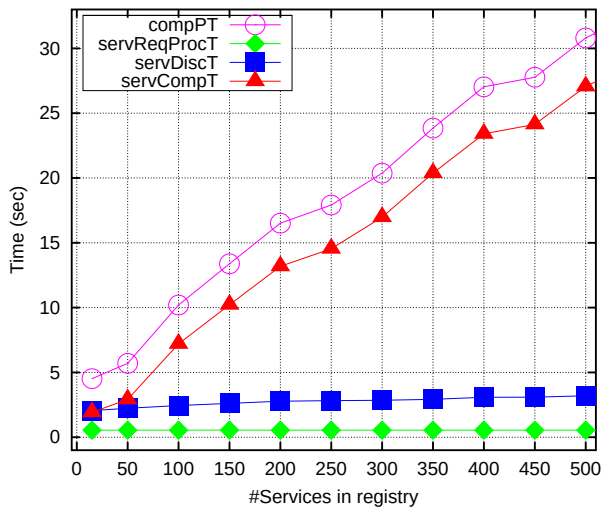


Figure 6-7 Composition time

Although automatically generated services may not be coherent, or useful, they allow us to test the scalability of the service composition approach. Figure 6-7 shows that the average processing time taken in the different phases of the composition process, when the number of services in the registry is increased. The variance on the measured times was very small. The higher variance was observed in the first experiment, with only manually defined services, where the variance was of 5.5%. On the other experiments the variance was lower, bellow 1% of the average time displayed in the graph. The service request processing time (*servReqProcT*) remains constant, as the same service request has been used in all the experiments. The service discovery time (*servDiscT*) slightly increases as the number of services in the registry increases, since more services are handled and retrieved. The biggest increase on the processing time is observed in the compo-

sition time (*servCompT*). This increase can be justified in Figure 6-6, which shows the number of discovered services (*#discServs*) and number of services' *IOPE* parameters (*#IOPEs*) considered in the composition process. As the number of *IOPE* parameters (*#IOPEs*) increases more processing is necessary to create the *CLM* and to execute the composition algorithm. This happens because more semantic concepts are handled in the composition process, i.e., more semantic reasoning is required to support the different activities of the service composition phase. We can conclude from these results that the required semantic reasoning introduces the biggest overhead in the service composition process.

6.2.4 Confusion Matrix Metrics

To report on the confusion matrix-based metrics, we consider the situation where only manually defined services were present in the services registry. The service request used and service compositions generated are discussed in Section 6.1.2.

In the sequel we compute the confusion matrix-based metrics for the service discovery and service composition processes of the DynamiCoS framework.

Service Discovery

To create the service discovery confusion matrix we have used the measurements from the performed experiments. *True positives* (*TP*) are services found that are used in correct service compositions (*releDiscServs*); *False positives* (*FP*) is the set of services found that are not used in correct service compositions (*discServs* \ *releDiscServs*); *False Negatives* (*FN*) are services that were not found, but we know they exist and could be used to create valid service compositions; and *True negatives* (*TN*) are other services that exist and are used in other service compositions, which are not considered in this particular service discovery process.

Table 6-4 Confusion Matrix for DynamiCoS Service Discovery

		Actual Values	
		P	N
Classified Values	P'	4	1
	N'	0	-

Table 6-4 presents the confusion matrix for the DynamiCoS service discovery process for the *evaluation scenario 1*, from Appendix B.2. We consider that the discovery process returns one *false positive*, which corresponds to the *makeMedicalAppointmen-*

tAtHome service, which is not exactly what is specified in the service request, although it has some similarities. The service request specifies the need for a medical appointment in the nearest hospital, not at home. This discovery is a consequence of the full automation of the service discovery process, and the expressiveness of the goals ontology (*Goals.owl*), where we have only defined a semantic type *MedicalAppointment*, which does distinguish between a hospital and home medical appointments. This is one of the possible sources of "false positives" in semantic service composition approaches, namely the mismatch between the real semantics of a service and the conceptualisation of the domain.

Table 6-5 presents the confusion matrix metrics for the service discovery process of DynamiCoS.

Table 6-5 Discovery confusion matrix-based metrics

Metrics	*Discovery*
$Precision\ or\ PPV$	$4/5 = 0.8$
$Recall\ or\ TPR$	$4/4 = 1$
FDR	$1/5 = 0.2$
Acc^{TP}	$4/5 = 0.8$
SNR^{TP}_{FP}	$4/1 = 4$

Service Composition

To create the confusion matrix for the service composition process we use the measurements from the performed experiments. *True positives* (*TP*) are correctly found service compositions (*releServComps*); *False positives* (*FP*) is the set of service compositions found that are not correct (*discServComps* \ *releServComps*); *False Negatives* are service compositions that were not found, but we know they exist, i.e., they are in the *reference service compositions* of the evaluation scenario; and *True negatives* (*TN*) are other service compositions that exist but are not relevant for the service request issued.

Table 6-6 Confusion Matrix for DynamiCoS Service Composition

		Actual Values	
		P	N
Classified Values	P'	1	1
	N'	0	-

Table 6-6 presents the confusion matrix for the DynamiCoS service composition process. We can observe that the composition process managed to find all the possible correct compositions,

defined in the *reference service compositions* (*RSC*) of the evaluation scenario. Nevertheless, it also found a service composition that does not completely match the request the user issued, given that it proposes a service that makes a medical appointment at home and not in the nearest hospital.

Table 6-7 presents the confusion matrix-based metrics for the service composition process of DynamiCoS.

Table 6-7 Composition confusion matrix-based metrics

Metrics	*Composition*
$Precision\ or\ PPV$	$1/2 = 0.5$
$Recall\ or\ TPR$	$1/1 = 1$
FDR	$1/2 = 0.5$
Acc^{TP}	$1/2 = 0.5$
SNR^{TP}_{FP}	$1/1 = 1$

These results show that our approach is capable of finding suitable service compositions, although it also proposes service compositions that may not completely meet the user intentions. However, the user may get a set of service compositions and select the most suitable one. The resulting service compositions could be ranked, for example, in terms of their quality of semantic links (services outputs-inputs semantic links) between the services of the composition.

6.2.5 Time Metrics

We have used the automatically generated service collection (Appendix B.1), to compute the time-based metrics. In each experiment we increased the number of services in the service registry by 50. All the experiments have been repeated 10 times. The results presented in this section are the average measured values of the evaluation metrics on the performed experiments.

Table 6-8 presents the average *service composition processing time* (*compPT*) for each of the performed experiments. Furthermore, Table 6-8 also presents the time variations in each iteration (IterationTime). Based on these results we can compute the scalability of the service composition approach. We used Equation (5.6) to compute the scalability of our approach. The scalability metric measures the variation of the total processing time when the number of services are varied in the service registry. The scalability is inversely proportional to such variation, i.e., the smaller the variation the higher the scalability. In Table 6-8 we already computed the partial service composition processing time

Table 6-8 Average Dynamic Service Composition Time per Experiment

Ite.	#**services**	**compPT**	**IterationTime**
1	50	5.701	-
2	100	10.197	4.496
3	150	13.384	3.187
4	200	16.512	3.128
5	250	17.913	1.401
6	300	20.375	2.462
7	350	23.846	3.471
8	400	27.034	3.188
9	450	27.772	0.738
10	500	30.806	3.034

(*compPT*) variation per increment. Based on these variations we can perform the following computation:

$$
\begin{aligned}
Scalability &= \left(\frac{\partial compPT}{\partial \#servs} \right)^{-1} \\
&\simeq \left(\frac{1}{N-1} \sum_{i=2}^{N} \frac{compPT(i) - compPT(i-1)}{\#servs(i) - \#servs(i-1)} \right)^{-1} \\
&= \left(\frac{1}{10-1} \frac{25.105}{50} \right)^{-1} \\
&= 17.925
\end{aligned}
$$

The value of the scalability metrics is not really meaningful without a reference value for this metric. Nevertheless, the computed value can be used for later reference for the evaluation of other semantic service composition approaches.

6.3 Discussion

In this chapter we present the prototype implementation we have developed for the DynamiCoS framework. The developed prototype was used to evaluate the DynamiCoS framework, using the evaluation framework presented in Chapter 5.

The performed evaluation experiments have shown that DynamiCoS can find the correct service compositions for a given service request. However, DynamiCoS also finds other service compositions that do not exactly match the service request, al-

though they deliver similar functionality. This shows one of the limitations of automatic service composition approaches, namely the *gap* between what the user wants and the requirements the user expresses, or can express, in the service request. In DynamiCoS we assumed that the user defines a complete set of requirements for the service request, which can be seen as a declarative specification of the of the desired service (composition).

Another conclusion taken from our experiments is that handling many semantic services in the process of automatic service composition can be quite time consuming. In real world scenarios we expect that fewer services are handled in the service composition process. Nevertheless, the use of semantic information to enable automatic processing is an expensive computing task, which needs to be carefully considered when designing real-time systems, such as runtime service composition supporting systems.

Chapter 7

User-centric Service Composition Support

The DynamiCoS framework, presented in Chapter 4, addresses part of the issues we identified to design user-centric service composition approaches, namely the automation of the different phases of the service composition life-cycle. DynamiCoS defines a rigid workflow of activities to support the service composition process independently of the users being supported. However, users are heterogeneous and have different characteristics and requirements in the service composition process, which demands different workflows of supporting activities. In this chapter we extend the DynamiCoS framework towards an adaptable framework, A-DynamiCoS, which aims at supporting user-centric service composition by invoking the necessary DynamiCoS basic components, according to the characteristics and requirements of the user driving the service composition process.

This chapter is organised as follows: Section 7.1 presents an overview of the proposed architecture to support user-centric service composition; Section 7.2 introduces *commands* used to communicate the user intentions to the back-end supporting system in the service composition process; Section 7.3 introduces the *front-end user support* component, which is responsible for mediating the user interactions with the supporting system; Section 7.4 presents the *coordinator* component, which implements the user commands by invoking the basic DynamiCoS framework components; and finally Section 7.5 presents two examples to illustrate the flexibility of our user-centric service composition support.

7.1 Architecture Overview

A user-centric service composition supporting system has to shield the users from the service composition process details, so that users without advanced technical knowledge can drive and benefit from service composition processes. Shielding of the users from the service composition details can be achieved by automating the service composition process. The DynamiCoS framework presented in Chapter 4 addresses these two issues (shielding users and automation of the service composition process) to design user-centric service composition support. However, DynamiCoS and other related works [31] mainly focus on the process of automatically composing existing services given a set of requirements, neglecting how these requirements are captured for each user being supported in the service composition process. Capturing the user requirements is of particular importance in user-centric service composition processes, since users have different degrees of knowledge and are in different situations, which influence the way requirements for service composition can be gathered. For example, a user that knows the domain in which he is looking for services may specify a concrete and detailed set of requirements, while another user that has no familiarity with the domain requires more assistance in the task of specifying requirements. The DynamiCoS framework assumes that users are able to specify a detailed set of requirements in one interaction, i.e., it is assumed that users have familiarity with the domains in which they are seeking services. This assumption allowed us to define a fixed and inflexible workflow of service composition supporting activities, namely: 1) service request; 2) services discovery; 3) services composition; 4) service execution. This workflow is common to other existing automatic service composition approaches.

7.1.1 Flexible Workflow

Although in the DynamiCoS framework we have defined a rigid supporting workflow, its basic components address all the different phases of the service composition life-cycle. Therefore, and considering that the basic components are independent of each other, they can be used as basic components to support a flexible support in a user-centric service composition process. Based on this observation, we extended the DynamiCoS framework with flexible coordination that invokes the necessary basic components, as function of the user requirements and characteristics at each step of the service composition process. We refer to the resulting

framework as *A-DynamiCoS* (Adaptable-DynamiCoS).

7.1.2 Architecture Components

Figure 7-1 shows the architecture of the the A-DynamiCoS framework.

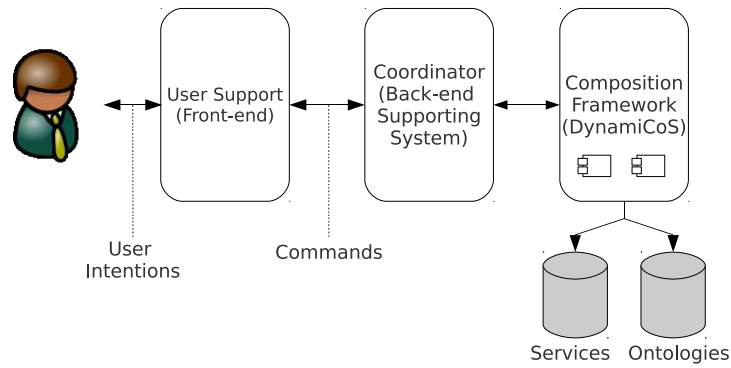


Figure 7-1 A-DynamiCoS architecture

The A-DynamiCoS framework extends the basic *Composition Framework* (DynamiCoS) with two components, namely the *coordinator* and *front-end user support*, to enable user-centric service composition by adapting the workflow of supporting activities of the service composition process according to the requirements and characteristics of the driving the composition process.

The *front-end user support* offers an interface for users to drive the service composition process. The aim of the user support component is to mediate the users' interactions with the composition process, shielding them from the actual details of the service composition process. To accomplish this, the user support component has to translate the user *intentions* at each step of the service composition process to *commands* that indicate which activities have to be performed in the service composition process. For example, the user support may be implemented as a simple web page that presents the user with a search box where he can write in natural language what service he is looking for. A request on this web page can be mapped into a service discovery command, which instructs the *coordinator* to carry out a service discovery activity to find services that deliver the requirements specified by the user.

The *coordinator* component is responsible for receiving the commands from the *user support* component, and map them onto the necessary invocations of the basic components of the composition framework, which deliver the required behaviour. In the example above, whenever the coordinator receives a command

for service discovery, it invokes the composition framework service discovery component, which queries the service registry for services that match the requested parameters. The matching services are then returned to the user support as result. The coordinator component aims at providing flexible support to users in the service composition process, allowing different commands to be issued as functions of the user intentions and needs at each step of the service composition. Different users may issue different commands in the course of the service composition process. This flexible coordination enables an on demand definition of the workflow of supporting activities, instead of having a rigid workflow to support all users, who possibly have different requirements and characteristics.

The *composition framework* component reuses the basic components of the DynamiCoS framework. The composition framework components have been implemented in a loosely coupled manner, which enables their invocation as required by the coordinator component to realise the different commands.

7.1.3 General Purpose Functionality

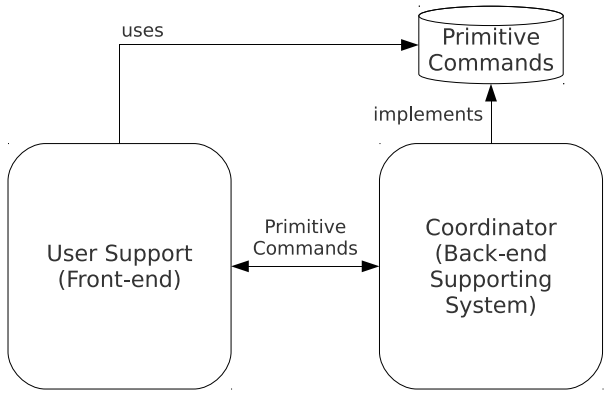
In A-DynamiCoS, the front-end user support component is defined as function of the characteristics and requirements of the target user population to be supported and application domain where services are to be composed. In contrast, the *back-end supporting system*, i.e., the coordinator and composition framework, is generic and can be reused in different application domains to support different types of users. As results the composition framework basic components provide all the basic functionality that is required to support the service composition process in different situations in different domains, possibly with different users. Only the order in which the basic components implementing the different service composition phases are used varies.

7.2 Commands

In A-DynamiCoS we have defined commands that instruct the back-end supporting system to execute the different activities of the service composition process.

Figure 7-2 depicts how these commands are used. The *back-end supporting system*, Coordinator, accepts a set of primitive commands that can be issued by the front-end. These commands are used to define the front-end user support, specifying the way

Figure 7-2
Commands



users are going to be supported in the service composition process.

Each command encapsulates a request for a given behaviour to be delivered by the back-end supporting system, more specifically by the basic components of the composition framework. Commands are encoded in messages, which can be interpreted by the coordinator in the back-end supporting system. Since several different commands can be issued we follow the *command pattern* [111] in this architecture to handle the different primitive commands. The command pattern gives a structured way to encapsulate different required behaviours in a single message format.

7.2.1 Command Types Dependency Graph

The service composition life-cycle supported by DynamiCoS (see Section 2.1.2) consists of a *service request* phase, a *service discovery* phase, a *service composition* phase and *service execution* phase. To support user-centric service composition we claim that the workflow of activities that support these phases has to be defined, on demand, according to the user that is being supported in the service composition process. This means that two different users may address these phases in different order, according to their requirements. Figure 7-3 presents two execution workflows, two different users drive the service composition processes differently.

User 1:

1. User specifies a *service request*, where he defines a detailed set of requirements for the service (composition) he needs;
2. The *service discovery* phase takes place, where candidate component services are discovered for each of the specified requirements;
3. User *selects* the relevant services for each of the requirements;

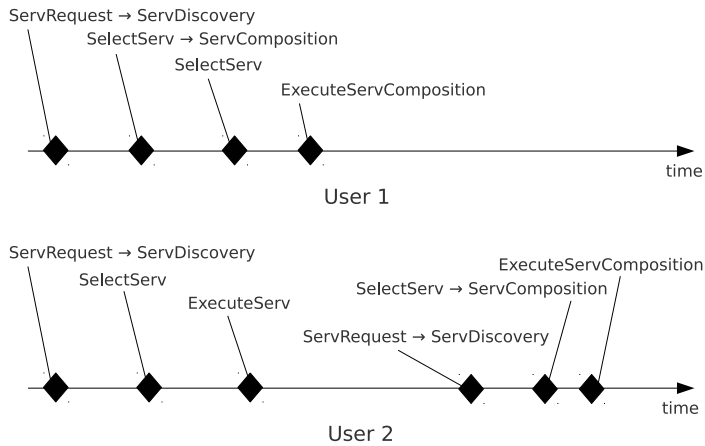


Figure 7-3 Two different execution workflows

4. The *service composition* phase takes place, where the discovered services are composed according to the user service request;
5. If multiple service compositions are proposed, the user *selects* one of the proposed services;
6. The user is presented with the interface to use the service (*service execution* phase).

User 2:

1. User specifies a *service request*, detailing the requirements for a service he wants at a given moment;
2. The *service discovery* phase for find services that satisfy the specified requirements takes place;
3. The user *selects* one of the services;
4. The user is presented with the interface to use the service (*service execution* phase);
5. Based on the result, the user decides to use another service, so he requests a new service (*service request*);
6. The *service discovery* phase takes place again to find services that can deliver the new requirements specified by the user;
7. The user *selects* one of the discovered services;
8. The user is presented with the new service interface for its usage (*service execution* phase). However, in this step some input values are automatically entered, or suggested to the user, because they are available from the execution of the first service. This process can go on, as long as the user requires.

Although the two users have different workflows of supporting activities, we can observe that they use the same basic activities, namely: *service request*, *service discovery*, *select*; *service compo-*

sition, *service execution*. This shows that we can define generic commands to support the different phases of the service composition process in different situations, where users with different requirements can be seamlessly supported by the same basic commands.

Although the same set of commands can be used, in different orders, across multiple situations to support different users, the order in which the commands can be invoked has to comply with some rules. For example, a service execution command can only be issued once a service has been discovered. To describe these *interdependencies* between commands we define a dependency graph [112] of command types. The dependency graph specifies the basic types of commands and their causal relations.

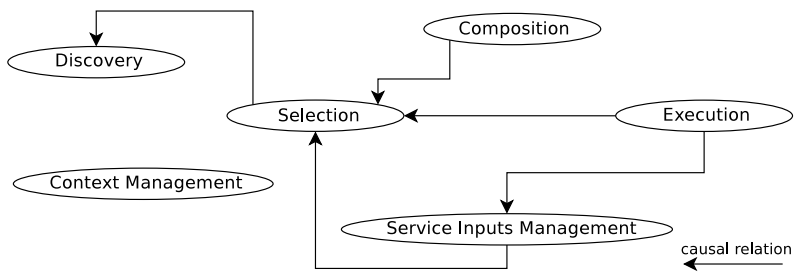


Figure 7-4 Command types dependency graph

Figure 7-4 presents the *Command Types Dependency Graph* we defined to develop our service composition support. It covers the typical phases of the services composition life-cycle and their causal relations. Furthermore, some other command types were proposed, namely the *Service Input Management*, which allows one to describe commands that are used to collect the values the user enters for the services execution, and *Context Management*, which allows one to manage information from the user context.

7.2.2 Primitive Commands

We refer to the commands that can be issued to the back-end supporting system coordinator as *primitive commands*.

A primitive command specifies a request for some functionality related with the service composition process. A primitive command defines a usage protocol, specifying its request and response messages. The command type of a primitive command defines its preconditions or dependencies on other primitive commands.

It is not reasonable to assume that the supporting system designer will envision all the possible primitive commands before the supporting system is deployed. Therefore, the supporting sys-

tem has to be extensible concerning the definition of new primitive commands, i.e., it should be possible to add new primitive commands in the supporting system later. A-DynamiCoS allows designers to add new primitive commands as long as they comply with the command types dependency graph. This guarantees that newly added primitive commands do not introduce undesired behaviours, which may lead the system to inconsistent states.

Examples of Primitive Commands

In our research we have identified several primitive commands. Most of these commands are directly related with the service composition life-cycle phases, such as: *ServTypeDiscovery*, discovers a service based on a given type; *SeleServ*, selects a service and adds it to a service composition; *ExecServ*, retrieves services ready for execution. Furthermore, we have also defined some auxiliary primitive commands, such as: *ValidateInputs*, stores the service inputs entered by the user if they are valid; *AddToContext*, stores user context information or service results in the execution context of the back-end supporting system.

All the primitive commands we have identified and used in the defined use cases where A-DynamiCoS was used are presented in Appendix D.2.

7.3 User Support

The *front-end user support* component defines the interface of the user with the back-end supporting system. This component is defined according to the target user population to be supported in the process of service composition and the usage scenario that is to be supported.

7.3.1 Component Architecture

Figure 7-5 presents the internal structure of the front-end user support component. It basically consists of two parts, namely *user interface* and *command flow*. The *user interface* defines the interaction points for the supported application, where the user intentions concerning the service composition process are collected. The *command flow* defines workflows of primitive commands defined to support the identified target user population to be supported.

This component translates *user intentions* into *primitive com-*

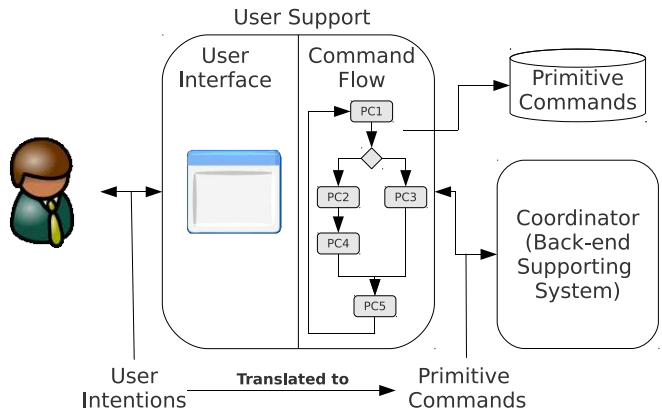


Figure 7-5 Front-end user support component internal structure

mands understandable by the back-end supporting system, allowing the user to drive the service composition process.

7.3.2 User Intentions

The front-end user support mediates the interactions between users and the service composition supporting systems, allowing the users to drive the service composition process as they need.

However, during the service composition process different users may have different requirements, different knowledge, different context, etc. These differences make the users drive the service composition process in different ways. These differences are reflected in different *user intentions* towards the service composition process.

The front-end user support is defined according to the application domain, the users and the service composition process it is supposed to support, or usage scenario. For example, in the e-government usage scenario, presented in Section 3.2.1, users are supported when consuming government electronic services. The user support in this situation defines an interface where users can find the services they need and get help to resolve the preconditions required to use these services. Preconditions are resolved by composing other services that can fulfil the missing preconditions. In this usage scenario users may have different intentions towards the service composition process, but somehow they are limited by the objectives that are defined for the application, in this case support users on consuming government electronic services.

To capture the essential activities that can be performed in a given usage scenario that makes use of service composition, the user support is defined by domain specialists according to the

application domain characteristics. Domains specialists know the domain and know the goal of the application being developed, to support a given usage scenario, which allows them to define a front-end user support suitable to capture the users intentions towards the service composition process.

7.3.3 Command Flows

In our approach we assume that a front-end user support designer (domain specialist) specifies *command flows* according to the target user population of the usage scenario being designed. Command flows should be determined to match the requirements of the target user population, so that these users can achieve their objectives by composing existing services. The usage scenario for which user support is being designed also influences the definition of the command flows. Therefore, command flows are defined based on two sources of requirements: 1) the usage scenario being designed (e.g., deliver citizens with government electronic services), and 2) the user population to be supported (e.g., citizens who require simple and easy to use mechanisms to assist them on using the different electronic services from the government).

A command flow is defined as a combination of primitive commands organised in a workflow of commands suitable to fulfil the requirements identified for a given user support. A command flow may define different execution flows, which allows to capture the needs of users with different characteristics, or intentions.

To guarantee that command flows respect the command types interdependencies, the *command types dependency graph* can be used. It provides the basic rules to be followed to define valid command flows.

Command flows are encoded in the user support application, namely in the *user interface*. The user interface defines the interaction points that users can follow, according to their intentions at a given moment of the service composition process. Multiple primitive commands can be accessible to the user at a given moment, which allows the user to decide which workflow of activities in the service composition process shall be followed at each moment. Although the user interface has to follow a prescribed command flow, the user does not have to be aware of the existence of this command flow, it can be made completely transparent to the user, as long as the front-end user support can capture the user intentions at each step of the service composition process.

Figure 7-6 shows an example of a command flow described as a UML activity diagram [113]. This command flow is similar to

the one defined to support the e-government use case, where citizens are supported to use government services. A user starts by defining which service he wants to use, which is translated into a *ServTypeDiscovery* command. Then the user gets a set of services, from which he selects one to be used, through the *SeleServ* primitive command, which communicates the selected service to the back-end supporting system. Then the user is presented with the service interface, where the different inputs of the service are requested. The user enters the inputs he knows. Once the user indicates that he entered all the inputs he knows, the *ValidateInputs* primitive command is used, which communicates to the back-end supporting system which inputs were introduced, and the ones the user can not provide. In case the user could not provide some inputs, the command flow goes to the *ResolveServ* command, which requests the back-end supporting system to discover services with outputs that can provide the missing inputs. This triggers a backwards-chaining service composition process, in which the user selects the services to be used to provide each of the missing input. Once all the services inputs are available, the service composition can be executed by issuing the *ExecServ* command.

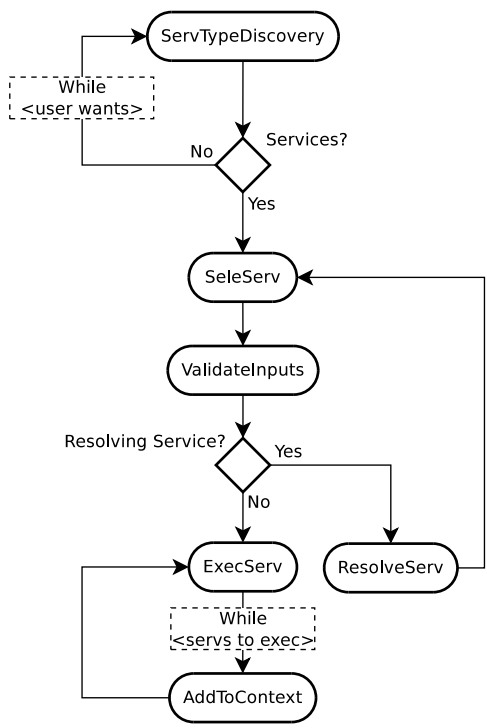


Figure 7-6 Command flow example

7.4 Coordinator

The *back-end supporting system* is composed by two components, namely the *coordinator* and the *composition framework*.

7.4.1 Component Architecture

Figure 7-7 depicts the internal structure of the coordinator. The coordinator contains several sub-components: *Interpreter*, *Validator* and *Executor*. Furthermore, it also includes the *Composition and Execution Context* manager, which basically holds the state of each user session being handled at each moment, to allow multiple user sessions to be supported simultaneously.

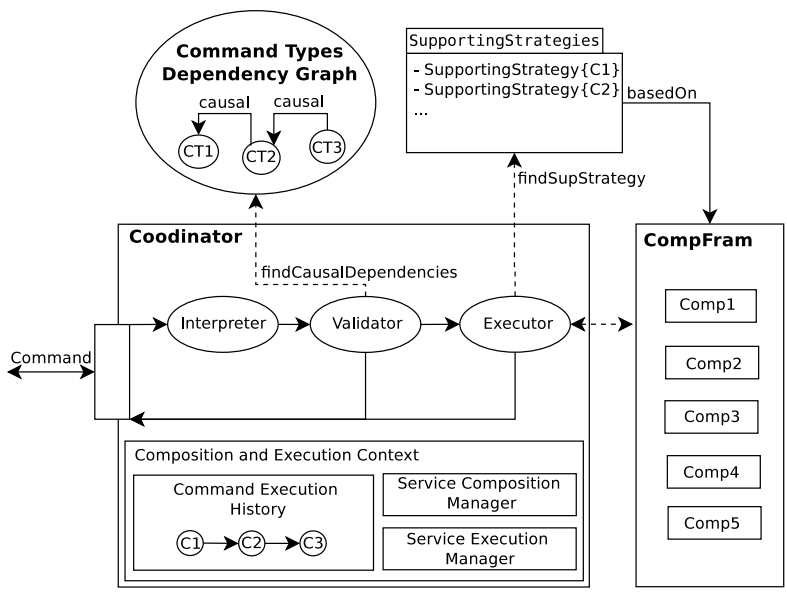


Figure 7-7 Architecture of A-DynamiCoS coordinator

The coordinator component gets a command, representing a user intention, and executes the *supporting strategy* that delivers the behaviour required by the user, by invoking the *composition framework* basic components in a given order, defined by the primitive command supporting strategy. The coordinator has been defined to be generic and extensible, i.e., it is independent of the *primitive commands* being processed, the *supporting strategies* that are executed and the *composition framework* components that are called to realise the supporting strategies. Furthermore, the coordinator enforces the soundness of the coordination being performed, namely by checking the order of execution of primitive commands against their dependencies, through the command

types dependency graph.

Command Pattern Implementation

The coordinator component can be accessed via *primitive commands* that are used to communicate different actions to the back-end supporting system.

The interactions between the front-end and back-end follows the command pattern [111]. For each primitive command we define a message with three parameters:

- *UserSessionID* identifies the user session being processed;
- *CommandID* identifies the type of primitive command issued to the coordinator;
- *CommandParameters* contains the parameters of this specific primitive command.

Once the coordinator receives a message containing a primitive command, this message is processed by the *Interpreter*, which performs two actions: 1) parses the message to extract the *UserSessionID*, *CommandID* and *CommandParameters*; and 2) opens the user session context associated to the *UserSessionID*, which is read from the *service composition and execution manager*.

Command Validation

Once a message is interpreted, i.e., the command type issued is identified, the coordinator proceeds to the *Command Validation*.

Figure 7-8 presents the procedure taken for the validation of issued primitive commands. The *Validator* verifies whether the issued interaction type is valid or not by inspecting the *command types dependency graph* for the causal dependencies of the issued primitive command. To achieve this, the coordinator uses the *command execution history*, which stores the set of primitive commands previously issued by the user.

In case the issued primitive command is considered invalid, i.e., its causal dependencies are not satisfied, the coordinator aborts the execution of its *supporting strategy*, and returns an error message to the front-end user support.

Command Execution

Once a command has been validated, the *Executor* component executes the *supporting strategy* that realises the primitive command indicated in the *CommandID*. The supporting strategy determines which basic components of the composition framework are going to be invoked, in which order, and which information is stored in

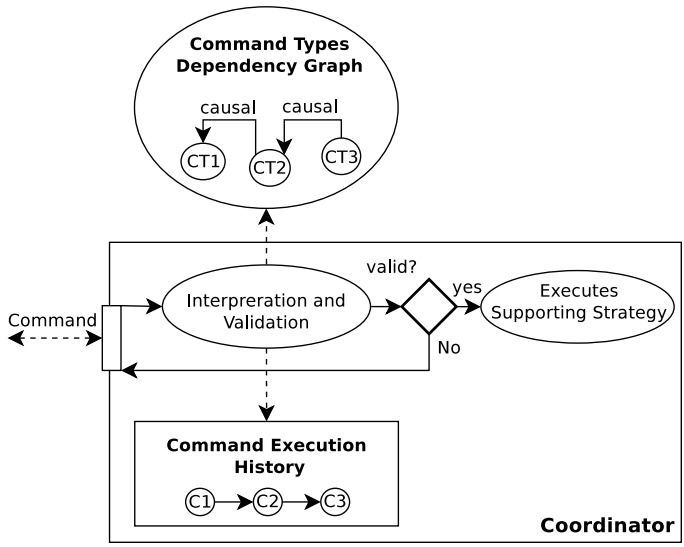


Figure 7-8
Command
validation

the composition and execution context of the user session.

7.4.2 Composition Framework Basic Components

The basic components of the composition framework have been defined in the *DynamiCoS* framework in Chapter 4. These components have been defined in order to automatically support the whole service composition process, aiming at shielding users from the service composition details.

The design of the composition framework components we have defined a component to support the activities of required to support each of the service composition life-cycle phases, for example, *ServiceDiscoverer* for service discovery and *ServiceComposer* for service composition. Furthermore, we have developed auxiliary components that support some activities of these two DynamiCoS components, namely the *SemanticReasoner*, which performs semantic reasoning on services parameters and interface semantic concepts, which are references to ontologies; and the *CLMManager*, which manages the CLM matrix. The CLM matrix stores all the possible semantic links between the services that are considered in the service composition process.

The composition framework components are *stateless*, i.e., they perform a given set of operations and do not keep any state. The state of the service composition is kept in a set of data structures, which hold the discovered services and service compositions being created, Figure 7-9. A service has a service interface, with in-

puts/outputs parameters, and has some other properties, namely goals, non-functional properties and natural language description of the services. The *CLM* stores all the possible semantic links between the outputs and inputs of the discovered services. Furthermore, the service composition, which is represented as a graph, is composed of multiple services.

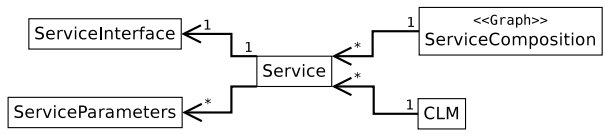


Figure 7-9 Composition framework data structures

7.4.3 Composition and Execution Context

The *Composition and Execution Context* keeps the data handled in the service composition process. Namely, it contains the information of the composition process, the information values from services' inputs and outputs, and information from the user context, which can be used as inputs for services. Some of the data structures from this component were adapted from the data structures of the DynamiCoS framework, namely to handle discovered services and service compositions.

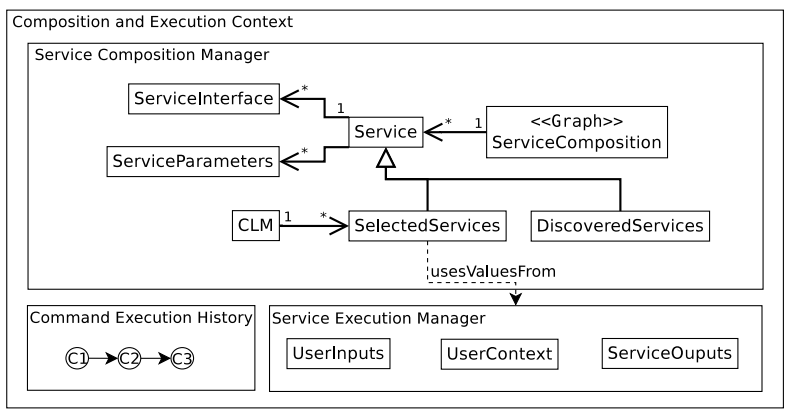


Figure 7-10 Composition and execution context manager

Figure 7-10 presents the data structures and components of the *composition and execution context* manager. This component has three basic sub-components: *Command Execution History*, *Service Composition Manager* and *Service Execution Manager*. Each of these components is discussed in the sequel.

Command Execution History

The *Command Execution History* component is responsible for keeping track of all the primitive commands issued by the front-end user support, and executed, in each user session. This component allows the coordinator to verify the correct order of execution of the primitive commands, guaranteeing in this way that the supporting system does not enter inconsistent states.

Service Composition Manager

The *Service Composition Manager* is responsible for storing the service compositions being created. This component implements the data structures shown in Figure 7-9. We have defined two other classes, the *SelectedServices* and the *DiscoveredServices*. *SelectedServices* stores the services that the user selects from the list of *DiscoveredServices*. The *SelectedServices* are services selected for composition, which the user wants to use (execute). The *SelectedServices* are used to define the *CLM*, which contains all the candidate services that can be used in service compositions.

Based on these data structures, the *supporting strategies* can perform the different activities required to support users in the service composition process. These data structures maintain the state of an instance of the composition process, or user session, while the basic composition framework components are used to perform the necessary processing in the service composition process, for example, adding a service to a service composition or discovering a new service.

Service Execution Manager

The *Service Execution Manager* is responsible for keeping the information values associated with the user and services execution context. Normally service composition approaches separate quite explicitly service (composition) creation from service (composition) execution. In these approaches, service compositions, once created, are deployed in an execution engine in oder to be executed. The execution engine manages all the service parameter values at runtime. Since we do not explicitly separate service composition design-time and runtime, in our user-centric approach, the service execution manager is integrated in the back-end supporting system. This design decision allows the framework to support a service composition process that alternates between the composition and execution of services, since it allows the framework to keep track of the service parameter values, while further

changes on the service composition are still possible.

Three main classes of data values are stored in the service execution manager: *UserInputs*, *UserContext* and *ServiceOutputs*. *UserInputs* store the values the user enters for the services execution, i.e., service inputs. *UserContext* stores values gathered from the user context, e.g., user location, which can be used as inputs for the execution of services of the service compositions. These values can be gathered at different moments. For example, user context can be gathered whenever a user starts to interact with the system, when the front-end user support is initialised, or whenever some change happens in the user environment, for example, when the user changes location. *ServiceOutputs* store the executed services outputs, which can then be used as inputs for the execution of other services.

Table 7-1 presents the information kept in each value stored the service execution manager.

Table 7-1 Service execution data structure

ValueID	Unique ID associated to a given data values.
SemanticalType	Semantic type of the stored data value.
SyntacticalValue	Syntactical type of the stored data values.
Value	Stored value.

Service Composition and Execution Process

In most of the service composition approaches in the literature [34] the service composition created is general and in principle is not developed for a specific end-user. However, in user-centric service composition scenarios, we assume that service compositions are created for a specific end-user, with a specific set of requirements. Users are heterogeneous and may have different requirements. We claim that the end-users need to play a central role in the service composition process. However, service end-users have different degrees of knowledge on the application domain and its services. This limits the way users can specify the service requirements and how they interact with the service composition supporting system.

Many users have some initial requirements, and require assistance to provide them with further information to learn about the application domain of the services being composed, so that they can take decisions and proceed in the service composition process. This lack of knowledge leads the users to the use of some services, to acquire further knowledge, while developing the service composition that will eventually deliver all the functionality required by the users. Figure 7-11 shows a simple example of a workflow of activities that can take place in our user-centric service composition

approach.

Figure 7-11 User-centric service composition activities workflow Execution

In this workflow of activities we can observe that the service execution and service composition activities take place in different moments and are interleaved. This scenario can describe the following workflow:

1. User looks for a service to satisfy requirement *R1*, for which *S1* was discovered;
2. User selects *S1*;
3. User executes *S1*;
4. Once the user executes *S1*, he learns some information and decides that he needs another service to satisfy a new requirement *R2*, he has just identified;
5. *S2* is discovered and selected by the user;
6. *S2* can reuse information from *S1* inputs/outputs, i.e., *S1* is composed with *S2*, resulting in the composition *S1S2*;
7. The resulting composition (*S1S2*) is then selected for execution;
8. *S1S2* is set for execution, which corresponds to the execution of *S2* by reusing some values from the *S1* inputs/outputs.

This simple example demonstrates that a user-centric service composition process is a dynamic process, and the service composition can be defined and extended dynamically, possibly leading to workflow executions where execution and composition of services is interleaved. Traditional service composition approaches with a design phase followed by an execution phase are not capable of supporting users in this type of service composition scenarios.

We have taken an architectural design decision in order to address this dynamic composition approach where service compositions are defined in an incremental fashion and services can be executed while the service composition is still being constructed. This decision is reflected in the fact that we do not perform service composition deployment, but assume that services from the service composition are executed one at a time, whenever they are commanded for execution and have the required information

values to be executed. To cope with this, we delegate the actual execution of services to the front-end user support component, although the back-end supporting system manages all the information values required for service execution and the results of the execution. The service results are stored in the back-end supporting system *composition and execution manager*, which can later be used by other services. In the example in Figure 7-11, the results from *S1* are used later in the execution of *S2*. Apart from allowing a more flexible composition and execution of services, this architectural decision allows us to make the back-end supporting system generic and independent of concrete service execution technologies. The drawback of this architectural decision is that the front-end user support has to handle the service execution process. This requires the user support to have the functionality necessary for invoking the services. Nevertheless, we consider that this decision is justifiable since the front-end user support is *application domain specific*, which allows us to define mechanisms to execute the services of the domain. This is beneficial as services can be defined with different implementation protocols, e.g.: RESTful web services [114], SOAP web services [11] or even proprietary APIs.

7.4.4 Supporting Strategies

Supporting strategies are defined to realise the different *primitive commands* as functions of the basic components of the composition framework and the composition and execution manager data structures (see Figure 7-12). Supporting strategies invoke the different composition framework components on demand, as they are required by the users being supported in the service composition process.

The basic assumption to realise such on demand invocation of the composition framework components is that these components are exposed independent of each other, which enables to call them whenever they are required. This allows to create different supporting strategies as functions of the basic components of the composition framework. Furthermore, the supporting strategies make use of the *composition and execution manager* data structures to store and handle services and service compositions, and information associated to services execution and user context.

New components can be added to the composition framework, or new primitive commands may be required. This allows one to define new supporting strategies which implement new behaviours whenever it is necessary. We have designed A-DynamiCoS to allow

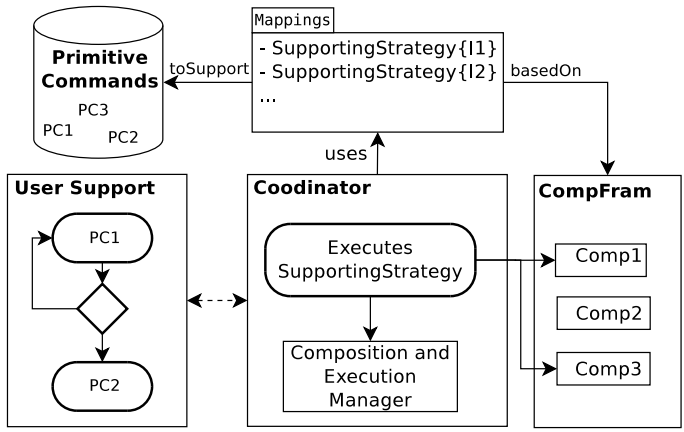


Figure 7-12 Support strategies

these extensions, while keeping the same *coordinator* dynamics, namely to *interpret* primitive commands, *validate* them, and then *execute* a supporting strategy to implement the support required for each primitive command. This makes A-DynamiCoS framework flexible/extensible and adaptable to new requirements.

The definition of supporting strategies can be made without having to know the behaviour of the coordinator component. The designer of primitive commands, and supporting strategies, only needs to know the composition framework components, and their interfaces, and the composition and execution manager data structures that hold the different information handled by the supporting strategies. Supporting strategies can be defined following the plugin pattern [115]. This approach would allow designers to define supporting strategies as plugins that can be deployed at runtime in the coordination *executor* at runtime whenever they are required.

Figure 7-13 shows the supporting strategy for the *ServTypeDiscovery* primitive command. This is one of the primitive commands we have defined in our research. The complete list of defined primitive commands is presented in Appendix D.

The basic actions taken in this supporting strategy are:

1. Discover matching services for each requested service type (*ServType*) using the *ServiceDiscoverer* composition framework component;
2. In case services have been discovered, add the services to the *Composition and Execution Context* of the user session instance, to the *DiscoveredServices* data structure;
3. Return a list of discovered services organised by the requested types or an empty list in case no service was discovered.

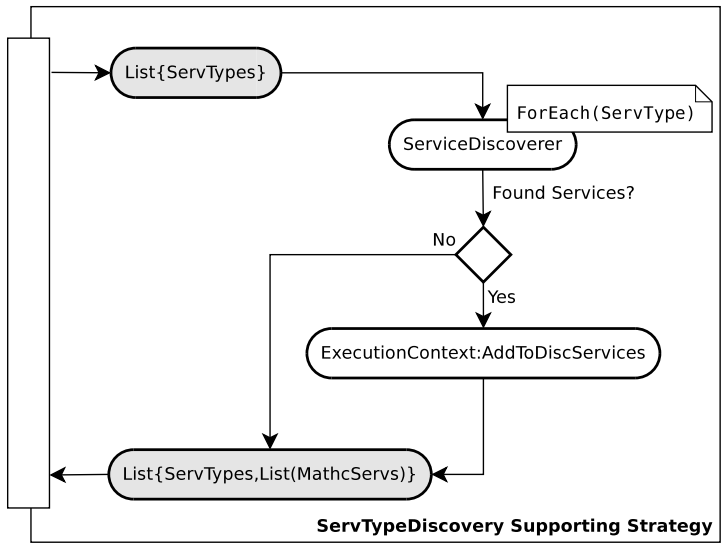


Figure 7-13 *ServType-Discovery* supporting coordination strategy

7.5 Example

In this section we provide two examples of service composition supporting workflows, or *command flows*. These examples show that different situations, usage scenarios and users, require different support in the service composition process. Furthermore, we also show that the A-DynamiCoS framework is capable of handling such heterogeneity.

We start by defining the support required in the front-end user support, according to the A-DynamiCoS framework. To achieve this, the user support makes use of basic primitive commands in order to define *command flows* that support the users in a given usage scenario from a given application domain.

The two examples represent two types of service composition approaches, namely *backwards-chaining service composition* and *forwards-chaining service composition*. We call them as *Objective-oriented flow* and *Incremental Compose-Execute flow*, respectively.

7.5.1 Objective-oriented Flow

This command flow defines a mechanism to support users that have a specific objective to achieve, which means they know which outcome they want by using some services. We define these services as *goal services*. However, users may require assistance to resolve the preconditions of the services that support such objectives. The preconditions of the *goal services* correspond to the

input information required to execute these services. To resolve the services' preconditions a *backwards chaining service composition* process is performed, starting from the goal services. In this flow, services are discovered, which have outputs that can provide values to resolve the preconditions (inputs) of services in the service composition and which the user is not able to provide. Figure 7-14 shows this command flow.

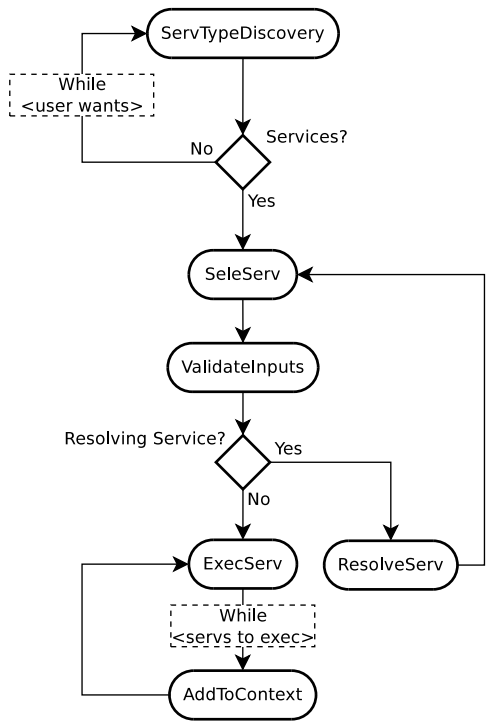


Figure 7-14 Objective-oriented flow

In this flow, the user first discovers the *goal services* by issuing the *ServTypeDiscovery* primitive command. Then, if any service is discovered, the user issues the *SeleServ* primitive command that allows the user to select one service or several services (the goal services). This command also adds the selected services to the service composition. Based on the user's selection, the system asks the user to input the information necessary for execution of the selected service. The user enters the information values that he knows, so that when *ValidateInputs* primitive command is issued this information is forwarded to the back-end supporting system. If the user has not been able to provide some of the inputs, the command flow issues the *ResolveServ* primitive command. This command discover services that have outputs that

match the unfulfilled inputs. Once services are discovered for each of the unfulfilled inputs, the command flow starts a selection activity again, informing the back-end of the services selected to resolve the unfulfilled inputs. This selection process triggers a backward chaining service composition in which the output of a discovered services is composed with the unfilled input of the service composition service. The resolution of unfulfilled inputs recurs until all the service composition inputs are available and validated, or fails in case there are not services to provide the unfulfilled inputs. Once all the service composition inputs are available the command flow proceeds by executing each of the services in the composition, according to the flow of execution created, using the *ExecServ* primitive command. Every time a service is executed, its results are added to the execution context, through the *AddToContext* primitive command. The stored information is used by the remaining services in the service composition, i.e., the services that have inputs composed with the outputs of the services already executed.

7.5.2 Incremental Compose-Execution Flow

This command flow defines a mechanism that allows to incrementally add new services to an existing service composition, where all the service composition services were already executed. This incremental composition allows the newly added services to make use of the input and output values of the already executed services of the service composition. This command flow enables users to incrementally learn from the execution of services, which can help them on deciding on the next action to be taken, or next service to be executed. Furthermore, it also allows users to reuse information from executed services in the new services selected for execution. This flow mimics a *forwards chaining* service composition process. Figure 7-1 shows this command flow.

The user starts the process by specifying which service he wants to execute, by using the *ServTypeDiscovery* primitive command. This command retrieves services that match the service type the user specified. From the set of retrieved services, the user selects one service, using the *SeleServ* primitive command. This command allows to communicate the selected service to the back-end supporting system, which adds the service to the service composition. As a response to this command, the back-end supporting system suggests possible inputs for the selected services, which are values from the previously executed services' input and output values or from the user context information. These suggestions are

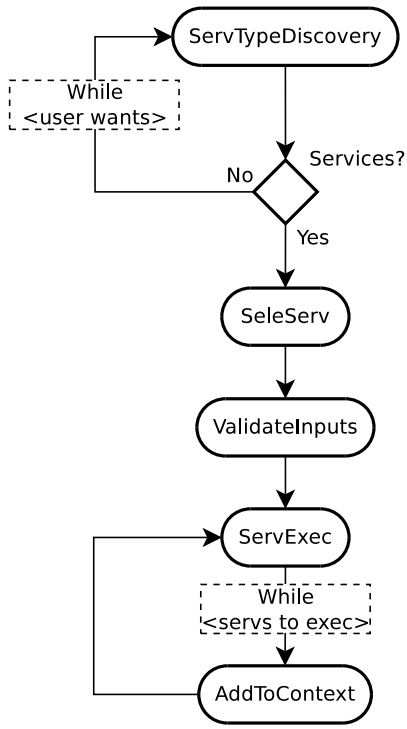


Figure 7-15 Incremental compose execute flow

made based on valid semantic matching between values from the execution context manager and the selected services' input parameters. The service input values that the user enters, or selects from the provided suggestions, are communicated to the supporting system through the *ValidateInputs* primitive command. This command establishes the input values to be used in the execution of the service. The user then requests the actual execution of the service using the *ExecServ* primitive command. Once the service is executed, the results from this execution are stored in the user composition and execution context, using the *AddToContext* primitive command, so that other services can use them later.

7.6 Discussion

In this chapter we present the A-DynamiCoS framework, which allows to support user-centric service composition processes. The A-DynamiCoS framework is defined as an extension of the DynamiCoS framework. The DynamiCoS framework defines a rigid supporting workflow of activities to support the service composi-

tion process, namely: 1) service request; 2) service discovery; 3) service composition; 4) service execution. However, different users may require different workflows of activities to support their service composition process. Although DynamiCoS provides a rigid workflow, it has independent basic components to address the different supporting of the service composition life-cycle phases. These basic components allowed us to design an extension to the DynamiCoS framework that consists of a flexible coordination, which makes use of the DynamiCoS basic components to deliver the functionality users require at each step of the service composition process.

To deliver such an adaptable support we have defined two additional components in the A-DynamiCoS architecture: *front-end user support* and *coordinator*. The *front-end user support* component is defined by application domain experts, and aims at providing users with a suitable interface to support them on the service composition process. To enable the definition of such user supports, we have used *commands*, which allow to communicate the user intentions to the back-end supporting system (i.e., Coordinator and Composition Framework). We describe the set of commands that can be used in the front-end user support as *primitive commands*. Primitive commands can be combined in different ways in order to define *command flows* suitable to the target user population of the usage scenario being designed. In the back-end supporting system, the coordinator component reacts to each primitive commands issued by the users by executing a *supporting strategy*. Supporting strategies are implemented as function of the basic composition framework components. To grant correct specification of *command flows* we define a *command types dependency graph*, which specifies the possible command types, and their causal relations. A primitive command is defined as an instance of one of the elements of the command types dependency graph.

By designing A-DynamiCoS with an adaptable coordination we grant that the service composition process can be driven on demand by users, as they require, instead of having a rigid and unique supporting workflow for multiple users. This flexibility allows to define appropriate support for users with different characteristics. One of the advantages of this is the definition of supporting command flows that allow to interleave the composition and execution of services. This is of high importance, as some user-centric service composition usage scenarios require such supporting workflow. Furthermore, and considering that the front-

end user support can be defined with intuitive interfaces, where command flows are embedded, users without technical knowledge can drive the service composition process and benefit from better personalisation of service delivery, as composition of existing services.

Chapter 8

A-DynamiCoS Implementation and Validation

In this chapter we discuss the implementation and validation of our approach to support user-centric service composition, the *A-DynamiCoS* framework. We have implemented the A-DynamiCoS framework prototype as an extension of the DynamiCoS prototype in order to deliver the required adaptable support. To validate A-DynamiCoS we define an evaluation strategy for user-centric service composition approaches that focus on two aspects: applicability and performance. We evaluate applicability by using A-DynamiCoS to support the service composition process of two different use cases, in different application domains and with different types of users to be supported. We evaluate the performance of A-DynamiCoS by measuring the time taken to process the primitive commands used to support users in the service composition processes of both use cases where A-DynamiCoS was applied.

The chapter is organised as follows: Section 8.1 presents the details of the A-DynamiCoS framework prototype implementation; Section 8.2 introduces the evaluation strategy followed to evaluate the A-DynamiCoS framework; Section 8.3 and Section 8.4 present the two use cases used to validate our user-centric service composition approach; Section 8.5 presents the performance evaluation of A-DynamiCoS framework; and Section 8.6 discusses our evaluation results.

8.1 Implementation

The *A-DynamiCoS* prototype was developed as an extension of the previously developed prototype of the *DynamiCoS* framework, dis-

cussed in Chapter 6. In this section we focus on the presentation of the new components added, namely the *Coordinator* component and the constructs developed to support the adaptability of the A-DynamiCoS framework, namely the *Primitive Commands* and *Supporting Strategies*.

8.1.1 A-DynamiCoS Overview

Figure 8-1 presents the overall architecture of our implementation, showing the application domain specific part and the generic part of the framework.

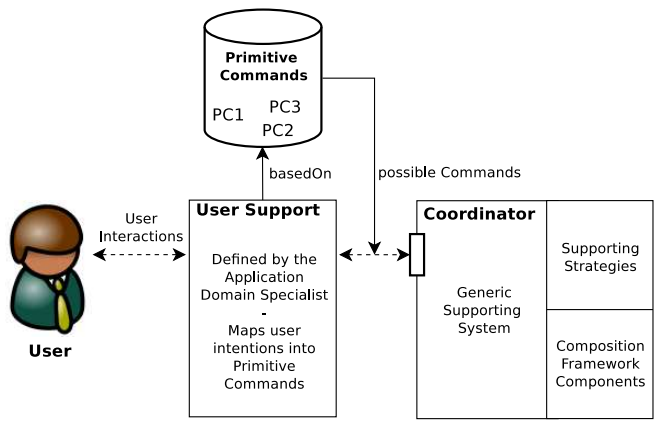


Figure 8-1 A-DynamiCoS prototype architecture

A-DynamiCoS domain specific elements, *front-end user support*, are developed by domain specialists. Domain specialists are able to develop the front-end user support, using the mechanisms and technologies more appropriate to support the target user population on the usage scenario being developed. Therefore, we do not specify a generic front-end user support for all the possible domains to be supported in the service composition process, since for each domain a specific and optimised front-end should be developed.

In contrast, the A-DynamiCoS *back-end supporting system* is defined as a generic system part, which is independent of the user support, and application domain specific parts that provide users with an interface to the service composition process. This allows the *A-DynamiCoS* back-end supporting system to support different front-ends, possibly defined by different domain experts, from different domains, with different types of users to be supported. To cope with this flexibility we expose the A-DynamiCoS back-end supporting system as a web service. This web service is stateful and can manage multiple user sessions, possibly from

different front-end user supports, in parallel.

The A-DynamiCoS back-end supporting system web service has been implemented in Java. The web service exposes one operation (*Coordinate()*) that has one input message with three parameters; namely *UserSessionID*, *CommandID* and *CommandParameters*. This web service has been developed in accordance with the *command design pattern* [111], in order to support the communication of multiple messages, representing different primitive commands, through the same interface on the back-end supporting system. Appendix D gives details about the A-DynamiCoS interface.

8.1.2 Primitive Commands

In the *A-DynamiCoS* prototype, primitive commands are defined using XML Schema Definitions (XSD) [116]. The primitive commands XSD define the request and response messages protocols. Based on these definitions, the front-end user support designers (domain specialists) can create the request messages for each primitive command used in the front-end user support. These definitions are also used in the A-DynamiCoS coordinator to generate the necessary functionality to handle the messages of each primitive commands, i.e., to process the received messages and then to create the response messages according to the primitive commands issued.

The definitions of primitive commands can be made available to interested parties, which can use them to design their user support strategy, namely to define the *command flow* to support their users. Furthermore, if other primitive commands are to be developed, the corresponding message definitions have also to be developed. Since our prototype was mainly developed in Java, it was easier to first define the XSD and derive automatically the code to unmarshal the primitive command request XML messages and marshal their response XML messages. We used tooling provided by the JAXB[1] project in order to generate this code.

8.1.3 Command Flows

Command flows are domain specific. They are defined according the requirements of the domain by combining primitive commands. Primitive commands have interdependencies that limit the possible combinations of primitive commands to create command flows. We specify these design rules as a *dependency graph*,

[1] http://jaxb.dev.java.net/

which the designers of front-end user supports should comply with when defining their command flows. Although the command types dependency graph imposes limitations for the definition of command flows, we do not enforce these limitations directly on the implementation on the front-end user support. The task of verification of correct definitions of command flows is delegated to the developer of the front-end user support. However, the back-end supporting system verifies the constraints imposed by the dependency graph each time a primitive command is received, guaranteeing the consistency of the command flow execution. In case the preconditions of a given requested primitive command are not met, the *A-DynamiCoS* coordinator does not allow the defined command flow to proceed, reporting that the requested command has violated the dependency graph rules. This information can be used by the user support developer to fix the command flow.

We did not develop any tool to validate command flows at design-time. However, it should not be difficult to build a tool that checks command flows, based on the rules specified in the dependency graph, allowing the user support developers to validate the specified command flows before deploying them.

8.1.4 Adaptive Coordinator

The *A-DynamiCoS back-end supporting system* was implemented as a single component, which is the *coordinator* component that implements the parts presented in Section 7.4. Figure 8-2 presents the elements of the coordinator.

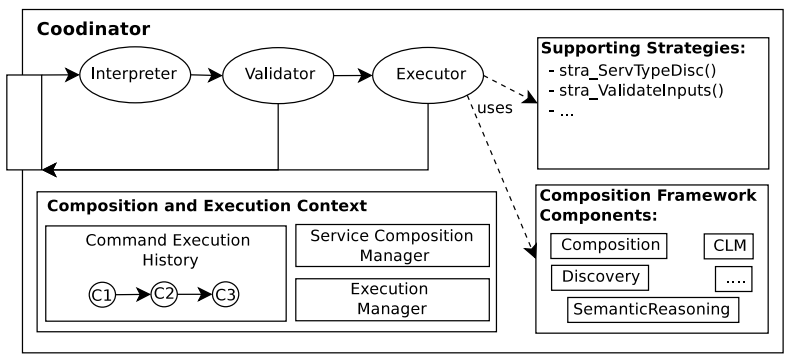


Figure 8-2 A-DynamiCoS coordinator implementation

The coordinator has been implemented in Java and exposed as web service. When a primitive command message is received, the coordinator *interprets* it and checks which user session the primitive command belongs to, using the message *UserSessionID* parameter. If a session for this user exists, it loads the user session,

the *Composition and Execution Context* instance, and proceeds to the *validation* of the primitive command. Otherwise, the coordinator creates a new user session. After that, the coordinator validates the requested primitive command against the *Commands Execution History* by using the command types dependency graph. Once the issued command is validated, the coordinator proceeds to the *execution* of the corresponding supporting strategy.

Supporting Strategies

Supporting strategies are implemented as Java methods of the Coordinator Class. The methods are called when a primitive command is received, immediately after its validation. All the supporting strategies share a common workflow of activities:

1. Unmarshal the *CommandParameters* from the primitive command request message, which contain the parameters of the primitive command;
2. Performs the supporting strategy, as a combination of calls to the DynamiCoS basic components;
3. Marshal the response message to be sent back to the front-end user support.

Composition Framework Components

The A-DynamiCoS composition framework components reuse the basic components defined in the DynamiCoS framework, Section 6.1. These components are implemented in Java and exposed as Java classes, which can be used by the different supporting strategies to implement the logic of the primitive commands.

Composition and Execution Context

The *composition and execution context* manager has been implemented as a list of objects that define the data structures that store the different elements of the user session: service compositions, discovered services, execution context (values), the user's context and the interaction execution history. Each time a new user session is initiated, a new composition and execution context session is initiated. The data structures that hold the different data handled in a user session is presented in Section 7.4.3.

8.2 Evaluation Strategy

In chapter 5 we discuss the problem of evaluation of semantic service composition approaches in detail, which was used to evaluate the DynamiCoS framework. We consider that the evaluation of a user-centric service composition approach has also to consider other dimensions, namely how the approach supports different users, who require different support in the service composition process. In this chapter we evaluate the A-DynamiCoS framework, which is an extension of the DynamiCoS framework. The evaluation mainly focuses on the measuring the suitability of approaches to support different application domains usage scenarios and users with different requirements and characteristics.

We consider that service composition platforms have a front-end and back-end supporting system. Our evaluation mainly concentrates on the back-end supporting system, which is the central focus of the research performed in this thesis. Furthermore, we also evaluate some time performance aspects of A-DynamiCoS, in order to verify whether the approach can be applied to support runtime service composition situations.

8.2.1 Applicability Evaluation

To measure the applicability, or suitability, of user-centric service composition approaches in different conditions, one has to define different use cases with different requirements. This is motivate by the fact that users are heterogeneous and different usage scenarios require different support on the service composition process.

To evaluate A-DynamiCoS in terms of applicability we have used the framework to support two different use cases of service composition support, in two different application domains, *e-government* and *entertainment* with different types of users. These use cases were studied and developed separately by two master students [117] [118] on their final graduation projects. The same A-DynamiCoS back-end supporting system was used to support both use cases. The domain specific parts, or the *front-end user support*, were developed by the students. The use cases are presented in Section 8.3 (*e-government*) and Section 8.4 (*entertainment*).

8.2.2 Performance Evaluation

User-centric service composition approaches are particularly suitable for runtime service composition situations, to support users

on getting personalised services as composition of existing services. To support runtime service composition approaches have to deliver *real-time responses* to the different supporting actions of the service composition process.

To capture this we measure the processing time of the different commands (*time-per-command*) used in the course of the experiments made on the two use cases to evaluate A-DynamiCoS. We mainly focus on evaluating the performance of the back-end supporting system, but the measured *time per command* counts the end-to-end time, from the moment the user issues a command in the front-end user support until it receives a response from the back-end supporting system. In Section 8.5 we present the performance evaluation of the A-DynamiCoS approach.

8.3 Use Case: E-Government

The e-government use case [117] defines a usage scenario that aims at facilitating the access of citizens to government electronic services. This specific usage scenario is expected to benefit from user-centric service composition approaches to attend specific requirements from citizens, given that different citizens, have different characteristics and requirements for the public services to be used.

In this use case we developed a web portal for accessing government services. This portal concentrates different public service from different departments of a government into one place where users can be assisted to make use of them. Users (citizens) normally have limited knowledge on the e-government application domain, and have limited information on the services that are available in the domain, since e-government is a large domain, with many services and bureaucratic processes. To cope with this, the supporting system has to assist users on discovering services and resolving preconditions of the services that they want to use. Furthermore, considering that users may have very limited technical skills they require very simple and intuitive interfaces to interact with the system, invoke services, communicate problems, request information from the system, or request assistance on resolving services' preconditions.

Because of the users characteristics mentioned above, we have defined the following workflow of activities to support citizens on the process of finding and using public electronic services:

1. A user specifies his *goal service* in a simplified interface, which

collects the type of service the user needs;
2. The system retrieves possible services that match the type of the user request;
3. A user gets information concerning the discovered services, which allows him to select a service for usage;
4. In case the user selects a service, he gets further information on the preconditions and/or inputs required for using the service. If the user does not know some of the preconditions or inputs, he may request assistance to resolve them. This assistance consists on finding services with outputs that provide the unfulfilled preconditions and/or inputs as output. This process can be repeated several times, while the user requires it. If no information or services are of assistance to the user to provide the unfulfilled preconditions and/or inputs, the user can be put in contact with a civil servant that can assist him to resolve the issues he is facing when using a particular public electronic service;
5. Once the user resolves all the preconditions and inputs of the goal service, he may make use of the service.

8.3.1 Command Flow

To develop a front-end user support that can deliver this workflow of activities in A-DynamiCoS, we have developed a command flow similar to the *objective-oriented flow*, discussed in Section 7.5.1. The objective-oriented flow defines a flow of commands that starts with the definition of *goal service(s)* of the service composition, and if necessary assists users to compose services backwards to resolve the goal service, or other services inputs that the user is not able to provide. This process recurs until all of the required inputs of the services of the composition are available. When all the service composition inputs are available, the service composition can be executed.

Figure 8-3 presents the command flow we defined to support this use case. This *command flow* uses the *primitive commands* introduced in Section 7.2.2 as follows:

1. *ServTypeDiscovery*: this primitive command instructs the back-end supporting system to discover services based on the service type(s) specified by the user. If no services are discovered, the users is prompted again with the interface to specify, or refine, which service typ(e) he wants, by being informed that no services were found that meet the issued service request. In case services are found, the user receives a list of discovered services;

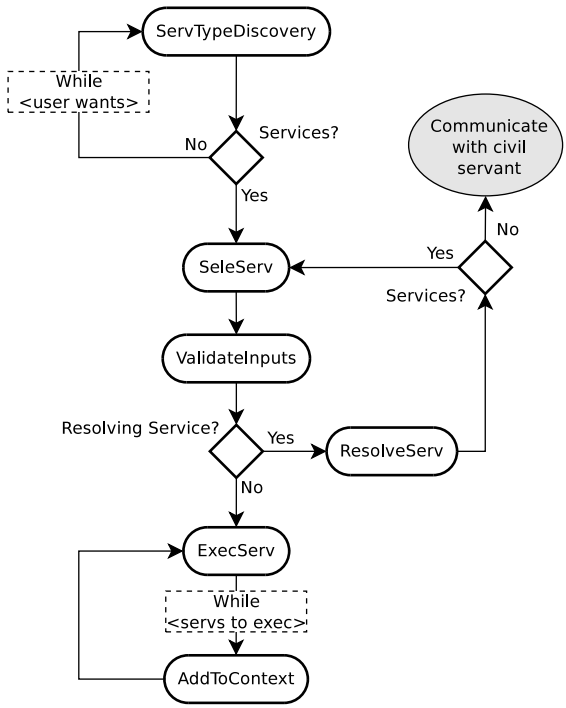


Figure 8-3 E-Government command flow

2. *SeleServ*: the user selects one or multiple services to be used, based on the information that describes the discovered services. The selected service(s) is the *goal service(s)*, i.e., the service(s) the user wants to use. The user selection triggers the *SeleServ* command, which informs the back-end supporting system that a specific service was selected by the user. At the back-end supporting system, the selected service is added to the service composition, which instantiates a new service composition;
3. *ValidateInputs*: after the user selects a service, he gets the preconditions and inputs require to use the service. At this point, the user can provide the necessary service inputs. In case the user is not able to fulfil some inputs or preconditions of the service, he can request further information about the preconditions or inputs or indicate them for getting assistance on resolving them. Then the *ValidateInptus* primitive command is issued. This command communicates the information entered by the user to the back-end supporting system. This information is stored at the back-end supporting system for later usage when the service is to be executed. The preconditions and inputs that are indicated for assistance are set as

unfulfilled, which means that they still need to be entered by the front-end user support or resolved by another service output. The unfulfilled inputs are reported back to the user as a response message of the *ValidateInputs* primitive command;

4. *ResolveServ*: if *unfulfilled inputs* are returned from the *ValidateInputs* command, the command flow proceeds with the *ResolveServ* command. This command discovers services that have outputs that semantically matches the unfulfilled preconditions or inputs. If services are found that match the unfulfilled preconditions or inputs, these are communicated to the user, which issues a *SeleServ* command to indicate which service is to be considered for resolving the unfulfilled precondition/input. In the back-end this *SeleServ* command triggers a composition between the service with unfulfilled preconditions/inputs and the newly discovered and selected service that provides an output, which corresponds to a backwards-chaining service composition. This composition is invisible to the user, since it takes place on the back-end supporting system. In case no services are found to resolve a given service precondition or input, the user is directed to a civil servant that may help him find the missing information. The process of resolving a service's preconditions and inputs recurs until all the services in the composition have their preconditions and inputs available, so that all the component services of the composition can be executed;
5. *ExecServ*: the execution of the component services is triggered by the *ExecServ* primitive command. This command allows to query the back-end supporting system for the next services in the service composition that are ready to be executed. A-DynamiCoS delegates the actual execution of the services of the service composition to the front-end user support, although the back-end still manages the process of which services are ready for execution, and the results (output information) of the services. Therefore, the front-end should be capable of calling the different component services that can be part of a service composition. To manage the execution of a service the back-end supporting system is queried using the *ExecServ*, which inspects the defined service composition, and retrieves the next service that can be executed, i.e., the services from the service composition that have all the precondition and input values available;
6. *AddToContext*: once a service is executed in the front-end user support, the resulting values are reported to the back-end

supporting system using the *AddToContext* primitive command. The *AddToContext* primitive command changes the status of the executed service to *executed*, and stores the service output results in the *composition and execution context* manager. After that, the *ExecServ* command is invoked again to request the next service, which can be a service that uses an output of the service that was executed in the previous iteration. The *ExecServ* primitive command is issued again until all the services of the service composition are executed.

8.3.2 Example

In the following we present an example of service composition support in the e-government domain. The example shows a possible procedure for a handicap citizen to order a parking place nearby his house. This procedure can be achieved by the following steps:

1. The user request a handicap parking place. Upon his request, he gets a list of services that may fulfil the service request. The user selects the *OrderHandicapParkingPlace* service, after which the user gets the service interface for using this service. The service requires the following information: *PermitPeriod*, *LicenseID*, *CarRegistrationStatus*, and *HandicapCardStatus*. We assume that the user only is able to provide the *PermitPeriod* and *LicenseID*, although he understands the other inputs. The user asks for help to the system for resolving the missing inputs;
2. The first input to be resolved from the *OrderHandicapParkingPlace* service is the *CarRegistrationStatus*. This input can be delivered by the *CarRegistrationValidation* service, which the user selects for usage. The *CarRegistrationValidation* service interface requires one input (*VehiclePlate*) which the user is able to provide;
3. The second input to be resolved from the *OrderHandicapParkingPlace* service is the *DriversLicenseStatus*. This input can be resolved by the *DriversLicenseValidation* service, which the user selects. The *DriversLicenseValidation* service interface requires two inputs: *LicenseID* and *AddressStatus*. For the *LicenseID* the user gets a suggestion, given that he has already entered it in the first step, in the *OrderHandicapParkingPlace* service. The user gets this information as suggestion since these two inputs are of the same semantic type (*IOTypes.owl#LicenseID*);
4. The user is not able to provide the other input of the *DriverLicenseValidation* service, the *AddressStatus*, so he requests

assistance to resolve it. This input can be resolved by the *AddressValidation* service, which the user selects. This service requires the user to enter his *Address* and *CitizenID*. The user is able to provide both inputs, i.e., the process proceeds with the resolution of the remaining unfulfilled inputs of the *OrderHandicapParkingPlace* service. To manage this process of resolving multiple services, the front-end user support keeps track of the services to be resolved in a FILO (*First In Last Out*) queue, which is used to manage the issued *ResolveServ* primitive commands. When the queue is empty, the process proceeds to the execution phase;

5. The last input to be resolved from the *OrderHandicapParkingPlace* service is the *HandicapCardStatus*. This input can be delivered by the *HandicapCardValidation* service, which the user selects. This service requires the user to input the *AddressStatus* and *CitizenID*. The *AddressStatus* is provided to the user as a suggestion, from the outputs of a previously selected service *AddressValidation*. The proposed suggestion is not yet the value of the *AddressValidation* service output, but rather a semantic type matching, since the *AddressValidation* service has not yet been executed at this stage. The *CitizenID* is provided as a suggestion, from the value the user entered in the first step of the process. At this moment, all the service composition services inputs are available, i.e., the service composition can be executed.

Figure 8-4 shows the service composition process described in the previous example.

Once the *goal service* (*OrderHandicapParkingPlace*) has all its preconditions/inputs available, and also all the services in the service composition created to resolve the *OrderHandicapParkingPlace* service, the execution phase can start. The execution follows the service composition graph, which defines the order of execution of the services in the composition. In this case, the order of service execution is:

1. {*CarRegistration validation* | *Address validation* | *CarRegistration validation*};
2. *HandicapCard validation*;
3. *OrderHandicapParking place*.

The service composition and execution flow presented above considers specific needs of a specific citizens, namely a citizen that lacks several certificates and information to make use of the *OrderHandicapParkingPlace* service. Other citizens requesting the same service may have completely different requirements. For ex-

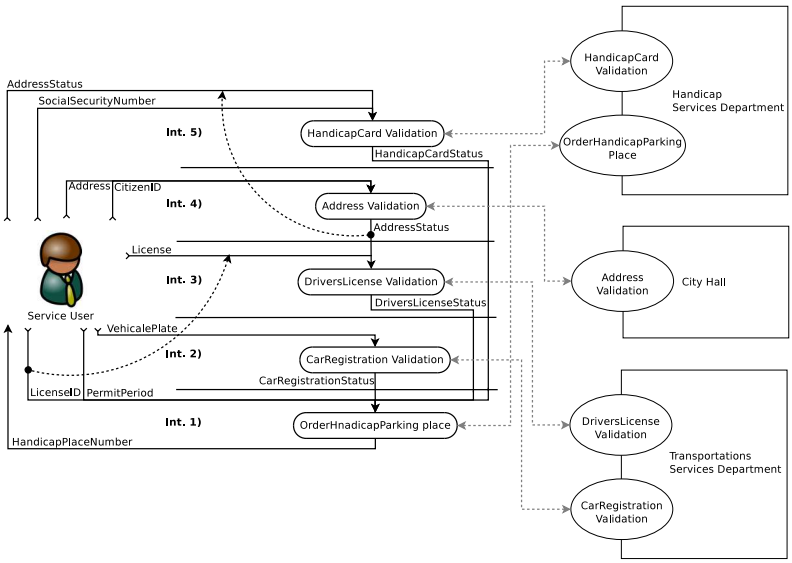


Figure 8-4 E-Government example

ample, someone that has all the certificates required to use the *OrderHandicapParking place* may use it directly, without requiring further assistance on getting them. This illustrates the need of user-centric service composition approaches as a mechanism to support different users to achieve their objectives.

8.3.3 Prototype

Figure 8-5 presents the architecture of the developed prototype, mainly showing the front-end user support part.

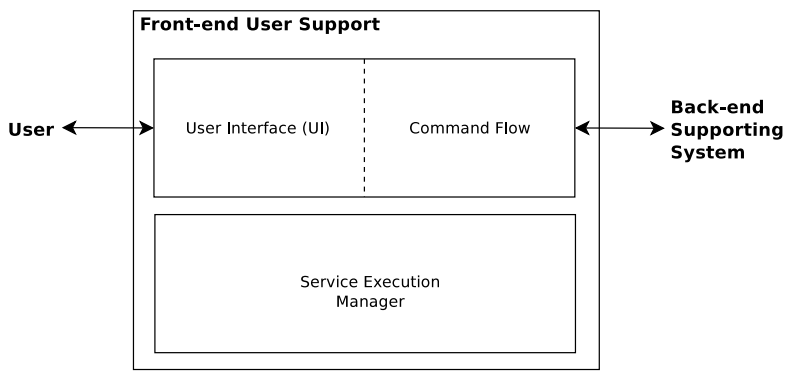


Figure 8-5 E-Government prototype

The user support developed in this use case consists of two main components: *User Interface/Command Flow* and *Service*

Execution Manager.

The *User Interface* implements the presentation functionality for the user in terms of a web interface and encodes the command flow shown in Figure 8-3. The user interface has four different *stages*, or screens:

1. *User Question*: the user specifies the service he wants (or *goal services*);
2. *Services and Selection*: matching services are retrieved and presented to the user. The user can select the service that fits better his requirements. The selected service is the *goal service*;
3. *Services Interface*: service preconditions and inputs are presented to the user in order to be filled in. In case the user is not able to provide some input or precondition the system assists the user on finding services to provide the unfulfilled inputs/preconditions;
4. *Results*: when all the services of the service composition are resolved, i.e., they have all its preconditions/inputs available, the service composition can be executed. At this stage each of the services is executed by using the *Service Execution Manager* component of the *front-end user support* to call the different services of the service composition, according to the information provided by the back-end supporting system on the next services to be executed.

The prototype web interface is shown in Figure 8-6. The prototype of the front-end for this use case was developed in PHP [119]. The prototype has scripts that generate the XML representation of the primitive commands. The command flow built in the front-end determines the dynamics of the implementation, so that commands can dynamically be generated according to the command requested by the user and its specific arguments.

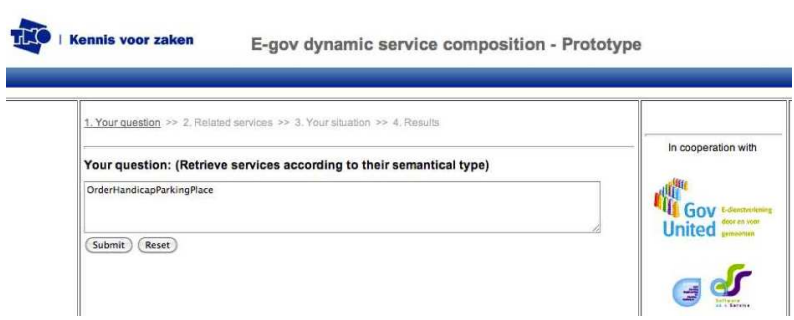


Figure 8-6 E-government user interface

8.3.4 Results

We evaluated our prototype by verifying whether the required user support, and its respective command flow could be supported by the back-end supporting system. Further evaluations were performed in [117]. From the performed evaluations we could can that the A-DynamiCoS back-end supporting system can support this use case.

8.4 Use Case: Entertainment

The Entertainment use case [118] aims at assisting users on planning leisure activities whenever they want or need. This use case can also benefit from user-centric service composition approaches because of the different requirements different users have at distinct moments when looking or planning leisure activities.

In this use case, users are presented with a web interface where multiple services can be used to plan different activities and also obtain diverse information concerning the activities and other related events. We assume that users have limited knowledge on the entertainment application domain and its services. We also assume that users cannot handle complex interfaces to support the service composition and execution process, since they may have limited technical skills. Furthermore, the devices used to carry the service composition and execution process, for example, mobile phones, also impose limitations on the type of interface. We also assume that users can have multiple *goals*, which can change overtime as users make use of services in the domain. For example, a user may use a service to look for cultural activities in a given location, and based on the results of the search, he can decide to go to a certain event and since the event finishes at dinner time, he can also decide, at that moment and based on the information he learned about the time and location of the activity, to book a table in a restaurant nearby the place where the activity takes place. This simple example shows that user goals may evolve overtime as the user makes use of services.

We have defined the following workflow of activities to support the usage scenario discussed above:

1. The user defines a *goal* to describe a given service he wants at a given moment, such as, for example find a restaurant in a certain place;
2. The system retrieves a list of possible services to satisfy this *goal*;

3. The user gets information about the discovered services, and selects one service based on information describing the service. This is the *goal service*;
4. Upon the selection of a service, the user gets the service interface to use the service. In case the user can enter all the preconditions/inputs he provides them and the service is set to be executed, otherwise he can request assistance to solve some preconditions/inputs that he does not know. The help on solving a given service input is provided by a service that has outputs that can deliver the unfulfilled precondition/input. Once the newly selected service is executed its results become available for the *goal service*. This represents a composition of two services. In case the user does not know some inputs of the second service, a new third service can be discovered to solve the second service unfulfilled inputs. This process recurs until the user is able to provide all the services inputs or there are no service to resolve a given unfulfilled input;
5. Once a user executes a given *goal service*, e.g.: find restaurant service, he gets the initial interface, where he can request other services. When new services are requested, the previously executed services information, and their inputs/outputs values, is all available in the user execution context, which makes it possible to reuse it for the execution of other services in the future.

8.4.1 Command Flow

To develop a front-end user support that can deliver this workflow of activities, we have implemented a *command flow* similar to the *Incremental Composition-Execution (ICE) flow* discussed in Section 7.5.2. The incremental Composition-Execution flow helps users to find, compose and execute services incrementally, as they are required. This is a *forward service composition approach*, in which at the beginning of the service composition process the user may not know, or may not want to define, all the services that he is going to use in the future. The decision of what service to use next may be taken on the fly as the user is making use of services. This justifies the interleaving between service composition and execution activities. The composition of a newly selected service with services previously executed facilitates the execution of the newly selected service by reusing information values (inputs and outputs) from previously executed services.

Figure 8-7 presents the command flow defined to support this

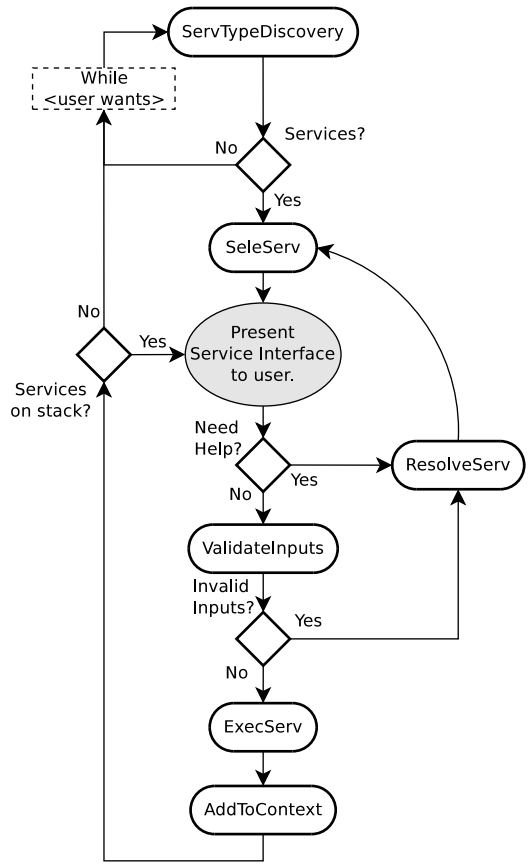


Figure 8-7 Entertainment command flow

use case. This *command flow* uses the *primitive commands* introduced in Section 7.2.2 as follows:

1. *ServTypeDiscovery*: when the user starts using the system to request assistance for getting a given service, he makes use of the service discovery facility, indicating the service he wants, or the *goal service*. This is translated into a *ServTypeDiscovery* primitive command, which instructs the backend supporting system to discover all the services that match the service type (semantic type) specified in the user request. This results in a list of services, which can be empty in case no services are discovered to match the user service request properties. If no services are returned, the user is redirected to the initial page, and the user is informed that no services were found;
2. *SeleServ*: if services are found, the user gets a list of services, and some information describing them. Based on this infor-

mation, the user decides which service better fits his specified needs, and selects one of these services. Once a service is selected, a *SeleServ* primitive command is issued to the back-end supporting system. This command adds the selected service to the service composition and the user composition and execution context in the back-end supporting system;

3. *Present Service Interface*: the service interface specifies the precondition and input parameters a service requires to be used. For each parameter of the interface, the user may get value suggestions. Value suggestions are taken from the user execution context, and may be from inputs and outputs of services previously executed, or from the user's context. The user selects suggestions or enters new values for each precondition/input of the service interface;
4. *ResolveServ*: if the user is not able to provide some of the service inputs he can request help to resolve them. This triggers the *ResolveServ* primitive command. This command retrieves all the services that can delivery an output that semantically match the type of the service precondition/input that the user is trying to resolve. The *ResolveServ* command offers several services. The user selects one service to resolve the precondition/input from the previously selected service, using the *SeleServ* primitive command;
5. *ValidateInputs*: for each service that is retrieved to resolve another service, the user gets the interface of the new service. The new service interface is removed whenever a service has all the required preconditions/inputs, and these are validated through the *ValidateInputs* primitive command. In case some precondition/input is unfulfilled or the user indicates that requires help on resolving a given precondition/input the command flow proceeds with the *ResolveServ* primitive command to find services to resolve the preconditions/inputs in question. This process recurs until all the preconditions/inputs of the composition services are available, or there are no services to provide the unfulfilled inputs;
6. *ExecServ*: when all the service preconditions/inputs of the service composition are available the command flow proceeds to the *ExecServ* primitive command. This command is used to query the back-end supporting system for the services that can be executed. The returned services are executed;
7. *AddToContext*: in the next step, the back-end supporting system receives the *AddToContext* command, which communicates that a service was executed. Furthermore, this com-

mand also communicates the results, or outputs, of the service, which is stored in the user session *composition and execution context*. These values enable the execution of the services that require such service outputs. The execution of services (step 6 - *ExecServ*) and the storing of executed service results (step 7 - *AddToContext*) recur until all the services of the composition have been executed;

8. *Initial Supporting Interface*: after executing all service composition services, namely the *goal service*, the user returns to the initial interface, which allows him to request new services. The services that are requested afterwards, can use the values from the *composition and execution context*, which means that if a service precondition/inputs is semantically related to values from the user composition and execution context, such values are presented as suggestions for the service preconditions/inputs. This facilitates the reuse of information, relieving the user from keeping track of all the information handled in the whole service composition and execution process and from re-entering repeatedly information.

8.4.2 Example

In the following we present an example of service composition support in the entertainment domain. The example shows a possible procedure for a user to plan a set of activities to arrange a weekend visit to Amsterdam. This procedure can be performed in the following iterations, where in each iteration at least one service is executed:

1. The user wants to find a hotel in Amsterdam. He queries the system for services to find hotels in Amsterdam. The user is presented with the *Kayak* service, which allows users to find Hotels in the Amsterdam area. The user is presented with this service interface, which enquires the user for: *City*, *Checkin_ date*, *Checkout_ date*, *Guests* and *Rooms*. The user is able to provide all this information, which means that the service can be executed. The result of the service is a list of hotels that match the user query for the hotel he is looking for. The user can then select one of the matching results, which gives the user more detailed information about the hotel, namely, *Address* and *City*. This information is stored in the user execution context;
2. Once the user has selected a hotel, he decides to look for cultural activities taking place in Amsterdam in the period of his stay, so that he can attend one of these activities. As a

result, the user gets the *Ikdoe* service, which provides information about leisure and cultural activities in the Netherlands. To use the Ikdoe service, the user has to provide the following information: *Keyword, Category, Start_ date, End_ date, City_ name*. Given that the user has already used other services, and entered some information of the same semantic type he gets suggestions for some of the Ikdoe service inputs, namely for the *Start_ date*, *End_ date* and *City_ name*. The user then selects the dates and city (Amsterdam) where he is going to spend the weekend. The user is able to enter the remaining service inputs, which allows to execute the service. The service provides the user with a list of activities in Amsterdam in the specified period. The user selects one or more activities to get more detailed information about, namely: *ActivityType*, *Location* and *Date*. The selected activities are stored in the user's execution context;

3. The user found some interesting activities in the city centre, and since they take place at night he decides that it may also be interesting to look for a restaurant in the city centre. To find restaurants, the user is presented with two services: *Iens* and *Seatme*. He selects the *Seatme* service, which provides information about restaurants in Amsterdam. The user gets the service interface, which requires the following inputs: *City* and *Kitchen*. Provided that the user has already entered the city before (Amsterdam), and this is stored in his composition and execution context, he gets Amsterdam as a suggestion for the city input. The user inputs the type of kitchen he wants, which completes the necessary information to execute the service. The service is executed and the user is then presented with a list of restaurants, from which he selects one. Based on the user selection he gets further information about the restaurant, namely its *address*. Information on the selected restaurant is saved in the user execution context information;
4. Before travelling to Amsterdam the user also decides to check the weather forecast in Amsterdam. To check the weather forecast the user gets the *Buienradar* service, which provides weather forecast in the Netherlands. To use this service the user has to input the location for which he wants to check the weather forecast. This input is of semantic type *Coordinates*, which the user is not able to provide, so that the user asks help to the system to resolve this input. This triggers the use of the *ResolveServ* primitive command, which retrieves the

Google Geocode service. This service provides the coordinates of a given place based its address. The user gets two suggestions for address, namely the address of the hotel and the restaurant. He selects the hotel location, and the two services are executed, first the *Geocode* then the *Buiendradar*, which returns the weather forecast information;

5. Finally, the user requests a service to get the directions from his home to the hotel in Amsterdam. He gets the *Google Directions* service, which requires the source address and the destination address. The user gets suggestions for both addresses, however he only selects the address of the destination, entering his home address in the service's source address. As a result, the user gets a textual description of the route he has to follow to reach the hotel in Amsterdam from his home.

Figure 8-8 shows the service composition process described in the previous example.

This example demonstrates that the service composition is incrementally created, and at each iteration the service that is added to the composition is also executed. In case the user is not able to provide some input information, he asks for help to resolve the input, as in the example of the *Buienradar* service execution, in which the *Geocode* service was used to resolve its input, *Coordinates*.

All the services presented in this example are real web services and were invoked in the course of our experiments.

8.4.3 Prototype

To develop the user support prototype we have followed a MVC (Model-View-Controller) design pattern [120]. In the *model* we store some information about the user session, in the *view* we define the user interface and in the *controller* we encode the command flow designed to support this use case.

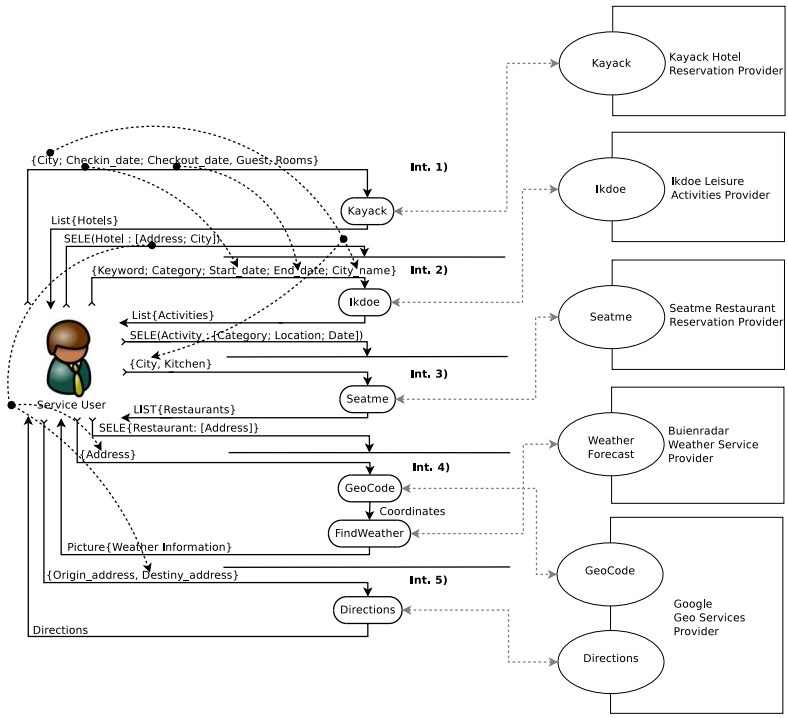
Figure 8-9 presents the architecture of the developed prototype.

To define the user support, in addition to the MVC components, we have also defined a component to handle service execution namely the *Service Execution Manager*.

The user interface on the user support component is divided into different screens:

1. *Initial Screen*: presents all the service types that exist in the domain using intuitive graphical icons. Users can select a service type and also see their execution context, namely which services were already executed;
2. *Service Selection*: when the user requests a service type, he

Figure 8-8 Entertainment example



gets a list of services of that type. The user can select a service from this list, and this service is considered for execution;

3. *Service Interface*: the user gets the interface of the selected service, i.e., the service preconditions and inputs. The user can enter the required information to use the service. Furthermore, the user can also request assistance to resolve unfulfilled unknown preconditions/inputs. Such request for assistance provides the user with new services (for selection) that can provide the precondition/input in question;
4. *Results*: once the user can provide all the service preconditions and inputs, the services is set for execution. After the execution, the user gets the service results which he can select in case multiple alternative results are presented.

Figure 8-10 shows a screenshot of the initial screen of our prototype. On the top part of the screen the information about the services already used, and composed, is shown. In this concrete case the user has made use of a service of the type "Reserve restaurant", to reserve a place in the "Casa Brazuca"; another service of type "Find activity" to find the "what is in your mind?!" activity; and one of type "Book hotel" to book a hotel room in the "Renais-

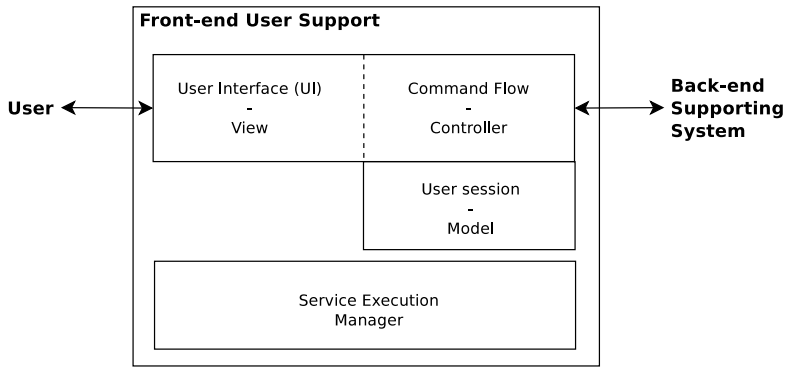


Figure 8-9 Entertainment prototype

sance Amsterdam Hotel". All this information is kept in the user session context.

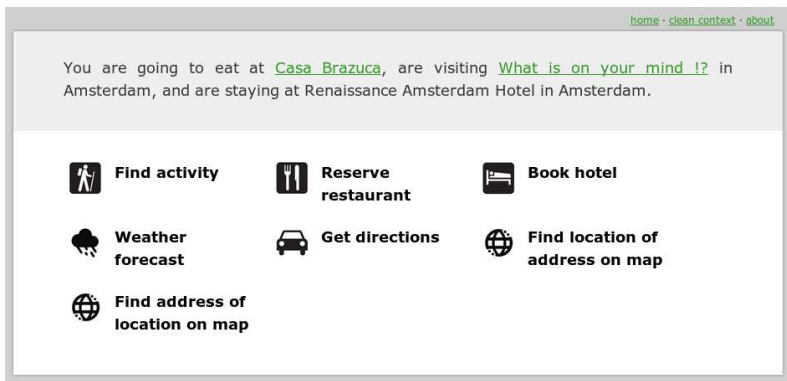


Figure 8-10 Entertainment initial screen user interface

The prototype for this use case was developed using Ruby on Rails [121].

8.4.4 Results

The developed prototype and respective supporting command flow was supported by the A-DynamiCoS back-end supporting system. This was observed in several tests, including the one presented in the previous section. Other experiments were performed and reported in [118].

Although, the focus of our research has been on the back-end supporting system, in this use case we have performed some usability tests. Users were asked to make use of the developed system to plan some leisure activities. We observed that several features of the user interface were not familiar to the users. For

example, the notion of "resolving an input" that the user does not know, which was indicated with a question mark in front of a service input, was not clear. Furthermore, users could not understand why a new service interface was opened when they made use of the question mark. These results show that users require training on the user-centric service composition environment proposed to be able to use of it or further modifications of the interface have to be made to provide users with more familiar user interface constructors (or interaction points) to handle the different primitive commands used to drive the service composition process. Furthermore, we have developed simplified client web interfaces for the services considered in the usage scenario. These simplifications make the client interface less intuitive for the users than the original and dedicated web services interface, on the services website. Further research on how to integrate this type of back-end supporting system functionality into more intuitive interfaces, or possibly the services original website, is required. This may imply that instead of defining a central place where the user performs the composition process, the service composition process can be addressed in a peer-to-peer fashion, where the user navigates through the different websites of the services that he needs, while the back-end supporting system keeps track of the service composition and the execution context. However, this implies that each service provider has to perform the extra effort of including the integration with the A-DynamiCoS back-end supporting system.

8.5 Performance Evaluation

The A-DynamiCoS performance evaluation concentrates on measuring the processing times taken to support the different primitive commands used in the evaluation use cases presented in the previous sections.

Bellow we present the primitive commands used in the command flows of the our two use cases. We executed the different presented primitive commands, sending them from the front-end user support to the back-end supporting system. In these experiments the client and server machine (containing the A-DynamiCoS web service) were in our department. During the experiments the A-DynamiCoS contained the services (Appendix C) defined for both evaluation use cases of A-DynamiCoS. The performed experiments were repeated and we present the average values of the measured

time per command, for each of the presented commands. The variance of the measured time per commands in the different experiments was very small, always lower than 5%.

Data

Table 8-1 and Table 8-2 show, respectively, the primitive commands, in order of appearance, and processing times of the commands used in the e-government and entertainment use case's examples presented in the previous sections. Figure 8-11 and Figure 8-12 plot the processing times in terms of the different issued primitive commands.

Table 8-1 Average *time-per-command* for e-government use case example

Ite.	Primitive Command	T(ms)
1	*ServTypeDiscovery(orderHandicapParkingPlace)*	1452
2	*SeleServ(OrderHandicapPlace)*	880
3	*ValidateInputs(OrderParkingPlace)*	118
4	*ResolveServ(CarRegistrationStatus))*	400
5	*SeleServ(CarRegistrationValidation)*	703
6	*ValidateInputs(CarRegistrationStatus)*	112
7	*ResolveServ(DriversLicenseStatus)*	342
8	*SeleServ(DriversLicenseValidation)*	771
9	*ValidateInputs(DriversLicenseValidation)*	111
10	*ResolveServ(AddressStatus)*	320
11	*SeleServ(AddressValidation)*	888
12	*ValidateInputs(AddressValidation)*	111
13	*ResolveServ(HandicapCardStatus)*	254
14	*SeleServ(HandicapCardValidation)*	912
15	*ValidateInputs(HandicapCardValidation)*	111
16	*ExecServ*	130
17	*AddToContext(HandicapCardValidation / AddressValidation / CarRegistrationValidation)*	111
18	*ExecServ*	122
19	*AddToContext(DriversLicenseValidation)*	112
20	*ExecServ*	143
19	*AddToContext(OrderParkingPlace)*	114

Analysis

The higher processing times were observed for the *ServTypeDiscovery* and the *SeleServ* primitive commands. These primitive commands perform several semantic reasoning operations. The *ServTypeDiscovery* finds all the services that have types that are semantically related with the requested type. The *SeleServ* adds a service to the CLM and to the service composition graph, which

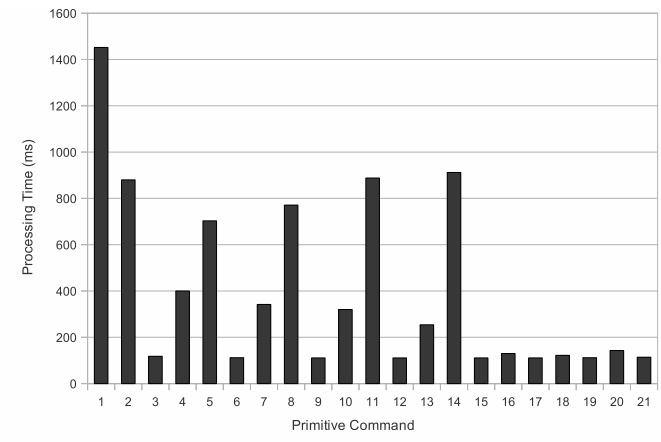


Figure 8-11 E-government primitive commands processing time

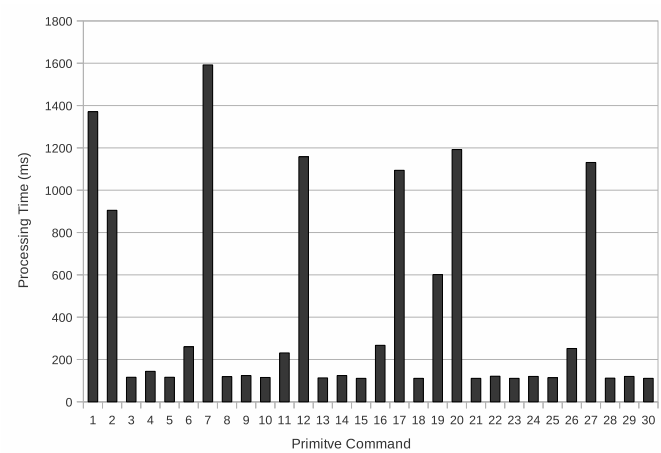


Figure 8-12 Entertainment primitive commands processing times

requires several semantic reasoning operations for updating the CLM. These semantic reasoning operations require intensive processing. The first primitive command, *ServTypeDiscovery* primitive command, takes longer than the other primitive commands of the same type issued later because the first primitive command has to open a new user session, which requires the instantiation of several data structures. The other primitive commands presented relatively low processing times, in the order of 100 to 300 ms, which represents a very quick response, i.e., the delivery of real time responses to the issued commands. We also observed that the user execution context information did not influence considerably the processing time of the different commands. This can be ob-

Table 8-2 Average *time-per-command* for entertainment use case example

Ite.	Primitive Command	T(ms)
1	*ServTypeDiscovery(FindHotel)*	1371
2	*SeleServ(Kayack)*	905
3	*ValidateInputs(Kayack)*	116
4	*ExecServ*	144
5	*AddToContext(Kayack)*	116
6	*ServTypeDiscovery(FindActivities)*	260
7	*SeleServ(Ikdoe)*	1592
8	*ValidateInputs(Ikdoe)*	119
9	*ExecServ*	124
10	*AddToContext(Ikdoe)*	115
11	*ServTypeDiscovery(FindRestaurant)*	231
12	*SeleServ(Seatme)*	1158
13	*ValidateInputs(Seatme)*	113
14	*ExecServ*	124
15	*AddToContext(Seatme)*	111
16	*ServTypeDiscovery(FindWeatherForecast)*	267
17	*SeleServ(FindWeather)*	1094
18	*ValidateInputs(FindWeather)*	111
19	*ResolveServ(Coordinates)*	601
20	*SeleServ(GeoCode)*	1192
21	*ValidateInputs(Geocode)*	111
22	*ExecServ*	121
23	*AddToContext(GeoCode)*	111
24	*ExecServ*	120
25	*AddToContext(FindWeather)*	114
26	*ServTypeDiscovery(FindDirections)*	252
27	*SeleServ(Directions)*	1131
28	*ValidateInputs(Directions)*	112
29	*ExecServ*	120
30	*AddToContext(Directions)*	111

served on the *ExecServ* primitive command, which basically was constant for the different times that this command was issued, although more information was being handled as new *ExecServ* commands were issued. This happens because simple queries are made to the user execution context, basically requesting the value of the service inputs of a service to be executed next.

8.6 Discussion

In this chapter we present the implementation and validation of our framework for user-centric service composition, A-DynamiCoS.

To evaluate our framework we have focused on two main di-

mensions: *applicability* and *performance*. The first considered the application of A-DynamiCoS to support different use cases, in different application domains with different types of users. The second concentrated on evaluating the time performance of A-DynamiCoS to support the different primitive commands required in the different experiments performed during the A-DynamiCoS evaluation.

From the developed use cases (e-government and entertainment) we can conclude that A-DynamiCoS allows to support different usage scenarios, or front-end user supports, based on the same generic back-end supporting system. This conclusion is taken from the fact that the two use cases were developed using the same set of primitive commands, although used in different orders and command flows, and supported by the same back-end supporting system. Such results demonstrate the flexibility that A-DynamiCoS offers to support primitive commands on demand, when they are required. The only limitations, or constraints, imposed on the order of the commands issued is imposed by the *command types dependency graph*. Although A-DynamiCoS offers support to both use cases seamlessly, we could observe that some of the functionality provided by the primitive commands defined in the user interfaces, or the front-end user support, were not very intuitive for the users driving the service composition process. Further research is needed to improve this integration and usability.

From the measured time-per-command we observed that most of the primitive commands are processed in the order of few hundreds of milliseconds. A-DynamiCoS provided real time responses to the commands issued in the evaluation use cases, which is of high importance for user-centric service composition approaches, as they are specially interesting approaches for runtime service composition, where real time response is very important. Further experiments with higher number of users and services may need to be conducted in order to evaluate further the scalability of the framework.

With A-DynamiCoS we manage to overcome some of the problems identified when we have used a fully automatic service composition approach (DynamiCoS framework), namely lowering processing times and better personalisation of the service composition process. The lower processing times are related with the fact that in A-DynamiCoS the service composition process is created by using different primitive commands, which divides the processing time into different parts. This improves the response to the users

driving the service composition process, since they do not feel longer delays on the processing of their commands. Furthermore, since the process of service composition consists of multiple flows of commands the user issues the commands he requires on demand, whenever required, which makes the service compositions more personalised, when compared with the ones proposed by the DynamiCoS framework.

Chapter 9

Conclusions and Future Work

This chapter discusses the conclusions and contributions of this thesis. Furthermore, it also discusses directions of future work in the area of user-centric service composition.

This chapter is organised as follows: sections 9.1 presents a general discussion on the work performed in this thesis; Section 9.2 shows the most important research contributions of this thesis; and Section 9.3 discusses further directions of work.

9.1 General Discussion

Nowadays people are increasingly using network-based services in different situations with a variety of computing devices. Moreover, they often make use of a specific combination of services to address a given set of requirements they have. The process of finding the right services and combine them in the right way is complex and requires users to manage different information themselves, while using the services independently of each other. Service composition approaches allow to overcome some of these problems. Service composition can shield the users from many technical details and free them from performing tedious and error-prone tasks such as handling information between the execution of the different required services. Service compositions are generally created at design-time to satisfy a given set of requirements of a target user population. However, users are heterogeneous, have different characteristics and requirements, which means that normally there is a gap between the provided service composition, designed for a general user population, and the concrete requirements of a specific user. To achieve better personalisation of the service delivery, through service composition, we claim that the specific requirements of the service end-user have to be taken into account to define the service composition. To address this, the composi-

tion has to be created when the user requires it, possibly at runtime. To support such on demand service composition process we cannot assume that a professional service developer handles the service composition process on behalf of the end-user. Therefore, the user must have a more active role in the service composition process, with appropriated automated support to assist him in the different steps of the service composition process. We define this service composition process as *user-centric service composition*.

This thesis focuses on the design of systems to support user-centric service composition. Towards the design of such systems, we first studied how users, possibly with limited technical knowledge, can play an active role in the service composition process, without the mediation of professional service developers (Chapter 3). Several issues were identified related to the fact that users are heterogeneous with respect to their functional requirements for the services to be composed and their technical skills to drive the service composition process. Such requirements and characteristics demand for different types of service composition supporting systems. We mapped these requirements into two general design issues for user-centric service composition supporting systems, namely *automation* and *adaptability*. These two design issues define the two main research questions posed in this thesis, which we address next.

– **RQ1:** *What mechanisms are required to automate the service composition process so that non-professional users can drive the service composition process, possibly at runtime?*

To address the automation of the service composition process we have developed a framework to support dynamic composition of services, the DynamiCoS framework (Chapter 4). DynamiCoS is composed of several components, which automate the different service composition life-cycle phases. Such automation enables runtime service composition support, which allows to drive the service composition process according to specific requirements of end-users at a given moment. Automatic service composition also enables to shield of users from the service composition technical details. This is a very important aspects, since end-users may not have the required technical knowledge to handle all the details of the service composition process. Although shielded from the service composition details, the (end-)users have to play a more active role in the service composition process, specifying the different requirements for the service composition. To automate the service composition life-cycle phases, the DynamiCoS framework

uses semantic services and ontologies. Semantic information enabled us to define automated service discovery mechanisms and an algorithm for automatic service composition. We have developed a prototype of the DynamiCoS framework (Chapter 6) to evaluate its application to support dynamic service composition processes and its scalability under different circumstances. The prototype has shown that the developed framework allows to support automatically the different phases of the service composition process. The automatic service composition process scales, when small number of semantic services are handled in the service composition process. However we have observed that when large sets of semantic services are manipulated in the service composition process the processing time can increase significantly, which may compromise the real time response requirements system.

– **RQ2**: *How to support different types of users, i.e., users with different characteristics, such as, for example, application domain knowledge, services knowledge and technical skills?*

DynamiCoS defines a rigid workflow of activities, supported by its basic components, to automatically support the service composition life-cycle phases. This workflow of activities is the same for all the users being supported by the framework. DynamiCoS assumes that users can provide a detailed declarative specification of the requested service parameters (IOPE, goals, non-functional properties), which then are used in the supporting workflow in the following order: 1) discovering candidate services that fulfil each of the service request parameters; 2) automatically compose the discovered services in order to create a composite service that fulfils all the requirements the user specified. This workflow may be appropriate for some types of users, specially users that are familiar with the application domain and its services. However, it is not suitable for users that do not have such familiarity with the application domain where they are seeking services or for users that do not have a clear specification of all the service requirements. To overcome this, we have extended the DynamiCoS framework with a flexible coordination. This extension, the A-DynamiCoS framework (Chapter 7), allows to make use of the DynamiCoS basic components in a flexible manner, as they are required by the user being supported in the service composition process, i.e., a user-centric service composition process. To gather the user intentions at each step of the service composition process we have defined a new component in the framework, the *front-end user support* component. This component is developed by domain ex-

perts, who know the target user population that is to be supported in a given usage scenario. The front-end user support is defined based on the use of *primitive commands*, which allow to translate user intentions, at each step of the service composition process, into supporting strategies realised by the basic components of the composition framework, in the back-end supporting system. The front-end user support can be designed in different fashions, with different user interfaces, as long as in the background the primitive commands are used to communicate the user intentions at each step of the service composition process. To validate our user-centric service composition approach we have applied it to support two different use cases (Chapter 8), from two different application domains (*e-government* and *entertainment*) with different types of users. These use cases had different front-end user support components and command flows, according to the needs of their target user populations and to the usage scenario to be supported. The same back-end supporting system was used to support the two use cases, consisting of the same set of primitive commands issued in different orders and command flows. From this study we could show the applicability of our user-centric service composition approach to support users with different requirements and characteristics.

9.2 Research Contribution

Our research contributes to the area of user-centric service composition in two directions:

1. *Dynamic Service Composition Support*, by defining a framework to support the automation of the service composition life-cycle phases;
2. *User-centric Service Composition Support*, by defining a framework that provides a flexible and adaptable service composition support, as function of the characteristics and requirements of the user driving the service composition process.

In the sequel we discuss these contributions in detail.

9.2.1 Dynamic Service Composition Support

Automatic Service Composition

We have defined the DynamiCoS framework [88] [122] (Chapter 4) which defines different components to automate the different phases of the service composition life-cycle [23]. The framework

provides support for the publication of basic component services to be used in the service composition process. Published services are represented in the framework in a language-neutral formalisms, which allows developers of basic services to describe their services in different existing languages, as long as the description languages have an interpreter in the DynamiCoS publication mechanism. Furthermore, and since DynamiCoS uses ontologies to semantically describe the services, services have to be semantically annotated with the framework domain ontologies. Such domain ontologies are the key to enable automatic service discovery and composition. We have developed an automatic service composition algorithm [92], which composes services in order to fulfil all the requirements specified by the user in the service request. The service request is a declarative specification (of the parameters) of the desired service.

Evaluation of Semantic Service Composition

We have developed an evaluation framework for semantic service composition approaches [123] (Chapter 5). A major problem for the evaluation of semantic service composition approaches is the lack of large and suitable real world semantic service collections to be used in the evaluation process. We propose the creation of semantic service collections as a combination of manually defined, and meaningful, semantic services with large sets of automatically generated services. Several metrics were proposed for the evaluation, namely *confusion matrix*-based metrics, which measure the quality of the service compositions proposed by service composition approaches, and time-based metrics, which measure the scalability of the approaches when the number of services in the registry is varied.

9.2.2 User-centric Service Composition Support

User Characterisation

To define supporting systems for user-centric service composition we started with the premise that users are heterogeneous, which means that they require different support during the service composition process. We propose a user characterisation and classification based on the user knowledge, namely *application domain knowledge*, *services knowledge* and *technical knowledge* [83] (Chapter 3). By studying the users according to these dimensions of knowledge, one can characterise the target user population of a given usage scenario that makes use of service composition. This

allows one to identify and define a support appropriate to the target user population. The main objective is to define a support system that can provide mechanisms to complement the knowledge users are lacking during the service composition process.

User-centric Service Composition Framework

To support user-centric service composition we have developed the A-DynamiCoS framework (Chapter 7). This framework aims at supporting the service composition process in a flexible manner, as a function of the requirements and characteristics of the user driving the service composition process. To provide such a flexible support we have defined a coordinator that invokes the basic composition framework components as function of the user requirements at each step of the service composition process. The execution workflow of supporting activities is not rigid, as in the DynamiCoS framework, it is defined on demand according to the commands issued by the users in the *front-end user support*. The front-end user support is developed by domain specialists, which define the appropriate the appropriate interface and *command flows* to support the target user population of the usage scenario. The command flows define workflows of primitive commands, which may consist of multiple flows, in order to provide suitable support to the different users being supported. The back-end supporting system is domain-independent, which means that it can support different front-end user supports, possibly defined to support different usage scenarios, from different application domains.

Incremental Service Composition-Execution

User-centric service composition is a dynamic process, in which service compositions are defined in an incremental manner by the users driving the service composition process (Chapters 3, 7). This incremental process is motivated by fact that users require multiple interactions with the supporting system to command the service composition process, and to acquire the necessary knowledge in order to proceed in the service composition process. Many existing service composition approaches consider a service composition life-cycle where service compositions are first defined, by a professional service developer, and then deployed so that end-users can make use of it. We claim that user-centric service composition processes, the composition and execution phases may need to be interleaved, multiple times, in order to allow users to achieve their

very specific needs. Services, or partially defined service compositions, may need to be executed to allow users to learn about the domain and increase their knowledge. Based on such increase of knowledge, the user may take further decisions on how to proceed with the service composition process. We have presented different *command flows* to support such processes, for example the *Incremental Compose-Execute* command flow and *Objective-oriented* command flow (Section 7.5).

9.3 Directions of Further Research

Service composition is a very active area of research (Chapter 2). However, the topic of user-centric service composition is a relatively unexplored area of research. So far, service composition research mainly concentrated on the functional aspects of the service composition process, namely how to optimise the service discovery and composition processes. Future research should also consider a more active participation of the (end-)user in the service composition process, and how service composition supporting systems can cope with users with different characteristics and requirements. In the following we present some possible directions of future work:

- *Composition-Execution Interleave*;
- *Primitive Commands and Supporting Strategies*;
- *User Context*;
- *Optimisation of Service Discovery and Composition*;
- *Semantic Services*;
- *Evaluation Methodologies*;
- *Human-Computer Interaction.*

Composition-Execution Interleave

In our research we have identified that user-centric service composition is a dynamic process, where service compositions are defined in an incremental fashion. The *incremental compose-execute* command flow is an example of such a process. This command flow assumes that users compose new discovered services with an existing service composition from which all the services were already executed. The result of this new service composition is executed, and then the user may decide to proceed further in the service composition process.

Further research is required, mainly to support the execution of partially defined service compositions. In our research we dele-

gate service execution process to the front-end user support component. The developer of the front-end user support component has to define the mechanisms required to support the execution of services of a given service composition. They are executed one-by-one, although the management of this process is handled by the back-end supporting system. Alternatively, generic clients could be developed, which, based on the service implementation protocol (SOAP [124], REST [114], etc.), would be deployed in the back-end supporting system to handle the service composition process. This development would simplify the front-end users support component, facilitating the integration of the back-end supporting system in different usage scenarios, with different front-end user supports, which can use services implemented in different technologies and using different implementation protocols.

Primitive Commands and Supporting Strategies

To support flexible coordination and support users according to their requirements and characteristics we have defined two basic elements in our framework for user-centric service composition: *primitive commands* and *supporting strategies*. *Primitive commands* define the basic elements that can be used to define front-end user support *command flows* and manage their interaction with the back-end supporting system. These commands trigger the execution of *supporting strategies*, which implement a given behaviour by invoking the basic components of the composition framework. These elements provide the flexible mechanisms that enable gathering the user intentions to drive the service composition process, as the user requires, i.e., in a user-centric fashion.

Supporting strategies are always defined as function of the basic *composition framework components* and the *composition and execution manager* data structures. The basic components perform the different computations associated with the service composition process, and the data structures allow to store the information handled in the service composition process. Considering this, the development of primitive commands and respective supporting strategies could be performed without requiring the designers to have extensive knowledge on the internals of the A-DynamiCoS back-end supporting system. Such developments would facilitate the development of new primitive commands corresponding to new actions in the front-end user support.

User Context

The user context information may play an important role in situations where users discover, compose and execute services on the fly. This information may be captured and beneficially used in the service composition and execution processes.

In our research we have considered that the user context information can be stored in the user session *composition and execution manager*. The context information stored is annotated with a given semantic type, as a reference to a semantic concept from a framework domain ontology. Whenever a selected service has an input of a semantic type related to context information values stored in the user session, the matching values are suggested to the user. In our approach we suggest all the semantically matching values. However, optimisations may be made by reasoning on the user situation. For example if the user is looking for a restaurant and he is in the centre of a city, and many location values are stored in the user context, the most probable location is his current location. Such user context models and reasoning on the user context will allow to provide users with more meaningful suggestions.

Optimisation of service discovery and composition

The process of service discovery and composition can be optimised based on non-functional properties, user preferences and user context information. This information can be handled in the back-end supporting system in order to rank the found services and service compositions. For example, if the user specifies that he wants always the cheapest services, such non-functional property can be used to rank the discovered services and generated service compositions. Furthermore, research also needs to be conducted on multi-objective functions optimisation. For example, users may specify several non-functional properties, and have several preferences and context information, which can be used for the optimisation of the service discovery and composition processes. These developments will allow further personalisation of the service delivery process, complementing the functional approach proposed in this thesis.

Semantic Services

We use semantic services in the different phases of the service composition process. The ontologies referred to in the semantic service descriptions represent domain conceptualisations. Formal-

isation of conceptualisations are required to enable automation in the discovery and composition processes.

However, semantic services are not yet widely employed. The task of semantically describing services is still expensive and requires experts to define ontologies beforehand. Research on creating mechanisms to simplify the definition of ontologies and annotation of semantic services is needed. This will enable the widespread use of semantic service composition approaches. We claim that such approaches are a key factor for the creation of user-centric service composition support.

Evaluation Methodologies

In this thesis we have introduced a framework for the evaluation of semantic service composition approaches. Nowadays the existing collections of semantic services are small and do not possess the required properties for the evaluation of semantic service composition approaches. To address this, we have developed a collection of semantic services, where some services are manually defined and have clear semantics, but most of the services are automatically generated, without clear semantics. Although this service collection allows to test different aspects of semantic service composition approaches, we consider that the use of semantic service collections completely based on real world services will allow to perform a more comprehensive evaluations.

To evaluate user-centric service composition approaches we have mainly focused on measuring the applicability and scalability of our approach. These evaluations mainly concentrate on the back-end supporting system, which is the focus of this thesis. To complement these evaluations, empirical evaluations and usability tests with users should also be performed in order to measure the impact of such supporting systems on the users being supported on the service composition process. These evaluations may help on improving further the design of user-centric supporting systems, specially to define suitable user interfaces that make use of the mechanisms defined in this thesis.

Humand-Computer Interaction

In this thesis we focused on the development of a flexible back-end supporting system for user-centric service composition. In the proposed framework we define a *front-end user support* component, which is responsible for intermediating the communication between users and the back-end supporting system.

We did not research mechanisms to optimise the definition of front-end user supports, for example which graphical user interface elements are better suited to implement the different primitive commands. Further research is necessary to identify how the different primitive commands can be mapped to user interface interaction points in order to optimise the user experience in the usage scenarios that make use of service composition.

Appendix A

Ontologies

Figure A-1 *IO-Types.owl* ontology

Figure A-2
Core.owl
ontology

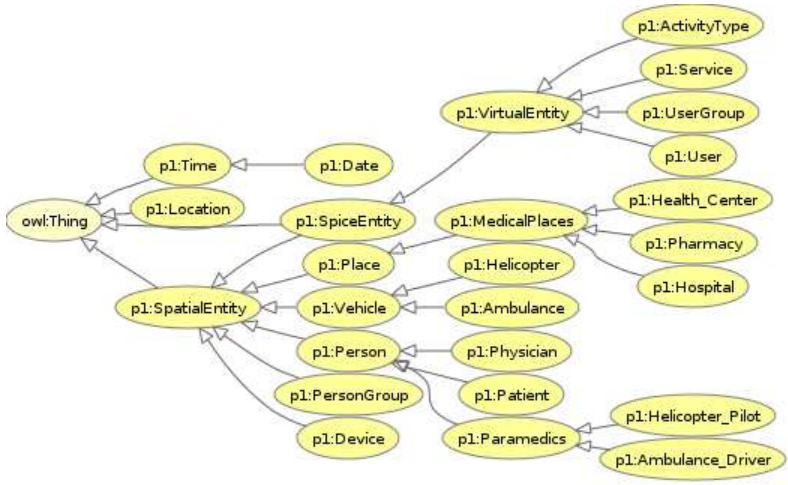


Figure A-3
NonFunctionaProperties.owl
ontology

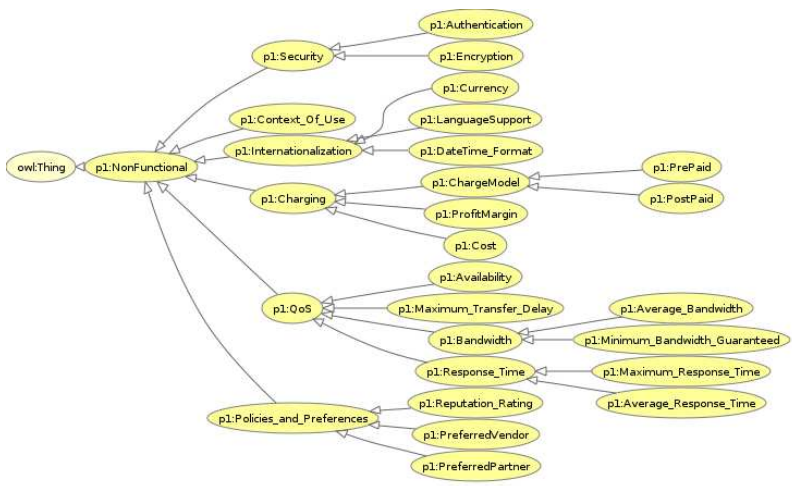
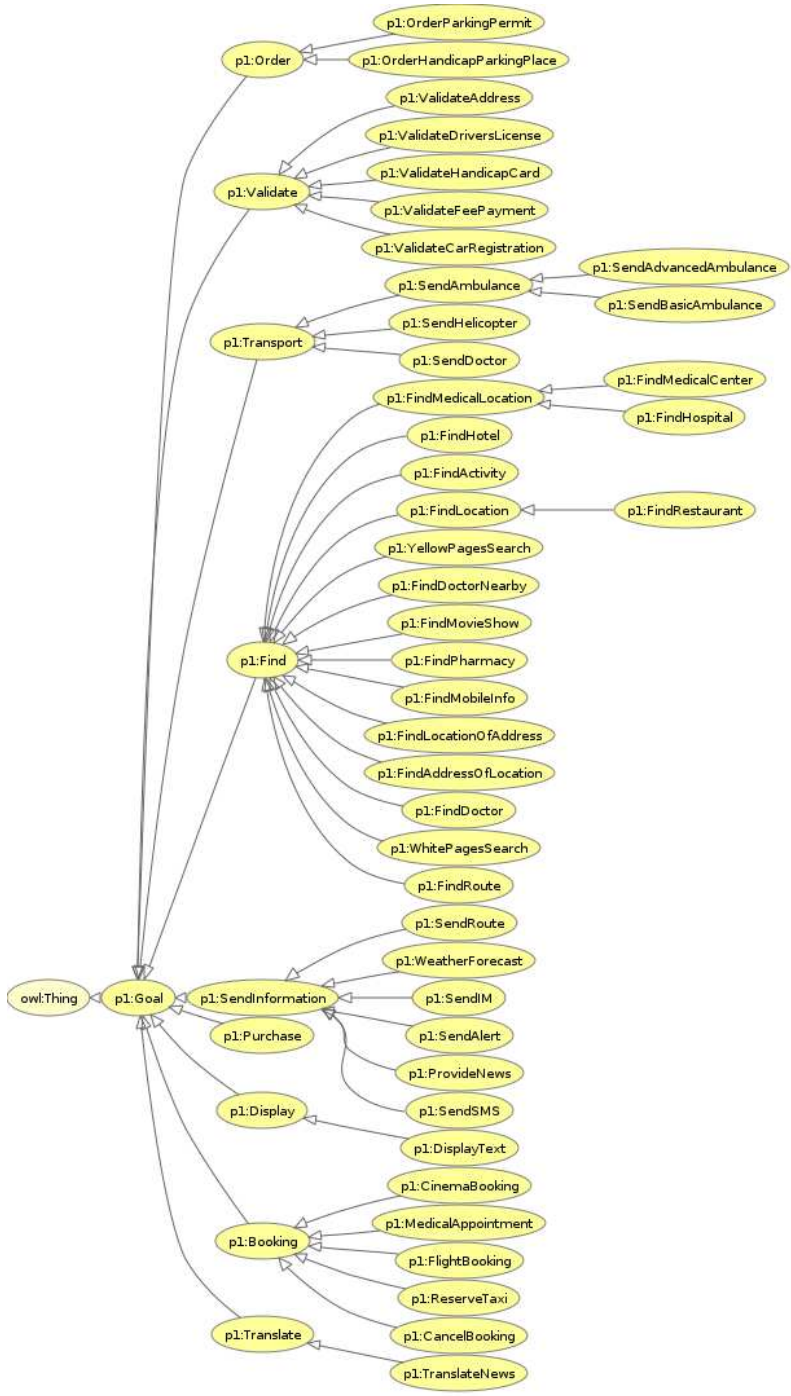


Figure A-4
Goals.owl
ontology



Appendix B

Evaluation Scenarios

B.1 Service Collections

We consider services in the domain of *e-health*. These services aim at supporting different activities to assist users, with health conditions, on their daily life.

B.1.1 Manually defined services

These services were all manually created and semantically annotated by humans, in a master student graduation project [125]. The collection consists of 13 services. The services support different actions of the user in the e-health domain, such as: find medical locations, make medical appointments, request medical transportation, find locations and routes, and find doctors in hospitals. These services are described using the SPATEL language [89]. These services are semantically annotated with semantic concepts that are references to the ontologies presented in Appendix A. In the following we present a description of the services used. We use a simplified representation of the SPATEL service description in the following. The complete SPATEL service descriptions, and further detail on the evaluation scenarios, are given in `http://dynamicos.sourceforge.net/eval-fram.html`.

AlertService

The *AltertService* consists of one service, (*alertContactPerson*), which sends an alert to the contact person of the patient, given the cell number patient and the hospital where he is or is being transported.

The SPATEL description of this service is presented in the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="AlertService">
```

```
    <service name="AlertService" semPattern="GQIO">
      <ownedOperation name="alertContactPerson">
        <ownedParameter name="number" semType="IOTypes.owl:
            CellNumber" direction="in" instanceType="String"/>
        <ownedParameter name="hospital" semType="core.owl:
            Hospital" direction="in" instanceType="String"/>
        <ownedParameter name="confirmation" semType="IOTypes.
            owl:Alert" direction="return" instanceType="String
            "/>
        <semTag name="OperationGoal" semType="Goals.owl:
            SendAlert" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="1"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="5" isDynamic="false
            "/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

LocationService

The LocationService consists of three services:

1. *locateUser*: returns the coordinates of the user, given his phone number;
2. *findRoute*: returns the route given the destination address and the number of the subscriber;
3. *findAddress*: returns the postal address of a given set of coordinates.

The SPATEL description of this service is presented in the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary >
  <nestedPackage name="LocationService">
    <service name="LocationService" semPattern="GQIO">
      <ownedOperation name="locateUser">
        <ownedParameter name="coordinates" semType="IOTypes.
            owl:Coordinates" direction="return" instanceType="
            float[]"/>
        <ownedParameter name="number" semType="IOTypes.owl:
            CellNumber" direction="in" instanceType="String"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindLocation" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="2"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="12" isDynamic="false
            "/>
      </ownedOperation>
      <ownedOperation name="findRoute">
        <ownedParameter name="number" semType="IOTypes.owl:
            CellNumber" direction="in" instanceType="String"/>
        <ownedParameter name="address" semType="IOTypes.owl:
            Address" direction="in" instanceType="String"/>
        <ownedParameter name="route" semType="IOTypes.owl:
            Route" direction="return" instanceType="String"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindRoute" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="3"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="4" isDynamic="false
            "/>
```

```
      </ownedOperation>
      <ownedOperation name="findAddress">
        <ownedParameter name="coordinates" semType="IOTypes.
            owl:Coordinates" direction="in" instanceType="
            float[]"/>
        <ownedParameter name="address" semType="IOTypes.owl:
            Address" direction="return" instanceType="String
            "/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindAddressOfLocation" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="6"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="7" isDynamic="false
            "/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

AppointmentService

The AppointmentService consists of two services:

1. *makeMedicalAppointment*: which, based on the patient and physician returns a medical appointment in medical location;
2. *makeMedicalAtHomeAppointment*: which, based on the patient and physician returns a medical appointment at the patients home.

The SPATEL description of this service is presented in the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary >
  <nestedPackage name="AppointmentService">
    <service name="AppointmentService" semPattern="GQIO">
      <ownedOperation name="makeMedicalAppointment">
        <ownedParameter name="doctor" semType="core.owl:
            Physician" direction="in" instanceType="String"/>
        <ownedParameter name="patient" semType="core.owl:
            Patient" direction="in" instanceType="String"/>
        <ownedParameter name="appointment" semType="IOTypes.
            owl:MedicalAppointment" direction="return"
            instanceType="String"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            MedicalAppointment" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="7"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="40" isDynamic="false
            "/>
      </ownedOperation>
      <ownedOperation name="makeMedicalAtHomeAppointment">
        <ownedParameter name="doctor" semType="core.owl:
            Physician" direction="in" instanceType="String"/>
        <ownedParameter name="patient" semType="core.owl:
            Patient" direction="in" instanceType="String"/>
        <ownedParameter name="appointment" semType="IOTypes.
            owl:MedicalAppointment" direction="return"
            instanceType="String"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            MedicalAppointment" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value
            ="15" isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="50" isDynamic="false
            "/>
```

```
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

TransportService

The TransportService consists of three services:

1. *sendAmbulance*: which sends an ambulance, given the coordinates of the location where the patient should be picked up, and the hospital to where he must be transported to;
2. *sendAdvancedAmbulance*: which sends an advanced ambulance, given the coordinates of the location where the patient should be picked up, and the hospital to where he must be transported to;
3. *sendHelicopter*: which sends an helicopter ambulance, given the coordinates of the location where the patient should be picked up, and the hospital to where he must be transported to.

The SPATEL description of this service is presented in the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary >
  <nestedPackage name="TransportService">
    <service name="TransportService" semPattern="GQIO">
      <ownedOperation name="sendAmbulance">
        <ownedParameter name="originCoordinates" semType="
            IOTypes.owl:Coordinates" direction="in"
            instanceType="float[]"/>
        <ownedParameter name="destinationHospital" semType="
            core.owl:Hospital" direction="in" instanceType="
            String"/>
        <ownedParameter name="confirmation" semType="core.owl:
            Ambulance" direction="return" instanceType="String
            "/>
        <semTag name="OperationGoal" semType="Goals.owl:
            SendBasicAmbulance" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value
            ="100" isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" isDynamic="false"/>
      </ownedOperation>
      <ownedOperation name="sendAdvancedAmbulance">
        <ownedParameter name="originCoordinates" semType="
            IOTypes.owl:Coordinates" direction="in"
            instanceType="float[]"/>
        <ownedParameter name="destinationHospital" semType="
            core.owl:Hospital" direction="in" instanceType="
            String"/>
        <ownedParameter name="confirmation" semType="core.owl:
            Ambulance" direction="return" instanceType="String
            "/>
        <semTag name="OperationGoal" semType="Goals.owl:
            SendAdvancedAmbulance" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value
            ="200" isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="1500" isDynamic="
            false"/>
      </ownedOperation>
```

```
      <ownedOperation name="sendHelicopter">
        <ownedParameter name="originCoordinates" semType="
            IOTypes.owl:Coordinates" direction="in"
            instanceType="float[]"/>
        <ownedParameter name="destinationHospital" semType="
            core.owl:Hospital" direction="in" instanceType="
            String"/>
        <ownedParameter name="confirmation" semType="core.owl:
            Helicopter" direction="return" instanceType="
            String"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            SendHelicopter" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost"  value
            ="1000" isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="500"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

WhitePages

The WhitePages consists of one service *findDoctor*, which finds a doctor, given a speciality and a hospital name.

The SPATEL description is presented in the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="WhitePages">
    <service name="WhitePages" semPattern="GQIO">
      <ownedOperation name="findDoctor">
        <ownedParameter name="speciality" semType="IOTypes.owl
            :MedicalSpeciality" direction="in" instanceType="
            String"/>
        <ownedParameter name="medicalcenter" semType="core.owl
            :MedicalPlaces" direction="in" instanceType="
            String"/>
        <ownedParameter name="doctor" semType="core.owl:
            Physician" direction="return" instanceType="String
            "/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindDoctor" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="2"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="4" isDynamic="false
            "/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

YellowPages

The YellowPages consists of three services:

1. *findHospital*: which finds the nearest hospital given a set of coordinates;
2. *findMedicalCenter*: which finds the nearest medical centre given a set of coordinates;

3. *findPharmacy*: which finds the nearest pharmacy given a set of coordinates.

The SPATEL description is presented in the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary >
  <nestedPackage name="YellowPages">
    <service name="YellowPages" semPattern="GQIO">
      <ownedOperation name="findHospital">
        <ownedParameter name="coordinates" semType="IOTypes.
            owl:Coordinates" direction="in" instanceType="
            float[]"/>
        <ownedParameter name="hospital" semType="core.owl:
            Hospital" direction="return" instanceType="String
            "/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindHospital" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="3"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="2" isDynamic="false
            "/>
      </ownedOperation>
      <ownedOperation name="findMedicalCenter">
        <ownedParameter name="coordinates" semType="IOTypes.
            owl:Coordinates" direction="in" instanceType="
            float[]"/>
        <ownedParameter name="medcenter" semType="core.owl:
            Health_Center" direction="return" instanceType="
            String"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindMedicalCenter" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="3"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="2" isDynamic="false
            "/>
      </ownedOperation>
      <ownedOperation name="findPharmacy">
        <ownedParameter name="coordinates" semType="IOTypes.
            owl:Coordinates" direction="in" instanceType="
            float[]"/>
        <ownedParameter name="pharmacy" semType="core.owl:
            Pharmacy" direction="return" instanceType="String
            "/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindPharmacy" kind="goal"/>
        <nonFuncTag semType="NonFunctional.owl:Cost" value="3"
             isDynamic="false"/>
        <nonFuncTag semType="NonFunctional.owl:
            Maximum_Response_Time" value="2" isDynamic="false
            "/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

B.1.2 Automatically Generated Services

The collection of services proposed in the previous section allows to test some aspects of the service composition approaches, namely the quality of proposed service compositions. However, and due to its size, it does not allow to test other aspects, such as the scalability of the approaches. To overcome this, we have developed a tool (*RandServGen*) to automatically generate semantic

services. This tool generates a given number of services, specified by the tool user, annotated with random semantic concepts from the framework ontologies. The generated services have also a random number of $IOPE$ parameters, according to limits specified by the tool user. The selection of semantic concepts for the service interface parameters ($IOPE$) and service goals (G) is random, i.e., we use the ontologies semantic concepts and select randomly concepts for each of the parameters. The IOPEs parameters are collected from the ontologies that describe these parameter types ($IOTypes.owl$ and $Core.owl$), while the goals are collected from the goal ontology ($Goals.owl$) defined in the framework. To support our experiments we have generated 500 services automatically. All these artifacts are presented and available in `http://dynamicos.sourceforge.net/eval-fram.html`.

We create 10 sets of 50 services each. These sets are to be incrementally added to the service registry, when evaluating the service composition approaches according to the time-based metrics, namely the scalability.

B.2 Evaluation Scenarios

To develop the evaluation scenarios, we consider only the manually defined semantic services from the *e-health* services collection. This decision was taken because the randomly generated services do not have clear semantics, since their parameters are randomly generated.

Based on the considered services, we can defined several evaluation scenarios, i.e., service requests (SR) and respective matching reference service compositions (RSC). In the following we present two possible evaluation scenarios:

- **SR1**: Make a medical appointment for a given medical speciality in the nearest hospital;
- **SR2**: Find the nearest medical locations;

For service requests $SR1$ we have identified one reference service composition (RSC), presented in Figure B-1. For the service request $SR2$ we have identified two service reference compositions (RSC), presented in Figure B-2.

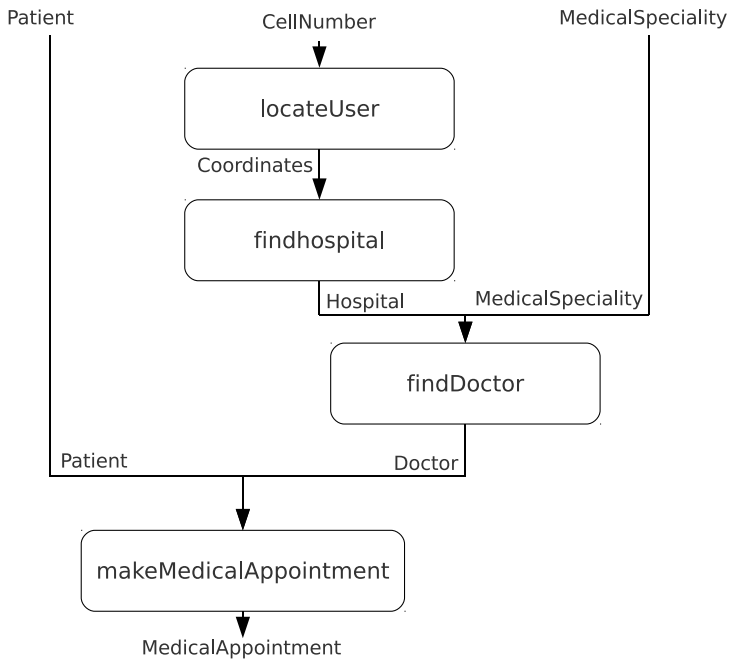


Figure B-1 Reference Service Compositions for $SR1$

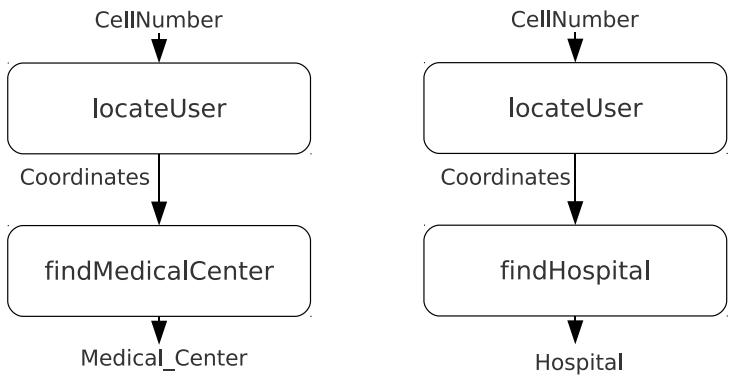


Figure B-2 Reference Service Compositions for $SR2$

B.2.1 Evalaution Resources

The resources presented in the previous section are available in the DynamiCoS webpage[1]. The manually defined services, the automatically generated services and the ontologies used for the annotation of the semantic services. These resources can be down-

[1] `http://dynamicos.sourceforge.net/eval-fram.html`

loaded and used for the evaluation of other semantic service composition approaches.

Appendix C

A-DynamiCoS Use Cases Services

C.1 E-Government

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="addressValidation">
    <service name="addressValidation" semPattern="GQIO">
      <ownedOperation name="addressValidation">
        <ownedParameter name="address" semType="IOTypes.owl:
            Address" direction="in" instanceType="String"/>
        <ownedParameter name="citizenId" semType="IOTypes.owl:
            citizenId" direction="in" instanceType="Integer"/>
        <ownedParameter name="confirmation" semType="IOTypes.
            owl:addressValidationStatus" direction="return"
            instanceType="Boolean"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            ValidateAddress" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary
  <nestedPackage name="carRegistrationValidation  ">
    <service name="carRegistrationValidation" semPattern="GQIO
        ">
      <ownedOperation name="carRegistrationValidation">
        <ownedParameter name="vehiclePlate" semType="IOTypes.
            owl:Vehicle_Plate" direction="in" instanceType="
            String"/>
        <ownedParameter name="carRegistrationStatus" semType="
            IOTypes.owl:carRegistrationStatus" direction="
            return" instanceType="Boolean"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            ValidateCarRegistration" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="driversLicenseValidation">
```

```
    <service name="driversLicenseValidation" semPattern="GQIO
        ">
      <ownedOperation name="driversLicenseValidation">
        <ownedParameter name="licenseID" semType="IOTypes.owl:
            licenseId" direction="in" instanceType="Integer"/>
        <ownedParameter name="addressStatus" semType="IOTypes.
            owl:addressValidationStatus" direction="in"
            instanceType="Boolean"/>
        <ownedParameter name="driversLicenseStatus" semType="
            IOTypes.owl:driversLicenseStatus" direction="
            return" instanceType="Boolean"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            ValidateDriversLicense" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary >
  <nestedPackage name="feeValidation">
    <service name="feeValidation" semPattern="GQIO">
    <ownedOperation name="feeValidation">
        <ownedParameter  name="feeCode" semType="IOTypes.owl:
            feeCode" direction="in" instanceType="Integer"/>
        <ownedParameter  name="feeStatus" semType="IOTypes.owl
            :feePaymentStatus" direction="return" instanceType
            ="Boolean"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            ValidateFeePayment" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="handicapCardValidation">
    <service name="handicapCardValidation" semPattern="GQIO">
      <ownedOperation name="handicapCardValidation">
        <ownedParameter name="socialSecurityNumber" semType="
            IOTypes.owl:socialSecurityNumber" direction="in"
            instanceType="Integer"/>
        <ownedParameter name="addressStatus" semType="IOTypes.
            owl:addressValidationStatus" direction="in"
            instanceType="Boolean"/>
        <ownedParameter name="handicapStatus" semType="IOTypes
            .owl:handicapCardStatus" direction="return"
            instanceType="Boolean"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            ValidateHandicapCard" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="orderHandicapPlace">
    <servicename="orderHandicapPlace" semPattern="GQIO">
      <ownedOperation name="orderHandicapPlace">
        <ownedParameter name="licenseId" semType="IOTypes.owl:
            licenseId" direction="in" instanceType="Integer"/>
        <ownedParameter name="handicapCardValidation" semType
            ="IOTypes.owl:handicapCardStatus" direction="in"
            instanceType="Boolean"/>
```

```
        <ownedParameter name="driversLicenseStatus" semType="
            IOTypes.owl:driversLicenseStatus" direction="in"
            instanceType="Boolean"/>
        <ownedParameter name="carRegistration" semType="
            IOTypes.owl:carRegistrationStatus" direction="in"
            instanceType="Boolean"/>
        <ownedParameter name="parkingPermitPeriod" semType="
            IOTypes.owl:parkingPermitPeriod" direction="in"
            instanceType="Boolean"/>
        <ownedParameter name="handicapPlaceIssueNumber"
            semType="IOTypes.owl:handicapPlaceIssueNumber"
            direction="return" instanceType="integer"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            OrderHandicapParkingPlace" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="orderParkingPermit">
    <service name="orderParkingPermit" semPattern="GQIO">
      <ownedOperation name="orderParkingPermit">
        <ownedParameter name="addressValidationStatus" semType
            ="IOTypes.owl:addressValidationStatus" direction="
            in" instanceType="Boolean"/>
        <ownedParameter name="driversLicenseStatus" semType="
            IOTypes.owl:driversLicenseStatus" direction="in"
            instanceType="Boolean"/>
        <ownedParameter name="feePaymentStatus" semType="
            IOTypes.owl:feePaymentStatus" direction="in"
            instanceType="Boolean"/>
        <ownedParameter name="carRegistrationStatus" semType="
            IOTypes.owl:carRegistrationStatus" direction="in"
            instanceType="Boolean"/>
        <ownedParameter name="parkingPermitPeriod" semType="
            IOTypes.owl:parkingPermitPeriod" direction="return
            " instanceType="String"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            OrderParkingPermit" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

C.2 Entertainment

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="Buienradar">
    <service name="Buienradar" semPattern="GQIO">
      <ownedOperation name="rainForecast">
        <ownedParametername="location" semType="IOTypes.owl:
            Coordinates" direction="in" instanceType="float
            []"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            WeatherForecast" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="GoogleDirections">
    <service name="GoogleDirections" semPattern="GQIO">
      <ownedOperation name="findDirections">
        <ownedParameter name="origin" semType="IOTypes.owl:
            Address"
         direction="in" instanceType="String"/>
        <ownedParameter name="destination" semType="IOTypes.
            owl:Address" direction="in" instanceType="String
            "/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindRoute" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="GoogleGeocode">
    <service name="GoogleGeocode" semPattern="GQIO">
      <ownedOperation name="findLocation">
        <ownedParameter name="address" semType="IOTypes.owl:
            Address" direction="in" instanceType="String"/>
        <ownedParameter name="location" semType="IOTypes.owl:
            Coordinates" direction="return" instanceType="
            float[]"/>
        <semTag name="OperationGoal" semType="Goals.owl:
            FindLocationOfAddress" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="Iens">
    <service name="Iens" semPattern="GQIO">
      <ownedOperation name="findRestaurant">
        <ownedParameter name="city" semType="IOTypes.owl:City"
             direction="in" instanceType="String"/>
        <ownedParameter name="kitchen" semType="IOTypes.owl:
            KitchenType" direction="in" instanceType="String
            "/>
        <ownedParameter name="address" semType="IOTypes.owl:
            Address" direction="return" instanceType="String"
            />
        <semTag name="OperationGoal" semType="Goals.owl:
            FindRestaurant" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="Ikdoe">
    <service name="Ikdoe" semPattern="GQIO">
      <ownedOperation name="findActivity">
        <ownedParametername="keyword" semType="IOTypes.owl:
            Keyword" direction="in" instanceType="String"/>
```

```
        <ownedParameter name="category" semType="IOTypes.owl:
            ActivityType" direction="in" instanceType="String
            "/>
        <ownedParameter name="start_date" semType="IOTypes.owl
            :Date" direction="in" instanceType="Date"/>
        <ownedParameter name="end_date" semType="IOTypes.owl:
            Date" direction="in" instanceType="Date"/>
        <ownedParameter name="city_name" semType="IOTypes.owl:
            City" direction="in" instanceType="String"/>
        <ownedParameter name="category" semType="IOTypes.owl:
            ActivityType" direction="return" instanceType="
            String"/>
        <ownedParameter name="location" semType="IOTypes.owl:
            Coordinates" direction="return" instanceType="
            float[]" />
        <ownedParameter name="date" semType="IOTypes.owl:Date"
             direction="return" instanceType="Date" />
        <semTag name="OperationGoal" semType="Goals.owl:
            FindActivity" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="Kayak">
    <service name="Kayak" semPattern="GQIO">
      <ownedOperation name="findHotel">
        <ownedParameter name="city" semType="IOTypes.owl:City"
             direction="in" instanceType="String"/>
        <ownedParameter name="checkin_date" semType="IOTypes.
            owl:Date" direction="in" instanceType="String"/>
        <ownedParameter name="checkout_date" semType="IOTypes.
            owl:Date" direction="in" instanceType="String"/>
        <ownedParameter name="guests" semType="IOTypes.owl:
            NumberOfGuests" direction="in" instanceType="int
            "/>
        <ownedParameter name="rooms" semType="IOTypes.owl:
            NumberOfRooms" direction="in" instanceType="int"/>
        <ownedParameter name="address" semType="IOTypes.owl:
            Address" direction="return" instanceType="String"
            />
        <ownedParameter name="city" semType="IOTypes.owl:City"
             direction="return" instanceType="String" />
        <semTag name="OperationGoal" semType="Goals.owl:
            FindHotel" kind="goal"/>
      </ownedOperation>
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary>
  <nestedPackage name="Seatme">
    <service name="Seatme" semPattern="GQIO">
      <ownedOperation name="findRestaurant">
        <ownedParameter name="city" semType="IOTypes.owl:City"
             direction="in" instanceType="String"/>
        <ownedParameter name="kitchen" semType="IOTypes.owl:
            KitchenCategory" direction="in" instanceType="
            String"/>
        <ownedParameter name="address" semType="IOTypes.owl:
            Address" direction="return" instanceType="String"
            />
        <semTag name="OperationGoal" semType="Goals.owl:
            FindRestaurant" kind="goal"/>
      </ownedOperation>
    </service>
```

```
  </nestedPackage>
</spatel:ServiceLibrary>
```

Appendix D

A-DynamiCoS Usage

D.1 A-DynamiCoS Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://coord.dynamicos"
  xmlns:ns1="http://org.apache.axis2/xsd" xmlns:ns="http://
      coord.dynamicos"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns
      :wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:xs
      ="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:
      mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
  <wsdl:types>
    <xs:schema attributeFormDefault="qualified"
      elementFormDefault="qualified" targetNamespace="http://
          coord.dynamicos">
      <xs:element name="Coordinate">
        <xs:complexType>
          <xs:sequence>
            <xs:element minOccurs="0" name="CommandID"
              nillable="true" type="xs:string" />
            <xs:element minOccurs="0" name="CommandParameters"
              nillable="true" type="xs:string" />
            <xs:element minOccurs="0" name="UserSessionID"
                nillable="true"
              type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="CoordinateResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element minOccurs="0" name="return" nillable="
                true"
              type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="CoordinateResponse">
    <wsdl:part name="parameters" element="ns:
        CoordinateResponse">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="CoordinateRequest">
    <wsdl:part name="parameters" element="ns:Coordinate">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="CoordinatorPortType">
    <wsdl:operation name="Coordinate">
      <wsdl:input message="ns:CoordinateRequest" wsaw:Action="
          urn:Coordinate">
      </wsdl:input>
```

```
      <wsdl:output message="ns:CoordinateResponse" wsaw:Action
          ="urn:CoordinateResponse">
      </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="CoordinatorSoap11Binding" type="ns:
      CoordinatorPortType">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="Coordinate">
      <soap:operation soapAction="urn:Coordinate" style="
          document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Coordinator">
    <wsdl:port name="CoordinatorHttpSoap11Endpoint" binding="
        ns:CoordinatorSoap11Binding">
      <soap:address
        location="http://localhost:8088/CoordinatorWS/services
            /Coordinator.CoordinatorHttpSoap11Endpoint/" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

D.2 Primitive Commands Definitions

D.2.1 AddToContext

Table D-1 *AddToContext* Primitive Command

CommandID	*ADD_TO_CONTEXT*
Description	Adds information (service output/result, user context info) to the user execution context.
Command Type	Context Management

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://dynamicos/interactions/AddToContext"
  xmlns:tns="http://dynamicos/interactions/AddToContext"
  xmlns:basictypes="http://dynamicos/interactions/BasicTypes"
  elementFormDefault="qualified">
  <import schemaLocation="BasicTypes.xsd"
    namespace="http://dynamicos/interactions/BasicTypes">
  </import>
  <!-- Request Message -->
  <element name="AddToContextRequest">
    <complexType>
      <sequence>
        <choice>
          <element name="ServiceIdToAdd" type="string"
              maxOccurs="unbounded" />
          <element name="ValueToAdd" type="basictypes:
              ContextValue" maxOccurs="unbounded" />
        </choice>
      </sequence>
    </complexType>
  </element>
```

```
  <!-- Response message -->
  <element name="AddToContextResponse">
    <complexType>
      <sequence>
        <element name="Result" type="string" />
      </sequence>
    </complexType>
  </element>
</schema>
```

D.2.2 ServTypeDiscovery

Table D-2 *ServType-Discovery* Primitive Command

CommandID	*SERV_TYPE_DISCOVERY*
Description	Discovers services based on a service type (concept on the goal domain ontology).
Command Type	Discovery

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://dynamicos/interactions/
    ServTypeDiscovery"
  elementFormDefault="qualified" xmlns="http://www.w3.org
      /2001/XMLSchema"
  xmlns:tns="http://dynamicos/interactions/ServTypeDiscovery"
  xmlns:basictypes="http://dynamicos/interactions/BasicTypes">
  <import schemaLocation="BasicTypes.xsd"
    namespace="http://dynamicos/interactions/BasicTypes">
  </import>
  <!-- Request Message -->
  <element name="ServTypeDiscoveryRequest">
    <complexType>
      <sequence>
        <element name="ServiceType" type="basictypes:
            ServiceType"/>
      </sequence>
    </complexType>
  </element>
  <!-- Response message -->
  <element name="ServTypeDiscoveryResponse">
    <complexType>
      <sequence>
        <element name="MatchingService" type="basictypes:
            ServiceTypeMatching'' maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

D.2.3 SeleServ

Table D-3 *SeleServ* Primitive Command

CommandID	*SELE_SERV*
Description	Selects a service from a set of services to consider it for composition/execution.
Command Type	Selection, Composition

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://dynamicos/interactions/SeleServ"
   xmlns:tns="http://dynamicos/interactions/SeleServ"
   xmlns:basictypes="http://dynamicos/interactions/BasicTypes"
   elementFormDefault="qualified">
   <import schemaLocation="BasicTypes.xsd"
      namespace="http://dynamicos/interactions/BasicTypes">
   </import>
   <!-- Request Message -->
   <element name="SeleServRequest">
      <complexType>
         <sequence>
            <element name="ServiceSelectionInfo" type="basictypes:
                 ServiceSelection" maxOccurs="unbounded" />
         </sequence>
      </complexType>
   </element>
   <!-- Response message -->
   <element name="SeleServResponse">
      <complexType>
         <sequence>
            <element name="ServiceInfo" type="basictypes:
                 ServiceInfo" maxOccurs="unbounded"/>
         </sequence>
      </complexType>
   </element>
</schema>
```

D.2.4 ValidateInputs

Table D-4 *ValidateInputs* Primitive Command

CommandID	*VALIDATE_INPUTS*
Description	Communicates the inputs entered by the user to the back-end supporting system.
	Out: *List{InvalidInputs}*
Command Type	Service Inputs Management

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://dynamicos/interactions/
        ValidateInputs"
   xmlns:tns="http://dynamicos/interactions/ValidateInputs"
   xmlns:basictypes="http://dynamicos/interactions/BasicTypes"
   elementFormDefault="qualified">
   <import schemaLocation="BasicTypes.xsd"
      namespace="http://dynamicos/interactions/BasicTypes">
   </import>
   <!-- Request Message -->
   <element name="ValidateInputsRequest">
      <complexType>
         <sequence>
            <element name="InputToValidate" type="basictypes:
                 InputValidation" maxOccurs="unbounded" />
         </sequence>
      </complexType>
   </element>
   <!-- Response message -->
   <element name="ValidateInputsResponse">
      <complexType>
         <sequence>
            <element name="ServiceValidation" type="basictypes:
                 ServiceValidation" maxOccurs="unbounded" />
         </sequence>
      </complexType>
```

```
  </element>
</schema>
```

D.2.5 ResolveServ

Table D-5 *ResolveServ* Primitive Command

CommandID	*RESOLVE_SERV*
Description	Finds services that can provide outputs to match the selected inputs/preconditions of a service previously selected by the user.
Command Type	Discovery

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://dynamicos/interactions/ResolveServ"
  xmlns:tns="http://dynamicos/interactions/ResolveServ"
  xmlns:basictypes="http://dynamicos/interactions/BasicTypes"
  elementFormDefault="qualified">
  <import schemaLocation="BasicTypes.xsd"
    namespace="http://dynamicos/interactions/BasicTypes">
  </import>
  <!-- Request Message -->
  <element name="ResolveServRequest">
    <complexType>
      <sequence>
        <element name="InputToResolve" type="basictypes:
            ServiceValidation" maxOccurs="unbounded" />
      </sequence>
    </complexType>
  </element>
  <!-- Response message -->
  <element name="ResolveServResponse">
    <complexType>
      <sequence>
        <element name="MatchingService" type="basictypes:
            ResolvingServiceMatching" maxOccurs="unbounded" />
      </sequence>
    </complexType>
  </element>
</schema>
```

D.2.6 ExecServ

Table D-6 *ExecServ* Primitive Command

CommandID	*EXEC_SERV*
Description	Returns the services from the service composition that can be executed.
Command Type	Execution

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://dynamicos/interactions/ExecServ"
  xmlns:tns="http://dynamicos/interactions/ExecServ"
  xmlns:basictypes="http://dynamicos/interactions/BasicTypes"
  elementFormDefault="qualified">
  <import schemaLocation="BasicTypes.xsd"
```

```
    namespace="http://dynamicos/interactions/BasicTypes">
  </import>
  <!-- Request Message -->
  <element name="ExecServRequest">
    <complexType>
      <sequence>
        <element name="ExecServParams" type="string" />
      </sequence>
    </complexType>
  </element>
  <!-- Response message -->
  <element name="ExecServResponse">
    <complexType>
      <sequence>
        <element name="ServiceInfo" type="basictypes:
            ServiceInfo"
          maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

References

[1] Howard Rheingold. Tools for Thought: The History and Future of Mind-Expanding Technology. MIT Press, 2000.

[2] Steven Levy. Hackers: Heroes of the Computer Revolution. Dell Publishing Co., Inc., 1994.

[3] Mark Weiser. The computer for the twenty-first century. Scientific American, (265(3):94-104), September 1991.

[4] Tim Berners-Lee, Robert Cailliau, and Bernd Pollermann. World-wide web: The information universe. Communications of the ACM, 37:76–82, 1992.

[5] Tim Berners-Lee. Www: Past, present, and future. Computer, 29:69–77, October 1996.

[6] Andrew Tanenbaum and Maarten Van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, 2001.

[7] Hamid Motahari-Nezhad, Bryan Stephenson, and Sharad Singhal. Outsourcing Business to Cloud Computing Services: Opportunities and Challenges. HP Laboratories, 2009.

[8] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing. Communications of the ACM, 46:25–28, 2003.

[9] Dimitrios Georgakopoulos and Michael P. Papazoglou. Service-Oriented Computing. The MIT Press, 2008.

[10] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0. http://www.w3.org/TR/wsdl20/, June 2007.

[11] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web services: concepts, architectures and applications. Springer-Verlag, 2004.

[12] Michael P. Papazoglou. Web Services: Principles and Technology. Prentice Hall, 2007.

[13] Schahram Dustdar and Mike P. Papazoglou. Services and service composition - an introduction. IT - Information Technology, 50(2):86–92, 2008.

[14] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. Internet Computing, IEEE, 8(6):51 – 59, 2004.

[15] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. International Journal on Web Grid Services, 1(1):1–30, 2005.

[16] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services, version 1.1, 2003.

[17] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. Computer, 40(11):38–45, 2007.

[18] Thomas Erl. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, 2005.

[19] Chad Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. Technical report, OASIS, October 2006.

[20] Chris Peltz. Web services orchestration and choreography. Computer, 36:46–52, 2003.

[21] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barret. Web services choreography description language version 1.0, 2005.

[22] Alistair Barros, Marlon Dumas, and Arthur H.M. ter Hofstede. Service interaction patterns. In Business Process Management, volume 3649 of Lecture Notes in Computer Science, pages 302–318. Springer Berlin / Heidelberg, 2005.

[23] Eduardo Gonçalves da Silva, Jorge Martinez López, Luís Ferreira Pires, and Marten van Sinderen. Defining and prototyping a life-cycle for dynamic service composition. In International workshop on architectures, concepts and technologies for service oriented computing, pages 79–90, July 2008.

[24] Jian Yang and Mike. P. Papazoglou. Service components for managing the life-cycle of service compositions. Information Systems, 29(2):97 – 125, 2004.

[25] Guo-Qing Zhang, Guo-Qiang Zhang, Qing-Feng Yang, Su-Qi Cheng, and Tao Zhou. Evolution of the internet and its cores. New Journal of Physics, 10(12), 2008.

[26] Antonio Gulli and Alessio Signorini. The indexable web is more than 11.5 billion pages. In Special interest tracks and posters of the International conference on World Wide Web, WWW 05, pages 902–903. ACM, 2005.

[27] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. Scientific American, May 2001.

[28] Nicola Guarino. Formal ontology and information systems. In International Conference on Formal Ontologies in Information Systems, pages 3–15. IOS Press, June 1998.

[29] Natalya F. Noy and Deborah L. mcguinness. Ontology development 101: A guide to creating your first ontology, 2001.

[30] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. IEEE Intelligent Systems, 16(2):46–53, 2001.

[31] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In International Workshop on Semantic Web Services and Web Process Composition, pages 43–54, 2005.

[32] Atif Alamri, Mohamad Eid, and Abdulmotaleb El Saddik. Classification of the state-of-the-art in dynamic web services composition techniques. International Journal of Web Grid Services, 2(2):148–166, 2006.

[33] Ravi Khadka and Brahmananda Sapkota. An evaluation of dynamic web service composition approaches. http://eprints.eemcs.utwente.nl/18346/, 2010.

[34] Rodrigo Mantovaneli Pessoa, Eduardo Gonçalves da Silva, Marten van Sinderen, Dick Quartel, and Luís Ferreira Pires. Enterprise interoperability with soa: a survey of service composition approaches. In International Workshop on Enterprise Interoperability, pages 32–45, September 2008.

[35] Ruoyan Zhang, Ismailcem Budak Arpinar, and Boanerges Aleman-Meza. Automatic composition of semantic web services. In International Conference on Web Services, pages 38–41, 2003.

[36] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In International Semantic Web Conference, pages 333–347, 2002.

[37] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Composition of semantic web services using linear logic theorem proving. Information Systems, 31(4):340–360, 2006.

[38] Mark H. Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew V. McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, and Katia P. Sycara. Daml-s: Web service description for the semantic web. In International Semantic Web Conference, pages 348–363, 2002.

[39] Freddy Lécué and Alain Léger. A formal model for semantic web service composition. In International Semantic Web Conference, LNCS, vol. 4273, pages 385–398, 2006.

[40] Freddy Lécué and Alain Leger. Semantic web service composition based on a closed world assumption. In European Conference on Web Services, pages 233 –242, 2006.

[41] Kunal Verma, Karthik Gomadam, Amit P. Sheth, John A. Miller, and Zixin Wu. The meteor-s approach for configuring and executing dynamic web processes. Technical report, University of Georgia, Athens, June 2005.

[42] Abhijit A. Patil, Swapna A. Oundhakar, Amit P. Sheth, and Kunal Verma. Meteor-s web service annotation framework. In International conference on World Wide Web, WWW '04, pages 553–562. ACM, 2004.

[43] Kunal Verma, Kaarthik Sivashanmugam, Amit Sheth, Abhijit Patil, Swapna Oundhakar, and John Miller. Meteor-s wsdi:

A scalable p2p infrastructure of registries for semantic publication and discovery of web services. Information Technology and Management, 6:17–39, January 2005.

[44] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint driven web service composition in meteor-s. In IEEE International Conference on Services Computing, pages 23–30. IEEE Computer Society, 2004.

[45] Kaarthik Sivashanmugam, John A. Miller, Amit P. Sheth, and Kunal Verma. Framework for semantic web process composition. International Journal Electronic Commerce, 9:71–106, January 2005.

[46] Simela Topouzidou. Service oriented development in a unified framework (sodium), final rerport. Technical report, IST-FP6-004559, 2007.

[47] A. Tsalgatidou, G. Athanasopoulos, M. Pantazoglou, C. Pautasso, T. Heinis, R. Grønmo, Hjørdis Hoff, Arne-Jørgen Berre, M. Glittum, and S. Topouzidou. Developing scientific workflows from heterogeneous services. SIGMOD Rec., 35:22–28, June 2006.

[48] Cesare Pautasso. A Flexible System for Visual Service Composition. PhD thesis, ETH Zurich, Computer Science Department, 2004.

[49] J. Pathak, S. Basu, R. Lutz, and V. Honavar. Moscoe: A framework for modeling web service composition and execution. In International Conference on Data Engineering Workshops, 2006.

[50] J. Pathak, S. Basu, and V. Honavar. Composing web services through automatic reformulation of service specifications. In IEEE International Conference on Services Computing, volume 1, pages 361 –369, 2008.

[51] David Martin, Mark Burstein, Drew Mcdermott, Sheila Mcilraith, Massimo Paolucci, Katia Sycara, Deborah L. Mcguinness, Evren Sirin, and Naveen Srinivasan. Bringing semantics to web services with owl-s. World Wide Web, 10(3):243–277, 2007.

[52] Keita Fujii and Tatsuya Suda. Semantics-based dynamic service composition. IEEE Journal on Selected Areas in Communications, 23(12):2361 – 2372, 2005.

[53] Keita Fujii and Tatsuya Suda. Dynamic service composition using semantic information. In International Conference on Service Oriented Computing, pages 39–48, 2004.

[54] Srividya Kona, Ajay Bansal, and Gopal Gupta. Automatic composition of semanticweb services. In International Conference on Web Services, pages 150–158, 2007.

[55] Ajay Bansal, Srividya Kona, Luke Simon, and Thomas D. Hite. A universal service-semantics description language. In European Conference on Web Services, page 214, 2005.

[56] Duane Merrill. Mashups: The new breed of web app, 2006. IBM Web Architecture Technical Library, http://www.ibm.com/developerworks/xml/library/x-mashups.html.

[57] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. Towards service composition based on mashup. In IEEE Congress on Services, pages 332 –339, 2007.

[58] Xuanzhe Liu, Gang Huang, and Hong Mei. Towards end user service composition. Computer Software and Applications Conference, Annual International, 1:676–678, 2007.

[59] Xuanzhe Liu, Gang Huang, and Hong Mei. A user-oriented approach to automated service composition. In IEEE international conference on Web services, pages 773–776, Sept. 2008.

[60] Jun Han, Yanbo Han, Yan Jin, Jianwu Wang, and Jian Yu. Personalized active service spaces for end-user service composition. In IEEE International Conference on Services Computing, pages 198–205, Sept. 2006.

[61] Y. Han, H. Geng, H. Li, J. Xiong, G. Li, B. Holtkamp, R. Gartmann, R. Wagner, and N. Weissenberg. Vinca: A visual and personalized business-level composition language for chaining web-based services. In International Conference on Service-Oriented Computing, volume 2910 of Lecture Notes in Computer Science, pages 165–177. Springer Berlin / Heidelberg, 2003.

[62] Juan C. Yelmo, Rubén Trapero, José M. Álamo, Juergen Sienel, Marc Drewniok, Isabel Ordás, and Kathleen Mccallum. User-driven service lifecycle management — adopting internet paradigms in telecom services. In International conference on service-oriented computing, pages 342–352. Springer-Verlag, 2007.

[63] Volker Hoyer, Katarina Stanoesvka-Slabeva, T. Janner, and C. Schroth. Enterprise mashups: Design principles towards the long tail of user needs. In IEEE International Conference on Services Computing, volume 2, pages 601 –602, 2008.

[64] Volker Hoyer and Katarina Stanoevska-Slabeva. The changing role of it departments in enterprise mashup environments. In International Service-Oriented Computing Conference Workshops, volume 5472 of Lecture Notes in Computer Science, pages 148–154. Springer Berlin / Heidelberg, 2009.

[65] B.F. Schmid and M.A. Lindemann. Elements of a reference model for electronic markets. In Hawaii International Conference on System Sciences, volume 4, pages 193 –201 vol.4, 1998.

[66] Tobias Nestler. Towards a mashup-driven end-user programming of soa-based applications. In 10th International conference on information integration and web-based applications & services, pages 551–554. ACM, 2008.

[67] Tobias Nestler, Marius Feldmann, Andre Preußner, and Alexander Schill. Service composition at the presentation layer using web service annotations. In International Workshop on Lightweight Integration on the Web. CEUR workshop proceedings, 2009.

[68] Agnes Ro, Lily Shu-Yi Xia, Hye-Young Paik, and Chea Hyon Chon. Bill organiser portal: a case study on end-user composition. In International workshops on web information systems engineering, pages 152–161. Springer-Verlag, 2008.

[69] Evren Sirin, James A. Hendler, and Bijan Parsia. Semi-automatic composition of web services using semantic descriptions. In Workshop on Web Services: Modeling, Architecture and Infrastructure, pages 17–24, 2003.

[70] Michael K. Smith, Deborah McGuiness, Raphael Volz, and Christopher Welty. Web Ontology Language (OWL) guide, version 1.0. W3C, 2002.

[71] Sonia Ben Mokhtar, Anupam Kaul, Nikolaos Georgantas, and Valérie Issarny. Efficient semantic service discovery in pervasive computing environments. In International conference on middleware, pages 240–259. Springer-Verlag New York, Inc., 2006.

[72] Sonia Ben Mokhtar, Anupam Kaul, Nikolaos Georgantas, and Valérie Issarny. Cocoa : conversationbased service composition for pervasive computing environments. International Conference on Pervasive Services, 0:29–38, 2006.

[73] Quan Z. Sheng, Boualem Benatallah, and Zakaria Maamar. User-centric services provisioning in wireless environments. Communications ACM, 51(11):130–135, 2008.

[74] Mike Papazoglou and Klaus Pohl. Longer term research challenges in software and services. Technical report, European Commission, 2008.

[75] Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten van Sinderen. Dynamic composition of services: Why, where and how. In International Workshop on Enterprise Systems and Technology, pages 73–85, May 2008.

[76] Nguyen Thi Thanh Hai and Toshio Obi. Government transformation: The first step to integrate e-business into e-government. In Integrated E-Business Models for Government Solutions: Citizen-Centric Service Oriented Methodologies and Processes. IGI, 2009.

[77] Horn Hauschild. Top Priority E-Government Guidelines for heads of public agencies. B.S.I, 2004.

[78] United Nations. United Nations e-government Survey 2008. New York : United Nations, 2008.

[79] Anamarija Leben and Marko Bohanec. Evaluation of life-event portals: multi-attribute model and case study. In IFIP international working conference on Knowledge management in electronic government, pages 25–36. Springer-Verlag, 2003.

[80] Gartner. Gartner highlights key predictions for it organisations and users in 2008 and beyond, January 2008.

[81] R. L. Ackoff. From data to wisdom. Journal of Applied Systems Analysis, 16:3–9, 1989.

[82] M. Zeleny. Management support systems: towards integrated knowledge management. Human Systems Management, 7:59–70, 1987.

[83] Eduardo Gonçalves da Silva, Luis Ferreira Pires, and Marten van Sinderen. On the design of user-centric supporting service composition environments. Information Technology: New

Generations, Third International Conference on, pages 666–671, 2010.

[84] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow;. In International Conference Advanced Information Systems Engineering, volume 1789 of Lecture Notes in Computer Science, pages 13–31. Springer Berlin / Heidelberg, 2000.

[85] Dipanjan Chakraborty and Anupam Joshi. Dynamic service composition: State-of-the-art and research directions. Technical report, University of Maryland, 2001.

[86] Freddy Lécué, Eduardo Gonçalves da Silva, and Luís Ferreira Pires. A framework for dynamic web services composition. In Workshop on Emerging Web Services Technology, November 2007.

[87] Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten van Sinderen. Supporting dynamic service composition at runtime based on end-user requirements. In International workshop on user-generated services, November 2009.

[88] Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten Sinderen. Towards runtime discovery, selection and composition of semantic services. Computer Communications, 34(2):159 – 168, 2011. Special Issue: Open network service technologies and applications.

[89] Joao Paulo Almeida, A. Baravaglio, M. Belaunde, P. Falcarin, and E. Kovacs. Service creation in the SPICE service platform. In Wireless World Research Forum meeting on "Serving and Managing users in a heterogeneous environment", November 2006.

[90] Christophe Cordier, Francois Carrez, Herma van Kranenburg, Carlo Licciardi, Jan van der Meer, Antonietta Spedalieri, and Jean-Pierre Le Rouzic. Addressing the challenges of beyond 3G service delivery: the SPICE platform. In International Workshop on Applications and Services in Wireless Networks, 2006.

[91] Mazen Shiaa, Paolo Falcarin, Alain Pastor, Freddy Lécué, Eduardo Gonçalves da Silva, and Luís Ferreira Pires. Towards the automation of the service composition process: case study

and prototype implementations. In Ict-mobilesummit conference, stockholm, sweden, pages 1–8. IIMC International Information Management Corporation, 2008.

[92] Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten Sinderen. An algorithm for automatic service composition. In International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing, pages 65–74, July 2007.

[93] Freddy Lécué. Web Service composition : Semantic Links based approach. PhD thesis, Ecole des Mines de Saint-Etienne, October 2008.

[94] Yasmine Charif and Nicolas Sabouret. An overview of semantic web services composition approaches. Electronic Notes in Theoretical Computer Science, 146(1):33 – 41, 2006. International Workshop on Context for Web Services.

[95] Ulrich Küster, Holger Lausen, and Birgitta König-Ries. Evaluation of semantic service discovery - a survey and directions for future research. In Workshop on Emerging Web Services Technology, November 2007.

[96] Charles J. Petrie, Holger Lausen, and Michal Zaremba. Sws challenge - first year overview. In International Conference on Enterprise Information Systems, pages 407–412, 2007.

[97] Yasser Ganjisaffar and Hadi Saboohi. Semantic web service test collection (sws-tc). http://projects.semwebcentral.org/projects/sws-tc/, 2006.

[98] David Martin, Mark Burstein, Erry Hobbs, Ora Lassila, Drew Mcdermott, Sheila Mcilraith, Srini Narayanan, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. Owl-s: Semantic markup for web services. Technical report, W3C, November 2004.

[99] Ulrich Küster and Birgitta König-Ries. On the empirical evaluation of semantic web service approaches: Towards common sws test collections. In IEEE International Conference on Semantic Computing, pages 339–346, 2008.

[100] Ernest Cho, Sam Chung, and Daniel Zimmerman. Automatic web services generation. In Hawaii International Conference on System Sciences, pages 1–8, 2009.

[101] Seog-Chan Oh, Hyunyoung Kil, Dongwon Lee, and Soundar R. T. Kumara. Wsben: A web services discovery and composition benchmark. In International Conference on Web Services, pages 239–248, 2006.

[102] Ron Kohavi and Foster Provost. Glossary of terms. Machine Learning, 30(2–3):271–274, 1998.

[103] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., 1999.

[104] Eclipse Foundation. Eclipse modelling framework (emf). http://www.eclipse.org/modeling/emf/.

[105] Apache. Apache juddi. http://ws.apache.org/juddi/.

[106] Luc Clement, Andrew HatelyClaus von Riegen, and Tony Rogers. Universal description discovery and integration (uddi) version 3.0. http://uddi.org/pubs/uddi_v3.htm, October 2004.

[107] Sean Bechhofer, Raphael Volz, and P. Lord. Cooking the semantic web with the owl api. In International Semantic Web Conference, pages 659–675, October 2003.

[108] Evren Sirin, Bijan Parsia, Bernardo C. Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. Web Semantics: Science, Services and Agents on the World Wide Web, 5(2):51–53, June 2007.

[109] jGraphT. jgrapht: Java graph library. http://www.jgrapht.org/.

[110] jGraph. jgraph: Graph visualisation. http://www.jgraph.com/.

[111] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley, January 1995.

[112] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In International conference on Software engineering, ICSE 92, pages 392–411. ACM, 1992.

[113] Marlon Dumas and Arthur ter Hofstede. Uml activity diagrams as a workflow specification language. In UML 2001:

The Unified Modeling Language. Modeling Languages, Concepts, and Tools, volume 2185 of Lecture Notes in Computer Science, pages 76–90. Springer Berlin / Heidelberg, 2001.

[114] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine, 2000.

[115] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., 2002.

[116] W3C. Xml schema. http://www.w3.org/XML/Schema, 2001.

[117] Joni Hoppen dos Santos. Public service improvement using runtime service composition strategies. Master's thesis, University of Twente, 2010.

[118] Edwin Vlieg. Usage patterns for user-centric service composition. Master's thesis, University of Twente, 2010.

[119] The PHP Group. Hypertext preprocessor (php). http://www.php.net, 2010.

[120] Trygve Reenskaug. Models Views Controllers. Technical report, Xerox PARC, 1979.

[121] The Rails Core Team. Ruby on rails. http://rubyonrails.org, 2010.

[122] Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten Sinderen. On the support of dynamic service composition at runtime. In International Service Oriented Computing Conference Workshops, volume 6275 of Lecture Notes in Computer Science, pages 530–539. Springer Berlin / Heidelberg, 2010.

[123] Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten Sinderen. A framework for the evaluation of semantics-based service composition approaches. In IEEE European Conference on Web Services, pages 66–74. IEEE Computer Society Press, November 2009.

[124] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Simple object access protocol (soap) version 1.2. http://www.w3.org/TR/soap12-part1/, April 2007.

[125] Jorge Martínez López. Dynamic service composition in an innovative communication environment. Master's thesis, University of Twente, 2008.

Author Publications

Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten Sinderen. Towards runtime discovery, selection and composition of semantic services. Computer Communications, 34(2):159 – 168, 2011. Special Issue: Open network service technologies and applications.

Antonio Madureira, Frank den Hartog, Eduardo Gonçalves da Silva, and Nico Baken. Model for trans-sector digital interoperability. In International Conference on Interoperability for Enterprise Software and Applications, Coventry, UK, pages 123–133. Springer Verlag, April 2010.

Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten van Sinderen. On the design of user-centric supporting service composition environments. pages 666–671. IEEE Computer Society, 2010.

Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten Sinderen. On the support of dynamic service composition at runtime. In International Service-Oriented Computing Conference Workshops, volume 6275 of Lecture Notes in Computer Science, pages 530–539. Springer Berlin / Heidelberg, 2010.

Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten Sinderen. A framework for the evaluation of semantics-based service composition approaches. In IEEE European Conference on Web Services, pages 66–74. IEEE Computer Society Press, November 2009.

Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten van Sinderen. Supporting dynamic service composition at runtime based on end-user requirements. In International workshop on user-generated services, November 2009.

Luiz Bonino da Silva Santos, Giancarlo Guizzardi, Renata Guizzardi-Silva Souza, Eduardo Gonçalves da Silva, Luís Ferreira

Pires, and Marten van Sinderen. Gso: Designing a well-founded service ontology to support dynamic service discovery and composition. In Internation Enterprise Distributed Object Computing Conference Workshops, pages 35–44. IEEE Computer Society, September 2009.

Luiz Bonino da Silva Santos, Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten van Sinderen. Towards a goal-based service framework for dynamic service discovery and composition. In International Conference on Information Technology: New Generations, pages 302–307. IEEE Computer Society, 2009.

Laura Daniele, Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten van Sinderen. A soa-based platform-specific framework for context-aware mobile applications. In IFIP WG5.8 Workshop on Enterprise Interoperability, volume 38 of Lecture Notes in Business Information Processing, pages 25–37. Springer Verlag, 2009.

Rodrigo Mantovaneli Pessoa, Eduardo Gonçalves da Silva, Marten van Sinderen, Dick Quartel, and Luís Ferreira Pires. Enterprise interoperability with soa: a survey of service composition approaches. In International Workshop on Enterprise Interoperability, pages 32–45, September 2008.

Eduardo Gonçalves da Silva, Jorge Martinez López, Luís Ferreira Pires, and Marten van Sinderen. Defining and prototyping a life-cycle for dynamic service composition. In International workshop on architectures, concepts and technologies for service oriented computing, pages 79–90, July 2008.

Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten van Sinderen. Dynamic composition of services: Why, where and how. In International Workshop on Enterprise Systems and Technology, pages 73–85. INSTICC Press, May 2008.

Freddy Lécué, Eduardo Gonçalves da Silva, and Luís Ferreira Pires. A framework for dynamic web services composition. In Emerging Web Services Technology, Volume II, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 59–75. Birkhauser Basel, 2008.

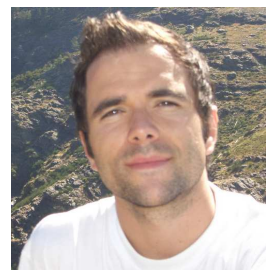
Mazen Shiaa, Paolo Falcarin, Alain Pastor, Freddy Lécué, Eduardo Gonçalves da Silva, and Luís Ferreira Pires. Towards the automation of the service composition process: case study and prototype implementations. In Ict-mobilesummit conference, pages

1–8. IIMC International Information Management Corporation, 2008.

Freddy Lécué, Eduardo Gonçalves da Silva, and Luís Ferreira Pires. A framework for dynamic web services composition. In Workshop on Emerging Web Services Technology, November 2007.

Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten Sinderen. An algorithm for automatic service composition. In International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing, pages 65–74. INSTICC Press, July 2007.

About the Author



Eduardo Manuel Gonçalves da Silva was born in Vila Real, Portugal, in 1981. He obtained his "Licenciatura" degree in Electronics and Computer Engineering from the Faculty of Engineering of the University of Porto (Porto, Portugal) in 2005. He participated in the ERASMUS students mobility program for one year in Gothenburg, Sweden, where he also obtained his Master's degree in Digital Communication Systems and Technology from the Chalmers Technical University. His master thesis was done at the Computer Engineering and Networks Laboratory (TIK), ETH-Zurich (Zurich, Switzerland), in the area of Service Management in Mobile Ad-hoc Networks.

Since 2007 he is a Ph.D. Associate at the Centre of Telematics and Information Technology (CTIT), University of Twente (Enschede, The Netherlands). He has been involved in the FP6 European project IST-SPICE. His research and professional interests are: distributed applications, service-oriented computing, dynamic service composition, and semantic web, with special interest on how to combine these for the design and development of user-centric service composition supporting systems. He was teaching assistant in the course of Service-Oriented Architectures with Web services for three years. He has also supervised three master students on their graduation projects. He co-authored more than fifteen international publications, including workshop and conference papers, journal articles and book chapters, and developed multiple prototypes. Furthermore, he serves as reviewer in several international journals and conferences.

Eduardo is passionate about technology, specially Internet technology. He likes to research and find innovative and pragmatic solutions for concrete problems. On his free time he likes to watch good movies, play football, travel and read.

Notes