Dynamic Service Composition in an Innovative Communication Environment

University of Twente
Enschede - The Netherlands

Jorge Martínez López

Software Engineering.
Faculty of Electrical Engineering, Mathematics and Computer Science.

University of Twente

Thesis submitted for the degree of

M.Sc. Telematics

October 2008

Graduation Committee:

1. Dr. Luís Ferreira Pires (EEMCS / SE)

2. Dr. Ir. Marten van Sinderen (EEMCS / IS)

3. M.Sc. Eduardo M. Gonçalves da Silva (EEMCS / SE)

Abstract

With the increasing usage of mobile telecommunication devices we are observing the emergence of new approaches to ease the development and deployment of brand new innovative telecommunication services. One of the proposed approaches is the creation of new services by composing already existing ones, we call this process *service composition*. Service composition facilitates the creation of innovative services that suit the end-user needs, with a reduced time-to-market and better component reuse.

The most common approach to perform service composition is at design time, by a service developer. But with the emergence of new technologies, such as semantically annotated services, one can envision new ways to perform service composition. If the process of service composition is performed on demand, automatically, triggered by a user service request, it is considered as dynamic service composition.

To identify all the requited phases in a dynamic service composition process, a service creation-life cycle was created. In this Master project we developed mechanisms to support the service publication and service discovery phases of the life-cycle. These mechanisms were integrated with the dynamic service composition framework developed at the University of Twente.

To test the proposed mechanisms and evaluate their performance, we created a library of services. These evaluations allowed us to conclude the suitability of the proposed mechanisms to support dynamic service composition.

The work performed in this Master thesis has been done in the context of the European FP6 SPICE Project.

A mis padres.

A Marta.

Acknowledgements

This work is supported by the European SPICE project (IST-027617) and the Dutch Freeband A-MUSE project (BSIK 03025).

Preface

I would like to say thanks the people that have helped and supported me in this Master project.

First of all, I would like to thank my project supervisors at the University of Twente: Eduardo Silva, Luis Ferreira Pires and Marten van Sinderen, for their valuable help, advice and guidance during my work in this project. I also want to thank my supervisor at the Universidad Carlos III de Madrid, Iria Estevez for her interest in the project and her kindness.

Thanks also to my parents for supporting me during my study period in the UT. Thanks to my dear Marta, for her emotional support, specially in the tough moments, and for the wonderful time we are having together. There are many other people I would also like to thank, in no special order: Jan Schut, for "opening the door" of the university; my big family, and specially my grandparents Isaac, Concha, Marcos and Visi; my former ESN boardmates, for their "pallomeri" spirit; my studymates Edu and Jorrit, for being a good reason to go to the lectures; some of my former flatmates, for the nice times with "gezelligheid en Grolsch"; the big 399 family; my labmates, for their company during endless hours; the former ASNA group, for sharing more than the lunch time; and at last, but not at least, my friends everywhere in this small world.

Contents

List of acronyms

ACE	Automatic Composition Engine
API	Application Programming Interface
AS	Application Server
CLM	Casual Link Matrix
EMF	Eclipse Modeling Framework
GQIO	Goals, QoS, Inputs and Outputs
GQIOPE	Goals, QoS, Inputs, Outputs, Preconditions and Effects
IDE	Integrated Development Environment
IMS	IP Multimedia Subsystem
IOPE	Inputs, Outputs, Preconditions and Effects
J2EE	Java 2 Enterprise Edition
MDA	Model-Driven Architecture
NFP	Non-Functional Properties
OMG	Object Management Group
OWL	Web Ontology Language
PIM	Platform-Independent Model
PSM	Platform-Specific Model
PSTN	Public Switched Telephone Network
PSTN	Public Switched Telephone Network
QoS	Quality of Service
SAWSDL	Semantic Annotations for WSDL

LIST OF ACRONYMS

SCE	Service Creation Environment
SEE	Service Execution Environment
SER	SIP Express Router
SIP	Session Initiated Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SPATEL	SPice Advanced service description language for TELecommunication services
SPICE	Service Platform for Innovative Communication Environment
UDDI	Universal Description, Discovery and Integration
UML	Unified Modelling Language
URL	Uniform Resource Locator
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Service Description Language
WSML	Web Service Modeling Language
WSMO	Web Services Modeling Ontology
WTP	Web Tools Platform

1

Introduction

This chapter gives an overview of the thesis background. Section 1.1 presents the motivation. Section 1.2 introduces the context of this project, which is embedded in the European SPICE (Service Platform for Innovative Communication Environment) project. The goals of the Master project are described is section 1.3, and the followed approach is presented in 1.4. The structure of this thesis is presented in section 1.5.

1.1 Motivation

The increasingly wide usage of mobile communications and Internet indicates the growing importance of telecommunications in everyone's life. As a consequence of this importance governments are changing the legislation in order to increase competition in the telecommunications sector, allowing the emergence of new operators in the market. For instance, there are many new Mobile Virtual Network Operators (MVNOs) that use access networks of other operators. The fast development of new features in computing and communication devices and software has enabled the creation of new attentive, personalised and context-aware services which were not available before. For instance, now it is possible to make video calls, browse the Internet, communicate via instant messaging and use presence services in mobile devices. As a result, the revenue generated by telecom operators is increasingly dependent on new services. Moreover there is a big pressure to increase this revenue, due to the operators huge investment in network equipment and 3G licenses auctions.

For this new market circumstances a suitable definition of "service" is given in [21]: "a service is a provider/client interaction that creates and captures value". This definition is quite broad as it can be applied to several domains like medicine, commerce and agriculture. There are also many facets of a service that can be considered, like business, technical, logistic, commercial, etc. In this thesis the provider/client interactions are expected to happen in computer systems, with special stress on mobile platforms.

Much effort has been put at the creation of tools that ease the creation, development and deployment of new services. Operators are seeking for the next "killer service" that will meet some users' demand and make a big profit. Well known examples of "killer services" are e-mail and instant messaging. Some research is being done on the "killer environment" that would allow such a "killer service" to be developed and deployed. An environment consists of a set of tools that facilitate the development task as, for example, in Java development the Eclipse Platform provides an Integrated Development Environment (IDE) [48]. Eclipse can be considered a "killer environment" due to its huge success and acceptance (Forrester claims that more than 65% of Java developer teams use Eclipse [54]) and Eclipse has been extended with plugins that allow software development in other languages like C++ and PHP. Software development, testing and execution tools are provided by Eclipse in its integrated and extendable platform. A similar environment is desirable for service creation and execution.

Some of the key features that need to be provided by this desirable service creation and execution environment are:

- *Flexibility*: this environment should not impose unnecessary restrictions on the services that can be developed and has to follow open standards to grant interoperability.
- *Personalisation*: end-users should be able to create and execute services customised to his needs. Every end-user is different and so are his needs.
- *Agility*: reduce the time-to-market for new services, allowing business to rapidly create new innovative services. A way to achieve agility is reusing existing services. Services could be built upon already available well-tested more basic service components.

1.2 Context

One of the trends in the software development industry nowadays is service-orientation, namely the principles specified in the Service-Oriented Architecture (SOA). In SOA, the foundation for the architecture is the service concept. In the SOA context, a service [19]:

- is a logical representation of a business activity, e.g., billing customer, searching for flight schedule or making an offer request;
- is self-contained;
- may be composed with other services.

1.2.1 Web Services

Nowadays the most popular technology to build SOA-based systems is Web Services [16, 18]. A Web Service could be defined as an executable component that can be accessed remotely using a protocol (i.e., SOAP). The interaction capabilities of a web service are specified in an interface described in a format understandable by a software agent. The actual implementation is hidden for the other parties (i.e. a Web service acts as a black box). Web Services are normally defined in WSDL (Web Service Description Language) [5]. WSDL describes the abstract messages (input and output parameters) that can be exchanged between entities. It also indicates how to access the service in terms of network protocol and message format (binding) [18]. Thus, WSDL describes the interface of the web service and the way it can be invoked (access point or end-point). For telecommunications services like voice calls, messaging and presence, there are standardised Web Services interfaces called Parlay X [13, 49], which are described in a set of WSDL files. The technology used to actually implement the Web Services, JSLEE, formerly known as JAIN SLEE, is gaining popularity at the moment. JSLEE is an industry-supported Application Programming Interface (API) standard specification based on Java 2 Enterprise Edition (J2EE) [51] .

WSDL defines request / response interactions that are not enough to model stateful long-running multiparty processes. To perform more complex interactions, multiple web services may have to be orchestrated. Orchestrations are multiparty interactions among Web Services. The Web Services Business Process Execution Language (WS-BPEL) is used to define Web Services orchestrations, modelling the interaction of different Web Services which in turn are defined by their own WSDL descriptions [30].

1.2.2 Need for semantics

Tim Berners-Lee envisioned [4] software agents that automatically handle service descriptions and compose them without human intervention. Semantics are needed to enable this automatic composition of services. For instance, an input parameter in a service can be a String. For a computer, a String is just a sequence of alphanumeric characters, while for a human being there is a semantic meaning attached to the String, which can be a telephone number, a flight number or the name of a city. In order to use semantics, first we need to map real world concepts onto a structure that can be computer understandable. These structures are called ontologies [20]. Ontologies contain a set of concepts, their properties and relations. The concepts are normally structured in a taxonomy forming categories.

Semantic languages can be used to describe ontologies. An example is the Web Ontology Language (OWL) [26]. WSDL does not support semantics. OWL-S [25] is an ontology that is used to semantically describe services, specially web services. Another semantic language for services description is WSML (Web Service Modeling Language) [9], which is based on WSMO (Web Services Modeling Ontology) [14].

1.2.3 The SPICE Project

The Service Platform for Innovative Communication Environment (SPICE) project [38] aims at fostering the dissemination of attentive and personalised mobile services, amongst others, by allowing service developers to define semantically-annotated services. A language called SPATEL [39] has been defined within the SPICE Project. SPATEL enables the definition of semantically-annotated services that can be later executed.

Two main environments are being developed in the SPICE project:

- A Service Creation Environment (SCE), which consists of developer tools that allow service creation and composition.
- A Service Execution Environment (SEE), which includes the Application Server (AS), in which services run, and also the lower lever infrastructure, like the network and its services.

The University of Twente contributes on the development of the Automated Composition Engine (ACE), which is a key component of the SPICE SCE. The goal of ACE is to enable dynamic composition of services.

One of the building blocks of the ACE is the composition algorithm, which automatically performs service compositions from existing services according to a specified semantic service request issued by a service developer or an end-user.

1.3 Goals of the Master project

The goals of this project have been:

1. to create artifacts that enable the development and evaluation of dynamic composition approaches using the semantically-annotated service descriptions;
2. to propose guidelines of usage for these artifacts;
3. to test the developed artifacts with a set of examples;
4. to consider and test integration issues with the other components developed in the SPICE project; and
5. to evaluate the suitability and performance of the ACE composition algorithm.

1.4 Approach

The following approach has been followed in this project:

1. Study literature on the current status and the service composition technology of the SPICE project, other approaches and related work. The knowledge of the project documentation allows one to understand the motivations for the design. Literature study on service composition provides knowledge about similar approaches, their advantages and their shortcomings.
2. Set up the environment for creating and executing services. The SPICE project developed an environment with tools that facilitate the creation of services. These tools were used to build an experimental testbed.
3. Understand and report on how to create a service, and develop guidelines for service creation, publication and discovery. These guidelines provide a systematic methodology to create and publish services and discover them later.

4. Develop and test a service publication and discovery library. Service publication and discovery are needed for service composition. Although these components were planned in the SPICE project, the provided implementation was not suitable so we had to develop our own.

5. Integrate the publication and discovery library in the ACE.

6. Evaluate the performed integration.

1.5 Structure of the thesis

This thesis is further structured in the following chapters:

- Chapter 2, "Relevant Concepts and Technologies", provides information about the technologies that have been used in this project, like description languages, orchestration languages, application servers, etc.

- Chapter 3, "SPICE Architecture", introduces the SPICE project in which this Master project is embedded.

- Chapter 4, "Service Creation Life Cycle", describes how services are created in SPICE, and how they can be automatically composed.

- Chapter 5, "Design and Implementation", explains how the artifacts used in the project were developed, namely the service creation, publication and discovery mechanisms.

- Chapter 6, "Case Study", shows some examples of services, how services can be requested and how services can be created as compositions of existing services. In this chapter an evaluation of the created artifacts is also performed.

- Chapter 7, "Final Remarks", provides the conclusions of this project and gives directions for future work.

2

Relevant Concepts and Technologies

This chapter presents an overview of the concepts and technologies that are relevant for this Master thesis.

Section 2.1 presents ontologies and the languages to represent them (semantic languages). Section 2.2 elaborates on languages used to describe services. Section 2.3 explains how services can be made available to other parties through a service registry. Section 2.4 introduces the concept of orchestration. Section 2.5 presents the Model-driven Architecture approach, which has been followed in many parts of the SPICE project. Section 2.6 describes some other relevant implementation tools and environments.

2.1 Ontologies and Semantic Languages

Semantic languages are used to describe ontologies. Ontologies are used in this project to build a model describing the domain where services are used, in a machine readable form.

2.1.1 Ontologies

Traditionally the term “ontology” designated the philosophical field that studies the “being” or the “existence”. In the context of computer and information science, an ontology defines “a set of representational primitives with which to model a domain of knowledge or discourse” [17]. These primitives are:

1. Classes or sets, which are collections of objects (terms).

2. Attributes or properties that class members may have.

3. Relationships or relations between the class members.

Tim Berners-Lee introduces [4] the following concepts:

1. A taxonomy, which is a classification of terms and the relationships among them (thus forming a hierarchy of classes).

2. Inference rules (axioms), which are formal constrains that allow one to "reason" over the terms in the taxonomy.

Because the axioms, the relationships and the properties are formally specified, the reasoning process can be performed by a computer software. This software is called a "reasoner". Examples of a reasoner is FaCT++ [50] and Pellet [6].

Figure 2.1 shows a simple ontology example taken from one of the ontologies provided by the SPICE Project. The root term is `SpatialEntity` and all the subclasses inherit from it. A `SpatialEntity` can be defined as an entity that has a `hasLocation` property, which is a link to an instance of the class `Location`. The figure shows the inheritance "is a" relationship between the concepts with arrows labelled with the "is a" legend. There is also an arrow to show the relationship between the `hasLocation` property and the `Location` class. It is possible to reason over the ontology because the relationship among the concepts is formally specified. From the example, it can be inferred that both a `Motorbike` and an `Ambulance` are `Vehicles`, but a `Motorbike` is not a `Car`.

Semantics play an important role in the approach followed in this project. For instance, by semantically annotating inputs and outputs of service operations it is possible to use a reasoner to automatically evaluate the matching between inputs and outputs of the different services involved in a service composition.

2.1.2 OWL

OWL, the Web Ontology Language [52], is a semantic language that aims at enabling automatic processing of information by applications, mainly in the context of the Semantic Web [26].

There are three variants of the OWL semantic language:

1. `OWL Lite` allows making classifications with simple constrains, i.e., the cardinality (number of instances) of an element in a relation can only be 0 or 1.

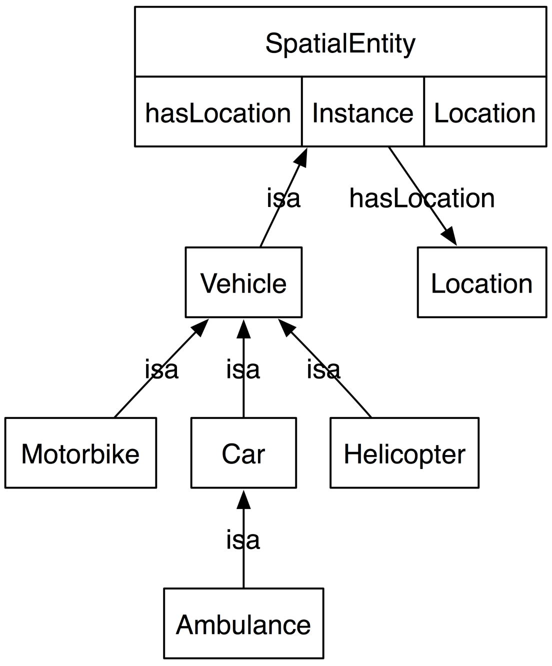


Figure 2.1: Ontology example

2. `OWL DL` allows maximum expressiveness while keeping completeness (all conclusions are computable) and decidability (reasoning can be performed in a finite time).

3. `OWL Full` imposes no constrains, but finite computable conclusion is not guaranteed.

Some language features supported by OWL are [52]:

- *Class*: defines a group of individuals that share the same properties. OWL has subclasses (inheritance). There are two special classes: *Thing* ($\top$), which is the superclass of all the classes and the class of all individuals; and *Nothing* ($\bot$) that has no individuals and is the subclass of all classes. The example ontology in Figure 2.2 has three class "Person", "Male" and "Female".

- *rdf:subClassOf*: is a relation used to define the hierarchy of classes, by declaring a class as subclass of another class. In the example "Male" and "Female" are subclasses of "Person".

- *Individual*: is an instance of a class, and properties may be used to relate one individual to another. Figure 2.2 shows three instances: "Peter", "Mary" and "Jane".

- *rdf:Property*: states the relationships between two class members, or between individuals and a data value. An example of the former is the property called “hasChild” in class “Person”; this property is used to describe the relationship between two individuals of class “Person”. An example of the later value is the “hasAge” of a “Person”. Both “Peter” and “Mary” have the property “hasChild” referencing “Jane”.

- *rdfs:subPropertyOf*: is used to define hierarchies of the properties. An example would be the properties “hasSon” and “hasDaughter” inherited from “hasChild” if the child is male or female, respectively.

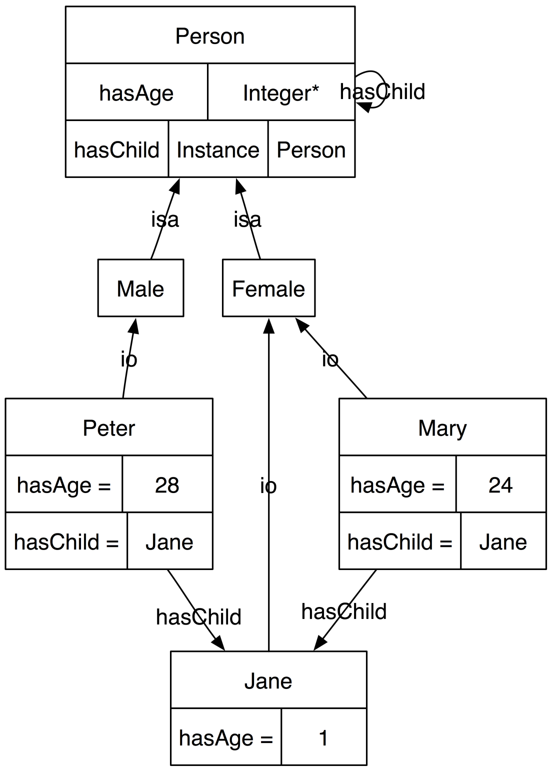


Figure 2.2: People ontology

There are many tools available that allow one to work with OWL specifications. A popular tool is Protégé [40], which provides functionality for editing ontologies, but also allows one to use a reasoner for reasoning on the ontologies.

2.1.3 WSML

The Web Service Modelling Language (WSML) [10] is used to model the Web Service Modelling Ontology (WSMO) and the web services that make use of it. WSML was designed due to some conceptual limitations of OWL [8]. For instance, OWL Lite and OWL DL have limited expresiveness and OWL DL is built upon RDF schemas but it is inconsistent with the original definition of the RDF's primitives. WSML developers were then concerned with the interoperability of ontologies described in the different languages, thus they focused on compatibility with the RDF schemas.

There are five WSML variants with different purposes; varying from the simplest WSML-Core to the most complete WSML-Full. The WSML syntax consists of two parts:

1. Conceptual syntax used to model ontologies, goals, web services and mediators.
2. Logical expressions that refine the models using a logical language.

WSML is not used within the SPICE Project. OWL is the language chosen and applied in SPICE.

2.2 Service Description Languages

In the Service-Oriented Architecture (SOA) services are required to provide a public interface in order to be found and executed. By exposing only an interface, implementation details are hidden from the point of view of the service users. Service description languages are needed to describe service interfaces and also service compositions.

2.2.1 WSDL

In Web Services, the service interface is typically defined in a standard XML modelling language format called Web Service Description Language (WSDL) [5]. The WSDL specification of a service contains information about the data types, messages used by the web service and the operations offered by the service. In WSDL, operations are grouped in endpoints at which messages are exchanged. Each endpoint contains definitions of abstract messages and exchange formats and the concrete message exchange protocol specification used for binding (connecting and invoking the service). Although many message exchange protocols can be used according to the WSDL standard, the most popular is SOAP (Simple Object

Access Protocol) [27]. One major drawback of WSDL is that it lacks semantic annotations support, hence the need for additional description languages. On the other hand, WSDL is quite important for web services' deployments.

One meaningful example of WSDL definition is Parlay X [13, 49], which is a set of Web services open standard definitions focused on telecommunication services. The Parlay Group provides WSDL files that can be used to build web services for tasks like making phone calls, sending SMS messages, communicating through instant messaging and presence management. Definitions of supporting services such as security and charging are also provided. The SPICE Project plans to use Parlay X for describing telecommunications services.

2.2.2 OWL-S

OWL-S [25] is an ontology of services meant to be part of a web-based semantic services' framework. OWL-S is sometimes referred to as a *language* (because an ontology defines a vocabulary) that combined together with OWL can be used to describe services [25].

OWL-S developers mention three tasks that have motivated the development of OWL-S [25]:

1. *Automatic web service discovery*: OWL-S allows one to advertise the properties and capabilities of the service, in a way that it would be possible for an automatic agent to find a service that fulfills a human-made request.

2. *Automatic web service invocation*: Inputs and outputs of services are semantically annotated, which enables a software agent to request the service execution by itself, by providing the necessary input parameters and understanding the output of the execution. OWL-S supports the semantic definition of the IOPE (Input, Outputs, Preconditions and Effects) parameters for each service.

3. *Automatic web service composition and interoperation*: OWL-S provides a description for each service based on goals, as well as precondition and effects for the execution of each service. This information can be used to compose individual services. The resulting service composition and data flows can be also modelled in OWL-S.

The structure of OWL-S upper ontology is presented in Figure 2.3. This structure shows the three types of knowledge about a service that can be represented using OWL-S [25]:

1. *ServiceProfile*: each service contains a `ServiceProfile` that can be used to advertise the capabilities of the service.

2. *ServiceModel*: a service has the property "describedBy", linking the service to a `ServiceModel` that describes how the service can be used.

3. *ServiceGrounding*: a service has the property "supports" to link the service to a `ServiceGrounding` that describes mechanisms (typically the network address and transport protocol) used to access the service.

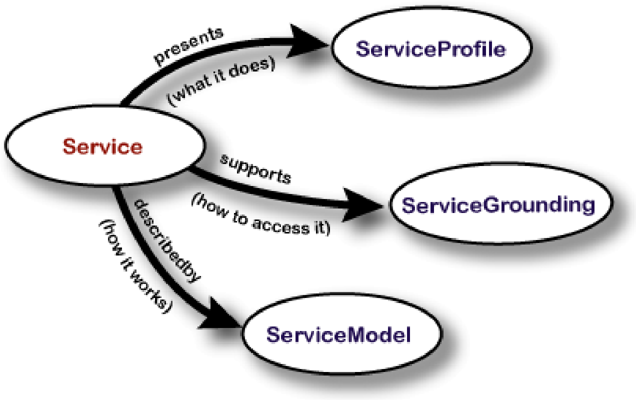


Figure 2.3: OWL-S Service Ontology

2.2.3 WSDL-S

WSDL-S [1] is a proposal of IBM and the University of Georgia for adding semantic annotations to WSDL, hence the "-S" suffix. The WSDL-S specification copes with two disadvantages of OWL-S: inputs and outputs have to be defined twice (syntactic and semantic definitions) and OWL-S is tightly coupled to OWL, while WSDL-S allows another semantic languages to be used.

WSDL-S served as base for the development within the W3C's SAWSDL workgroup. This workgroup developed semantic annotation support in the WSDL 2.0 standard [53].

2.3 Service Registry

Once a service has been modelled, developed and deployed, for instance, using WSDL, its interface can be made available for service clients to invoke the service, for instance using the WSDL definition of the service. However, before being able to invoke a service interface, service clients need to find and locate the service. A service registry contains a list of the available services and their bindings (locations) as well as the businesses offering these services. Thus, a service registry, if organised in terms of business entities, is the equivalent of the Yellow and White Pages in telephone directories.

In Web Services, the most popular repository technology is UDDI (Universal Description, Discovery and Integration) [29]. The UDDI registry is implemented as a web service.

A UDDI repository contains one or many `BusinessEntities`. `BusinessEntities` represent companies like "Google, Inc." or "University of Twente". Each `BusinessEntity` may have `BusinessServices`. A `BusinessService` is the UDDI entity that represents a service. Each `BusinessService` has a name, a unique `ServiceKey` identifier and a `BusinessKey` that points to the `BusinessEntity`.

Each `BusinessService` can have one or many `BindingTemplates` that provide information about the endpoint of the service; for instance, in Web services, there is typically one `BindingTemplate` that contains the URL of the WSDL description file.

The most basic structure in the UDDI repository is the `tModel` (technical Model). A `tModel` is meta-information that contains an unique `tModelKey`, a name, a value and a URL pointing to a description document with more information about the `tModel`.

In order to provide more powerful searches, some data elements, e.g. `BusinessService`, may have a `CategoryBag` element. This data structure is used to categorize the parent data structure in a given taxonomy. A `CategoryBag` contains a list of `KeyedReferences`, which in turn are name-key pairs with an optional link to a `tModel`. This optional `tModel` can be used as qualifier in a namespace.

Figure 2.4 shows the UDDI data structure explained before. More detailed information is available in [22] and in the UDDI specification [7].

2.4 Orchestration Languages

Although services usually consist of simple stateless request-response interactions, sometimes services have to support complex stateful longer term processes involving many par-

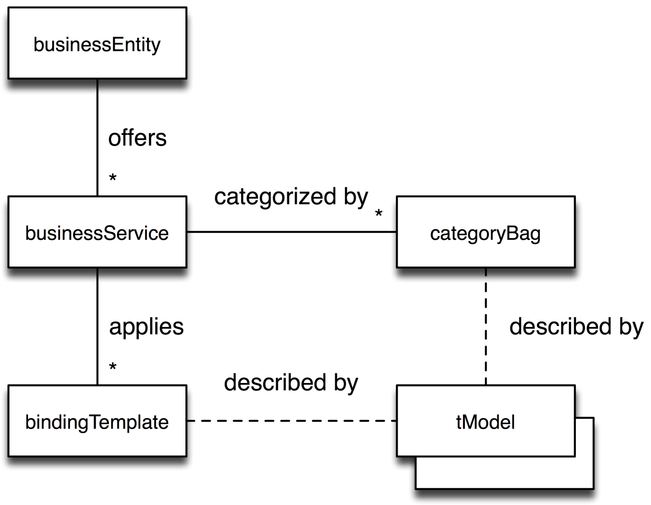


Figure 2.4: UDDI Data structure

ties. One example of these kind of process is orchestration. The orchestration is an executable process that defines the flow of information between the services and the sequence in which the services are executed. A complementary concept is choreography, which only defines the rules and messages used in the interaction between the services but it does not define a sequence of service operations to execute.

Orchestrations are described in orchestration languages. The Web Services Business Process Execution Language (WS-BPEL) [30] is the most popular orchestration language, and was developed by combining previous orchestration languages developed by IBM and Microsoft. OASIS [30] is the organisation in charge of coordinating the development of WS-BPEL. WS-BPEL can supports two kinds of processes: abstract processes that can be used to model business processes, mainly by defining the messages that are exchanged without giving implementation details; and executable processes that have all the detailed information necessary to be executed by an orchestration engine.

2.5 Model-Driven Architecture

Modelling is important in the system development process because it gives a better understanding of the system at a high-level of abstraction without the need of dealing with implementation details. Model-driven Architecture (MDA) [32] is a way of developing applications through modelling proposed by the Object Management Group (OMG) .

In MDA, a platform-independent model (PIM) is defined using a modelling language. A PIM contains the high-level definition of the application logic and the behaviour of the system under development. A PIM is independent of the technical implementation details. Using a transformation, the PIM can be mapped onto platform-specific models (PSMs) that cover the underlying implementation technology aspects. In this way, business functionality and implementation technology are decoupled in the design process.

It is possible to define the abstract syntax of the language used to define models as a model, and that called a metamodel. Figure 2.5 shows a PIM model and a PSM model. Both models have their own metamodels (PIM-M and PSM-M). If there is a generic transformation between the two metamodels then it is in principle possible to generate a PSM from a PIM.

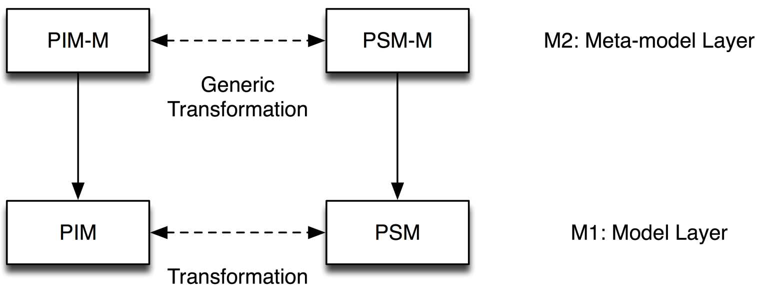


Figure 2.5: PIM and PSM

The MDA metamodelling architecture has a total of four layers, as shown in Figure 2.6. Layer M0 contains the entities in the real world. Models (PIM and PSM) are positioned at layer M1. The most popular metamodelling language is the UML (Unified Modelling Language)[33] . UML allows one to create models by drawing diagrams, i.e., class diagrams, user cases, sequence diagrams, etc. Metamodels like UML are in layer M2. In layer M3, MDA defines the MOF as the metametamodel with which metamodels can be defined.

Also in Figure 2.6 the Eclipse Modelling Framework (EMF) [46] is shown. EMF has the same principles of MDA, but it uses Ecore as metametamodel instead of the MOF. EMF Metamodels can be represented as XML schemas, UML diagrams or annotated Java interfaces. EMF differs from the OMG'S MDA technologies in that EMF is oriented at Eclipse plug-in development while MDA is a general purpose architecture.

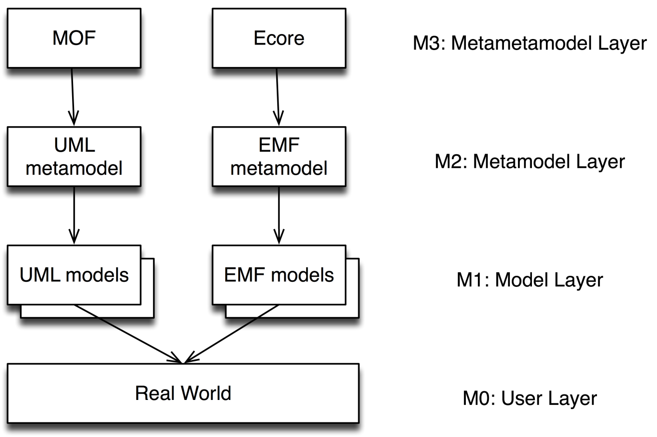


Figure 2.6: MDA four-layered architecture

2.6 Other Supporting Technologies

Services can be modelled starting from their requirements, by first defining the model in platform-independent service description languages and then further transforming these models into platform-specific models. A PSM takes into account the technologies in which the service is built. This section presents some implementation platforms in which services can be built and deployed.

2.6.1 Application Servers

An Application Server is a software that supports the deployment and execution of applications. An Application Server provides a runtime environment for components while it keeps the complex business logic and data sources hidden from the component user. Application Servers also provide access and interconnection to different middleware technologies. Java 2 EE and .Net applications are typically used within Application Servers, and those applications can be also reached through Web services.

Examples of Application Servers are Apache Tomcat [45] and Red Hat's JBoss [35].

2.6.2 Eclipse Web Tools Platform

Given a WSDL specification, it is possible to generate Java code from it using the Eclipse Web Tools Platform [47, 48]. The generated code includes all the classes needed to develop and deploy a Web Service, as well as skeleton code meant to be completed to provide the actual service implementation. Once the generated code has been completed to fulfill the service intended task, Eclipse WTP can deploy the service in a running Application Server, where it can be executed. The Eclipse WTP also provides a wizard for publishing the WSDL service description in a UDDI repository.

2.6.3 JSLEE

JSLEE [42], which stands for Java APIs for Integrated Networks Service Logic Execution Environment, is a Java EE standard API specification meant for developing telecommunications services [42, 51].

The main differences between JSLEE and a traditional enterprise application environment are [31]:

- JSLEE is asynchronous and event-oriented instead of synchronous and call-oriented.
- JSLEE is capable of handling high frequency of events. Those events are more fine-grained than in other platforms.
- The JSLEE components are lighter and their lifetime is shorter.
- While JSLEE is processing-intensive (frequent events) and uses different data sources, traditional environments use heavily a single database.

In a JSLEE environment services are built into Service Building Blocks, which are the counterpart of Java 2 EE's EJB. A special kind of component, the Resource Adaptor, allows communication with other components like instant messaging servers, SIP servers or SMS gateways.

MobiCents [36] is a JSLEE Application Server that runs on top of the JBoss Application Server [35].

2.6.4 IP Multimedia Subsystem - IMS

IMS (IP Multimedia Subsystem) [12] enables IP communication across a telecom operator's network. IMS allows the transmission of multimedia data (voice, video, audio) between subscribers or from content providers to subscribers. IMS aims at easing the deployment and use of multimedia services. Those services are executed in Application Servers, and IMS filters and routes the data traffic (service requests and responses) between the subscriber's terminal and the Application Servers. IMS eases the integration of supporting functions like authentication and charging.

IMS puts a strong focus on interoperability. For instance, it uses standard protocols, like, e.g. SIP (Session Initiated Protocol) [28].

Open IMS Core [15] is an open source reference IMS implementation from the Fraunhofer FOKUS Institute for Open Communication Systems. Its only purpose is to provide an IMS environment for research and testing, which means that Open IMS Core is not suitable for a production environment.

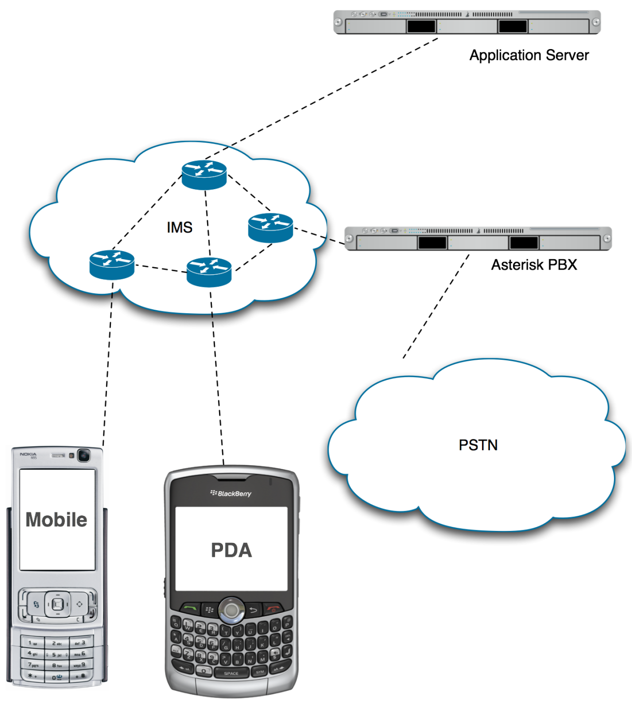


Figure 2.7: IMS architecture

Figure 2.7 shows the role of IMS in a service provisioning scenario. The cloud in the figure represents the IMS network. Inside the cloud there are components that perform tasks such as filtering, authentication and the routing of messages within the network. Mobile clients, represented at the bottom, use SIP to communicate with the rest of the network. Application Servers like MobiCents are connected to the IMS using SIP or Diameter [15], which is the protocol used inside the IMS network. In the case of MobiCents, the interface to the IMS network is provided by a Resource Adaptor.

2.6.5 Other tools

Asterisk [11] is an open source Private Branch eXchange (PBX). Asterisk is used within the SPICE Service Execution Environment as a SIP gateway between the IMS network and the Public Switched Telephone Network (PSTN). Figure 2.7 illustrates how the different components are interconnected.

A presence server is an application that shows the status (online, busy, away, etc.) of a subscriber, as seen by other subscribers. OpenSER [34] is an open source implementation of SIP Express Router (SER) and is used as a presence server.

3

The SPICE Architecture

This chapter presents the SPICE Architecture, in which this Master project is embedded. The SPATEL language is also presented. This language was used in the creation of services in this thesis.

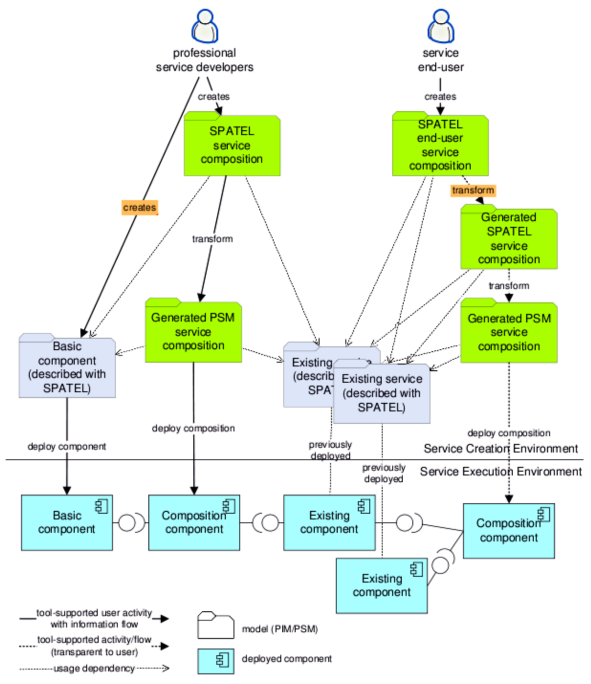
Section 3.1 depicts an overview of the architecture. Section 3.2 introduces the SPATEL language, which is used to define models. Section 3.3 gives an overview of the Service Execution Environment (SEE).

3.1 Architecture Overview

One of the main goals of the SPICE (Service Platform for Innovative Communication Environment) Project is facilitating service development, deployment and execution. The Service Creation Environment (SCE) assists in the development process while the Service Execution Environment (SEE) takes care of the deployment and the execution. Both environments are composed of different tools that are described in this chapter. The SPICE architecture is heavily inspired by the Model Driven Architecture (MDA) approach which was presented in Section 2.5.

Following the MDA approach, the SCE is mainly composed of tools that work with models (Platform-Independent and Platform-Specific Models), while the SEE works with deployable and executable components that are modelled in the SCE.

Figure 3.1 shows an architectural overview of the SPICE Project. The upper part of the picture shows the SCE models, while the lower part depicts the SEE entities. We explain Figure 3.1 from the perspective of the two system users shown as persons in the figure.



professional service developers
service end-user
creates
SPATEL service composition
SPATEL end-user service composition
transform
Generated SPATEL service composition
Generated PSM service composition
Basic component (described with SPATEL)
Existing service (described with SPATEL)
deploy component
deploy composition
previously deployed
Service Creation Environment
Service Execution Environment
Basic component
Composition component
Existing component
tool-supported user activity with information flow
tool-supported activity/flow (transparent to user)
usage dependency
model (PIM/PSM)
deployed component

Figure 3.1: SPICE Architecture overview [2]

System user on the left is a service developer. A service developer can create basic components. Each basic component models an individual service. The model of each service is produced using the SPATEL language, which is introduced in Section 3.2. A basic component has a real implementation in the form of an executable basic component in the SEE. The service developer can also create a service composition, which consists of basic service components or already existing services (also described in SPATEL). A service composition is transformed onto a generated PSM service composition, which is executable in the SEE.

The system user on the right is a service end-user. End-users can request their own compositions based on existing services. This composition request should be produced in any user-friendly language including natural language, and is transformed to SPATEL in order to be processed. The resulting composition is transformed into a PSM that models a composition component that runs on the SEE.

Figure 3.2 shows the components of the SCE. The developer studio and end-user studio provide the development tools for developers and users. The ACE supports the automatic composition of services. A service deployer component deploys the services created in the

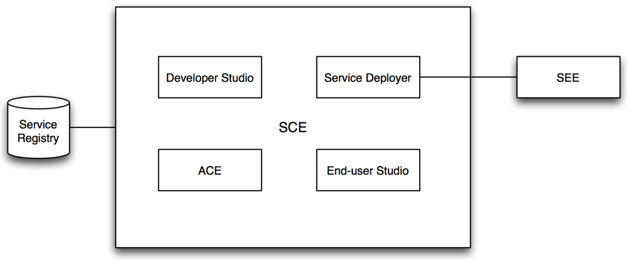


Figure 3.2: Service Creation Environment overview

SCE in the SEE. The SCE communicates with the service registry that contains the definitions of the services.

This Master final project focuses on the support of the service composition process.

3.2 The SPATEL description language

SPICE has developed the SPice Advanced service description language for TELecommunication services (SPATEL), which is a key part of the Service Creation Environment. SPATEL can be used to model services, both basic "atomic" services and service compositions.

3.2.1 Application scenarios and formalisms

SPATEL offers two different formalisms: a developer formalism and an end-user formalism. An end-user formalism is offered because the SPICE project wants to allow end-users to create their own customised services based on previously available basic services. The end-user formalism is more restricted than the developer formalism due to the lower level of expertise of end-users and the need to keep the capabilities bounded for security reasons. The SPATEL user formalism is not relevant for this Master project.

Another key point of SPATEL is that it supports the description of the functional parameters (Goals, Inputs, Outputs, Preconditions and Effects) and non-functional properties (QoS, costs, etc) by using semantic annotations. These semantic annotations can be defined as references to terms in ontologies. The SPICE project provides a set of ontologies that is meant to suit the most common use cases in the telecommunication domain.

Figure 3.3 shows how SPATEL models relate to the general architecture presented in Figure 3.1. A service developer can create SPATEL models of newly created services or of al-

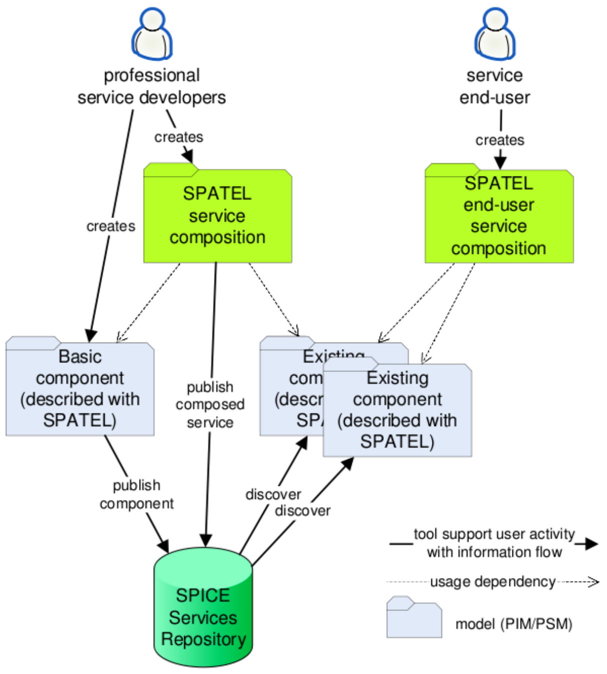

professional service developers
service end-user
creates
creates
creates
SPATEL service composition
SPATEL end-user service composition
Basic component (described with SPATEL)
publish composed service
Existing component (described with SPATEL)
Existing component (described with SPATEL)
publish component
discover
discover
SPICE Services Repository
tool support user activity with information flow
usage dependency
model (PIM/PSM)

Figure 3.3: Service Creation overview [2]

ready existing ones. The developer can also model a service composition based on models of basic services. These composed services can be made available as services themselves. The end-user can generate a service request that may be fulfilled by a service composition. A service composition can be modelled in SPATEL. In order to actually execute the composition, a Platform-Specific Model is needed. While service compositions generated by services developers can be published in the service registry and be made available to other users, service compositions created by end-users cannot be stored in the service registry because end-users lack the expertise or could harm the system intentionally.

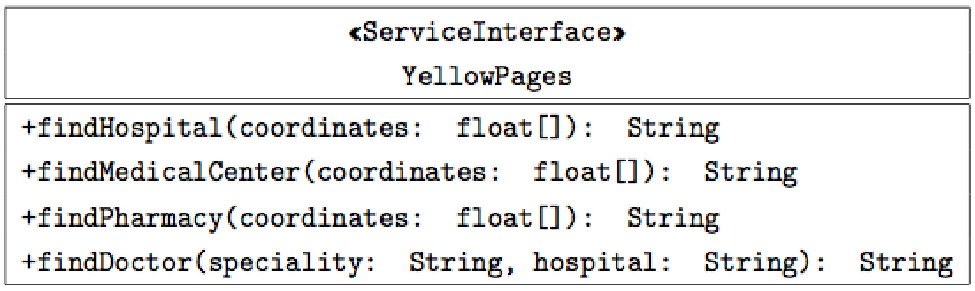
Taking into account the end-user, the ultimate goal is to enable natural language service requests that are analyzed and translated to SPATEL in a transparent way, possibly taking into account the end-user context information, like, e.g. user location and terminal capabilities, which can help determining which services are more suitable for an enhanced user experience.

3.2.2 Views

The SPATEL developer formalism supports two views: the internal view and the external view.

In the external view, a service is represented by its `ServiceInterface`, which is the list `ServiceOperations` that the service provides. A `ServiceInterface` can also contain a list of atributes although this feature does not make much sense. Each `ServiceOperation` contains information about inputs, outputs, preconditions and effects. In the context of telecommunication services it is quite common for an operation to be triggered by an event like receiving a phone call or a message and operations also generate those kind of events. A `ServiceContract` can be used to model a service choreography. The `ServiceContract` is formally specified as a simplified state-machine (protocol state machine).

Although services may have multiple service operations we do not consider in this work more complex cases in which invocations have to be ordered or depend on each other.



```
«ServiceInterface»
YellowPages

+findHospital(coordinates: float[]): String
+findMedicalCenter(coordinates: float[]): String
+findPharmacy(coordinates: float[]): String
+findDoctor(speciality: String, hospital: String): String
```

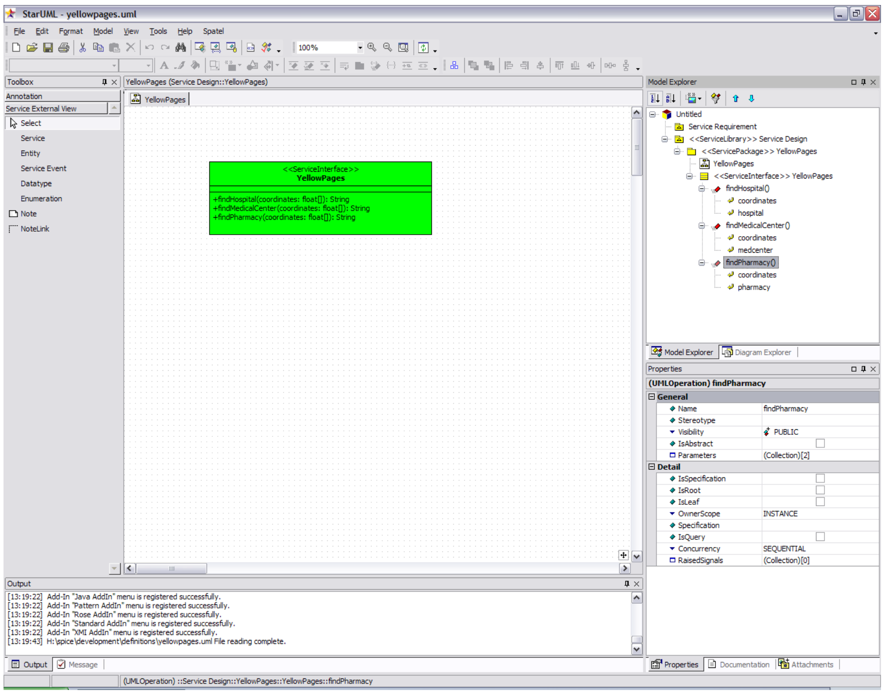
Figure 3.4: Yellow Pages service interface

Figure 3.4 shows an example of a `ServiceInterface` of a Yellow Pages service. The interface has no attributes, but four `ServiceOperations`. The first three are used to find medical locations (hospital, medical center and pharmacy) close to a certain location. They take a float array as input (coordinates) and returns a String (the name and address of the place). The last one takes a speciality and a hospital (Strings) and returns a doctor. Each input and output parameter has a name and a data type.

A `ServicePackage` is a collection of `ServiceInterfaces`. `ServicePackages` can be further grouped in a `ServiceLibrary`.

The internal view deals with the definition of the component that performs the operations defined in the external view. For instance, the same component can offer different external views or alternative implementations. Also, the functionality of a component may be provided by an orchestration of sub-components; then the internal view consists of a structural definition of the service's subcomponents and a state-machine diagram that defines the behaviour of the service.

SPATEL has a graphical representation based on UML diagrams. This graphical representation can be serialised into XML using XMI. The SPICE project provided a plugin for the StarUML [41] application that allows one to edit SPATEL models using the developer formalism. Figure 3.5 shows a screenshot of StarUML.



<<ServiceInterface>>
YellowPages

+findHospital(coordinates: float[]): String
+findMedicalCenter(coordinates: float[]): String
+findPharmacy(coordinates: float[]): String

(UMLOperation) findPharmacy

General	
Name	findPharmacy
Stereotype	
Visibility	PUBLIC
IsAbstract	☐
Parameters	(Collection)[2]
Detail	
IsSpecification	☐
IsRoot	☐
IsLeaf	☐
OwnerScope	INSTANCE
Specification	
IsQuery	☐
Concurrency	SEQUENTIAL
RaisedSignals	(Collection)[0]

```
[13:19:22]  Add-In "Java AddIn" menu is registered successfully.
[13:19:22]  Add-In "Pattern AddIn" menu is registered successfully.
[13:19:22]  Add-In "Rose AddIn" menu is registered successfully.
[13:19:22]  Add-In "Standard AddIn" menu is registered successfully.
[13:19:22]  Add-In "XMI AddIn" menu is registered successfully.
[13:19:43]  H:\spice\development\definitions\yellowpages.uml File reading complete.
```

(UMLOperation) ::Service Design::YellowPages::YellowPages::findPharmacy

Figure 3.5: StarUML

3.2.3 Semantic Annotations

The following SPATEL language elements can be semantically annotated:

- *Goals* of the `ServiceInterface` and its `ServiceOperations`;
- *Inputs* and *Outputs* of the `ServiceOperations`;
- *Preconditions* that need to be fulfilled before the service operation can be executed;

- *Effects* that the execution of the service operation has on other entities;
- *Non-Functional Properties (NFPs)* also QoS parameters: maximum response time, availability, reliability, etc. Cost, authentication and security parameters are also considered as NFPs.

We consider the example of a flight reservation service to illustrate the SPATEL language elements. This flight reservation service is supposed to have a “Book flight” service operation in which:

- The goal is to book a flight;
- The inputs are the name of the passenger, the flight number and the desired seat;
- The output is the confirmation code;
- The precondition is having enough money in the bank account to pay for the ticket;
- The effect is that the flight ticket is booked. This effect is the the precondition for a complementary cancellation service operation: a ticket can only be cancelled if it has been booked previously.
- Non-Functional Properties could be the fee of using the service operation (not the ticket cost) or a maximum processing time.

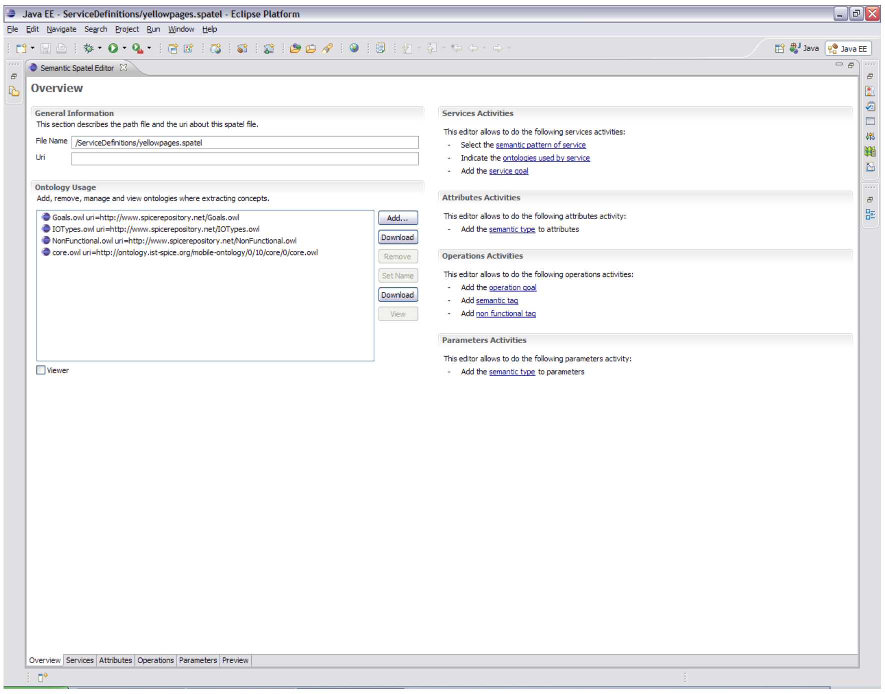
`ServiceOperations` can be annotated using two patterns:

1. GQIO: only Goals, Quality of Service, Inputs and Outputs are annotated.
2. GQIOPE: all elements are annotated.

The semantic terms used in the service need to be defined in an ontology. Ontologies are defined in semantic languages like the ones presented in Chapter 2. OWL is the language of choice within the SPICE project. A semantically-annotated service in SPATEL contains the URL (Uniform Resource Locator) that links to the ontology definition. Nevertheless, in order to achieve interoperability a common ontology should be used by all related services for overlapping concepts. Section 4.2.2 shows an example of semantic annotation, based on an example service model shown in Section 4.2.1. In order to support more services the ontologies need to be extended. Protégé [40] can be used to edit the ontologies. Protégé allows one to load an ontology and add new classes.

The ontologies provided by SPICE and enhanced by us are shown in Appendix A.

The SPICE project developed a SPATEL plugin for StarUML [41], which is a graphic UML editor. Once a model has been created in StarUML it can be exported (serialized) as XML. A semantic editor has been developed as an Eclipse plugin. This editor provides a graphical user interface that facilitates the annotation process. Figure 3.6 shows a screenshot of the Eclipse semantic annotation editor.


Java EE - ServiceDefinitions/yellowpages.spatel - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Java
Java EE
Semantic Spatel Editor
Overview
Add...
Download
Remove
Set Name
Download
View
Viewer
Overview Services Attributes Operations Parameters Preview

General Information
This section describes the path file and the uri about this spatel file.
File Name /ServiceDefinitions/yellowpages.spatel
Uri

Ontology Usage
Add, remove, manage and view ontologies where extracting concepts.
Goals.owl uri=http://www.spicerepository.net/Goals.owl
IOTypes.owl uri=http://www.spicerepository.net/IOTypes.owl
NonFunctional.owl uri=http://www.spicerepository.net/NonFunctional.owl
core.owl uri=http://ontology.ist-spice.org/mobile-ontology/0/10/core/0/core.owl

Services Activities
This editor allows to do the following services activities:
- Select the semantic pattern of service
- Indicate the ontologies used by service
- Add the service goal

Attributes Activities
This editor allows to do the following attributes activity:
- Add the semantic type to attributes

Operations Activities
This editor allows to do the following operations activities:
- Add the operation goal
- Add semantic tag
- Add non functional tag

Parameters Activities
This editor allows to do the following parameters activity:
- Add the semantic type to parameters

Figure 3.6: Eclipse semantical annotations editor

3.2.4 Translators

Following an MDA principle, a model described in a particular modelling language can be also transformed into another model using another model language if there is a transformation available defined in terms of the metamodels of these languages.

The SPICE Project aims at building translators that can transform a SPATEL model to another model representation, like WSDL for basic services or WS-BPEL for service compositions.

Currently it is possible to translate SPATEL to WSDL by means of an on-line form [3]. Due to the lack of semantic support in WSDL, the annotations are lost in the transformation.

However, tools are available to generate Java skeleton code from a WSDL file, like, e.g, the Eclipse Web Tools Platform [47].

3.3 Service Execution Environment

The Service Execution Environment (SEE) is a reference set of tools that can be used for running SPICE services' executable components. These executable components can be generated from a model by using a transformation, following a MDA approach. There could be many different transformations that lead to different implementations.

The components are implemented using the technologies presented in section 2.6. The SPICE Project has agreed on the following technologies as reference implementation of the SEE:

- Apache Tomcat [45] is used to execute general purpose services.
- MobiCents [36], a JAIN SLEE Application Server, is used to run telecommunication services like making a phone call or sending a SMS message.
- Open IMS Core [15] is an open source implementation of an IMS network environment and is used to manage an connect MobiCents and the end-users.
- Asterisk [11] is used as gateway to a traditional phone network (PSTN, Public Switched Telephone Network).

3.3.1 Service Registry

The Service Registry developed in SPICE Project provided is based on UDDI. The UDDI implementation chosen for the SPICE Project is jUDDI [44], the most popular open source UDDI implementation. jUDDI is written in Java and runs in the Tomcat Application Server. The repository back-end is implemented as a database, for which the MySQL database server can be used. jUDDI can be accessed and managed using its own API. Moreover, since UDDI is a standard, it is also possible to use alternative libraries to interact with it, like JAXR [43].

Unfortunately, the registry implementation originally available in the SPICE Project did not support SPATEL, but WSMO. Furthermore, service definitions were hardcoded. For these reasons we had to drop it and implement the registry ourselves, also based on jUDDI.

3.3.2 SPATEL Engine

The SPATEL Engine is meant to be one of the most important components of the SEE. The SPATEL Engine takes a service request written in SPATEL, discovers the relevant services, composes them and executes the composition. Thus, the SPATEL Engine plays the role of the coordinator or orchestrator in the architecture. The aim of the SPATEL Engine is to be driven only by SPATEL models, i.e., it would not be necessary to transform a SPATEL composition into WS-BPEL in order to execute it.

4

Service Creation Life Cycle

This chapter presents the Service Creation Life-cycle proposed in the SPICE project. The Service Creation Life-cycle starts with the creation of a basic service and ends with the selection of service compositions that fulfill the service request.

Section 4.1 gives a brief overview of the life-cycle. Service Creation is covered in Section 4.2. Section 4.3 depicts the service publication process. Section 4.4 introduces the service request functionality. Service discovery and composition are described in Section 4.5. Service composition selection is covered in Section 4.6.

4.1 Overview

The SPICE architecture defines the role of the Service Creation Environment as the component where services and compositions are modelled. Service creation in the SCE follows a life-cycle which is shown in Figure 4.1 [37].

The life-cycle starts with a service developer that creates a service and creates a model of the service. The service developer can also create a model of an already existing service, allowing legacy services to be used in compositions. Once the model, or the service description document, of the service is created, this model is published in a service registry.

A service developer or an end-user can create a service request according to his needs in terms of goals, inputs, outputs, etc. This request can be represented in a formal language in the case of the developer, or in a more abstract formalism (e.g., natural language) in the case of the end-user. The request is used to discover the relevant services in the registry. The composition mechanism tries to compose these services in order to satisfy the service

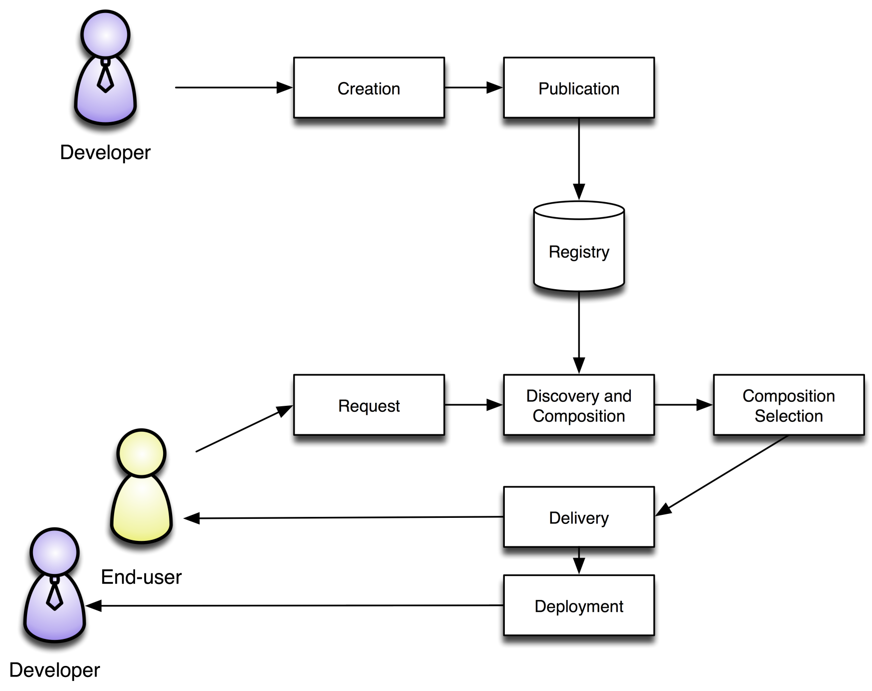


Figure 4.1: Service Life-cycle

request, and if the composition is possible one or more compositions are returned. An experienced service developer can choose the most suitable composition. End-users may not be able of making a choice, so a composition is chosen for them, either by using the user's preferences or some available user context information. Once a composition is chosen, it can be invoked by the user, or in the case of the service developer, he can make the composition available to others by deploying the composed service [37].

4.2 Service Creation

Service creation has three steps:

1. *Service modelling*: a model of the service is created.
2. *Semantic annotation*: the model of the service is semantically annotated using terms from a common ontology.
3. *Service development*: the service is actually developed, implemented and deployed, making it available for users.

The latest step, service development, can also be the first one if a bottom-up approach is followed.

4.2.1 Service Modelling

Using a modelling language a model of the service is created. Within this project the SPATEL language [39] was used to create service models (see Section 3.2). In order to keep things as simple as possible we did not consider complex service behaviours. Thus an internal view representing the state diagram is not required, and the service external view is enough. The external interface of a service has one or more service operations and may have attributes.

In SPATEL, the service model can be serialized to a XML document by using a transformation. The root element of the XML document is the `Service Library`. A `Service Library` may contain a `Service Package`, which in turn contains the services. In the yellow pages service example shown in Figure 3.4 the generated XML is:

```
<?xml version="1.0" encoding="UTF-8"?>
<spatel:ServiceLibrary xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:spatel="http://
    istspice.org/spatel/1.0.0/Spatel" name="Service␣Design">
  <nestedPackage xsi:type="spatel:ServicePackage" name="YellowPages">
    <service ...>
     ...
    </service>
  </nestedPackage>
</spatel:ServiceLibrary>
```

Each service has service operations, and service operations have parameters. The following example shows the definition of the “findHospital” service operation in the yellow pages service example. The operation has two input parameters and one output parameter:

```
<service name="YellowPages">
        <ownedOperation xsi:type="spatel:ServiceOperation" name="findHospital">
        <ownedParameter xsi:type="spatel:ServiceParameter" name="coordinates"
            direction="in" instanceType="float[]"/>
        <ownedParameter xsi:type="spatel:ServiceParameter" name="hospital"
            direction="return" instanceType="String"/>
    </ownedOperation>
...
</service>
```

4.2.2 Semantic Annotation

So far the service has been described using the external view of the service interface. The external view contains the name of the operations and some information about the parameters like the name, the direction and the data type. Semantic annotation is the process of adding references to terms in an ontology in service operations and parameters:

1. *Services* and *Service Operations* may be annotated with Goals, Preconditions, Effects and Non-Functional Properties.

2. *Parameters* may be annotated with their semantic types (reference to concepts in an ontology).

Semantic annotations are made using the semantic editor developed by the SPICE Project. This editor is a plugin for the Eclipse development environment. The output of the editor is a semantically annotated SPATEL serialized XML document.

The following example shows how the yellow pages service can be annotated with the "YellowPagesSearch" goal:

```
<service name="YellowPages" semPattern="GQIO">
      <ownedOperation ...>
        ...
      </ownedOperation>

      <semTag xmi:id="id" name="ServiceGoal" semType="Goals.
          owl:YellowPagesSearch" kind="goal"/>
      <ontology xmi:id="id" name="Goals.owl" uri="http://www.example.com/Goals.
          owl"/>
      <ontology xmi:id="id" name="IOTypes.owl" uri="http://www.example.com/
          IOTypes.owl"/>
      <ontology xmi:id="id" name="NonFunctional.owl" uri="http://www.example.com
          /NonFunctional.owl"/>
      <ontology xmi:id="id" name="core.owl" uri="http://www.example.com/core.owl
          "/>
    </service>
```

The "semTag" element is used to semantically annotate the service interface. The "semType" attribute refers to a term in the ontology and is formatted as the name of the ontology, a semicolon and the term in the ontology. The "kind" attribute defines which kind of annotation is it (goals, precondition, effects). The service interface contains references to the URLs of the ontologies from which the concepts were taken (in the example, the URL to "Goals.owl", "IOTypes.owl", etc.).

Service operations may be also annotated, like in the example below:

```
<ownedOperation xsi:type="spatel:ServiceOperation" name="findHospital">
        <ownedParameter xsi:type="spatel:ServiceParameter" name="coordinates"
            semType="IOTypes.owl:Coordinates" direction="in" instanceType="float
            []"/>
        <ownedParameter xsi:type="spatel:ServiceParameter" name="hospital"
            semType="core.owl:Hospital" direction="return" instanceType="String"
            />
        <semTag xmi:id="id" name="OperationGoal" semType="Goals.owl:FindHospital
            " kind="goal"/>
        <nonFuncTag xmi:id="id" semType="NonFunctional.owl:Cost" category="
            Charging" value="3" isDynamic="false" criterion="Cost"/>
        <nonFuncTag xmi:id="id" semType="NonFunctional.owl:Maximum_Response_Time
            " category="QoS" value="2" isDynamic="false" criterion="ResponseTime
            "/>
</ownedOperation>
```

The example shows that the Service Operation is annotated with a goal (in principle independent of the service interface goal) and two non-functional properties: cost and maximum response time. The terms are taken from the ontologies declared in the service interface.

Operation parameters can also be annotated with terms of an ontology, like this:

```
<ownedParameter xsi:type="spatel:ServiceParameter" name="coordinates" semType="
    IOTypes.owl:Coordinates" direction="in" instanceType="float[]"/>
```

4.2.3 Service Development

Once the service model is created, the next step is to transform it into an platform-specific model that can be made executable. The SPICE project provides a tool that allows one to perform SPATEL to WSDL transformation.

This WSDL specification can be easily translated into Java stubs and skeletons using the Eclipse Web Tools Platform [47].

It is also possible to work the other way around. Assuming that the WSDL specification is available, it is possible to translate it to SPATEL. This enables the integration of legacy services.

4.3 Service Publication

Once the services are defined and implemented, the next step is to publish them so other parties can invoke them when needed. The service is then stored in a registry that can be queried.

We use a UDDI-based service registry. Services are stored as `BusinessServices` in the registry. We developed a component that is able to publish a service in an UDDI registry given its SPATEL definition. This component extracts the information provided in the SPATEL definition and stores it using UDDI entities.

The design and implementation of this component is reported in Section 5.3.

4.4 Service Request

The service developer or the end-user can make a service request. We developed our own service request metamodel. The service request contains the GIOPE parameters (Goals, Inputs, Outputs, Preconditions and Effects) of the requested service. Each parameter of the service request is a reference to a term in an ontology. Service requests can be serialized to a XML document as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceRequest xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="
    http://localhost/docs/servicediscovery.ecore">
  <goal>http://www.example.com/core.owl#FindHospital</goal>
  <input>http://www.example.com/IOTypes.owl#Coordinates</input>
  <output>http://www.example.com/IOTypes.owl#Hospital</output>
</ServiceRequest>
```

The XML shows a service request with three parameters: a goal (FindHospital), an input (Coordinates) and an output (Hospital). Each parameter contains a reference to an ontology, in the form of the URL to the ontology file, and a "#" symbol followed by the semantic term.

We developed our own service request metamodel to have a simple language to make service requests, without dealing with the complexity of SPATEL which also lacked documentation about how to represent service requests.

4.5 Service Discovery and Composition

The SPICE Project aims at enabling dynamic and automatic service composition. This means that minimal user intervention should be required to obtain a service composition. A computer algorithm performs the composition, being the user task just to make a service request and check the results.

The composition process consists of the following steps [24]:

1. Discovery of relevant services based on the service request.

2. Construction of the Casual Link Matrix, which is used to evaluate the matchings between inputs and outputs of the discovered services.
3. Generation of the composition graphs that meet the service request, if feasible.
4. Finally, the output is a specification of the service composition written, for example, in WS-BPEL.

4.5.1 Service Discovery

Given a service request the Composition Factory performs a goal-based service discovery as function of the requested goals. Because the goal is part of an ontology, a reasoner allows one to find semantically related goals in the ontology that can also fulfill the service request. The result is a list of relevant services.

4.5.2 Casual Link Matrix

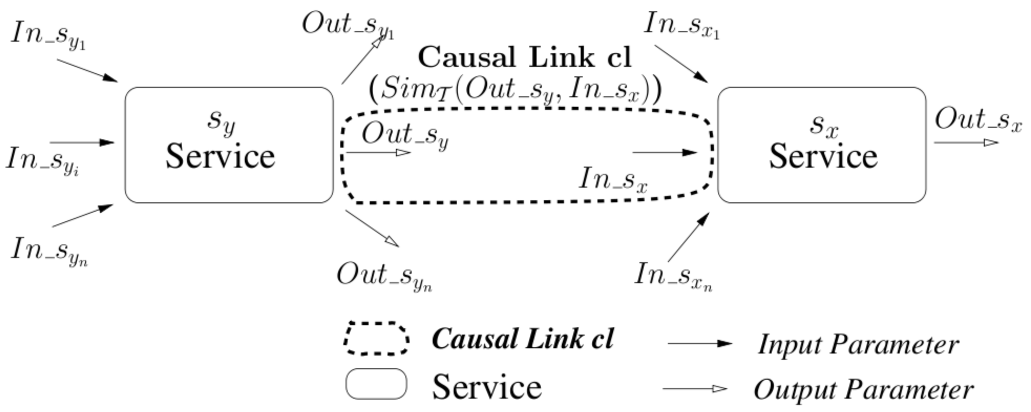


Figure 4.2: Casual Link [24]

Once the relevant services have been discovered, a Casual Link Matrix (CLM) [23] is built using the available services and the semantic links between their inputs and outputs. The parameter's semantic annotations are terms in an ontology $\mathcal{T}$. Each pair input-output is called a casual link and each cell of the CLM represents a specific casual link. The matchmaking function $Sim_{\mathcal{T}}(In\_s_x, Out\_s_x)$ evaluates the quality of the semantic link between an input $In\_s_x \in \mathcal{T}$ and $Out\_s_x \in \mathcal{T}$. $Sim_{\mathcal{T}}$ can have the following values:

- *Exact* ($\equiv$) if both input and output parameters are equivalent concepts in the ontology; formally $\mathcal{T} \models In\_s_x \equiv Out\_s_x$ (the ontology models that In_s_x and Out_s_x are equivalent).
- *PlugIn* ($\sqsubseteq$) if Out_s_x is a sub-concept of In_s_x; formally $\mathcal{T} \models Out\_s_x \sqsubseteq In\_s_x$ (the ontology models that Out_s_x is a subset of In_s_x).
- *Subsume* ($\sqsupseteq$) if Out_s_x is a super-concept of In_s_x; formally $\mathcal{T} \models In\_s_x \sqsubseteq Out\_s_x$ (the ontology models that In_s_x is a subset of Out_s_x).
- *Intersection* ($\sqcap$) if the intersection of Out_s_x and In_s_x is satisfiable; formally, $\mathcal{T} \not\models Out\_s_x \sqcap In\_s_x \sqsubseteq \perp$ (the ontology does not model that the intersection of In_s_x and Out_s_x is a subset of Nothing).
- *Disjoint* ($\perp$) if Out_s_x and In_s_x are incompatible; formally, $\mathcal{T} \models Out\_s_x \sqcap In\_s_x \sqsubseteq \perp$ (the ontology models that the intersection of In_s_x and Out_s_x is a subset of Nothing).

According to the definition given in [24], a casual link is valid if and only if the matchmaking function $Sim_{\mathcal{T}}(In\_s_x, Out\_s_x)$ is not a Disjoint matchmaking.

An extended Casual Link Matrix (CLM^+) can be defined by adding to a CLM the NFP information of the different services.

4.5.3 Composition Graph

After performing the service discovery and constructing the CLM^+, the next step is building the composition graph. The composition graph [24, 37] is a graph where the services correspond to the nodes and the casual links correspond to the edges. The composition algorithm starts from the service request outputs and builds the graph backwards in the direction of the input. Preconditions and effects can also be taken into account. A valid composition has to provide the requested inputs and outputs, but also has to meet all the requested goals. The algorithm has a preference for Exact matchings, but other acceptable casual links (PlugIn, Subsume and Intersection) are possibly considered. Disjoints casual links are not valid and are not taken into account.

NFPs are also considered in this step. The algorithm may prune a particular graph branch if the NFP constrains are not met, for instance, if the accumulated costs are higher than a predefined limit.

4.6 Service Composition Selection

It is possible to have many composition graphs as results of the composition process. In this case the NFPs should be considered. To select compositions, the Service Composition Selection component calculates the global NFPs of each graph; that procedure is called the aggregation of NFPs properties.

The aggregation is performed in a different way depending on the considered property. For instance, the cost of a service composition is the sum of the costs of the individual components. But considering throughput, the aggregated throughput is the minimum of all the throughputs of the basic services in the composition.

Once the aggregated NFPs are calculated a service composition must be chosen. The procedure differs for each of the roles that are considered within the SPICE project. An end-user is presented with the most suitable composition according to his preferences, like minimum cost, minimum response time, maximum availability, etc. A service developer can choose the composition himself based on the presented aggregated NFPs, or may possibly be provided with a ranked list given a set of preferences he specified.

5

Design and Implementation

This chapter introduces the design and prototype implementation of the components presented in the previous chapters, specially the service creation, publication and discovery components.

Section 5.1 introduces the artifacts that were developed. Section 5.2 elaborates on how services are defined using models. Section 5.3 presents the way of publishing services in a service registry. Section 5.4 shows the discovery and composition of services.

5.1 Developed artifacts

In this Master project we designed mechanisms and implemented a prototype that supports the service creation life-cycle introduced in the previous chapter (Figure 5.1).

Considering each block:

1. *Creation* is already supported by the SPICE-provided tools.

2. *Publication*: SPICE did not provide a suitable implementation for the publication functionality so a publication mechanism was developed from scratch.

3. The *registry* was implemented using jUDDI [44].

4. We developed the *request* functionality.

5. We developed the *discovery* of services in the registry. For the composition we used an already developed prototype.

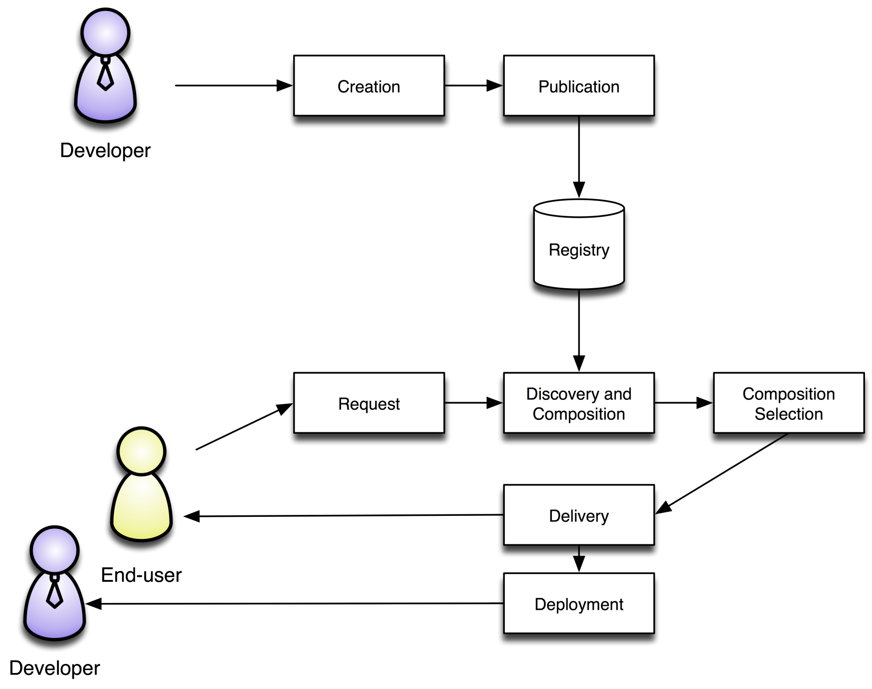


Figure 5.1: Service Life-cycle The service composition lifecycle

6. *Composition selection* and *delivery / deployment* functionality are left as future work due to time constrains.

In order to make some understandable and meaningful example compositions, all service examples were taken from the same domain, namely health services. These example services are assumed to be provided by a mobile phone operator working together with a health company that provides the actual health services. Clients of both companies can subscribe to these services. The business relationships among the parties are irrelevant and out of the scope of this project. The actual underlying implementation of these services is also out of scope.

5.2 Service Definition

Only the external view of the service is considered. Using the current available SPICE project tools [38] the definition process has two steps:

1. Definition of the service interface, described in Section 3.2.

2. Semantic annotation. A prerequisite for semantic annotation is that the needed terms are defined in a valid ontology, in our case a SPICE compliant ontology.

5.2.1 Semantic Annotation

StarUML does not support semantic annotations but there is a semantic annotation editor for Eclipse [48] developed by the SPICE Project. The usage of this editor is quite simple:

1. Load the SPATEL file containing the service definition (previously created with StarUML).
2. Load the OWL ontologies that are going to be used.
3. Select the goal of the service from the ontology.
4. Select the goal of each service operation.
5. Select the annotation of each parameter.
6. Select the NFP of the service operation, its criterion and its value.
7. Save the semantically annotated definition as a SPATEL XML file.

A drawback of the existing tools workflow is that once the SPATEL file has been semantically annotated it should not be edited with StarUML because the semantic annotations get lost.

An aim of the SPICE project is substituting Star UML by an Eclipse plugin. By doing this, the whole definition and annotation process would take place solely in the Eclipse environment.

5.3 Service Publication

Publishing a service description in a registry involves extracting the relevant information from the service definition and store it in the service registry in a way that facilitates later processing. In the SPICE project, service descriptions are modelled using SPATEL. The relevant information are the service operations, their name and their GQIOPE parameters (Goals, Non-functional properties, inputs, outputs, preconditions and effects). Moreover, the parameters are semantically annotated.

In order to extract information from the SPATEL XML documents, an alternative is to parse the XML document. However a metamodel for the SPATEL XML has been defined. Working directly with the SPATEL metamodel allows one to tackle the problem in a higher level of abstraction. Starting from the SPATEL metamodel, using the Eclipse Modeling Framework (EMF) [46], it is possible to generate a Java API, which is a set of interfaces and classes that allows editing and generating SPATEL definition models.

In order to do so, the first step is to create a new EMF Project in Eclipse and then importing into this project the Ecore file that stores the metamodel. A genmodel file is generated in this way. From the genmodel it is possible to generate different code components, ordered by increasing complexity:

1. Model code.

2. Edit code.

3. Editor code.

4. Test code.

5. All code.

We developed a class called `SpatelPublisher` that is able to load a SPATEL definition model. This model is loaded from a file that is given to the program as a runtime argument. Once the definition file is loaded into a `Resource`, it is possible to access all the elements of the model. In the developed version only elements of the external interface view are processed.

The key component of the `SpatelPublisher` program is the `SpatelSwitch` class. When loading the model file into the `Resource` it is possible to iterate over the elements of the model, but these elements are represented as general `EObjects`, being the specific type of each element unknown. `SpatelSwitch` contains a `doSwitch` method that accepts any `EObject` and call the `case` method corresponding to the actual type of the given object. In order to do some further processing, the `case` methods in `SpatelSwitch` needs to be overriden.

The following example shows the `caseServiceInterface` method:

```
//Service Interface
@Override
public Object caseServiceInterface(ServiceInterface si) {

        // For each Service Operation
        for (Operation ownedOperation : si.getOwnedOperation()) {
                doSwitch(ownedOperation);
        }
        return si;
}
```

This method is called if the `EObject` passed to the `doSwitch` function is a `ServiceInterface`. From the `ServiceInterface` it is possible to get the `OwnedOperations`, which are the `Service Operations`. This method then calls again the `doSwitch` to further process the elements.

From the results of parsing the SPATEL model, a `ServiceDefinition` object is created. This `ServiceDefinition` object contains the name, goals, inputs, outputs, preconditions, effects and non-functional properties of the service. Thus the role of the `SpatelPublisher` is to translate a SPATEL model to a Java object. This Java object provides the function `publish` that takes the information of the object and publishes it into a UDDI repository. The class diagram of the design can be seen in Figure 5.2.

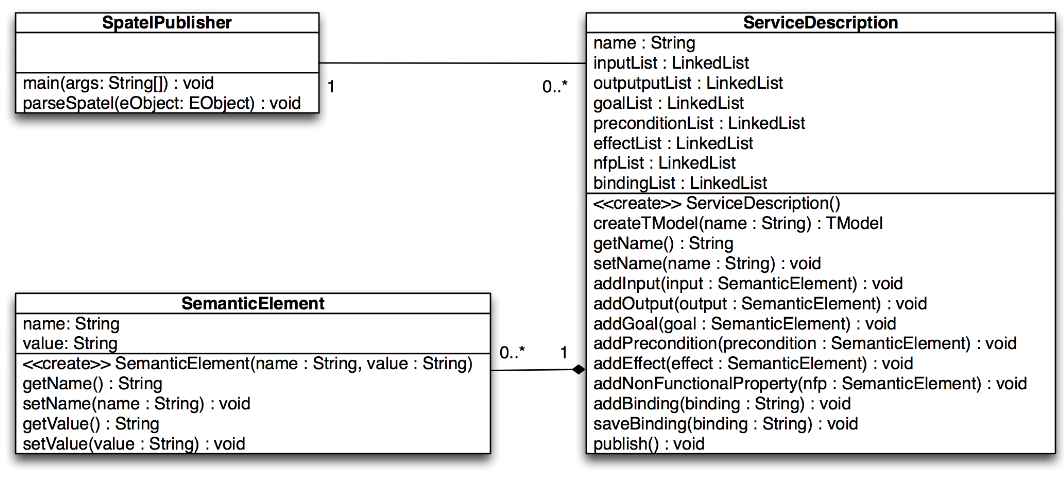

SpatelPublisher
main(args: String[]) : void
parseSpatel(eObject: EObject) : void
1
0..*
ServiceDescription
name : String
inputList : LinkedList
outputputList : LinkedList
goalList : LinkedList
preconditionList : LinkedList
effectList : LinkedList
nfpList : LinkedList
bindingList : LinkedList
<<create>> ServiceDescription()
createTModel(name : String) : TModel
getName() : String
setName(name : String) : void
addInput(input : SemanticElement) : void
addOutput(output : SemanticElement) : void
addGoal(goal : SemanticElement) : void
addPrecondition(precondition : SemanticElement) : void
addEffect(effect : SemanticElement) : void
addNonFunctionalProperty(nfp : SemanticElement) : void
addBinding(binding : String) : void
saveBinding(binding : String) : void
publish() : void
SemanticElement
name: String
value: String
<<create>> SemanticElement(name : String, value : String)
getName() : String
setName(name : String) : void
getValue() : String
setValue(value : String) : void
0..*
1

Figure 5.2: Spatel Publisher

This architecture was designed in such a way that it would be possible to replace the `SpatelPublisher` class and reuse the `ServiceDefinition`, given that the parameters annotation strategy (GQIOPE) remains the same. This allows one to use other languages than SPATEL for the service definition like, for instance, SAWSDL.

Method `publish` of `ServiceDefinition` publishes the service to the registry. To do so, first `tModels` for “goal”, “input”, “output”, “precondition”, “effect” and the “NFP” are created if they do not existe yet. The `BusinessEntity` is read from a configuration file, and if not present it is created. `ServiceOperations` are stored in the registry as `BusinessServices`. The bindings are also saved in the registry.

Each `BusinessService` can contain zero or more `CategoryBags`. Each `BusinessService` contains a `CategoryBag`. `CategoryBags` are used in UDDI to organize entities in categories. We used `CategoryBags` to store the parameter’s information. Each `CategoryBag` contains a set of `KeyedReferences`. Each `KeyedReference` contains an optional pointer to a `tModel`, a name and a value.

We used a `KeyedReference` for each parameter of the service operation. Within the `KeyedReference`, the name is the type of parameter being stored (goal, input, output, etc.) and the value is the URI of the semantic term. This URI is constructed using the URL of the ontology file, and the “#” symbol followed by a term of the ontology. The `KeyedReference`’s optional `tModelKey` takes the value of the `tModelKey` of the parameter type `tModel`: goal, input, output, etc. The NFPs are a special case. The name stores the ontology URI, as explained before, but appending the criterion attribute preceded by “`;criterion=`”. The value in the `KeyedReference` stores the value of the NFP, for instance, the cost in Euros.

Figure 5.3 shows the mapping between the service interface and its semantics annotations (on the left) and the UDDI entities (on the right).

A limitation of the current implementation is that it does not allow versioning of services. That means that if a new version of the service is published, the registry does not update nor overwrites the previous one but it publishes the new description keeping the old one as well.

The service registry used in this Master project was jUDDI [44], an open source implementation of a UDDI registry. jUDDI runs on top of the Tomcat Application Server and allows one to use MySQL as back-end database.

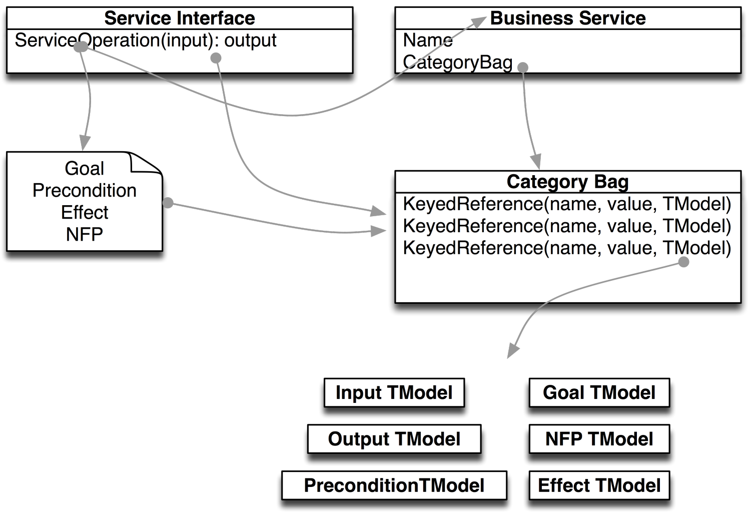


Figure 5.3: Mapping between service interface and UDDI entities

5.4 Service Discovery and Composition

The service discovery procedure can be divided in two steps: the definition of a service request and the service discovery itself.

5.4.1 Service Request

Service requests include semantically-annotated information about goals, preconditions, effects, inputs and outputs. This is modelled using Eclipse EMF [46]. EMF allows the creation of metamodels starting from an annotated Java interface. The interface used was:

```
/**
 * Metamodel for a service request.
 * @author jorgeml
 * @model
 */
public interface ServiceRequest extends EObject {

        /**
         * Goals of the model
         * @model
         */
        public EList<String> getGoal();

        // etc.

}
```

The elements to include in the metamodel are annotated with `@model`. There is a `get` method for each element of the metamodel. The example only shows the `getGoal` but there are equivalent methods for inputs, outputs, etc.

Using the EMF tools an Eclipse editor was generated. The editor eases the creation of model instances by providing a basic graphical user interface. This is the serialised form of a service request instance:

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceRequest xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="
    http://localhost/docs/servicediscovery.ecore">
  <goal>goal1</goal>
  <goal>goal2</goal>
  <input>input1</input>
  <output>output1</output>
</ServiceRequest>
```

This service request contains: two goals elements, “goal1” and “goal2”; one input element, “input1”; and one output, “output1”. In a real service request those elements are URIs that reference a term in an ontology.

5.4.2 Service Discovery

Service requests are processed by the `ServiceDiscoverer`. This component was also developed using EMF and the service request metamodel. The `ServiceDiscoverer` reads the service request file, extracts the parameters and makes a list of values of each of the parameters of the same kind (goals, inputs, outputs, etc.).

When the lists are made it performs a reasoning over the parameters’ ontologies and adds the subclasses of the requested elements to the list. The motivation is that the user may request a term which has subclasses. These subclasses will not be discovered, on the other hand they would fulfill the request. For example, a request for a `Price` input parameter is also fulfilled by the `PriceInEuro` and `PriceInDollars` classes. These reasoning is performed using the Pellet [6] reasoner.

Once the lists of requested parameters are complete, the `ServiceDiscoverer` searches for services that match any of the given values (logical OR) for each parameter. In the case of the previously given example, the `ServiceDiscoverer` first finds all the services that have either `goal1` or `goal2`. Then it searches services that have `input1` and then services that have `output1`.

Finally, a list of the matching services and their parameters is given, avoiding duplicates. The possibility to retrieve all the semantically-annotated parameters directly from the repository allows one to avoid mapping back to SPATEL or other service description language.

The prototype implementation is able to find any service given one of its parameters (or a superclass of it), avoiding duplicates. All other parameters from the discovered service are also extracted from the registry.

5.4.3 Service Composition

The `ServiceDiscoverer` was later integrated in the framework developed at the University of Twente for dynamic service composition. This framework offers the user a GUI that allows the creation of a service request. Upon user request, it will discover the relevant services by using `ServiceDiscoverer` functions and compose the services to match the service request.

6

Case Study

This chapter introduces the case study that has been used to evaluate the developed prototype. This case study consists of basic services and service composition scenarios.

Section 6.1 introduces the basic service examples. Section 6.2 presents some service composition examples. Section 6.3 shows the testing results on these service composition examples.

6.1 Basic Services

In this section we present the library of services that we developed for testing purposes. Each of the examples is represented by the corresponding service interface's figures. The semantic annotations (references to terms in an ontology) for input and output parameters are shown in the figures represented by arrows.

We defined some general purpose basic services, for example, Yellow / White Pages service or user location services. Other services were taken from the healthcare domain.

The first basic service that we defined is the Alert Service in Figure 6.1. Assuming that a subscriber has been taken into a hospital, the operation `alertContactPerson` sends an alert, for instance via SMS to the subscriber's contact person, given the subscriber identification number (i.e., the cell phone number) and the hospital. The operation returns an alert to the contact person.

Another basic service we defined is an Appointment Service (Figure 6.2). It takes two inputs, the identity of both patient and doctor, and returns an appointment. In order to make

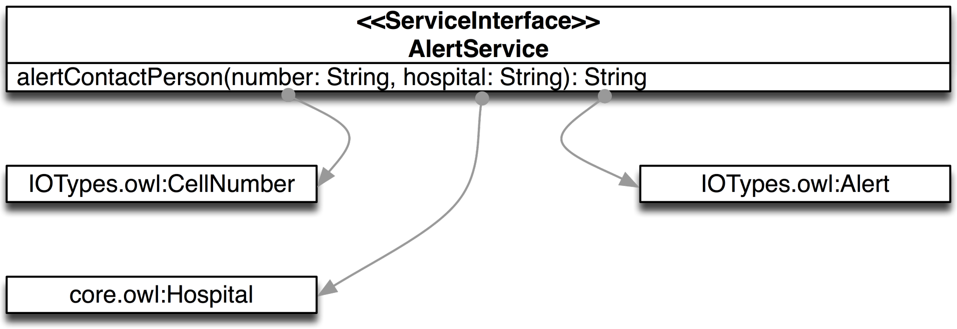


Figure 6.1: Alert Service

the appointment the service checks the agendas of both patient and doctor and chooses an available time slot for both parties. The output is an appointment.

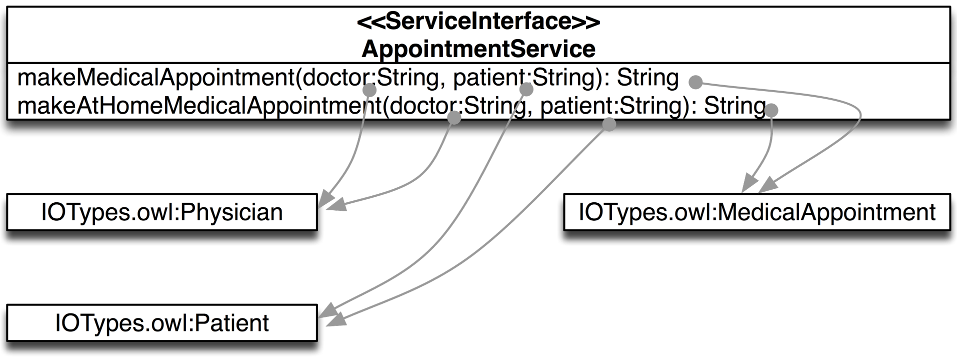

<<ServiceInterface>>
AppointmentService
makeMedicalAppointment(doctor:String, patient:String): String
makeAtHomeMedicalAppointment(doctor:String, patient:String): String
IOTypes.owl:Physician
IOTypes.owl:MedicalAppointment
IOTypes.owl:Patient

Figure 6.2: Appointment Service

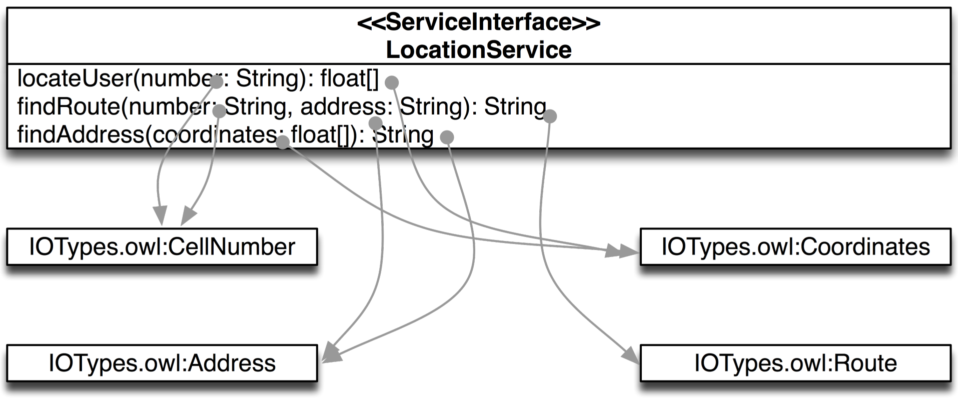
The Location Service shown in Figure 6.3 provides three operations.

1. `locateUser` returns the coordinates of a subscriber, given his phone number.

2. `findRoute` returns the route given the destination address and the number of the subscriber

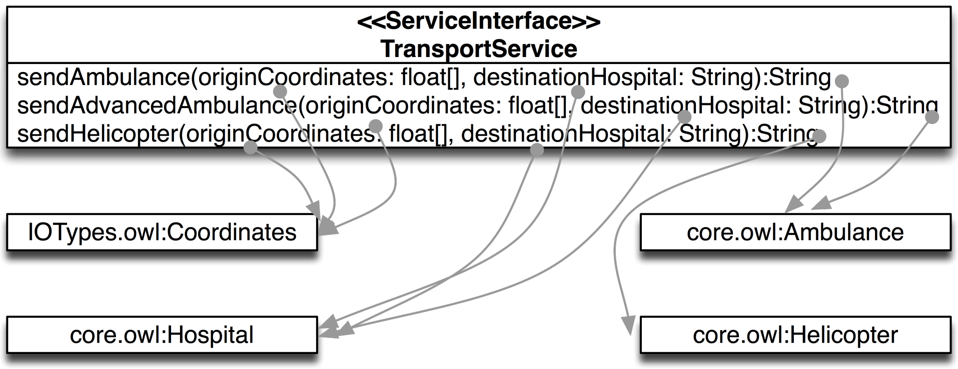
3. `findAddress` returns the postal address of the given coordinates.

A Medical Transport Service is also defined (Figure 6.4). It has three operations which take the origin coordinates and the destination hospital and returns a confirmation. A medical transport, an ambulance, an advanced ambulance or a helicopter is sent to transport the patient between the origin point and the destination hospital.



<<ServiceInterface>>
LocationService
locateUser(number: String): float[]
findRoute(number: String, address: String): String
findAddress(coordinates: float[]): String
IOTypes.owl:CellNumber
IOTypes.owl:Coordinates
IOTypes.owl:Address
IOTypes.owl:Route

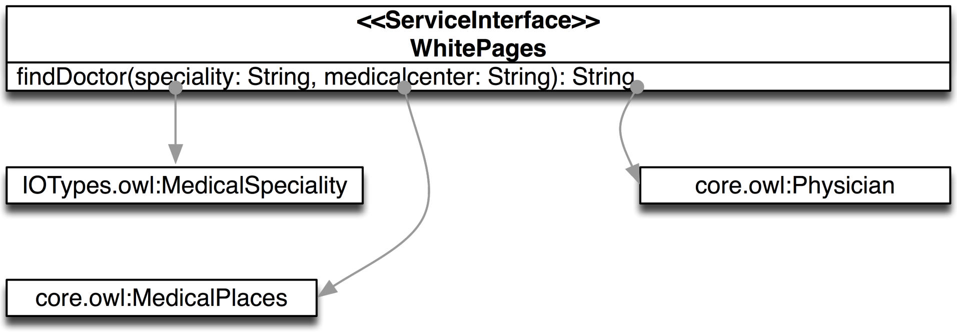
Figure 6.3: Location Service



<<ServiceInterface>>
TransportService
sendAmbulance(originCoordinates: float[], destinationHospital: String):String
sendAdvancedAmbulance(originCoordinates: float[], destinationHospital: String):String
sendHelicopter(originCoordinates: float[], destinationHospital: String):String
IOTypes.owl:Coordinates
core.owl:Ambulance
core.owl:Hospital
core.owl:Helicopter

Figure 6.4: Transport Service

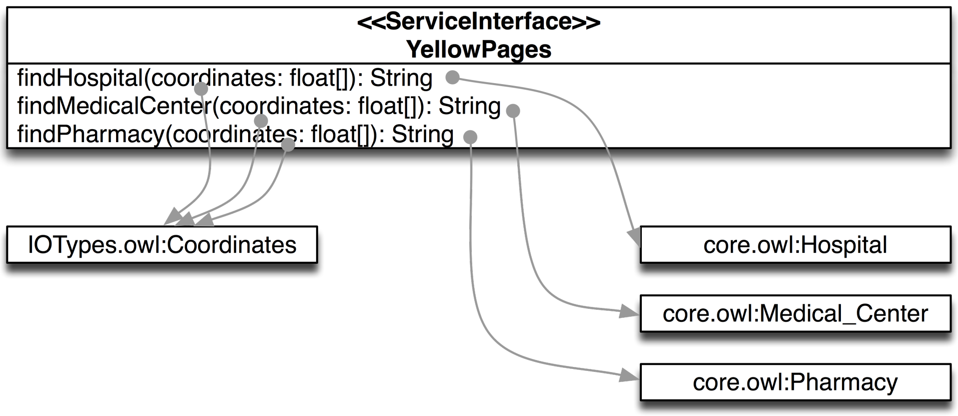
The White Pages service (Figure 6.5) only supports one operation, namely `finddoctor`. Given one speciality, for example neurology, and a hospital name, it returns the name of a doctor of this speciality who works in the given hospital.



<<ServiceInterface>>
WhitePages
findDoctor(speciality: String, medicalcenter: String): String
IOTypes.owl:MedicalSpeciality
core.owl:Physician
core.owl:MedicalPlaces

Figure 6.5: White Pages Service

The Yellow Pages service (Figure 6.6) has three operations that are used to locate the three kinds of medical facilities, namely Hospital, Medical Center and Pharmacy. It returns the closest one given the position of the subscriber.



<<ServiceInterface>>
YellowPages
findHospital(coordinates: float[]): String
findMedicalCenter(coordinates: float[]): String
findPharmacy(coordinates: float[]): String
IOTypes.owl:Coordinates
core.owl:Hospital
core.owl:Medical_Center
core.owl:Pharmacy

Figure 6.6: Yellow Pages Service

6.2 Service Composition Scenarios

This section discusses some example scenarios of composed services. These services are built by using the basic services of Section 6.1 as component services. These scenarios provide some insight in the potential of service composition. The actual implementation of the services falls out the scope of the project.

6.2.1 Emergency Service

Alice is walking on the street and she starts to feel an intense pain in her chest. Unfortunately she will have to try the new emergency service provided by her health insurance. She sends a SMS message to the insurance emergency number. They ask her phone operator to locate her. According to her position they find the nearest hospital. An ambulance team receives a SMS message with the location of Alice and the address of the hospital where they have to take her. At the same time Alice's husband receives an alert SMS message on his mobile telling him the hospital were she has been taken.

Fortunately she did not have anything serious and after one day under observation Alice could go home.

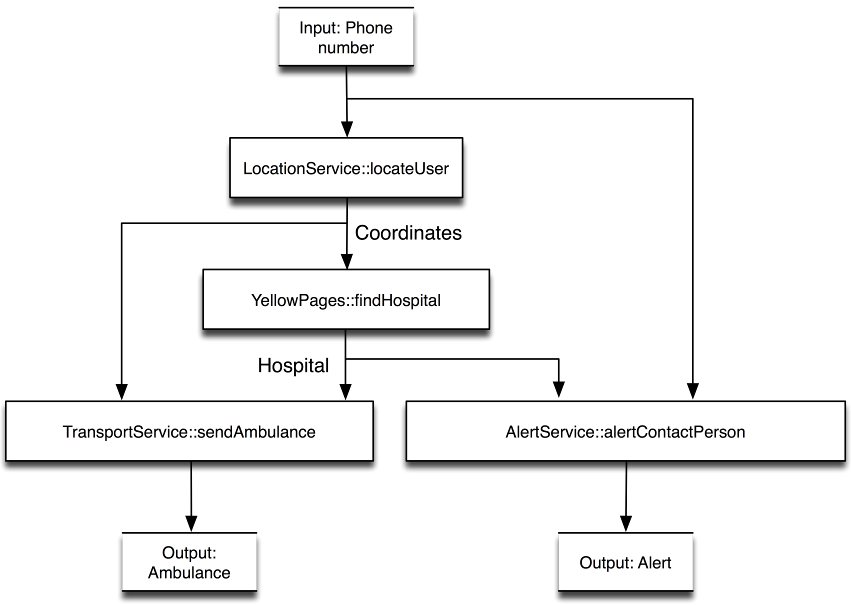

Input: Phone number
LocationService::locateUser
Coordinates
YellowPages::findHospital
Hospital
TransportService::sendAmbulance
AlertService::alertContactPerson
Output: Ambulance
Output: Alert

Figure 6.7: Emergency Service

Figure 6.7 shows the service composition. The user's position is provided by the `locateUser` service, given the user's mobile phone number. The output are the position coordinates. Those coordinates are used to find the nearest hospital.The coordinates and the name of the hospital are given to the `sendAmbulance` service, which arranges an ambulance. From the mobile number a previously given contact person can be found using the `alertContactPerson` service. An alert message containing the identification of the patient and the hospital can be then sent.

In this composition scenario, the `sendAmbulance` service can be replaced by the `sendAdvancedAmbulance` or `sendHelicopter` services, because their goals are similar. Thus, three possible compositions are possible. The choice depends on NFPs, like maximum response time (a helicopter is faster) or cost (an advanced ambulance is more expensive than a regular ambulance).

6.2.2 Appointment Service

Bob needs to go to the dentist to have a regular check-up. He sends a SMS message to the appointment service provided by his health insurance. The nearest medical centre is located, and an available dentist is selected. Then the Appointment Service checks the availability in the agendas of Bob and the selected dentist (e.g., by using Google Calendar) and sets up the appointment event.

The composition shown in Figure 6.8 shows how the nearest medical center (i.e., a physician office) is found using the user location. Given a speciality (i.e., dentist, in the scenario) and the medical center it is possible to find a suitable doctor by using the `findDoctor` service. `makeMedicalAppointment` is then used to look up both doctor and patient online agendas, find a suitable time slot and arrange the appointment.

6.2.3 Doctor At Home Service

While painting his apartment Charlie fell off the stairs and has a twisted ankle. He could arrange an ambulance to go to the hospital but decides to send an SMS to the new "Doctor at home" service that his insurance company provides. The phone operator will locate the position of Charlie and after checking that he is at home, a request will be sent to the nearest medical centre. Charlie will have to choose the medical speciality and the appointment will be made.

The composed service in Figure 6.9 is similar to the one shown in Figure 6.8, but in this case the appointment service is replaced by the appointment at home service.

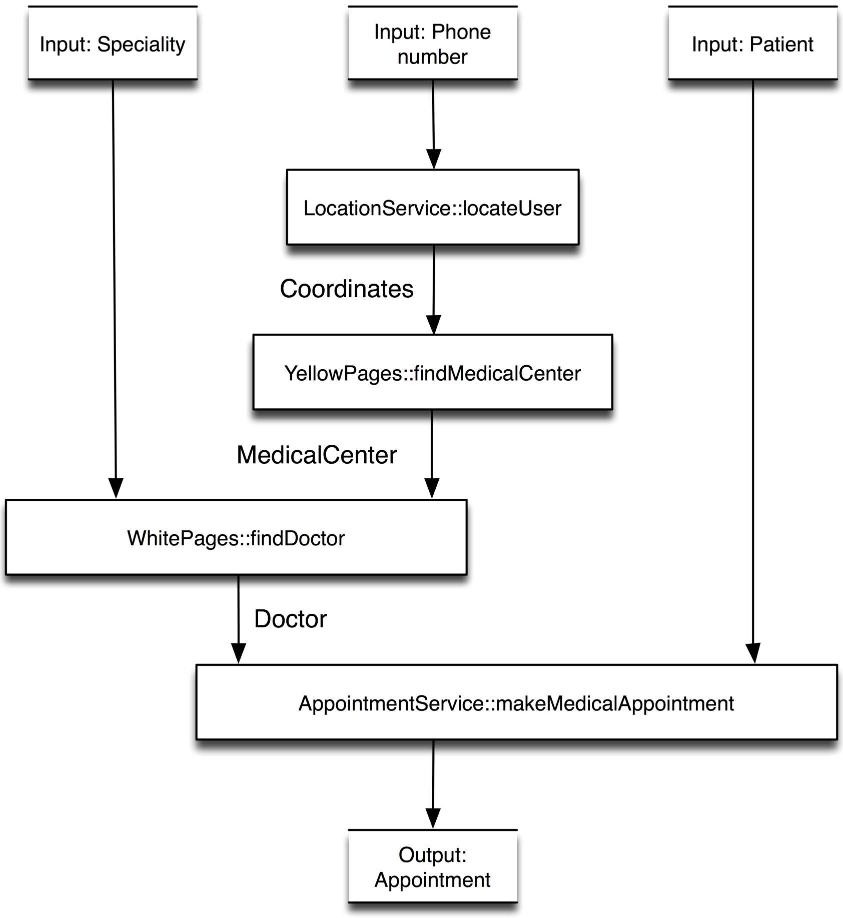


Figure 6.8: Appointment Service

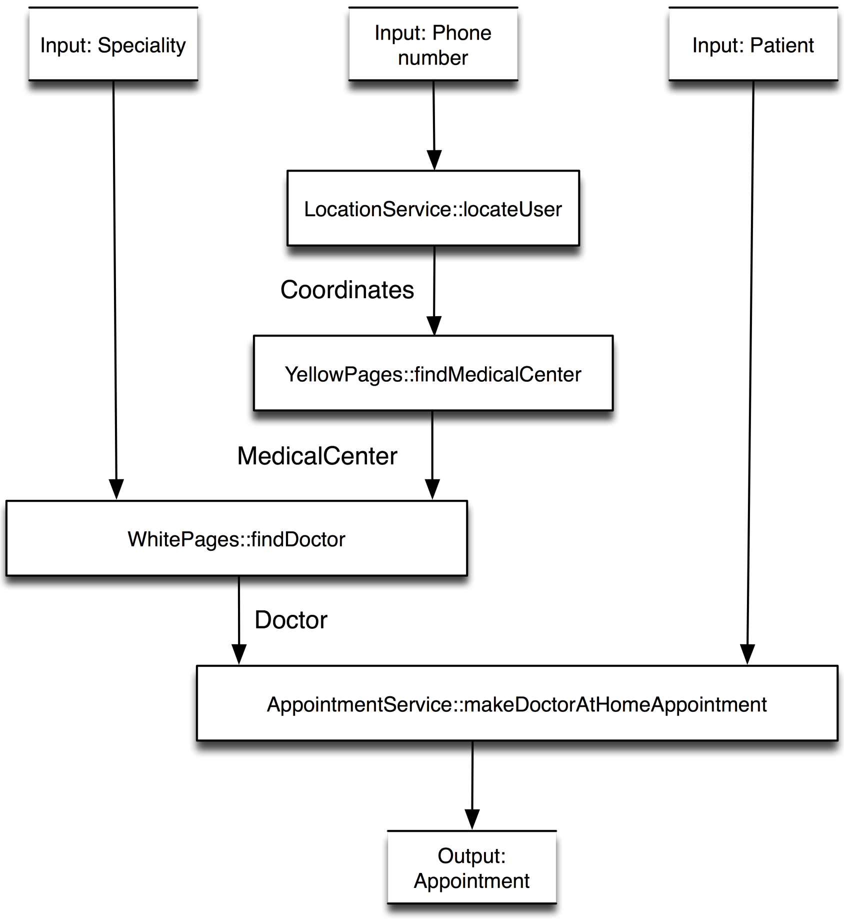


Figure 6.9: Doctor At Home Service

6.3 Service Composition Testing

In order to test the prototype all service descriptions have been published in the service registry. Then, we tried to generate the compositions described in the scenarios in Section 6.2. To generate the compositions, first a service request needs to be made. This service request must contain all the inputs and outputs of the desired composition, together with the goals to be achieved by the service compositions. Service requests can be stored in a file and passed as an argument or they can be created by using the graphical user interface shown in Figure 6.10.

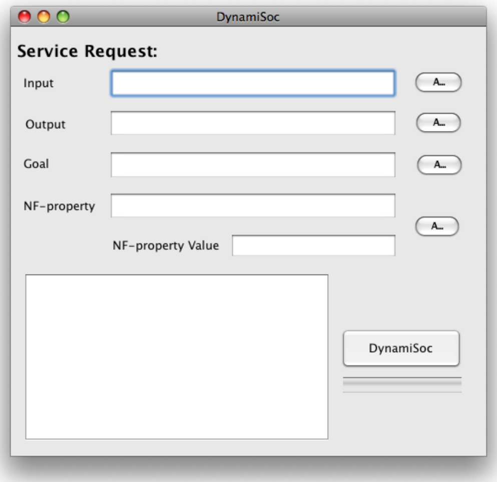


Figure 6.10: Service request graphical user interface

6.3.1 Emergency Service

The first composition that we tried is the Emergency Service, shown in Figure 6.7. The service request used was:

- Inputs:
  - `IOTypes.owl#CellNumber`
- Outputs:
  - `core.owl#Ambulance`

  - `IOTypes.owl#Alert`

- Goals:
  - `Goals.owl#SendAmbulance`
  - `Goals.owl#FindLocation`
  - `Goals.owl#FindHospital`
  - `Goals.owl#SendAlert`

`Goals.owl#FindLocation` is included in the service request because we want to use the user location to dispatch the ambulance.

The service request is used to discover the relevant services. These services are composed according to the algorithm explained in Section 4.5.The result of the composition is shown in Figure 6.11. Service operations are represented by boxes. The edges of the graph represent the input and output couplings.

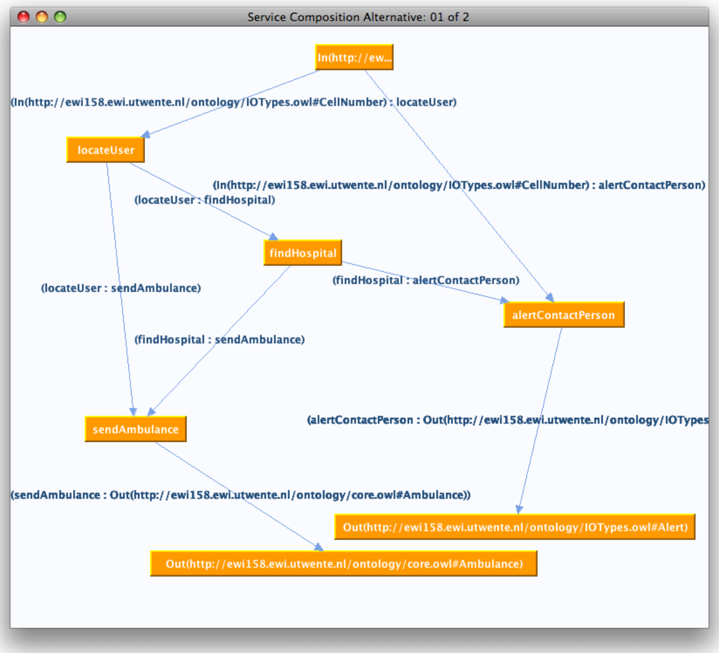

Service Composition Alternative: 01 of 2
In(http://ew...
(In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#CellNumber) : locateUser)
locateUser
(In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#CellNumber) : alertContactPerson)
(locateUser : findHospital)
findHospital
(findHospital : alertContactPerson)
(locateUser : sendAmbulance)
alertContactPerson
(findHospital : sendAmbulance)
(alertContactPerson : Out(http://ewi158.ewi.utwente.nl/ontology/IOTypes
sendAmbulance
(sendAmbulance : Out(http://ewi158.ewi.utwente.nl/ontology/core.owl#Ambulance))
Out(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#Alert)
Out(http://ewi158.ewi.utwente.nl/ontology/core.owl#Ambulance)

Figure 6.11: Emergency Service Composition

Since the request contains the requested goal `Goals.owl#SendAmbulance`, services that support subgoals of this goals (goals that are subclasses of this goal) are also included in the service request. This generates two compositions being generated, shown in Figure 6.11 and Figure 6.12, respectively. The later contains the service operation `SendAdvancedAmbulance`, which supports a subgoal of `Goals.owl#SendAmbulance`.

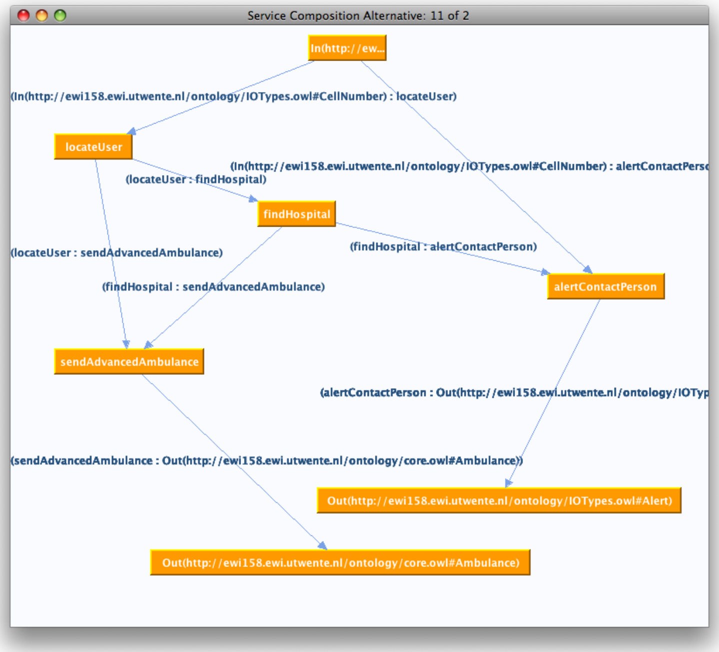

Service Composition Alternative: 11 of 2
In(http://ew...
(In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#CellNumber) : locateUser)
locateUser
(In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#CellNumber) : alertContactPers
(locateUser : findHospital)
findHospital
(locateUser : sendAdvancedAmbulance)
(findHospital : alertContactPerson)
(findHospital : sendAdvancedAmbulance)
alertContactPerson
sendAdvancedAmbulance
(alertContactPerson : Out(http://ewi158.ewi.utwente.nl/ontology/IOTyp
(sendAdvancedAmbulance : Out(http://ewi158.ewi.utwente.nl/ontology/core.owl#Ambulance))
Out(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#Alert)
Out(http://ewi158.ewi.utwente.nl/ontology/core.owl#Ambulance)

Figure 6.12: Emergency Service Composition

Both compositions comply with the service request. However, the non-functional properties differ in that the cost of the `SendAdvancedAmbulance` is higher.

6.3.2 Appointment Service

It is possible to generate the Appointment service composition shown in Figure 6.8, by using the following service request:

- Inputs:
  - `IOTypes.owl#CellNumber`
  - `IOTypes.owl#MedicalSpeciality`
- Outputs:

  - `IOTypes.owl#MedicalAppointment`

- Goals:
  - `Goals.owl#FindLocation`
  - `Goals.owl#FindMedicalCenter`
  - `Goals.owl#MedicalAppointment`
  - `Goals.owl#FindDoctor`

The composition graph is shown in Figure 6.13

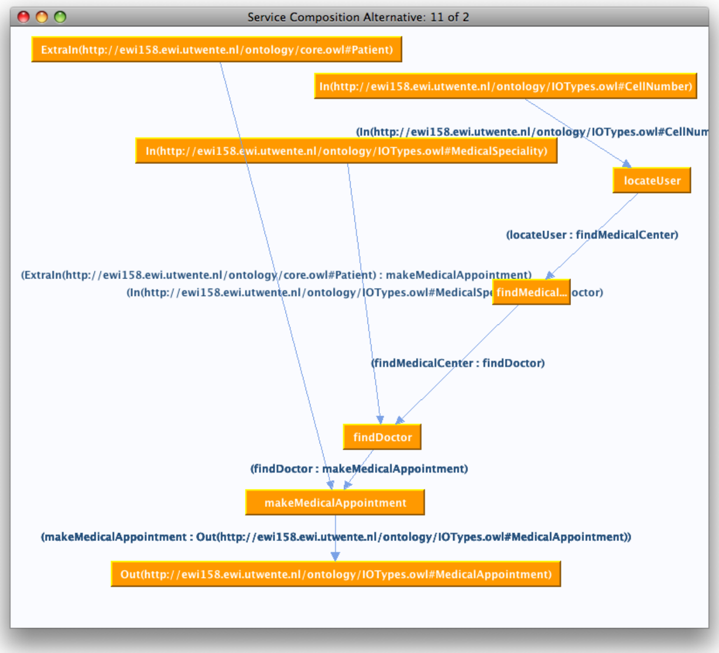

Service Composition Alternative: 11 of 2
ExtraIn(http://ewi158.ewi.utwente.nl/ontology/core.owl#Patient)
In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#CellNumber)
(In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#CellNum
In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#MedicalSpeciality)
locateUser
(locateUser : findMedicalCenter)
(ExtraIn(http://ewi158.ewi.utwente.nl/ontology/core.owl#Patient) : makeMedicalAppointment)
(In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#MedicalSpe
findMedical...
octor)
(findMedicalCenter : findDoctor)
findDoctor
(findDoctor : makeMedicalAppointment)
makeMedicalAppointment
(makeMedicalAppointment : Out(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#MedicalAppointment))
Out(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#MedicalAppointment)

Figure 6.13: Appointment Service composition result

6.3.3 Doctor At Home Service

An alternative service composition, shown in Figure 6.14, is also generated from the same service request as in the appointment service. This composition is generated because `makeAtHomeMedicalAppointm` has `AtHomeMedicalAppointment` as goal, which is a subclass of the requested `MedicalAppointment` goal. This service composition supports indeed the Doctor At Home scenario shown in Figure 6.9

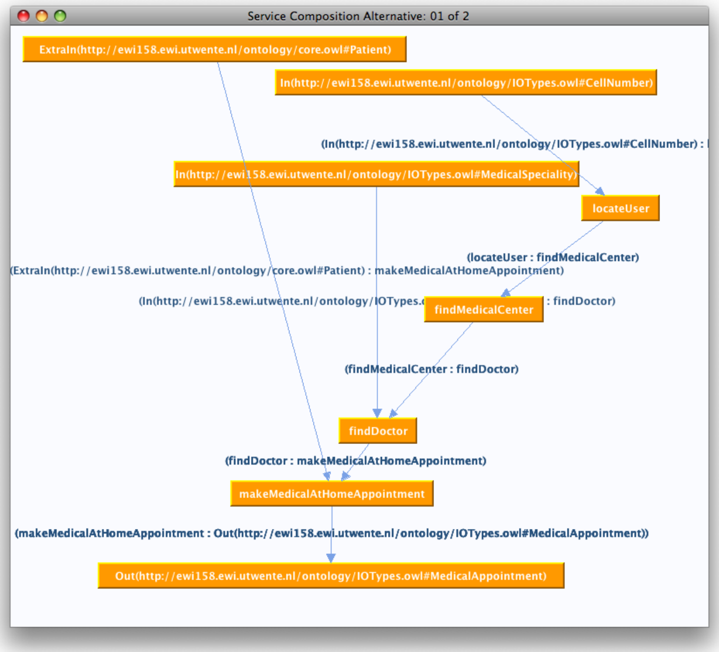

Service Composition Alternative: 01 of 2
ExtraIn(http://ewi158.ewi.utwente.nl/ontology/core.owl#Patient)
In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#CellNumber)
(In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#CellNumber) :
In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#MedicalSpeciality)
locateUser
(locateUser : findMedicalCenter)
(ExtraIn(http://ewi158.ewi.utwente.nl/ontology/core.owl#Patient) : makeMedicalAtHomeAppointment)
(In(http://ewi158.ewi.utwente.nl/ontology/IOTypes.
findMedicalCenter
: findDoctor)
(findMedicalCenter : findDoctor)
findDoctor
(findDoctor : makeMedicalAtHomeAppointment)
makeMedicalAtHomeAppointment
(makeMedicalAtHomeAppointment : Out(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#MedicalAppointment))
Out(http://ewi158.ewi.utwente.nl/ontology/IOTypes.owl#MedicalAppointment)

Figure 6.14: Doctor at home composition

6.3.4 Performance

In order to have an idea of the performance of the discovery process implementation and the composition process we measured the time that takes both tasks to complete in some of the scenarios presented before. The results are presented in the following table:

	Time (sec)		
Scenario	**Discovery Time**	**Composition Time**	**Total Time**
Emergency Service	3.8	2.02	5.82
Appointment Service	2.88	1.77	4.65
Doctor At Home Service	3.6	1.36	4.96

The time it takes to discover the services is mainly due to the time that takes to download the ontologies from the remote server where they are stored. Both Emergency Service and Appointment Service produce two different composition graphs each one, while the Doctor At Home Service produces only one composition graph.

7

Final Remarks

This chapter presents our conclusions about the work done in this Master project and guidelines for future work.

Section 7.1 presents our conclusions. Section 7.2 some directions for future developments.

7.1 Conclusions

In this Master project we have designed and implemented a service publication and service discovery mechanism to support dynamic composition of services.

The first task was the creation of a set of semantically annotated service descriptions, which allowed the testing of all other developed mechanisms. This was done using the SPATEL service description language, the language created in the SPICE project. Nowadays there are already several languages that allow one to describe semantically services, namely OWL-S, SAWSDL, WSML, among others. We concluded that the most common approach is the usage of ontologies written in OWL to perform the semantic description of services.

SPICE defined a service composition life-cycle. Having such a life-cycle stating the different phases of the service development and composition is essential to identify the different steps in the process, and then create the necessary mechanisms to address them. In thesis we mainly focused in the service creation, publication and discovery phases. From the literature studies performed, we concluded that not much work exist on the definition of a life-cycle for dynamic service composition. From the work performed in the definition of the life-cycle we published the following article [37].

The mechanisms defined for service creation, publication, request and discovery were successfully tested independently, using the created set of services.

Finally we have integrated successfully the developed service publication and service discovery mechanisms with the dynamic service composition framework. This integration was validated with the tests performed. We concluded the performance of the framework is suitable for the needs of a dynamic service composition at design time. However optimizations can be foreseen, which will be very useful to support runtime dynamic service composition. These optimizations can mainly be made in the semantic reasoning procedure, which will lead to reductions in the discovery and composition times.

7.2 Future Work

The service composition field has a lot of potential directions of work and there are still many issues that need to be addressed.

This project focused in semantic-based service composition. Nevertheless, syntactic-level data representation also needs to be considered. For instance, the way parameter data is structured and coded has to be considered. For example, the output of a service could be a phone number stored as a String (character representation) and the input of the next service could be also a phone number, but stored as a Long (numerical representation). While at semantic level there is an exact match, at syntactic level there is a mismatch. Our developed solution does not address this problem at the moment and improvements can be probably made, like, for example, syntactic type transformation.

Also the use of ontologies needs some attention. The composition possibilities are bound to the size of the ontology. It is not likely that a single global ontology can cover the needs of everyone and the most common and realistic scenario is that each company develops its own ontologies. These ontologies need to be shared with other business partners, which may already have their own ontologies. Total integration may not be feasible, for example, because conceptualization performed by the different companies is different. Hence the need of finding the mappings between the different ontologies is a really important issue.

This project is based on service descriptions that contain GQIOPE information. This means that a service request should include the desired goals, inputs, and outputs. An end-user may not be capable of making a complete request that fulfills his needs using the

GQIOPE approach. Thus, a potential direction of work is to find new approaches of describing services that ease the user requirement's analysis and enables composition based on more abstract mechanisms, such as natural language requests.

A limitation of the approach we followed to compose services is that it only composes services sequentially or in parallel. Other kind of compositions like loops or runtime choices are currently not supported. Furthermore, our services are based on request/response interactions, and service choreographies are not supported.

Only the service creation phase of the service life-cycle has been considered in this Master project. However, this is only a part of the whole service life-cycle. There are some issues that have not been addressed, for instance, service updates and versioning. As an example, when a service is updated with a new version it is possible that existing compositions are not longer valid.

It would be also really interesting to give support to context-aware services. Context-aware services consider user context information like location, time, presence or terminal capabilities. A service composition may be modified at runtime according to this context information. As an example, making or receiving phone calls is prohibited on board of trains in Japan. A context-aware service composition would avoid using voice-featured services but show the information on the screen when the user is on the train. In a context-aware environment services could be enabled and disabled dynamically according to user information or preferences.

7. FINAL REMARKS

References

[1] Rama Akkiraju, Joel Farrell, John Miller, and Meenakshi Nagarajan. **Web Service Semantics - WSDL-S**. W3c member submission, W3C, November 2005. Available from: http://www.w3.org/Submission/WSDL-S/. 13

[2] J. P. Almeida, A. Baravaglio, M. Belaunde, P. Falcarin, and E. Kovacs. **Service Creation in the SPICE Service Platform**. *Wireless World Research Forum meeting on Serving and Managing users in heterogeneous environments*, November 2006. 22, 24

[3] Mariano Belaunde. **Spatel Engine Online Site** [online]. February 2008. Available from: http://mdasoa.elibel.tm.fr/spatelengine/. 28

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. **The Semantic Web**. *Scientific American Magazine*, May 2001. Available from: http://www.sciam.com/article.cfm?id=00048144-10D2-1C70-84A9809EC588EF21. 4, 8

[5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. **Web Services Description Language (WSDL) 1.1**. W3C note, W3C, March 2001. Available from: http://www.w3.org/TR/wsdl. 3, 11

[6] Clark and Parsia, LLC. **Pellet: The Open Source OWL DL Reasoner** [online]. May 2008. Available from: http://pellet.owldl.org/. 8, 48

[7] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. **UDDI Version 3.0.2**. Uddi spec technical committee draft, OASIS, October 2004. Available from: http://www.uddi.org/pubs/uddi_v3.htm. 14

[8] Jos de Bruijn, Rubén Lara, Axel Polleres, and Dieter Fensel. **OWL DL vs. OWL flight: conceptual modeling and reasoning for the semantic Web**. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 623–632, New York, NY, USA, 2005. ACM. Available from: http://doi.acm.org/10.1145/1060745.1060836. 11

[9] Jos de Bruin. **The Web Service Modeling Language**. Technical report, WSMO, June 2006. Available from: http://www.wsmo.org/wsml/wsml-syntax. 4

[10] JOS DE BRUIN, HOLGER LAUSEN, RETO KRUMMENACHER, AND AXEL POLLERES. **D16.1v0.21 The Web Service Modeling Language WSML**, October 2005. Available from: http://www.wsmo.org/TR/d16/d16.1/v0.21/20051005/. 11

[11] DIGIUM. **Asterisk :: The Open Source PBX & Telephony Platform** [online]. May 2008. Available from: http://www.asterisk.org/. 20, 29

[12] ERICSSON. **IMS - IP Multimedia Subsystem** [online]. Available from: http://www.ericsson.com/technology/tech_articles/IMS.shtml. 18

[13] ERICSSON. **Parlay X web services** [online]. March 2007. Available from: http://www.ericsson.com/mobilityworld/sub/open/technologies/parlayx/index.html. 3, 12

[14] ESSI WSMO WORKING GROUP. **Web Service Modeling Ontology**. Available from: http://www.wsmo.org/. 4

[15] FRAUNHOFER FOKUS. **OpenIMSCore.org** [online]. 2008. Available from: http://www.openimscore.org/. 19, 29

[16] DONNA GRIFFIN AND DIRK PESCH. **A Survey on Web Services in Telecommunications**. *IEEE Communications Magazine*, **45**(7), July 2007. 3

[17] TOM GRUBER. **Ontology (Computer Science) - definition in Encyclopedia of Database Systems** [online]. June 2008. Available from: http://tomgruber.org/writing/ontology-definition-2007.htm [cited June 2008]. 7

[18] HUGO HAAS, DAVID BOOTH, ERIC NEWCOMER, MIKE CHAMPION, DAVID ORCHARD, CHRISTOPHER FERRIS, AND FRANCIS MCCABE. **Web Services Architecture**. W3C note, W3C, February 2004. Available from: http://www.w3.org/TR/ws-arch/. 3

[19] C. HARDING. **Definition of SOA** [online]. June 2006. Available from: http://opengroup.org/projects/soa/doc.tpl?gdid=10632 [cited December 2007]. 3

[20] JEFF HEFLIN. **OWL Web Ontology Language Use Cases and Requirements**. W3c recommendation, W3C, February 2004. Available from: http://www.w3.org/TR/webont-req/. 4

[21] IBM RESEARCH: SERVICES SCIENCES, MANAGEMENT AND ENGINEERING. **Services definition**. Available from: http://www.research.ibm.com/ssme/services.shtml. 1

[22] SATYA KOMATINENI. **Understanding UDDI and JAXR** [online]. February 2002. Available from: http://www.onjava.com/pub/a/onjava/2002/02/27/uddi.html [cited May 2008]. 14

[23] F. LECUE AND A. LEGER. **A Formal Model for Semantic Web Service Composition**. *Lecture Notes in Computer Science*, **4273**:385, 2006. Available from: http://iswc2006.semanticweb.org/items/Lecue2006uq.pdf. 37

[24] Freddy Lécué, Eduardo Silva, and Luís Ferreira Pires. **A Framework for Dynamic Web Services Composition**, November 2007. 36, 37, 38

[25] David Martin, Mark Burstein, Jerry Hobbs, and Ora Lassila. **OWL-S: Semantic Markup for Web Services**. W3c member submission, W3C, November 2004. Available from: http://www.w3.org/Submission/OWL-S/. 4, 12

[26] Deborah L. McGuinness, Chris Welty, and Michael K. Smith. **OWL Web Ontology Language Guide**. W3c recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/. Available from: http://www.w3.org/TR/owl-guide/. 4, 8

[27] Henrik Frystyk Nielsen, Marc Hadley, Anish Karmarkar, Noah Mendelsohn, Yves Lafon, Martin Gudgin, and Jean-Jacques Moreau. **SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)**. W3c recommendation, W3C, April 2007. Available from: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. 12

[28] Rogier Noldus and Ralf Keller. **Multi-access for the IMS network**. *Ericsson Review*, **2**:81–86, 2008. Available from: http://www.ericsson.com/ericsson/corpinfo/publications/review/2008_02/files/7_IMA.pdf. 19

[29] OASIS. **UDDI Online community for the Universal Description, Discovery and Integration** [online]. 2008. Available from: http://uddi.xml.org. 14

[30] OASIS Web Services Business Process Execution Language (WSBPEL) TC. **Web Services Business Process Execution Language Version 2.0**. Oasis standard, OASIS, April 2007. Available from: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 3, 15

[31] Phelim O'Doherty. **JAIN SLEE Principles** [online]. May 2003. Available from: http://java.sun.com/products/jain/article_slee_principles.html. 18

[32] OMG. **MDA** [online]. Available from: http://www.omg.org/mda/. 15

[33] OMG. **UML** [online]. Available from: http://www.uml.org. 16

[34] OpenSER.org. **OpenSER SIP Server** [online]. May 2008. Available from: http://www.openser.org/. 20

[35] Red Hat Middleware. **JBoss.com** [online]. 2007. Available from: http://www.jboss.com/. 17, 18

[36] Red Hat Middleware. **Mobicents** [online]. May 2008. Available from: http://www.mobicents.org-a.googlepages.com/index.html. 18, 29

[37] Eduardo Silva, Jorge Martínez López, Luís Ferreira Pires, and Marten van Sinderen. **Defining and Prototyping a Life-cycle for Dynamic Service Composition**. In *2nd Workshop on Architectures, Concepts and Technologies for Service Oriented Computing (ACT4SOC), Porto, Portugal*, Portugal, July 2008. INSTICC Proceedings. 31, 32, 38, 65

[38] SPICE Consortium. **FP6 IST Project SPICE** [online]. Available from: http://www.ist-spice.org. 4, 42

[39] SPICE Deliverable 5.1. **Advanced Language for Value Added services composition and creation**, August 2006. 4, 33

[40] Stanford Center for Biomedical Informatics Research. **The Protégé Ontology Editor and Knowledge Acquisition System** [online]. 2008. Available from: http://protege.stanford.edu/. 10, 27

[41] StarUML. **StarUML - The Open Source UML/MDA Platform** [online]. Available from: http://staruml.sourceforge.net/en/. 26, 28

[42] Sun Microsystems. **JSLEE and the JAIN Initiative** [online]. Available from: http://java.sun.com/products/jain/index.jsp. 18

[43] Sun Microsystems. **Java API for XML Registries (JAXR)** [online]. 2008. Available from: http://java.sun.com/webservices/jaxr/. 29

[44] The Apache Software Foundation. **Apache jUDDI - Web Services - JUDDI** [online]. 2007. Available from: http://ws.apache.org/juddi/. 29, 41, 46

[45] The Apache Software Foundation. **Apache Tomcat** [online]. January 2008. Available from: http://tomcat.apache.org/. 17, 29

[46] The Eclipse Foundation. **Eclipse Modeling Framework Project** [online]. April 2008. Available from: http://www.eclipse.org/modeling/emf/. 16, 44, 47

[47] The Eclipse Foundation. **Eclipse Web Tools Platform (WTP) Project** [online]. 2008. Available from: http://www.eclipse.org/webtools/ [cited May]. 17, 29, 35

[48] The Eclipse Foundation. **Eclipse.org home** [online]. May 2008. Available from: http://www.eclipse.org. 2, 17, 43

[49] The Parlay Group. **Parlay X Web Services Specifications**, June 2007. Available from: http://www.parlay.org/en/specifications/pxws.asp. 3, 12

[50] Dmitry Tsarkov. **OWL: FaCT++** [online]. May 2007. Available from: http://owl.man.ac.uk/factplusplus/. 8

[51] University of Otago. **About JAIN SLEE** [online]. December 2007. Available from: http://jainslee.org/slee/slee.html. 3, 18

[52] Frank van Harmelen and Deborah L. McGuinness. **OWL Web Ontology Language Overview**. W3c recommendation, W3C, February 2004. Available from: http://www.w3.org/TR/owl-features/. 8, 9

[53] W3C. **Semantic Annotations for WSDL Working Group** [online]. 2006. Available from: http://www.w3.org/2002/ws/sawsdl/. 13

[54] CARL ZETIE. **Eclipse Adoption Rates: Emerging Into The Light**. *Forrester Research*, July 2005. Available from: http://www.forrester.com/Research/Document/Excerpt/0,7211,37238,00.html. 2

Appendices

Appendix A

Ontologies

This chapter shows the graphical representation of the ontologies used within this Master project.

The core ontology (Figure A.1) contains terms related to people, places, devices, etc.

The goal ontology (Figure A.2) contains services' goals.

The IO types ontology (Figure A.3) contains input and output parameters.

The Non-Functional Properties ontology (Figure A.4) contains terms such as cost, bandwidth, response time, etc.

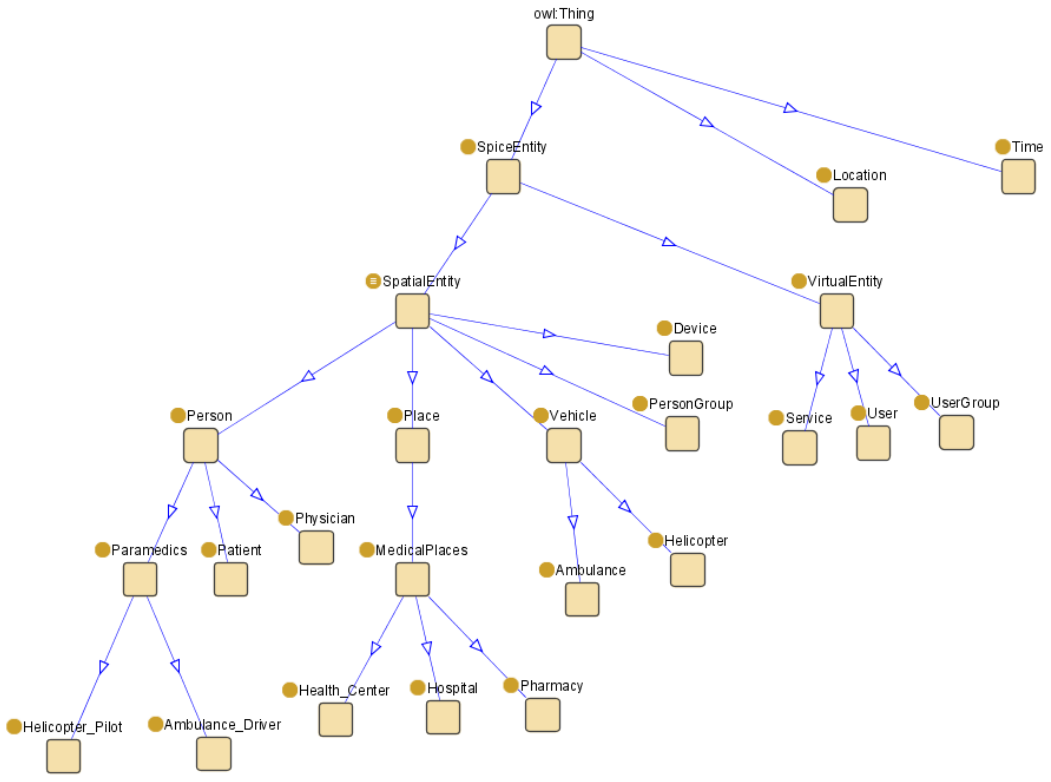


Figure A.1: Core ontology visualisation

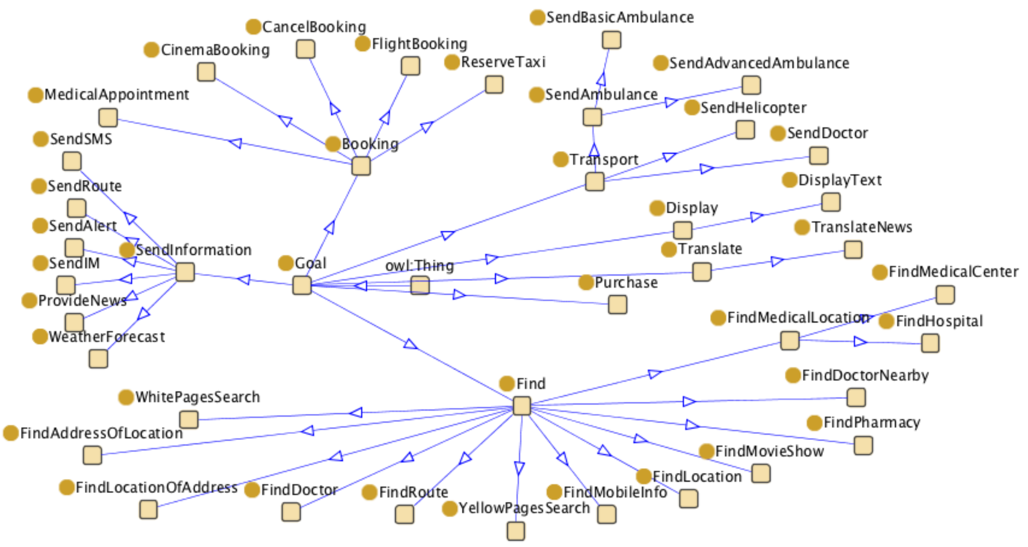


Figure A.2: Goals ontology visualisation

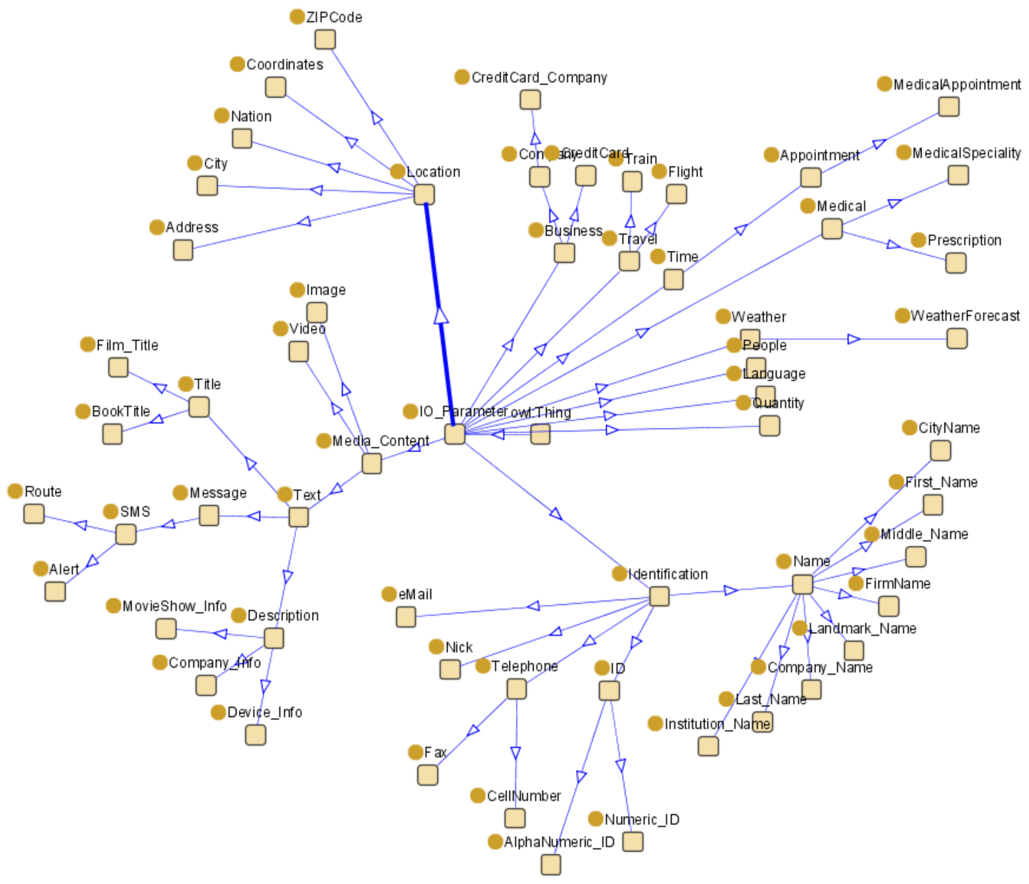


Figure A.3: IO Types ontology visualisation

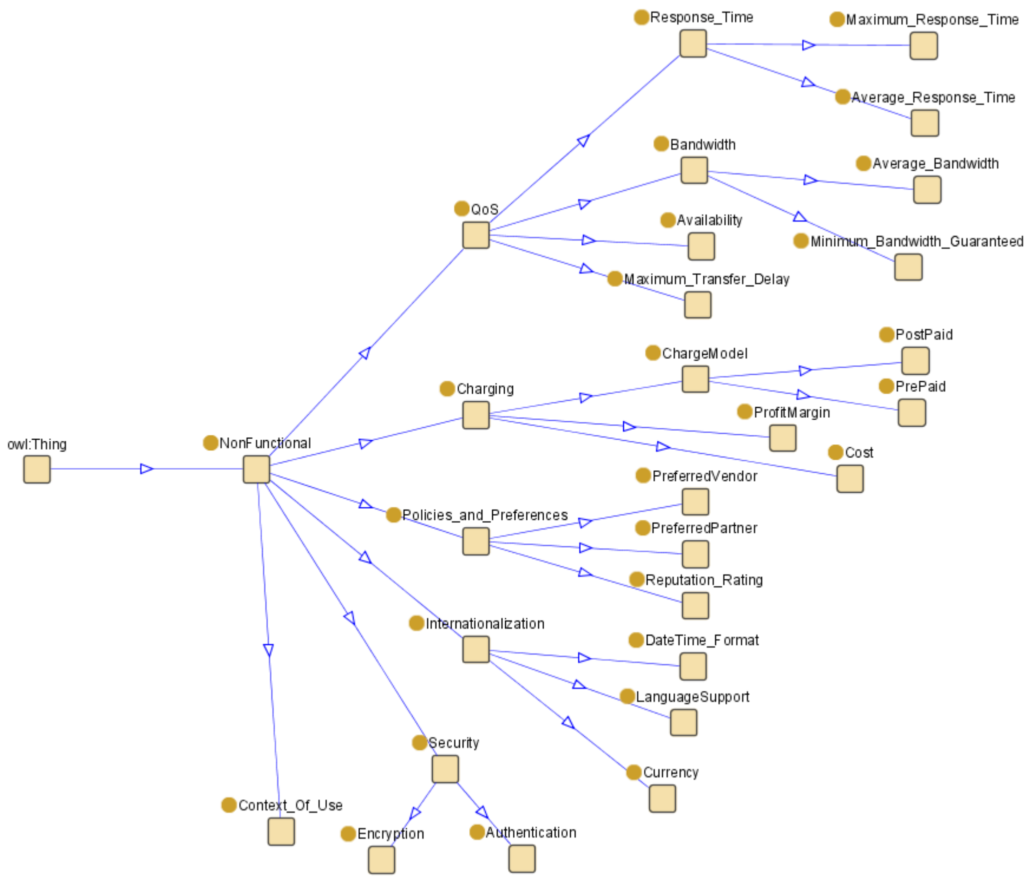


Figure A.4: Non Functional Properties ontology visualisation

License and Contact Information

License

This work is licensed under the Creative Commons Attribution 3.0 Netherlands License. To view a copy of this license, visit http://creativecommons.org/licenses/by/3.0/nl/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Contact Information

The author can be contacted via e-mail. Please send your message to the address jorgeml@jorgeml.net