Towards Runtime Discovery, Selection and Composition of Semantic Services

Eduardo Gonçalves da Silva, Luís Ferreira Pires, Marten van Sinderen

Centre for Telematics and Information Technology
University of Twente, The Netherlands
P.O. Box 217, 7500 AE Enschede

Abstract

Service-orientation is gaining momentum in distributed software applications, mainly because it facilitates interoperability and allows application designers to abstract from underlying implementation technologies. Service composition has been acknowledged as a promising approach to create composite services that are capable of supporting service user needs, possibly by personalising the service delivery through the use of context information or user preferences. In this paper we discuss the challenges of automatic service composition, and present DynamiCoS, which is a novel framework that aims at supporting service composition on demand and at runtime for the benefit of service end-users. We define the DynamiCoS framework based on a service composition life-cycle. Framework mechanisms are introduced to tackle each of the phases and requirements of this life-cycle. Semantic services are used in our framework to enable reasoning on the service requests issued by end users, making it possible to automate service discovery, selection and composition. We validate our framework with a prototype that we have built in order to experiment with the mechanisms we have designed. The prototype was evaluated in a testing environment using some use case scenarios. The results of our evaluation give evidences of the feasibility of our approach to support runtime service composition. We also show the benefits of semantic-based frameworks for service composition, particularly for end-users who will be able to have more control on the service composition process.

Email addresses: `e.m.g.silva@ewi.utwente.nl` (Eduardo Gonçalves da Silva), `l.ferreirapies@ewi.utwente.nl` (Luís Ferreira Pires), `m.j.vansinderen@ewi.utwente.nl` (Marten van Sinderen)

Key words:
Service-Oriented Computing, User-centric Service Composition, Semantic Services

1. Introduction

With the Internet becoming ubiquitous, the use of network-based application services is being increasingly adopted and it is expected to grow in the upcoming years [1]. This is being reflected in many technology developments and innovations, such as, for example, *Software as a Service* (SaaS) [2], *Internet of Services* [3] and *Cloud Computing* [4]. The proliferation of service-oriented systems [5] is leading to the emergence of large sets of services in different domains. At the same time, the use of mobile devices with fast data connections is increasing quite rapidly. In [6] it is reported that by 2013 more than 38% of the European population will access the Internet on their mobile device, which is an increase of 300% compared to the current situation.

These developments are allowing and *pushing* new, more adaptive and personalised application services where the users play an active role in the process of service creation. This is one of the main motivations behind the *Internet of Services.* However, users are not expected to create new services from scratch but to *aggregate* existing services to fulfil a set of user requirements. Supporting end-users in this kind of runtime service creation process is a complex undertaking. Different users have different preferences and request services in different context situations, which require different actions to be taken. Furthermore, end-users expect a high-level of abstraction in the service creation process, since they lack the technical knowledge to use advanced technical tools. This implies that automation has to be provided to support the end-user in the service creation process. We claim that this can be achieved by using semantic-based service composition approaches. We assume that if no single application service exists to provide a requested service to a user, a new composite service can possibly be created on-demand from existing services, considering the user preferences and context to personalise the service creation process. If a user request cannot be fully matched in the creation process, a partial fulfilment of the user request may be possible or an alternative suggestion that the user is not aware of may be proposed. We denote the creation of service compositions on-demand based on specific requirements as *dynamic service composition.* To support this approach, we developed a framework for dynamic service composition provisioning called *DynamiCoS*.

The *DynamiCoS* (**Dyna**mic **Com**position of **S**ervices) framework addresses all the phases and stakeholders of the dynamic service composition life-cycle. To allow automation of the composition life-cycle, semantic information is used based on ontologies (domain conceptualisations) to which the different framework stakeholders have to comply. The framework allows service developers to publish their semantically annotated services in a formalism that is neutral with respect to description languages and technologies, which consequently enables the use of different semantic service description languages. The composition process is likewise language-neutral, so that services described in different service description languages can be combined in the same service composition. DynamiCoS supports end-users in the service creation process, through automatic discovery, selection and composition of services based on the user service request. We make use of the notion of *goal* to describe and specify the activities (or operations) the services can perform and to capture the requirements the user wants to fulfil when executing a service. We argue that comprehensive frameworks, which address not only composition but also the other supporting phases of the life-cycle, are required to boost the use of automated mechanisms to support service composition, especially at runtime.

This paper is further organised as follows: Section 2 presents our dynamic service composition life-cycle; Section 3 presents our approach to address the dynamic service composition life-cycle, and the details of the DynamiCoS conceptual framework and prototype; Section 4 presents the testing environment and the methodology used to evaluate our approach; Section 5 presents and analyses the evaluation results; Section 6 provides an overview of related work; and Section 7 gives our conclusions and challenges for future work.

2. Dynamic service composition life-cycle

Figure 1 presents our dynamic service composition life-cycle, depicting the phases and stakeholders associated with the service composition process [7].

We consider two stakeholders in this life-cycle: *Service developer*, who publishes new services in the framework, which can be used as basic components on the composition process, or creates new service compositions at design-time using the framework support; and *End-user*, who makes use of

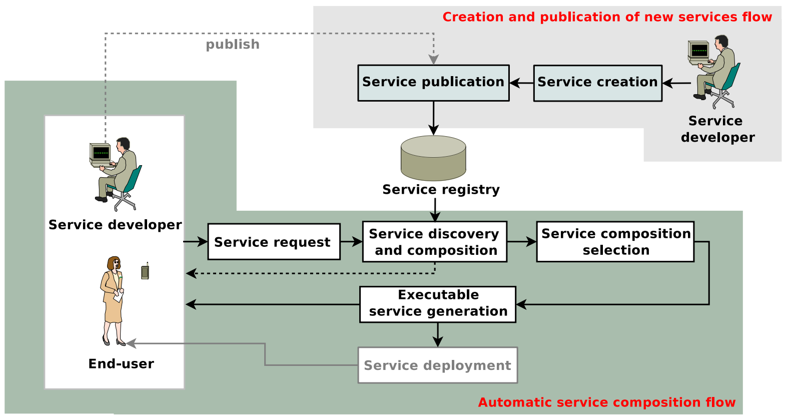

Creation and publication of new services flow
publish
Service publication
Service creation
Service developer
Service registry
Service developer
Service request
Service discovery and composition
Service composition selection
Executable service generation
End-user
Service deployment
Automatic service composition flow

Figure 1: Dynamic service composition life-cycle

automatic service composition mechanisms at runtime. The life-cycle has two main flows, namely *creation and publication of new services* and *automatic service composition*.

The *service creation* phase is performed by a service developer, who creates new services by programming new applications and makes them available as services, or builds a new service composition from existing services and makes the resulting composition available as a new service. The service creation phase also encompasses the definition of the service description document by the service developer. The service description document is then used in the *service publication* phase to publish the service functional and non-functional information in a service registry. The definition of a service description document, and specifically a semantic service description, is compulsory to enable automation in the service composition process. If services are not semantically described, then automation cannot be achieved in the service composition process.

In Figure 1, the *automatic service composition* flow assumes that an end-user or a service developer wants a new service to satisfy some specific requirements. We assume that an end-user has no technical skills on service composition and wishes a new service at *runtime*. A service developer is a user with technical skills on service composition, but wants to create a new service at *design-time*, in a faster (and more automated) way, given some specific requirements for a new service. The first phase of the automatic service composition process is the specification of a *service request*, where the user specifies requirements and preferences for the desired service. Once the service request is defined, the *service discovery and composition* phase takes place. Candidate services are discovered through the service registry interface according to some properties defined by the user. Services that match these properties are discovered from the service registry. Given the set of discovered services, an algorithm takes the user service request into account and builds candidate compositions. Interactions may take place with the user to guide and refine the composition process in case the algorithm cannot deliver compositions that fulfil the service request. Once more than one service compositions are found that match the user service request, the *service composition selection* phase takes place. In the case of an 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. In the case of a service developer, a ranked list of services that match the service request may be returned. The *Executable service generation* phase consists of the creation of an executable representation of the generated service compositions. This is necessary because usually the core composition process is performed using a formal representation of the service compositions that is not executable. In the *Service deployment* phase the selected composition is deployed to allow the execution and delivery of the created service.

3. DynamiCoS framework

DynamiCoS [1] (**Dynami**c **Co**mposition of **S**ervices) is a framework for the provision of dynamic service composition that supports all the life-cycle phases and stakeholders identified in Section 2. Figure 2 shows the DynamiCoS framework architecture, and indicates (between parenthesis) the technologies used in the implementation of the framework components of the DynamiCoS prototype platform. The work presented in this article extends our previous work discussed in [8, 7, 9].

3.1. Main characteristics

Automated support to the service composition process requires that the necessary information is *machine readable* and *understandable* without human intervention. In DynamiCoS, service description, publication, discovery

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

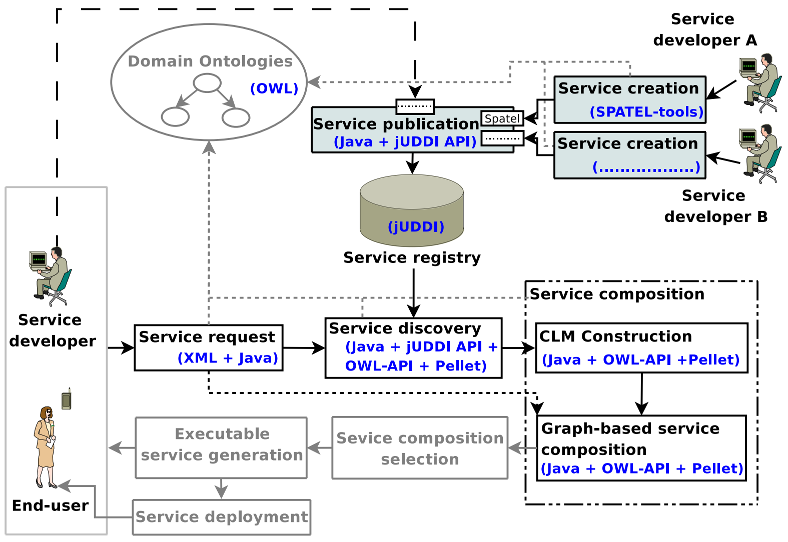


Figure 2: DynamiCoS framework

and composition are performed using semantic service descriptions. DynamiCoS applies ontologies that define the conceptualisations of the application domains being supported. These ontologies are used by all the framework components, and the service developers should have to comply with these ontologies when describing their services. These ontologies may be defined by different stakeholders, for the different domains, aiming at describing (conceptualising) the application domains supported by DynamiCoS.

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. Such representations allow us to create mappings between the specific languages for the representation of service and service compositions, and the internal representation in the framework. Language-neutrality allows different technologies to be used in the framework, as long as the necessary mappings are defined. To allow these mappings, the languages to be supported in the framework require interpreters to collect the necessary information from the service description, and publish it according the service representation formalism of the framework. A service is represented as a seven-tuple $s =< ID, I, O, P, E, G, NF >$, where ID is the service identifier, 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. We assume in this work that services are of type *request-response*, i.e., they consist of one atomic activity (operation). 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 to be coupled with itself.

3.2. Running Example

In order to explain the dynamics of the different components of the framework we introduce below a running example from the e-health and assisted living domain. This scenario is used only for the purpose of demonstration of our framework, and we made some assumptions to simplify the example. We define several services in the domain, namely *FindHospital*, which finds the nearest hospital given a location, *FindDoctor*, which finds a doctor given a hospital and a medical speciality, *LocateUser*, which locates a user given his telephone location and *MakeMedicalAppointment*, which makes an appointment between a patient and a doctor of a given hospital, among others. We wrote semantic service descriptions for these services, using the ontologies defined in the DynamiCoS framework. These semantic service descriptions are published in the service registry of the framework. Based on the knowledge on the domain we define a service request that leads to at least one service composition, namely that fulfils the requirement: *"Make a medical appointment at the nearest hospital"*. In Section 3.5 we give more details on how such a service request can be created.

Throughout the presentation of the DynamiCoS modules, we use this running example to illustrate how DynamiCoS deals with the different phases of the service composition life-cycle.

3.3. Service creation

We assume that whenever a service developer creates a new service "from scratch" 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.

Prototype: In our prototype we have used a language called Spatel [10] to describe services. Spatel is a language developed in the context of the European IST-SPICE project [11], where the development of the DynamiCoS framework was started. Spatel allows one to semantically annotate service operation *inputs*, *outputs*, *preconditions* and *effects*, to define *service goals*, and to define *non-functional properties* of a service. The ontologies used in the framework are described in OWL [12]. We used four ontologies: *Goals.owl*, which contains the services' supported goals and also goals that the user can specify in a service request; *NonFunctional.owl*, which defines non-functional properties to be used in a service description and a service request; *Core.owl* and *IOTypes.owl*, which are used to describe services and service request $IOPE$ parameters. These ontologies were defined in the context of the IST-SPICE project, but the framework is general enough to support the use of other ontologies, according to the domains to be supported.

3.4. Service publication

The DynamiCoS framework has a two-step service publication mechanism, to allow the support of different service description languages. First, there should be an interpreter for each supported service description language. The interpreter reads the service description document and extracts the necessary information for publication (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.

Prototype: Since in our prototype we use Spatel for service creation and description, we implemented a Spatel interpreter in order to import Spatel service description documents to our framework. This interpreter was created by using a Java API generated from the Spatel Ecore model with the Eclipse Modelling Framework (EMF). The service is then published in a UDDI-based service registry that has been extended to support the publication of semantic services. We use jUDDI [13] as service registry implementation, which is a Java-based implementation of the UDDI specification [14] for Web services. 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 of semantic annotations (I, O, P, E, G, NF) that describe a service in our framework.

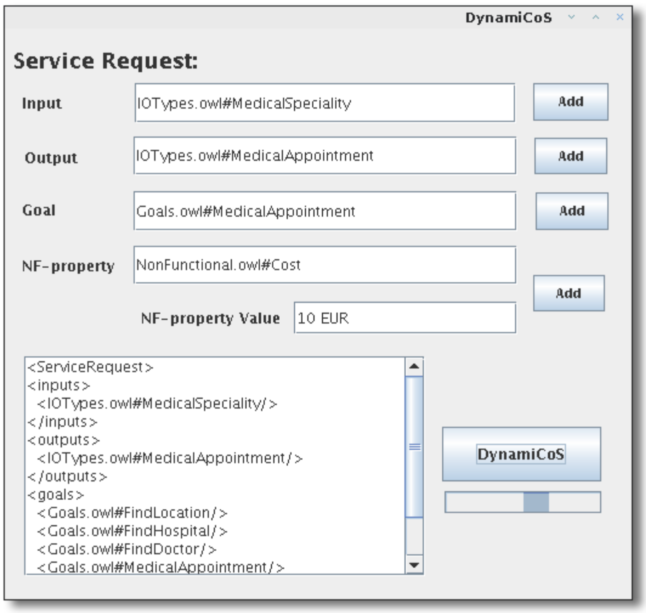
3.5. Service request

A service request consists of a set of semantic annotations (ID, I, O, P, E, G, NF) that describe *declaratively* the desired service properties. These semantic annotations are also used for service representation within the framework, and refer to concepts defined in the framework ontologies. However, to simplify the task of defining the service request, we filter the concepts available in the ontologies, presenting to the user only the concepts used to annotate services that are already published in the registry. This simplifies the definition of a service request. The interface provided to the user for service request specification may vary according to the target users, 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 [11] we have investigated the possibility of using *Natural Language* service requests [15], which gives users without technical knowledge the possibility of requesting services in a simple way.

Prototype: We have implemented two interfaces for the specification of service requests: a simple Java-based graphical interface (Figure 3) and a web-based interface (Figure 4). The aim has been to facilitate the access to the prototype for experimentation and testing. The interface can be accessed from the project webpage (http://dynamicos.sourceforge.net). The information introduced by the user in either interfaces is then transformed to an XML-based document that represents the *desired service*.

Running Example: Considering the running example introduced in Section 3.2, a service request can be created using the interfaces for service request specification. The definition of the service request is performed by the End-user or the Service developer, and to create the service request these users have to know the requirements of their services, 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. This service request can be represented in the following way:

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


DynamiCoS

Service Request:

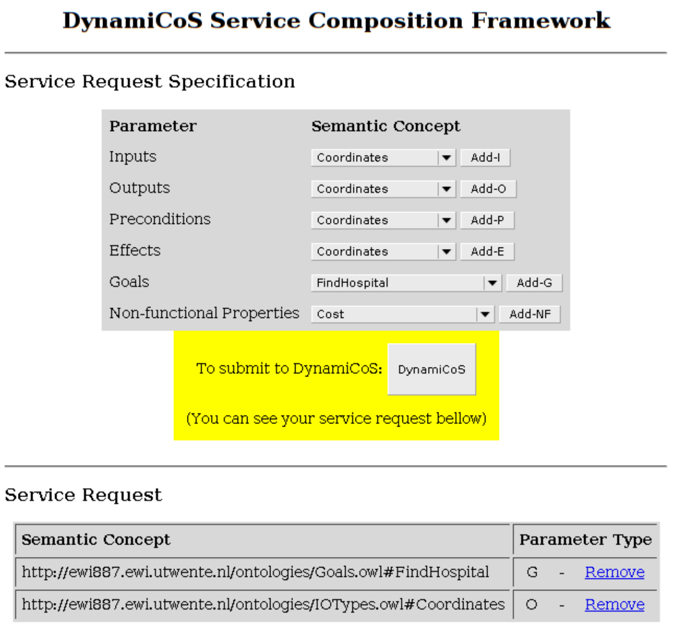
Input	IOTypes.owl#MedicalSpeciality
Output	IOTypes.owl#MedicalAppointment
Goal	Goals.owl#MedicalAppointment
NF-property	NonFunctional.owl#Cost
NF-property Value	10 EUR

```
<ServiceRequest>
<inputs>
  <IOTypes.owl#MedicalSpeciality/>
</inputs>
<outputs>
  <IOTypes.owl#MedicalAppointment/>
</outputs>
<goals>
  <Goals.owl#FindLocation/>
  <Goals.owl#FindHospital/>
  <Goals.owl#FindDoctor/>
  <Goals.owl#MedicalAppointment/>
```

Figure 3: Service Request GUI

3.6. Service discovery

DynamiCoS performs the discovery of 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 information. This assumption follows that the user defines *declaratively* which activities he wants the service to fulfil in the service request. The service discovery process consists of querying the service registry for all the services that *semantically* match the service request inputs, outputs, preconditions, effects ($IOPE$) and goals (G). 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


DynamiCoS Service Composition Framework

Service Request Specification

Parameter	Semantic Concept	
Inputs	Coordinates	Add-I
Outputs	Coordinates	Add-O
Preconditions	Coordinates	Add-P
Effects	Coordinates	Add-E
Goals	FindHospital	Add-G
Non-functional Properties	Cost	Add-NF

To submit to DynamiCoS: DynamiCoS

(You can see your service request bellow)

Service Request

Semantic Concept	Parameter Type
http://ewi887.ewi.utwente.nl/ontologies/Goals.owl#FindHospital	G - Remove
http://ewi887.ewi.utwente.nl/ontologies/IOTypes.owl#Coordinates	O - Remove

Figure 4: Web Service Request Interface

realise the goals that the user has defined for the service. Since DynamiCoS uses a semantic-based service discovery and composition, it discovers exact matches with the service request $IOPE$ and G semantic concepts, but also partial semantic matches are possible, 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, i.e., they make it possible to reason on the descriptions, so that even when two concepts are not equal, they may be related to each other, and thus relation can then be deduced.

Prototype: The service request XML document is analysed, and the $IOPE$ and G annotations are extracted from this document. The service registry is queried using these annotation through the jUDDI API Inquiry function for services with $IOPE$ and G that are semantically related to the service request $IOPE$ and G. To define the semantically related concepts we use the OWL-API [16] and Pellet [17].

Running Example: Considering the XML representation of the service request, if a goal-based service discovery is performed, the following services that semantically match the goals are retrieved:

```
<goal>Goals.owl#FindLocation</goals>
<goal>Goals.owl#FindHospital</goals>
<goal>Goals.owl#FindDoctor</goals>
<goal>Goals.owl#MedicalAppointment</goals>
```

Table 1 shows the services that have been retrieved in our running example.

Service	Input	Output
$locateUser$	IOTypes.owl#CellNumber	IOTypes.owl#Coordinates
$findHospital$	IOTypes.owl#Coordinates	Core.owl#Hospital
$findDoctor$	IOTypes.owl#MedSpeciality Core.owl#MedicalPlaces	Core.owl#Physician
$makeMedAppointment$	Core.owl#Physician Core.owl#Patient	IOTypes.owl#MedAppoint

Table 1: Discovered Services

3.7. Service composition

To perform service composition, DynamiCoS first organises the descriptions of the discovered services in a so called *Causal Link Matrix (CLM)* [18] [8]. The CLM stores all possible semantic connections, or *causal links*, between the discovered services input and output concepts. CLM rows (Equation 1) represent the discovered services input concepts ($DiscServs_I$). CLM columns (Equation 2) represent service inputs concepts plus requested service outputs ($ServReq_O$).

$$CLM_{rows} = DiscServs_I \quad (1)$$
$$CLM_{col} = DiscServs_I \cup ServReq_O \setminus (DiscServs_I \cap ServReq_O) \quad (2)$$

We place a service s in the row i and column j position if the service has an input semantically related with the input i of the CLM and an output semantically related with the column j semantic concept. We store the *Semantic Similarity* for these values in the matrix for each service. Four types of semantic matching are possible:

- **Exact** ($\equiv$) if the output parameter Out_s_y of s_y and the input parameter In_s_x of s_x are equivalent concepts; formally, $\mathcal{T} \models Out\_s_y \equiv In\_s_x$.
- **PlugIn** ($\sqsubseteq$) if Out_s_y is sub-concept of In_s_x; formally, $\mathcal{T} \models Out\_s_y \sqsubseteq In\_s_x$.

- **Subsume** ($\sqsupseteq$) if Out_s_y is super-concept of In_s_x; formally, $\mathcal{T} \models In\_s_x \sqsubseteq Out\_s_y$.
- **Disjoint** ($\perp$) if Out_s_y and In_s_x are incompatible; formally, $\mathcal{T} \models Out\_s_y \sqcap In\_s_x \sqsubseteq \perp$.

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. If this is not true we can request the users for refinements on the service request. The composition process is also simplified, since it consists only of the inspection of the CLM for services that match a given input/output.

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 the service request concepts (IOPE/G). The CLM is then inspected for services that provide the service request outputs. If there are services that provide the service request outputs, the algorithm starts by creating the initial matching nodes, otherwise it stops. If the service request outputs can be provided by the discovered services, 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 output concepts of services from the CLM matrix, or column concepts. 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 in the algorithm, 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 requested inputs, preconditions and goals from all the alternative service compositions are resolved.

Prototype: The CLM matrix is constructed by using the OWL-API [16], which allows one to handle and perform semantic inference in OWL

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 $CLM_{rows \cup colu} \supseteq ServReq_{I,O}$ then
       // Create new graph
 6     $activeG \leftarrow createNewGraph()$;
 7     $createInitialNodes()$;
 8     $openG \leftarrow activeG$;
 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 $activeG_{I,G} \supseteq ServReq_{I,G}$ then
13         $validG \leftarrow activeG$;
14         $openG \leftarrow openG \setminus activeG$;
15         $activeG \leftarrow openG^0$;
16         $activeN \leftarrow activeG_{openN^0}$;
17         break; // Goes to next $openG$
       // Checks CLM for services that match open inputs
18     foreach $semCon \in activeN_I$ do
19         if $CLM_{colu} \supseteq semCon$ then
20             $activeN \leftarrow CLMmatchingNode$;
21         else
22             $openG \leftarrow openG \setminus activeG$;
23             $activeG \leftarrow openG^0$;
24             $activeN \leftarrow activeG_{openN^0}$;
25             break; // No possible composition, goes to next open graph
       // Check if graph NF props comply with requested NF props
26     if $activeG_{NF} \cap ServReq_{NF} = \emptyset$ then
27         $openG \leftarrow openG \setminus activeG$;
28         break; // If Not, composition is not possible
       // prepare next cycle
29     $openN \leftarrow openN \setminus activeN$;
```

ontologies through a semantic reasoner, in our case Pellet [17], which is a reasoner for Description Logics (DL). The service composition algorithm is implemented in Java. We use jGraphT [2] to print out the resulting service

[2]http://jgrapht.sourceforge.net/

compositions (graphs).

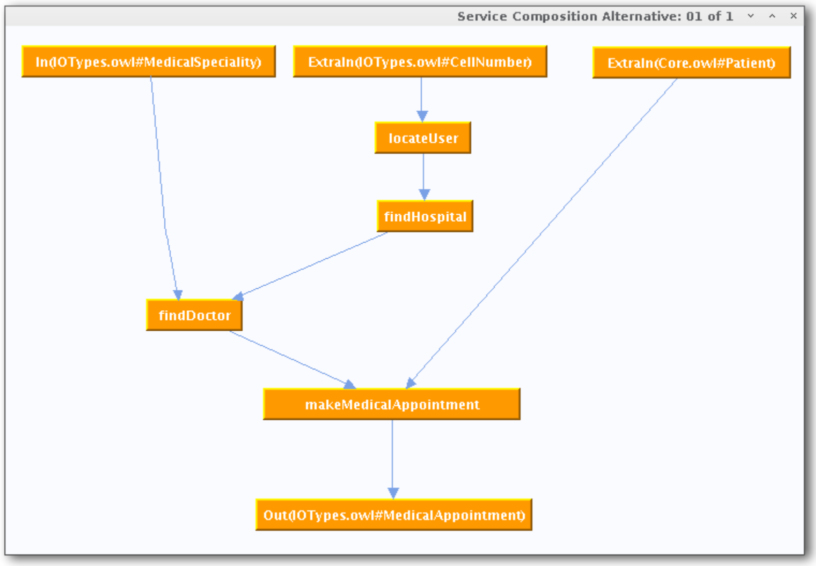
Running Example: For the set of discovered services of our running example, the CLM matrix in Table 2 has been constructed.

	cellNumber	Coordinates	MedSpeciality	MedPlaces	Patient	Physician	MedAppoint
CellNumber	0	$S1$,$\equiv$	0	0	0	0	0
Coordinates	0	0	0	$S2$,$\sqsubseteq$	0	0	0
MedSpeciality	0	0	0	0	0	$S3$, $\equiv$	0
MedPlaces	0	0	0	0	0	$S3$, $\equiv$	0
Patient	0	0	0	0	0	0	$S4$,$\equiv$
Physician	0	0	0	0	0	0	$S4$,$\equiv$

Service ID	Service Name
$S1$	locateUser
$S2$	findHospital
$S3$	findDoctor
$S4$	makeMedAppointment

Table 2: Causal Link Matrix

Figure 5 shows the service composition generated by our composition algorithm based on the CLM in Table 2.



Service Composition Alternative: 01 of 1
In(IOTypes.owl#MedicalSpeciality)
ExtraIn(IOTypes.owl#CellNumber)
ExtraIn(Core.owl#Patient)
locateUser
findHospital
findDoctor
makeMedicalAppointment
Out(IOTypes.owl#MedicalAppointment)

Figure 5: Generated service composition

4. Evaluation

We evaluated our framework through a prototype implementation. In the sequel we present the environment where we conducted the evaluation experiments, and the strategy, scenarios and metrics used in the evaluation process. We aimed at evaluating whether dynamic service composition is feasible with our approach and how our approach scales when the number of available services increases.

4.1. Evaluation environment

Figure 6 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, 2GHz), which performs all the service discovery, composition and delivery actions and the *DynamiCoS Registry* (Intel Pentium 4 2.4GHz, 512MB), which stores the services published by service developers.

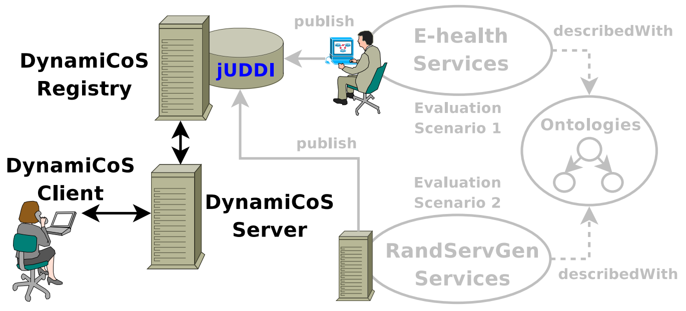


Figure 6: Evaluation environment

We have programmed the different components of the framework in Java. This allowed us to reuse many existing tools, such as: jUDDI, OWL-API and Pellet. 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.

4.2. Evaluation strategy

The evaluation of semantic-based service composition approaches is a topic not yet extensively addressed in the literature. The most relevant efforts somehow related to this kind of evaluation are the Semantic Web Services Challenge (SWS-Challenge)[3] [19] and Service Semantic Service Se-

[3]http://sws-challenge.org

lection (S3)[4] Contest. They provide sets of semantic services, which may be considered as a starting point to define collections to evaluate semantic-based service composition approaches. However, their focus is not originally on the evaluation of semantic-based service composition approaches, since the SWS-Challenge focuses on the evaluation of the mediation problem, while the S3-Contest mainly focuses on the problem of service discovery and matchmaking. The S3-Contest is the approach that provides the largest set of services (around one thousand semantic services). Both initiatives use service collections based on existing services or new services developed from scratch that execute some functionality. An alternative way to deal with the creation of service collections for the evaluation of semantic services collections is to generate services automatically, based on some user specifications and a set of ontologies, which can be used to semantically annotate the service parameters. To evaluate a service composition approach in a given services collection, service requests are also required to specify which service compositions are requested to the service composition approach. These service requests have to lead to some service composition, so that we can verify whether a composition approach is able to find suitable compositions or not.

4.3. Evaluation Scenarios

Based on the discussion above we have defined the following evaluation scenarios:

- *Evaluation Scenario 1 - Manually defined services*: services in the registry are all created and semantically annotated by *humans* (service developers). We assume in this scenario that all the services are meaningful. We have defined a set of 15 services in this scenario;

- *Evaluation Scenario 2 - Automatically generated services*: services in the registry are automatically generated by a tool we have implemented (*RandServGen*). 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 $IOPE$, according to limits specified by the tool user. This tool is implemented in Java, and creates SPATEL semantic service descriptions, which can then be published in the framework registry.

[4]http://www-ags.dfki.uni-sb.de/ klusch/s3/

The selection of semantic concepts for the service interface parameters (IOPEs) and service goals (G) is random, i.e., we collect the ontologies concepts and select randomly concepts from this collection. 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.

For both scenarios, the services we generated are from the e-health and assisted living domain. Some of the services from the evaluation scenario 1 are mentioned in Section 3.2. The service request used in the evaluation is the same as the one defined in Section 3.2.

4.4. Evaluation Metrics

The following metrics were used in the evaluation process:

- *Number of discovered services (#discServs)*: number of services discovered and used in the composition process;
- *Number of relevant services (#releDiscServs)*: number of discovered services (*#discServs*) used in the generated service compositions (*servComps*);
- *Number of IOPE (#IOPE)*: total number of inputs, outputs, preconditions and effects operation parameters of the discovered services. These are all the semantic concepts that can be composed in the service composition process;
- *Dynamic service composition time (dynaServCompT)*: sum of the time required to process a user service request (*servReqProcT*), to perform the discovery of services (*servDiscT*) and to perform composition (*servCompT*), as defined in Equation 3.

$$dynaServCompT = servReqProcT + servDiscT + servCompT \tag{3}$$

5. Results and analysis

To evaluate the performance of our framework on different situations we used the service request defined in Section 4.3 and varied the number of available services in the service registry. This allowed us to reason on the scalability of the composition process and to characterise the phases of the dynamic service composition process in terms of required processing time. In the first experiment we considered only the services defined manually, then we increased the number of available services in the registry by 50 in each experiment iteration. All the experiments have been repeated 20 times. The results presented in this section are the average values of the measured values of the evaluation metrics on the performed experiments.

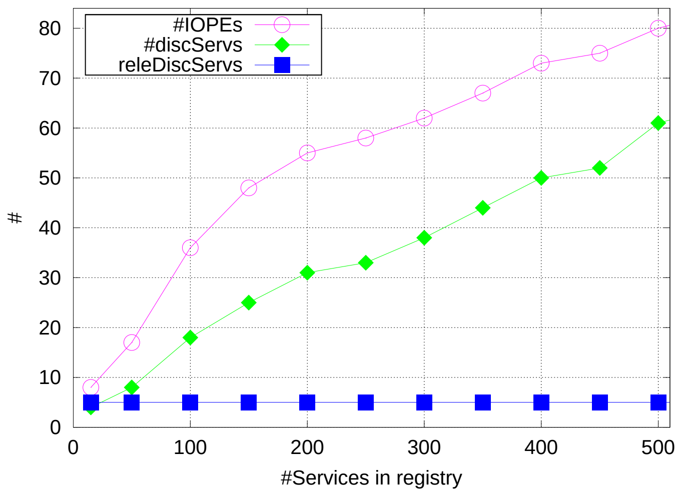


Figure 7: $IOPE$ and discovered services

In Figure 7 shows that the number of discovered services ($\#discServs$) and the total number of $\#IOPE$ parameters of these services increase with the number of available services. This almost *linear* increase is in our opinion not completely realistic, since just a very few number of discovered services is *relevant* for the created service compositions ($\#releDiscServs$). This happens because the automatically generated services combine randomly all the concepts of the used ontologies. The result of this process is a random set of services that are not meaningful for the service composition process, i.e., these services are not used in the generated service compositions. This exposes the main drawback of using automatic ($random$) generation of semantic service collections. In a more realistic situation, services with common goals would be annotated with concepts that are semantically close to each other, so that the number of $IOPE$ semantic concepts to handle and the dynamic service composition time would decrease.

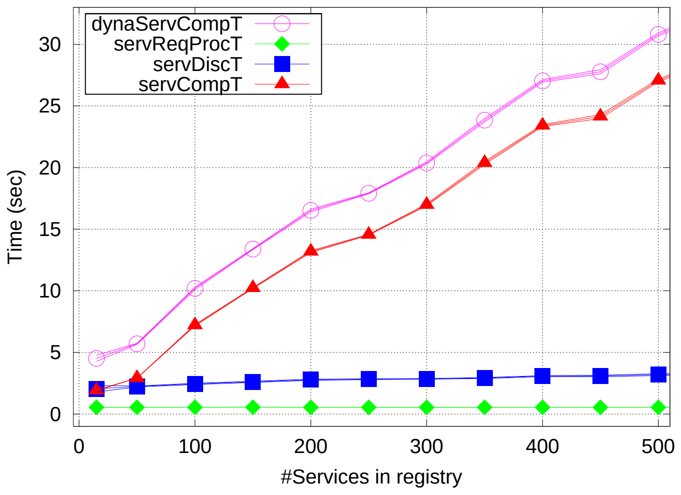


Figure 8: Composition time

However, the automatically generated service collection allows us to evaluate how much processing is required in the different phases of the automatic service composition process. Figure 8 shows the three different phases of the automatic service composition process, represented as the processing time taken in each of these phases: *service request* ($servReqProcT$); *service discovery* ($servDiscT$); and *service composition* ($servCompT$). These times represent the average measured values and their standard deviation, which was very small. The $servReqProcT$ remains constant, given that the same service request is used in all the performed experiments. The service discovery time ($servDiscT$) has a small increase as the number of services in the registry increases, since more services are handled and retrieved. However, the biggest increase on the processing time is observed in the composition time ($servCompT$). This increase can be justified in Figure 7, which shows the number of discovered services ($\#discServs$) and number of services' $IOPE$ ($\#IOPE$) considered in the composition process. Figure 7 shows that as the number of services' $IOPE$ ($\#IOPE$) increases, more processing is necessary to create the CLM and to perform the composition algorithm, since more concepts are handled in the composition process, i.e., more semantic reasoning is performed. We can conclude from these results that the required semantic reasoning introduces the biggest overhead in the framework automatic service composition process. Given this, we expect that enhancements in the parts that rely on semantic reasoning may improve the overall performance of the system.

Nevertheless, if we consider the case of only meaningful services (the most realist situation), compositions that match the user services request can be found in reasonable time. This proves the feasibility of our approach.

6. Related work

Dynamic service composition has received a lot of attention lately. Most of the existing approaches focus on a subset of the life-cycle phases presented in Section 2, mainly on the discovery and composition phases, while a few of them cover most of these phases. Furthermore, most approaches assume that the users of the composition environment have technical knowledge about the system, i.e., details of its interface and the composition process. In the following we present some important related work. However, many other approaches exist that may be related with DynamiCoS. We refer to [20, 21, 22, 23] for a comprehensive overview on existing approaches on service composition, specially dynamic and automatic based service composition approaches.

METEOR-S [24] is one of the most comprehensive frameworks for semantic-based service composition. The approach provides mechanisms for semantic annotation of existing services, service discovery, and service composition. However, METEOR-S focuses mainly on design-time creation of service compositions. It supports a *template-based* composition of processes based on the semantic service descriptions. At runtime, dynamic binding can be performed, depending on user preferences or QoS parameters. Our approach, as many others, has been inspired by some facilities of METEOR-S, but we target *on demand* runtime service composition creation, aiming at supporting end-users on the creation of service compositions at runtime. The creation process is then performed from scratch at runtime, based on a specific end-user requirements, as opposed to METEOR-S, which allows its users to develop service composition templates at design-time.

Fujii and Suda [25] propose a dynamic service composition approach that uses semantic information. They introduce a model (CoSMoS) to semantically represent services at different levels, namely at the data, semantic and logic levels. CoSMoS descriptions can be used as metadata to represent a service, or can be embedded in other languages, such as, e.g., WSDL [26].

The authors assume that a developer has to deliver a CoSMoS-based service description. In DynamiCoS, a service developer does not have to be aware of the framework's internal service representation. Our only requirements are that (i) the semantic service description language must follow the framework ontologies to semantically describe the services; and (ii) this language must have a DynamiCoS interpreter, so that it can be parsed and published in the framework. We consider that this separation of concerns is essential, and provides a more flexible and extensible framework, facilitating the support of additional languages.

Kona et al. [27] propose an approach to the automatic composition of semantic web services. Similarly to DynamiCoS, they propose 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 $IOPE$ 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 a service registry is available, and services are represented in USDL [28]. 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, mainly for the case of runtime service composition. Another difference is that DynamiCoS allows to describe services in terms of *goals*. The same conceptualisations are used to specify the user *intentions* for the services. This facilitates the discovery of matching services for the user request.

In [29] and [30], the authors explore the use of semantic services as basic constructs in pervasive environments. The different devices and functionality in the environment are exposed as services, and then these services are composed as they are required, to match the user needs. This approach differs from the approaches discussed before in the sense that they are limited to service delivery in pervasive environments. This allows the approach to identify the 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, possibly as composition of services available in their environment, whenever they need them. We follow a similar philosophy in DynamiCoS, and we specially focus on the type/properties of the user being supported in the design process. User-centricity is required to deliver the user with appropriated support, so that end-users can achieve their objectives. End-users are starting to play an increasingly active role on the definition of their services.

Several approaches are emerging to support end-users in the service composition process based on mashups [31, 32, 33, 34, 35]. These approaches offer intuitive graphical service representations that allow end-users to create their own services as compositions, normally in a web browser. We argue that these approaches are applicable in some composition scenarios for some types of end-users, but mainly knowledgeable users. If the end-user of the composition environment has some technical knowledge on the composition environment, has a clear idea of the service he wants and knows the application domain, these environments may be appropriate. However, if the end-user does not have a clear idea of the service he wants, but only knows the goals that the desired service has to fulfil, an approach similar to DynamiCoS may be more appropriate.

7. Final Remarks

In this paper we present our work on the support to runtime service delivery through automated service composition. Some approaches are emerging in which end-users are placed in the centre of the service creation, on demand at runtime. We present DynamiCoS, which is a framework for the dynamic and automated composition of services. Many frameworks that target the support of dynamic service composition have been proposed [20] [22], however most of these approaches mainly focus on the discovery and composition phases, targeting design-time service composition. DynamiCoS is a modular, extensible and service description language-neutral framework, which provides support to the whole dynamic service composition process. To develop DynamiCoS we have identified the different stakeholders and the required phases in the dynamic service composition life-cycle. We developed a conceptual framework to address these phases. DynamiCoS supports service creation and publication by service developers at design-time, the automatic service composition by end-users at runtime, and by service developers at design-time. Based on our conceptual framework, we developed a prototype implementation. We performed several tests on our prototype implementation to evaluate its performance and to validate the framework. We demonstrated that DynamiCoS is capable of providing *real-time* service delivery in case set of human-defined *meaningful* services are available to be used as component services in the service compositions. We conclude that automatic service composition using semantic-based techniques can be achieved in *controlled* environments. However, semantic reasoning is an expensive task in terms of processing time. We claim that semantic reasoning and the definition of ontologies are still the major obstacles to the usage of semantic-based service composition approaches. However, with this work we show that such semantic-based mechanisms facilitate the creation of automatic service composition at runtime, providing mechanisms to allow end-users to play a central role in the creation process of their services.

In the future we will investigate further the use of non-functional properties and the user context and preferences in the composition and composition selection processes. This information is expected to enable further optimisation and personalisation to the composition process. To collect this information, an infrastructure must be developed that gathers the user context transparently for the user. Furthermore, since our aim is to support end-users in the process of runtime service composition, we will investigate mechanisms to *guide* the user on the specification of the behaviour of the desired service, instead of asking the user for all the information in a single interaction. This is important, since the end-user normally is not capable of fully specifying the desired service, without having some feedback of available services and functionality. To tackle this, the process will require several interactions and a negotiation to match the user interests and the available services that may fulfil these interests. The generated service compositions need to be translated to an executable formalism, enabling a full runtime environment to be built. We will address the translation of our graph representation to an executable language, such as, e.g., WS-BPEL [36]. We plan to support other service description languages, such as, e.g., SAWSDL [37] and OWL-S [38], to implement more elaborate service composition scenarios and test the framework performance under these conditions. By extending the set of the supported languages we expect to obtain more realistic and *meaningful* testing services, by reusing existing services described in these languages. Furthermore, we also intend to extend the evaluation strategy proposed in this paper, aiming at the development of a generic evaluation framework for semantic-based service composition approaches. Our initial ideas towards this evaluation framework have been presented in [39]. This evaluation framework will allow us to evaluate and compare different approaches in an objective way. This should foster the identification of service composition issues and the help consolidating of this research area.

References

[1] Gartner, Gartner Highlights Key Predictions for IT Organisations and Users in 2008 and Beyond, 2008.

[2] M. Turner, D. Budgen, P. Brereton, Turning Software into a Service, Computer 36 (10) (2003) 38–44.

[3] M. Papazoglou, K. Pohl, Longer Term Research Challenges in Software and Services, Tech. Rep., European Commission, 2008.

[4] B. Hayes, Cloud computing, Commun. ACM 51 (7) (2008) 9–11.

[5] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall, 2005.

[6] Forrester, European Mobile Forecast: 2008 To 2013, 2008.

[7] E. Gonçalves da Silva, J. M. López, L. Ferreira Pires, M. J. van Sinderen, Defining and Prototyping a Life-cycle for Dynamic Service Composition, in: International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing, Portugal, 79–90, 2008.

[8] F. Lécué, E. Gonçalves da Silva, L. Ferreira Pires, A Framework for Dynamic Web Services Composition, in: Workshop on Emerging Web Services Technology, Germany, 2007.

[9] E. Goncalves da Silva, L. Ferreira Pires, M. J. van Sinderen, Supporting Dynamic Service Composition at Runtime based on End-user Requirements, in: Proceedings of the International Workshop on User-generated Services, UGS 2009, part of ICSOC 2009, Stockholm, ISSN 1613-0073, 2009.

[10] J. P. Almeida, A. Baravaglio, M. Belaunde, P. Falcarin, E. Kovacs, Service Creation in the SPICE Service Platform, in: Wireless World Research Forum meeting on "Serving and Managing users in a heterogeneous environment", 2006.

[11] C. Cordier, F. Carrez, H. van Kranenburg, C. Licciardi, J. van der Meer, A. Spedalieri, J.-P. L. Rouzic, Addressing the Challenges of Beyond 3G Service Delivery: the SPICE Platform, in: International Workshop on Applications and Services in Wireless Networks, 2006.

[12] M. K. Smith, D. McGuiness, R. Volz, C. Welty, Web Ontology Language (OWL) guide, version 1.0, W3C, URL `http://www.w3.org/TR/2002/WD-owl-guide-20021104/`, 2002.

[13] Apache, Apache jUDDI, http://ws.apache.org/juddi/, URL `http://ws.apache.org/juddi/`, 2008.

[14] L. Clement, A. H. von Riegen, T. Rogers, Universal Description Discovery and Integration (UDDI) Version 3.0, http://uddi.org/pubs/uddi_v3.htm, 2004.

[15] M. Shiaa, P. Falcarin, A. Pastor, F. Lécué, E. Goncalves da Silva, L. Ferreira Pires, Towards the automation of the service composition process: case study and prototype implementations, in: ICT-MobileSummit 2008 Conference Proceedings, Stockholm, Sweden, IIMC International Information Management Corporation, 1–8, 2008.

[16] S. Bechhofer, R. Volz, P. Lord, Cooking the Semantic Web with the OWL API, in: International Semantic Web Conference, 659–675, 2003.

[17] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2) (2007) 51–53.

[18] F. Lécué, A. Léger, A formal model for semantic Web service composition, in: International Semantic Web Conference, LNCS, vol. 4273, 385–398, 2006.

[19] C. J. Petrie, H. Lausen, M. Zaremba, SWS Challenge - First Year Overview, in: International Conference on Enterprise Information Systems, 407–412, 2007.

[20] A. Alamri, M. Eid, A. E. Saddik, Classification of the state-of-the-art in dynamic web services composition techniques, Int. J. Web Grid Serv. 2 (2) (2006) 148–166.

[21] S. Dustdar, W. Schreiner, A survey on web services composition, Int. J. Web Grid Serv. 1 (1) (2005) 1–30.

[22] J. Rao, X. Su, A Survey of Automated Web Service Composition Methods, in: International Workshop on Semantic Web Services and Web Process Composition, 43–54, 2005.

[23] R. M. Pessoa, E. Goncalves da Silva, M. J. van Sinderen, D. A. C. Quartel, L. Ferreira Pires, Enterprise Interoperability with SOA: a Survey of Service Composition Approaches, in: International Workshop on Enterprise Interoperability, 32–45, 2008.

[24] K. Verma, K. Gomadam, A. P. Sheth, J. A. Miller, Z. Wu, The METEOR-S Approach for Configuring and Executing Dynamic Web Processes, Tech. Rep., University of Georgia, Athens, URL `http://lsdis.cs.uga.edu/projects/meteor-s/techRep6-24-05.pdf`, 2005.

[25] K. Fujii, T. Suda, Dynamic service composition using semantic information, in: International Conference on Service Oriented Computing, New York, NY, USA, 39–48, 2004.

[26] R. Chinnici, J.-J. Moreau, A. Ryman, S. Weerawarana, Web Services Description Language (WSDL) Version 2.0, http://www.w3.org/TR/wsdl20/, 2007.

[27] S. Kona, A. Bansal, G. Gupta, Automatic Composition of SemanticWeb Services, in: International Conference on Web Services, 150–158, 2007.

[28] A. Bansal, S. Kona, L. Simon, T. D. Hite, A Universal Service-Semantics Description Language, in: European Conference on Web Services, Washington, DC, USA, 214, 2005.

[29] S. Ben Mokhtar, A. Kaul, N. Georgantas, V. Issarny, Efficient semantic service discovery in pervasive computing environments, in: Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, Springer-Verlag New York, Inc., New York, NY, USA, 240–259, 2006.

[30] S. Ben Mokhtar, A. Kaul, N. Georgantas, V. Issarny, COCOA : ConversationBased Service Composition for Pervasive Computing Environments, International Conference on Pervasive Services 0 (2006) 29–38.

[31] X. Liu, G. Huang, H. Mei, A User-Oriented Approach to Automated Service Composition, in: Web Services, 2008. ICWS '08. IEEE International Conference on, 773–776, 2008.

[32] J. Han, Y. Han, Y. Jin, J. Wang, J. Yu, Personalized Active Service Spaces for End-User Service Composition, in: Services Computing, 2006. SCC '06. IEEE International Conference on, 198–205, 2006.

[33] J. C. Yelmo, R. Trapero, J. M. Álamo, J. Sienel, M. Drewniok, I. Ordás, K. Mccallum, User-Driven Service Lifecycle Management — Adopting Internet Paradigms in Telecom Services, in: ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing, Springer-Verlag, Berlin, Heidelberg, 342–352, 2007.

[34] A. Ro, L. S.-Y. Xia, H.-Y. Paik, C. H. Chon, Bill Organiser Portal: A Case Study on End-User Composition, in: WISE '08: Proceedings of the 2008 international workshops on Web Information Systems Engineering, Springer-Verlag, Berlin, Heidelberg, 152–161, 2008.

[35] T. Nestler, Towards a mashup-driven end-user programming of SOA-based applications, in: iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, ACM, New York, NY, USA, 551–554, 2008.

[36] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana, Business Process Execution Language for Web Services, version 1.1, 2003.

[37] J. Kopecký, T. Vitvar, C. Bournez, J. Farrell, SAWSDL: Semantic Annotations for WSDL and XML Schema, in: IEEE Internet Computing, vol. 11, 60–67, 2007.

[38] D. Martin, M. Burstein, E. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, K. Sycara, OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S, URL `http://www.w3.org/Submission/OWL-S/`, 2004.

[39] E. Goncalves da Silva, L. Ferreira Pires, M. J. van Sinderen, A Framework for the Evaluation of Semantics-based Service Composition Approaches, in: Proceeding of IEEE European Conference on Web Services, ECOWS 2009, Eindhoven, IEEE Computer Society Press, Los Alamitos, 66–74, 2009.