UNIVERSITATEA **POLITEHNICA** DIN BUCUREŞTI
Facultatea Automatică şi Calculatoare
Departamentul de Calculatoare

Nr. Decizie Senat 11463 din 23.07.2012

TEZĂ DE DOCTORAT

Interoperabilitatea semantică a aplicațiilor din domeniul medical bazate pe agenți software şi servicii web

Semantic Interoperability in Healthcare Systems Based on Software Agents and Web Services

Autor: Ing. Sorin-Alexandru Cristescu

COMISIA DE DOCTORAT

Preşedinte	Prof. dr. ing. Theodor Borangiu	de la	Universitatea POLITEHNICA din Bucureşti
Conducător de doctorat	Prof. dr. ing. Florica Moldoveanu	de la	Universitatea POLITEHNICA din Bucureşti
Referent	Prof. dr. ing. Adina Magda Florea	de la	Universitatea POLITEHNICA din Bucureşti
Referent	Prof. dr. ing. Ioan Salomie	de la	Universitatea Tehnica Cluj-Napoca
Referent	Prof. dr. Ing. Costin Bădică	de la	Universitatea din Craiova

Bucureşti 2012

Semantic Interoperability in Healthcare Systems Based on Software Agents and Web Services

Table of Contents

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor, prof. dr. Florica Moldoveanu for her continuous support during my whole doctoral studies.
I would like to thank my friend and former Oracle colleague, Octavian Arsene, without whose enthusiasm for software agents I probably wouldn't have pursued this PhD.
Also many thanks go to my wife for her amazing patience during this whole period.

INTRODUCTION

1. Motivation

Traditionally, the medical institutions around the world have operated as autonomous entities, with little communication among each other. This has had a great impact on the quality of the medical act.

Think for example about the patient's medical record, which means in principle paper records of patient's medical history fragmented and spread all over the hospitals where the patient was investigated. For a certain diagnosis and treatment, a doctor usually relies on the patient's medical history, so it is extremely important to have access to it in an easy way, ideally via a centralized, reliable system that keeps the entire patient's history and automatically updates it when necessary.

Another important field which has manifested lack of communication and interoperability between medical institutions is patient monitoring. A patient dependent on dialysis typically needs to go to the hospital on a weekly basis and be connected there to a heavy dialysis machine.

A patient who has a heart attack on a street or at home has good odds to survive only if he/she can alert the emergency service and only if the latter comes on time and provides the proper approach to the patient's condition.

As the medicine progresses and the IT industry grows at an ever increasing pace, the intertwining between the two brings obvious benefits, with the ultimate goal of improving the quality of the medical act and thus the quality of life. Nowadays we talk about electronic patient record (EPR) as opposed to the fragmented information we are used to, we innovate in the area of tele-dialysis, which allows a dialysis dependent patient be treated at home, we already have proprietary protocols for remote monitoring of elderly people, hospitals and other medical institutions as well as the insurance companies communicate better with each other, etc.

Nevertheless, we are still far from an optimized flow of operations in the medical domain. What if an elderly patient who faints in the street or at home is monitored by a software entity, which autonomously alerts an electronic emergency service, which in turn informs the closest ambulance service, based on patient's symptoms and exact geographic coordinates ? No other human intervention is needed here until the medical staff operating the ambulances notices the alert and rush to the patient's location. Because the patient's symptoms are known to the ambulance staff, they can bring for example a defibrillator, potentially saving the patient's life. Moreover, as the patient is on the way, the reception of the hospital is also electronically alerted about the patient's condition, so

the right specialist is prepared to take over the case. The examples of interoperability could go on in the domain of tele-dialysis where the patients need to be remotely monitored, or in the domain of EPR, where various software entities could bring and put together the entire patient record, to be analyzed by a doctor during a consultation.

This document proposes an architecture based on software agents and web services that leverages semantic interoperability among medical entities in order to improve the medical flow of operations and ultimately to improve the quality of life. As the basis of this architecture we use existing standards and ontologies.

The first part focuses on the issues that are common to software agents and web services:

- annotating them with semantic information
- registering them
- semantically discovering them
- their communication

The second part deals with the proposal for a semantic search engine, as a means to register and discover the services (offered by both software agents and web services). With the annotations proposed in the first part, the searcher will use a measure for the match based on the so-called Tversky distance [36].

2. Scientific publications in connection with the thesis

Cristescu S., Moldoveanu F. *An Agent-Oriented and Service-Oriented Architecture in Medicine*. In: Annals of DAAAM for 2009 & Proceedings of the 20th International DAAAM Symposium, Published by DAAAM International, Vienna, Austria, 2009, pp. 1767-1769

Cristescu S., Moldoveanu F. *Patient Monitoring with Agents and Semantic Web.* In: Annals of DAAAM for 2010 & Proceedings of the 21th International DAAAM Symposium, Published by DAAAM International, Vienna, Austria, 2010, pp. 809-811

Cristescu S. *High Level Design of a Semantic Search Engine.* In: Annals of DAAAM for 2011 & Proceedings of the 22th International DAAAM Symposium, Published by DAAAM International, Vienna, Austria, 2011, pp. 567-568

Cristescu S., Moldoveanu F. *A Semantic Search Engine Implemented with Open Source*. In: International Conference on Innovative Technologies, IN-TECH 2012

Cristescu S., Moldoveanu F. *An Agent-Oriented and Service-Oriented Architecture in Medicine*, UPB Scientific Bulletin, ISSN 1454-234x, seria C, vol. 75, nr. 1/2013

Cristescu S., *Semantic Search and Invocation of Web Services*, Journal of Control Engineering and Applied Informatics, 2012 (submitted)

3. Thesis Outline

The thesis is split in two logical parts: Part 1 deals with the semantic aspects of agents and services interoperability, while Part 2 uses the ideas from Part 1 and proposes a semantic search engine as a mechanism to register, search and consume semantic services.

Chapter 1 formulates the issue at hand by looking at a potential scenario from the medical domain. Then it explains why software agents and web services are part of the solution. We also deal with a known issue in the current medical domain: the patient's medical history, also known as electronic patient record.

Chapter 2 deals with the cornerstone of semantic interoperability: ontologies. It also limits the research domain to the medical one, by looking at the HL7 standard. Then we explain how to introduce semantics in the equation – namely by using semantic annotations. We also detail the semantic interoperability by explaining how, once the services are annotated, they can be registered and semantically discovered. Finally, we present the role played by logic inferences in the world of communicating agents and services.

Once the services/agents are annotated, registered and discovered, they can be consumed, typically by other services/agents. Thus chapter 3 deals with the communication among services and agents. We also refer to RESTful services and their advantages. We mention how they can be added to the mix.

We start Part 2 with chapter 4, where we present the current research in the area of semantic matching.

In chapter 5 we introduce the idea of a semantic search engine as a means to register and discover the services/agents and also to replace the UDDI standard, not supported any more. Then we propose a design for the semantic search engine called *Semmed*.

Chapter 6 details the search algorithm performed by the proposed search engine. It also refers to ontology matching from the perspective of the search engine.

Finally, we draw the conclusions, referring also to our original contributions and future work.

We also provide an Appendix with some implementation details of the search engine components, as well as with an example of a JADE agent behavior which uses Jena logic inference.

Part 1: Standards for Semantic Interoperability of Medical Applications – Multiagent and Service Oriented Architectures

CHAPTER 1

BACKGROUND

1.1 The Case for Software Agents

Let's assume the following scenario:

"John, a 70 year old grandfather is taking his 2 year old grandson for a walk in the park. Suddenly John feels bad and collapses to the ground. There is nobody at that hour in the park. Luckily, John wears his intelligent watch, which, by the means of a software agent and wireless Internet access, alerts the emergency service. Namely, it sends the patient's personal data, his symptoms, his medical record as well as his current location. Knowing the patient's location, the emergency service looks up the closest available ambulance, by informing the ambulance dispatcher agents in the neighborhood. In less than 5 minutes, an ambulance comes with the defibrillator prepared and transports the patient to the closest hospital. On the way to the hospital, the ambulance agent gathers the patient's symptoms and medical history from the patient's agent and informs the hospital reception agent about the patient's arrival. Having received the patient's symptoms and his medical history, the hospital reception agent looks up an available specialist, who is later helped by his own agent in checking and diagnosing the patient. From this point on, the specialist's agent monitors the patient by communicating with his personal agent."

The patient's personal agent would help in the same way in other similar circumstances, e.g. a car accident: as soon as the patient's vital signs deteriorate, the agent can contact the emergency service. Moreover, an agent embedded in the car could also take action, e.g. informing the police, the insurance company, etc.

What are the properties of agents that make them suitable for such a scenario ?

1.1.1 Multi-Agent Systems

It's impossible to give a unique, unambiguous definition of a software agent. However, most researchers agree that an agent is a software program that acts on behalf of a user and it is different from other types of software because it can take decisions autonomously and because it might manifest certain characteristics that are found in intelligent beings, such as beliefs, desires, intentions (BDI).

Typically we are interested in several characteristics of the software agents that make them appropriate in the scenario described above and in general in the medical domain. These features are described in the next sections.

Another important concept is a multi-agent system (MAS), which is a system composed of multiple interacting agents that try to solve together problems that would be impossible (or very difficult) to solve by individual agents.

In this paper we deal with a system composed of agents and web services that communicate seamlessly. We focus here on the registration, semantic discovery and invocation of the agents and services, rather than on their cooperation in order to solve specific tasks. Nevertheless, the communication is a cornerstone for any further developments, without which a MAS is impossible to exist. On top of that we can add complex algorithms. Let's assume that three agents communicate at a certain moment: the patient's agent, the doctor's agent and the agent of the insurance company. Together they reach the *Nash equilibrium* [40] in a game theory problem that has as outcome the best decision of what to do with the patient: send him home with the prescribed treatment or hospitalize the patient and if so, for how many days. This is just an example of problems solved by a MAS and in the daily life there are many other problems to be solved, from fixing an appointment between a patient and his/her family doctor to collecting the pieces of the patient's medical record and bringing them together to the doctor for analysis. Discussing the Nash equilibrium further is outside the scope of this document, but it is one of the topics for future research.
Besides medicine, MAS are researched in many technical domains, most notably road traffic management [7] and logistics.
There are also several MAS platforms that allow users to develop multi-agent applications. The best known is the FIPA-compliant [14] open source JADE [1], which we also use in our first prototype (section 2.3). The latest version at the time of this writing, 4.1.1, released in November 2011, supports the Android operating system, which makes JADE even more suitable for the scenario described above: personal software agents can be now deployed on smart phones.

1.1.2 Autonomy

It is essential that an agent performs actions without user input. The scenario above relies on this important feature of agents, since the patient might be unconscious, yet his agent will alert the emergency service.
The agent senses the environment – in this scenario the environment is constituted by sensors stitched to the patient's body: the watch itself where the agent is installed and/or wireless sensors to monitor the heart and potentially other organs. Every few minutes the

agent collects health info from these sensors, uses a simple inference engine (see section 2.7) to derive new rules and if the symptoms deteriorate, it alerts the emergency service. Our prototype (section 2.3) does that using the open source JADE MAS framework and the Jena [35] inference engine.

1.1.3 Mobility

Mobility means moving the processing to the data, performing tasks on that data and returning with the result. This is typically useful in circumstances such as sporadic network connectivity and/or low bandwidth, which is exactly what we describe in the scenario above: John is in a park, where there is limited wireless Internet connectivity. In such a case, the agent autonomously finds the proper moment to migrate to the destination host, where it alerts the emergency service. Thus the combination of autonomy and mobility helps in dealing properly with such a scenario.

As mentioned before, our first prototype (section 2.3) is built on JADE, which supports mobile agents that migrate across machines or between processes on the same machine. Recently JADE introduced support for the Android operating system, thus enabling smart phones as hosts for software agents.

1.1.4 Intelligence

Another important feature of an agent that we exploit is the intelligence, i.e. the capability of an agent to learn from previous experiences and improve its solutions to certain problems.

Intelligence is a feature manifested both by a single agent in isolation and by a MAS. The former case is important in scenarios such as an agent helping an imaging specialist analyzing medical images. Such an agent could learn e.g. how to identify polyps on the colon and notify the medical specialist. The latter case is important in scenarios where the agents composing a MAS learn from experience and improve their contribution to the final solution, thus improving the quality of the overall solution.

JADE doesn't intrinsically support intelligent agents, but various third-party add-ons do. One of them is Jadex [54], an implementation of the BDI model (Beliefs, Desires, Intentions) – a well known model for cognitive intelligent agents.

Of course, nothing prohibits us from programming intelligence into our JADE agents, by employing any of the machine learning techniques: supervised learning, unsupervised learning or reinforcement learning.

A recent example of an intelligent agent is Apple's *Siri* [52], installed on iPhone 4S (and ported to other versions), which recognizes speech and answers back to any question. Sometimes, if the question is unclear or vague, Siri asks back for clarification. It learns from examples, and improves its behavior over time. As opposed to other so-called

agents, Siri doesn't simply extract keywords from your speech and looks them up, but rather tries to derive meaning, i.e. semantics [53].

1.2 The Case for Semantic Web

Web services have been around for a while now. We've been using them in many scenarios, from checking the online train schedule in a certain region to verifying our credit card credentials, to getting the weather predictions at our location or anywhere else in the world. Figure 6 shows how services are registered, discovered and consumed.
Semantic web is a relatively new kid on the block. So far it has been used heavily as a research topic, with too few applications in industry. But there's change on the horizon, with Google reportedly moving towards semantic web adoption.
The "regular" (i.e. non-semantic) web only contains the two bottom layers in Figure 1. The rest is specific to the semantic web.

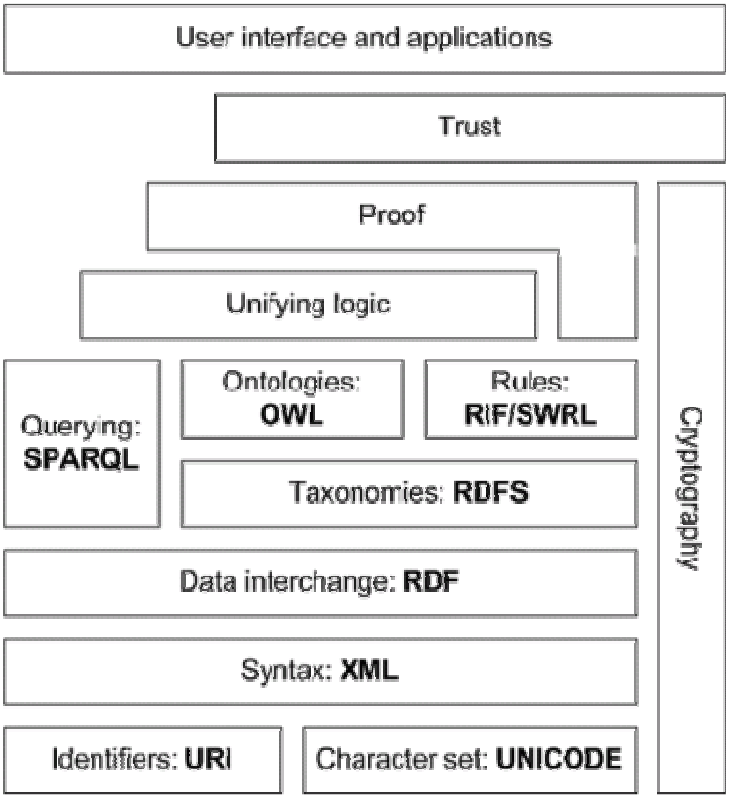


Figure 1: The famous "semantic web layer cake" (source: "Semantic Web – XML 2000", slide 10, Tim Berners-Lee, http//www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html)

The XML layer provides the syntax for content structure of the online documents, without associating any semantics with the data contained within.
RDF is a fundamental standard of the Semantic Web, expressing data models, which refer to objects and how they relate to each other. RDFS (RDF Schema) [48] extends RDF and is a general-purpose language for representing simple RDF vocabularies on the

web. OWL (Web Ontology Language) builds on RDFS and allows applications to process and interpret the content of information, as opposed to just presenting information to human readers. Thus OWL enhances the machine interpretability of the Web, by providing vocabulary additional to XML, RDF, RDFS and a formal semantics.
The top layers – unifying logic, proof and trust – are not yet entirely realized. Proof refers to the ability to prove that certain statements on the Web are true or false. This is based on logic – how was the answer derived. Trust is really what the name says – it enables autonomous communicating parties to achieve bi-lateral agreements based on reciprocal trust.

The scenario shown in the previous section refers to "agents" and "services". As opposed to an agent, a service can be regarded as a public entity, always available, so it can be invoked from anywhere at any time provided its address, its binding and its contract are known. Agents are rather "private" entities performing their tasks on behalf of the user. An agent can't be used just by anyone at any time.
In the above scenario for example, it makes more sense to implement the emergency entity as a service, since it is publicly available to everybody. Implementing it as an agent would mean making it accessible only to other agents (and used through communication). Making it a (web) service opens it up to many potential types of clients. Nevertheless, an agent can be wrapped as a web service and thus made public – feature supported by the JADE [1] platform, which we've chosen to implement our prototype (section 2.3).
Thus, choosing between agents and services in order to implement certain functionality is not essential. Conceptually, both agents and services perform tasks on some input data and return some results. What we consider essential though is the ability to seamlessly integrate them in one platform. We'll come to this point in the next sections.

A service is basically an operation that obeys to a certain contract. Due to the heterogeneous character of the medical world, we can't expect to have the same services with the same input and output types being used by all medical institutions. The communication between medical institutions needs semantic interoperability, which can be obtained by employing ontologies (see section 2.2). For example, when a certain hospital asks a service for the electronic patient record (EPR) of a certain patient, it needs to provide certain identification data for that patient and it needs to understand/interpret the received patient record. All pieces of exchanged information need to be expressed in ontologies understood by all involved parties. Likewise, when a doctor writes the diagnosis in the patient's medical record, the symptoms have to be mapped to certain ontology (e.g. SNOMED-CT [16]) such that they are understood by any other entity (human or not) which can interpret that ontology.

1.3 Integrating Agents and Services

The scenario shown in section 1.1 is described from the perspective of software agents – entities that help their owners (humans) in achieving certain goals. Agents are based on communication paradigms (exchanging knowledge) and performing tasks by migrating through the network: the algorithm moves to data. For example an agent would migrate to a medical data repository, retrieve some records, perform some computation and bring the result back to where it started from.

The other, more popular, paradigm for executing remote tasks over the Internet is the web service, in which case data moves to the algorithm.

As it will result from next chapters, the issues of semantically annotating, registering, discovering and consuming services and agents are similar. There are thus many similarities between agents and services used for semantic interoperability.

Due to the widespread adoption of web services, there is immense profit for the agents which consume them. The most obvious way to go is the integration of legacy web services developed by medical institutions into the new scenarios we are envisioning in this project. The legacy services (offered either by agents or web services) could be annotated with semantic annotations and thus the full power of semantic search can be leveraged.

1.4 Case Study: Electronic Patient Record

We've started this document with a scenario that shows how software agents and semantic web services could be used in a certain daily life situation. The applicability of multiagent systems and semantic web spans over a large spectrum of use cases. An interesting one is presented in this section.

For years now there have been discussions about the so-called Electronic Patient Record, serving basically as a history of the medical events for a patient. The main idea is that typically a patient is treated at various hospitals/practices during his lifetime and thus his medical history is spread over these medical institutions, without any relationships among them. In effect, the medical history is fragmented and thus there is no possibility to retrieve facts from the past that could help in the current diagnosis/treatment. The proposals are invariably related to a centralized system that either holds the entire medical history or knows how to retrieve the pieces on request.

Our proposal is, briefly, to allow the patient's personal agent to find the pieces of the patient's medical history from the distributed sources.
A typical examination in a hospital might have as result findings that comprise medical images, text, voice recordings, electrocardiograms, etc. All these should be part of the patient's medical history. Since the personal agent is embedded and thus its resources are strictly limited, it cannot gather and hold physically all these findings. Rather, the agent holds a summary of each medical event and references to specialized entities that are able to retrieve detailed information about the patient's medical history. These entities are the software agents of the hospital/medical institution reception. Each of them is responsible for managing a certain piece of the medical history, for example the date and time of the patient's visits to the hospital. Further, each such agent has references to other specialized agents (organized by the specialties of that hospital), each of them managing a corresponding piece of the patient's medical history. For example, the agent of the CT imaging department is able to retrieve the medical images that have been taken with the patient during a certain CT scan examination.

There are several issues that have to be carefully considered here.

1. How does the patient's personal agent find the other involved agents ?

First of all let's note that the proposed multiagent system is a hierarchical, layered one, in that an agent from a certain layer only needs to know some of the agents from the layer below. Thus, the personal agent only needs to know about the reception agents and not about the medical specialties agents. This is suggested in Figure 2. The "layering" described here is pretty logical, since, as the patient is taken to the hospital, his/her agent

communicates to the hospital reception agent, as described in the scenario of section 1.1. For example, the patient's personal agent gets to know about the hospital reception agents H1, H2, H3 and H4, since the patient has been taken to these four hospitals so far. The hospital agent H1 gathers its info from the specialist agents S1.1 and S1.2, corresponding to the two specialists who consulted the patient in the hospital H1.

There are several scenarios that show how agents get to know about each other:

a. The scenario described at the beginning of section 1.1 tells about the personal agent contacting the emergency agent/service. This one in turn contacts the ambulance dispatcher agent and makes the link between this one and the patient's agent. From the ambulance agent, the patient's agent finds out the hospital reception's agent (of the hospital where the patient is taken to).
b. A patient can decide to visit his family doctor and thus he/she informs the personal agent about this intention. In effect, the personal agent contacts the family doctor's agent in order to fix an appointment (it's assumed that the personal agent has been configured with the address of the doctor's agent; in case the latter changes, see bullet 2 below). Part of the patient's medical history falls under the responsibility of the family doctor's agent.
c. A patient can visit a hospital/medical institution following the family doctor's recommendation. In this case, the personal agent has already contacted the hospital reception agent, by searching for it in a registry under the hospital's name and address.
d. A spontaneous visit to the hospital. In such a case, his/her personal agent senses the environment: it queries periodically a GPS service to find out the patient's geographical coordinates, with which it queries a service that is able to tell what is the address and type of institution found at that address. If the institution is a hospital, the patient's personal agent will search a UDDI-type registry for the reception agent registered with that hospital.
e. Variation of d: the other way around – when entering the hospital, the patient's watch is scanned via a RFID-like chip by the reception agent. After finding out the patient's identity, the reception agent looks up his/her personal agent in a UDDI-like repository. So, it's the reception agent who contacts first the personal agent.

Note that these scenarios mimic closely the current situation, which involves human operators.

2. What happens when an agent changes its address ? How are the related agents informed ?

In this case, again, we'd just mimic the scenario which involves human operators: if you call your family doctor and notice the number has been changed, you look it up in a

repository based on the doctor's name and address. Similarly, if the address of the family doctor's address has been changed, the patient personal agent looks it up in a UDDI-like repository based on the doctor's name and address.

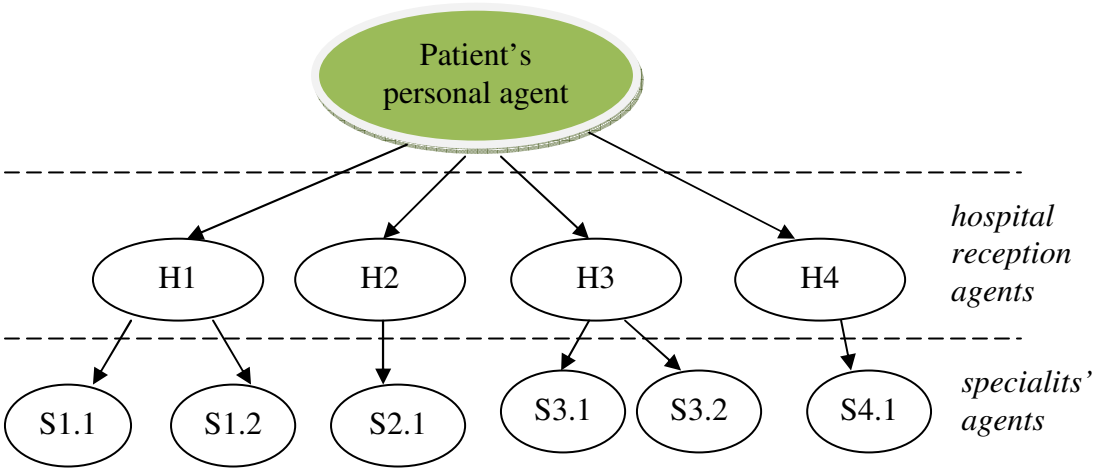


Figure 2: The layers that define what agents know

CHAPTER 2

SEMANTICS AND ONTOLOGIES

2.1 Evolution of Semantic Web

RDF (Resource Description Framework) [47] emerged from the need to identify the resources used over the Web. RDF describes statements consisting of triples: subject, predicate (or property) and object. A subject and a predicate are always resources, while an object is either a resource or a literal (value of a property). In fact, RDF is a data model: it describes data and metadata. Note that RDF is not a language, but just a conceptual abstraction. However, in order to be able to use it in practice, several notations are used: XML and non-XML based.

OIL (Ontology Inference Layer or Ontology Interchange Language) is a language defined on top of RDF which makes use of *description logics* to provide reasoning service and formal (logic-based) semantics. OIL is a European research result, while DAML (DARPA Agent Markup Language) is a US funding program which gave birth to a language similar in purpose to OIL.
DAML+OIL [45] is thus a joined effort developed on top of RDF in order to make it possible to describe ontologies.
OWL (Web Ontology Language) [46] derives from DAML+OIL and it is a formal W3C recommendation. Its main purpose is to define/describe ontologies.
OWL-S [44] builds on top of OWL and it is geared at defining ontologies for describing web services. More specifically, it enables the following tasks:

- automatic web service discovery
- automatic web service invocation
- automatic web service composition and interoperation

In parallel, another language has been developed by the DERI Institute in Innsbruck and Galway: WSML – Web Service Modeling Language [8]. Based on frame logic, the language is overly complex, abstract and so far it has remained an academic exercise. We are not going to deal with WSML in this paper.

Having a mechanism of describing ontologies, the question is how to use it in the world of web services? The W3C recommendation is SAWSDL [6], which builds on top of WSDL-S [49] and has as purpose to define a way to annotate WSDL [37] files with semantic annotations. WSDL-S accepts only annotations described in OWL. SAWSDL goes one step further and accepts any ontological language.

The rest of the document will make use of SAWSDL for semantic web service description and OWL for describing ontologies.
Sections 2.4 and 2.5 detail the similarities between agents and web services with regard to the standards they use to fix various issues. Table 1 is a preview of these topics.

Steps	*AOA*	*SOA*
Ontology integration	FIPA + HL7	SAWSDL + HL7
Semantic publication	DF + annotations/ Semantic search engine	UDDI + OWL/ Semantic search engine
Semantic discovery	Service Description + annotations/ Semantic search engine	UDDI + OWL/ Semantic search engine
Communication	FIPA-ACL + OWL	SOAP/REST + OWL
Ontologies based on HL7-RIM		

Table 1: Similarities between web services (SOA) and agents (AOA) with respect to semantic annotation, publication, discovery and communication

2.2 Ontologies

In the scenario presented in section 1.1, we've seen software agents communicating with each other and potentially with web services. But since agents and (web) services serve various purposes, are built by various companies and are configured in different ways, the environment we're facing is highly heterogeneous. Clearly, we can't assume that any two given agents will be able to communicate with each other just out of the box. Moreover, the same information can be represented in various ways. For example, the patient's location might be represented by an address or by two coordinates – latitude and longitude.

The solution here is semantic interoperability based on ontologies. Each communicating entity has to stick to certain ontology when creating a message. The communicating party has to understand this ontology or perform a translation to an ontology which it understands.

In an important contribution to the interoperability among medical information systems [3], the author highlighted the use of mobile agents and HL7-based ontology. HL7 [2] version 3 is a protocol created to enable semantic interoperability of medical applications. However, while an HL7-based ontology can be used for interoperability among medical institutions which keep their data in local formats, this will not be enough in a highly heterogeneous environment. We have to take into account that in the medical domain there are several stakeholders, namely patients, doctors, hospital managers, insurance companies, etc., each with its own goals and potentially employing various ontologies. Some of these ontologies will not even be medical, as suggested in the example with the patient's location. Another useful scenario is when agents make an appointment on behalf of their users, in which case they'd use a non-medical ontology.

So, in order to properly cover a scenario as the one presented in section 1.1, we need to take into account:

- various ontologies, including non-medical
- communication between parties that use different ontologies, which means *ontology mapping* is necessary

RIM

Reference Information Model is the backbone of semantic interoperability in HL7 v3. It consists of 5 basic concepts, as depicted in Figure 3: *Entity, Role, RoleLink, Act* and *ActRelationship*.

To better understand the role of RIM, think about the following issue: having an ontology (e.g. SNOMED [16] or LOINC [15]), why do I still need an information model ? Isn't a concept expressed in that ontology enough to be understood as is by other systems ?

The issue here is that a concept can mean various things, depending on the context. For example "blood pressure" might mean an observation, a diagnosis or maybe a problem (e.g. high value of blood pressure). An information model such as RIM is able to define this context.

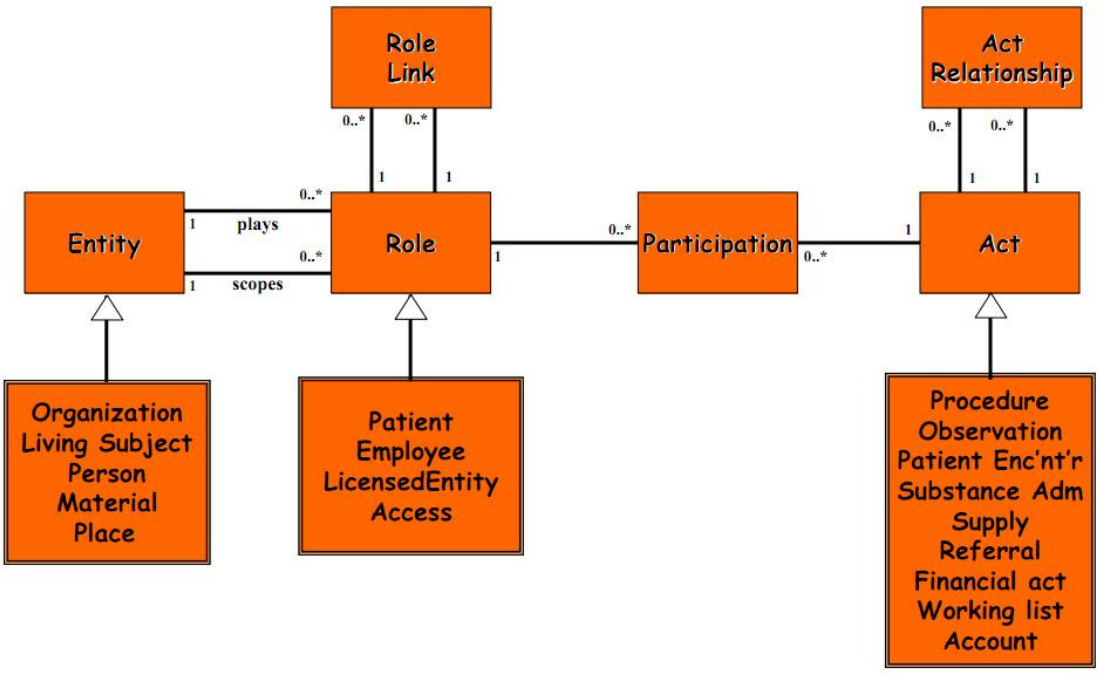


Figure 3: HL7 v3 RIM (Reference Information Model); source:
http://www.hl7.org/documentcenter/public/calendarofevents/himss/2009/presentations/Reference%20Information%20Model_Tue.pdf

It isn't difficult to put HL7 ontologies to work. Bhavna Orgun describes in [3] how she integrated an HL7-based ontology in JADE [1], the agent development framework we are also using for our prototype (section 2.3). The process is shown in Figure 4.
First a manual conversion from the official HL7-RIM UML models (Rational Rose) to RDF/OWL is necessary. The result OWL ontology is imported in Protégé [10] and from here, using the *OntologyBeanGenerator* plugin (owned by the Stanford University), the Java files are generated representing the FIPA-compatible ontology to be used in JADE.
We are using these ontology classes based on HL7-RIM in our prototype implemented with JADE.

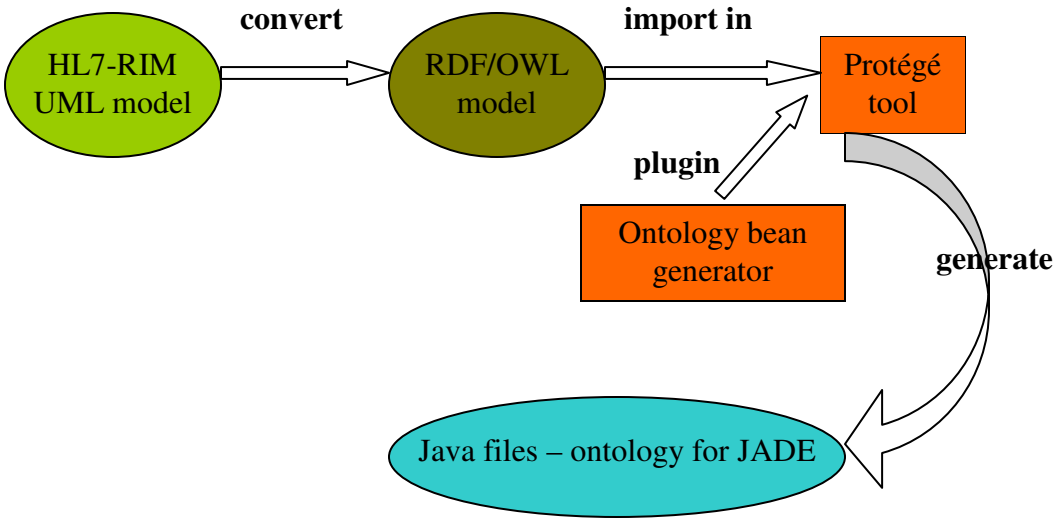


Figure 4: From HL7 UML model to ontology files for JADE

2.3 The First Prototype – Software Agents with JADE

In order to assess the feasibility of the scenario described in section 1.1, we've implemented a prototype with the main agents described in that scenario. Figure 5 illustrates these agents. Note that we've implemented the "Ambulance dispatcher service" as a web service, not as an agent. The others are all agents implemented using the JADE framework.

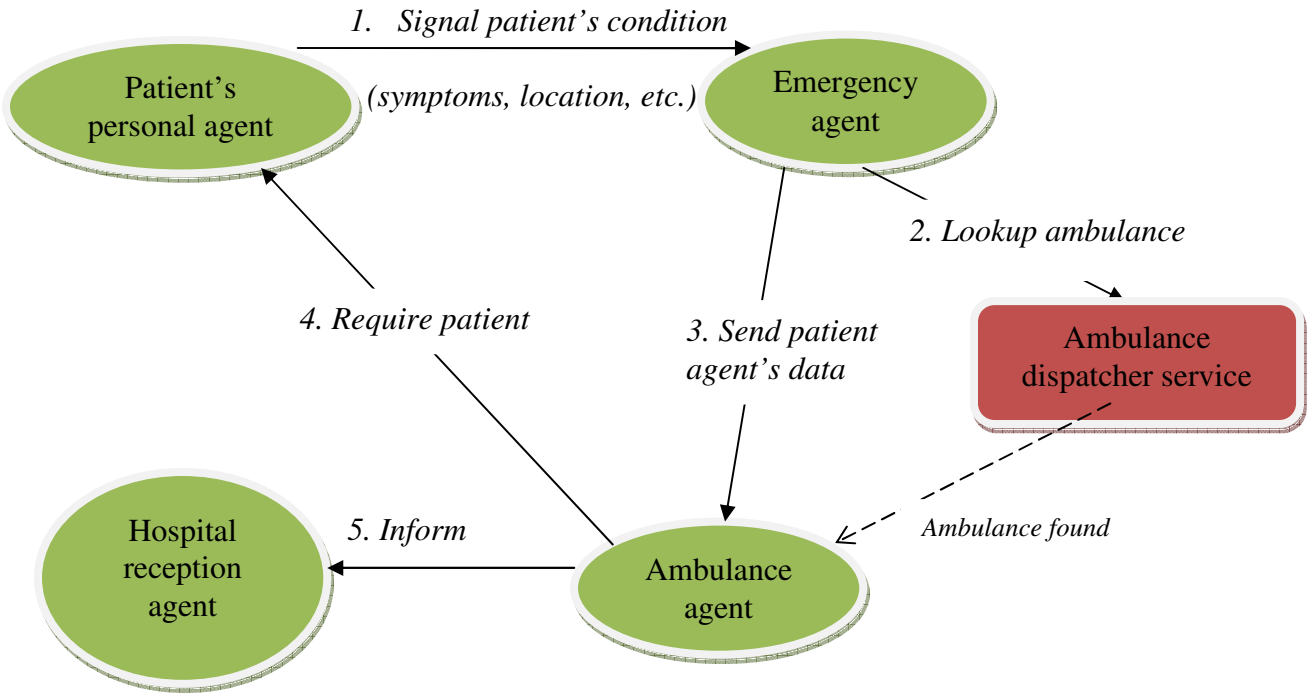


Figure 5: The main agents and services involved in the scenario presented in section 1.1

In JADE, each agent works by using one or more "behaviors", which can be started in parallel. In the Appendix we show how to implement such behaviors.
In our prototype we've hardcoded the address of the "Ambulance dispatcher service", so that the "Emergency agent" can contact it. This is not a realistic scenario and in the second prototype (section 5.4) we'll see how to correct it.
The "Patient's personal agent" simulates that it checks the patient's state of health periodically and, based on a set of rules, it infers whether it's needed to alert the

emergency service. The agent uses the reference engine Jena to do this inference. This is shown in the Appendix.
The agents know about each other via hardcoded addresses, which again, is not a realistic scenario. In the second prototype we'll show how the agents can find each other based on their semantics.
In JADE, the agents communicate to each other using the FIPA-ACL standard language. Each message needs to define a certain ontology. In our prototype we integrate the HL7-RIM medical ontology to JADE, as explained in section 2.2. Thus our agents' messages are based on this ontology. The common ontology is the key of understanding each other. In some domains, such as the medical domain, using a common, well-defined, ontology might work, but in practice it is more realistic to assume that several ontologies are used by the communicating agents. Sometimes an agent won't understand the ontology used by another, so a translation is needed (as between two persons trying to talk different languages to each other). This is the domain of ontology matching and we'll touch several aspects of it in this document.

Implementing the agents in JADE and making them call web service has been smooth and straightforward. Thus, the prototype demonstrates that it is realistic to build a system based on software agents and web services that interoperate in order to achieve a certain purpose. In our scenario the purpose is to give the best possible answer to the patient, i.e. find the closest ambulance, find the most appropriate specialist in the hospital, etc.
The prototype also shows how to employ logic (see also section 2.7) in order to make inferences based on concepts defined in certain ontologies and take decisions autonomously.
The second prototype will extend this one with semantic search capabilities, so that the agents find each other and the services they need to invoke. Next sections explain how to introduce semantics capabilities to agents and services.

2.4 Semantic Annotations

Semantic annotations have been around for some time now in various forms. Probably the most familiar form of annotation is a tag, i.e. a keyword or a group of keywords attached to a piece of text published online aiming to speed up the search, helping at the same time in finding more relevant and precise information.
However, there is hardly anything *semantic* in tagging. The real semantic annotation implies going at a deeper level and enhancing the unstructured data online with a context linked with a structured domain. The ultimate goal of semantic annotation is to enable not only better aimed search, but to transform a simple information retrieval (i.e. based on textual matching) into structured data retrieval (i.e. based on concepts from a domain).
Kim Semantic Annotation [4] and GoNTogle [5] are example of tools that allow users to annotate documents – online or not.

2.4.1 Semantic Annotations for Web Services

In the short scenario of section 1.1 we've shown that agents would need to interact with each other and potentially with (web) services. In section 2.2 we've argued that for this communication to take place, ontologies are necessary.
Let's focus now on web services. Traditionally, they are described in WSDL, using a format based on XML. This description is registered into a repository (traditionally UDDI-based) and thus a web service can be discovered and consumed, as shown in Figure 6. Or at least this is the theory. In practice the step of registering the service in a UDDI repository is skipped and instead the address of the service is hard-coded into the client. Also, a proxy is generated based on the WSDL file; the proxy is then compiled together with the client of the web service and thus the client can invoke the web service using the local proxy.

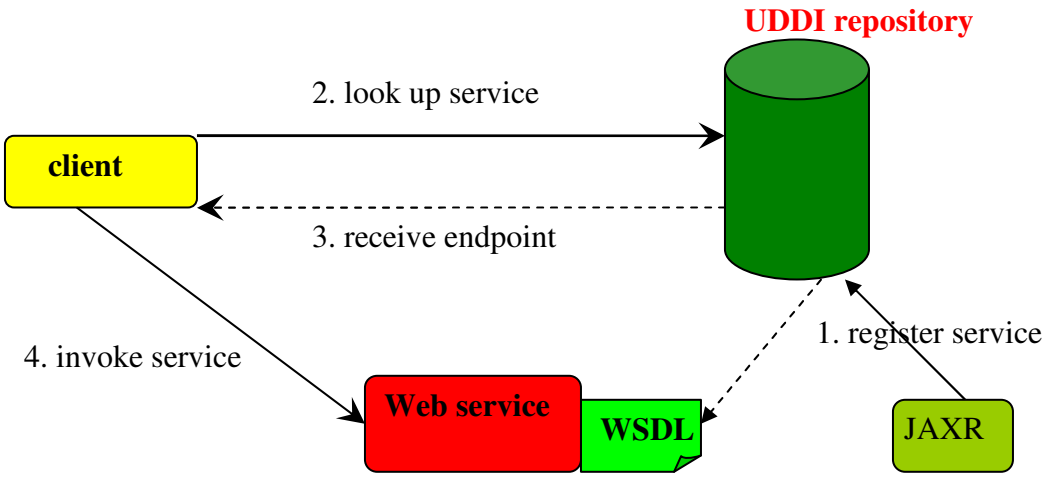


Figure 6: Discovering and consuming a web service

There are several issues with "standard" WSDL-described web services. First and foremost, WSDL is a purely syntactic representation of the web service operations with their parameters and results. WSDL doesn't say anything about the service behavior or about the semantics of its operations.

Just as one can add semantic annotations to a document (see previous section), web services can also be semantically annotated. The W3C recommendation is **SAWSDL** [6], which stands for *Semantic Annotations for WSDL and XML Schema*, based on the older W3C submission WSDL-S.
Briefly, SAWSDL provides means to semantically annotate certain elements of a WSDL.

Before delving into SAWSDL, let's take a look at other options of semantically describing a web service.
In section 2.1 we've mentioned the evolution of languages for describing/developing ontologies. We've seen that OWL-S is the latest ontology language and it can be used in web service description as follows:

1. choose a converter from WSDL to OWL-S, for example the open source WSDL2OWL-S [9]; OWL-S consists of four OWL files (service, profile, process, grounding), which should be generated by the converter; the grounding file is the link to the WSDL
2. choose a domain ontology; for example you can create a medical ontology using an editor such as Protégé [10] or simply reuse an existing ontology
3. in the OWL files you link the input/output parameters of the web service to entities from the ontology
4. validate the OWL files using a semantic validation tool

SAWSDL offers a much simplified semantic description of web services by allowing semantic annotations embedded in the WSDL. The annotated WSDL can still be registered in UDDI, just as any other standard WSDL.
There are two semantic annotation constructs used by SAWSDL:
1. *modelReference*, which specifies a mapping between a WSDL element or an XML Schema component and a concept in a certain semantic model; according to the W3C recommendation [6], "a model reference may be used with every element within WSDL and XML schema. However, SAWSDL defines its meaning only for wsdl:interface, wsdl:operation, wsdl:fault, xs:element, xs:complexType, xs:simpleType and xs:attribute"
2. *liftingSchemaMapping* and *loweringSchemaMapping*, which annotate XML Schema element declarations and type definitions to specify mappings between XML and semantic data

For example, let's take the scenario in section 1.1 and define the emergency service operation of giving advice based on patient's symptoms, medical record and location. This is a simplified excerpt from the WSDL description of such an operation:

```
<wsdl:binding name="EmergencyServiceSoapBinding"
type="es:EmergencyServicePortType">

    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetAdvice">
    <soap:operation
        soapAction="http://www.emergencyservice.org/GetPatientAdvice"/>
      <wsdl:input name="Patient">
        <soap:body use="literal"
 namespace="http://schemas.emergencyservice.org/GetPatientAdvice.xsd"/>
      </wsdl:input>
      <wsdl:output name="MedicalAdvice">
        <soap:body use="literal"
namespace="http://schemas.emergencyservice.org /GetPatientAdvice.xsd"/>
      </wsdl:output>
      <wsdl:fault>
        <soap:body use="literal"
namespace="http://schemas.emergencyservice.org /GetPatientAdvice.xsd"/>
      </wsdl:fault>
    </wsdl:operation>
 </wsdl:binding>
```

This excerpt defines the operation, with its input, output and fault messages. The actual types of the inputs, outputs and fault are defined in the XML Schema document GetPatientAdvice.xsd (but could also be embedded in the WSDL file):

```
    <xsd:schema
targetNamespace="http://namespaces.emergencyservice.org"
      xmlns:xsd="http://www.w3.org/1999/XMLSchema">

      <xsd:element name="Patient">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="string"/>
             <xsd:element name="addr" type="string"/>
<!– many other fields, corresponding to HL7-RIM ontology →
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="Location">
        <xsd:complexType>
```

```
          <xsd:all>
            <xsd:element name="latitude" type="string"/>
            <xsd:element name="longitude" type="string"/>
          </xsd:all>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="MedicalAdvice">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="hospital" type="string"/>
            <xsd:element name="location" type="#Location"/>
<!-- many other fields -->
          </xsd:all>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetPatientAdviceFault">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="errorMessage" type="string"/>
          </xsd:all>
        </xsd:complexType>
      </xsd:element>

    </xsd:schema>
```

This is the not annotated WSDL that describes the service. In order to give semantic meaning to this WSDL description and to the search process, one would add SAWSDL annotations, e.g.:

```
     <xsd:element name="Patient">
        <xsd:complexType sawsdl:modelReference="http://www.hl7.org/spec
/ontology/rim#Patient">
          <xsd:sequence>
            <xsd:element name="name" type="string"/>
             <xsd:element name="addr" type="string"/>
<!– many other fields, corresponding to HL7-RIM ontology →
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

The sawsdl:modelReference in this example says that the Patient type our service is using maps in fact to the Patient concept from the HL7-RIM ontology (note that the URL is fictional, it's meant only to illustrates the idea).

The SAWSDL annotations as described by the standard are the first step in the direction of semantic discovery. However, the discovery process remains relatively imprecise.

For example, the semantic annotation referred to by "modelReference" in the next example doesn't specify the nature of the semantic information, i.e. we don't know if "validPatientInfo" defines a precondition, a result or an effect.

```
<operation name="getPatientRecord" modelReference= "validPatientInfo
patientRecord">
```

The more semantic information we introduce in the matching process, the more accurate the matching is. There is ongoing research in the area of enhancing SAWSDL with more semantic information [20]. Typically, the extra information refers to IOPE = inputs, outputs, preconditions and effects.
According to YASA4WSDL [20], the example above could be rewritten to include "serviceConcept":

```
<operation name="getPatientRecord" serviceConcept=
"ServiceOntology;#precondition ServiceOntology;#effect" modelReference=
"validPatientInfo patientRecord">
```

In this example, by adding the *serviceConcept* we simply say that the first semantic data, *validPatientInfo*, is a precondition and the second one, *patientRecord*, is an effect of the operation. This dramatically improves the semantic search process and other potential operations, like composition of services.

RESTful Annotations

RESTful [51] services are hype right now, for good reasons. In a world heading towards smartphones and other mobile devices, having a supple, effective communication among them is essential. RESTful web services don't incur the overhead of the SOAP [50] messages (encoding/decoding the SOAP messages, but also the extra payload that comes with SOAP), making the communication between mobile devices and web services more efficient.
This document refers to agents installed on mobile devices communicating with each other and with web services, so it makes perfect sense to consider the RESTful approach here too, instead of (or in parallel with) the classic SOA approach. As it will be shown (see section 3.4), there are several other advantages in dealing with RESTful services.
Just like a SOA-style service is described by WSDL, a RESTful one has its **WADL** [18] counterpart. As we semantically annotate WSDL to obtain SAWSDL, we can also annotate WADL [31]. Such annotations for REST services are the basis for semantic registration and semantic discovery of these services.
Next sections will refer to REST as an alternative or a complement of SOA-based techniques, where applicable.

2.4.2 Semantic Annotations for Agents

The yellow pages service for agents is called Directory Facilitator (DF) and is described by a FIPA specification [14]. Figure 7 shows the DF in the context of the agent platform.

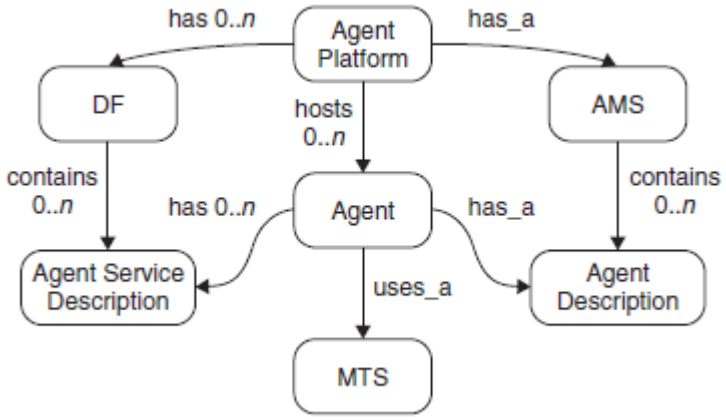


Figure 7: The Directory Facilitator (DF) as part of the Agent Platform (source: reference [1])

Note that even though the system of agents and services is open, the interactions among them are rather strict. In most of the cases the communication among agents takes a rather strict form, known as interaction protocol (IP) [17]. For example, a personal agent would always send the patient's symptoms and location to the emergency service, so the latter knows what to expect. What remains to be fixed is the potential difference in ontologies between the communicating parties.

Consequently, the registration that DF can offer as described by the FIPA standard [14] is sufficient in most of the cases.

In case agents need to look up each other in the DF, we will need an agent description that is general enough to support the type of open system we are dealing with, yet specific enough to be meaningful for the agent that is interested in such a description.

For example, below we show a hospital reception agent's description, registered in the DF. The typical service of such an agent is to receive the patient's symptoms and in turn look up a specialist within the hospital.

```
(df-agent-description
  :name
    (agent-identifier
      :name ReceptionHospitalXYZ@xyz.com
      :addresses (sequence iiop://xyz.com/acc))
  :services (set
    (service-description
      :name reception
```

```
      :type reception
      :ontology (set SNOMED)
      :properties (set
        (property
          :name "hospital reception area"
          :value 224890005)
        (property
          :name "temperature symptoms"
          :value 271399003)
      ....
       )))
  :protocol (set FIPA-Request FIPA-Query)
  :ontology (set SNOMED FIPA-Agent-Management)
  :language (set FIPA-RDF))
```

In this agent description, notice that SNOMED ontology is used. Among the interesting properties of the service offered by the hospital reception agent, we show here the "hospital reception area", which has the value 224890005 [16] in SNOMED. Then, we enumerate all symptom types that can be sent to this agent with their corresponding SNOMED code values. These properties (i.e. "hospital reception area", the symptoms) are in fact the *semantic annotations* we are employing in the case of agents. Note that we don't even need a standard extension (as was SAWSDL in case of web services) of the DF description to add semantic annotations. We'll see in section 2.5.3 that this is not enough in the general scenario. In effect, we'll propose an extension to FIPA standards.

A scenario of a personal agent that looks up a hospital reception agent goes as follows:
1. create the "profile", containing the ontology and the code value of the concept searched for; e.g., an agent would say "I'm interested in the agent that represents the code 224890005 in the SNOMED ontology"
2. the DF agent of the hospital platform looks up this *code - ontology* pair among the registered agents; if not found, it delegates to other DFs (DF is a federated service according to FIPA)
3. the profile should return exactly one match (each hospital has exactly one reception)
4. now the personal agent has the address/name of the hospital reception agent, so it can contact it directly; then, it will send the symptoms of the patient to the hospital reception agent; for example it will send the value 248427009, which is the code for "fever symptoms" in SNOMED; since this is a child of the concept "temperature symptoms", accepted by the hospital reception agent (see the description above), the latter will be able to match and interpret the code sent by the personal agent
5. all symptoms sent by the personal agent are matched and interpreted by the hospital reception agent (the "…." in the agent description above represent the entire list of symptoms according to the SNOMED terminology; some of them will be filled in by the personal agent)

6. after having the patient's symptoms, the hospital reception agent can look up a corresponding specialist in the hospital, based on the symptoms; it will give an answer to the patient's personal agent, containing the coordinates of the specialist's agent

Note that the steps above can be summarized into a custom *interaction protocol* (IP) [17], since it is a procedure that will be always repeated in such a scenario (i.e. personal agent looking up a hospital reception agent, sending the symptoms and receiving the specialist's agent coordinates).
If the ontologies used by the agents don't match, an ontology matching algorithm needs to be involved. This is the subject of sections 2.6 and 6.3.

In Figure 8 we show the steps 4, 5 and 6 in the scenario described above. Namely, the Initiator, in our case the personal agent submits a Request FIPA communicative act to the Participant – in this case the hospital reception agent – asking it for the most appropriate specialist based on the patient's symptoms. The hospital reception agent answers with either *Not Understood* communicative act if the symptoms are not understood even after employing a potential ontology matcher or with *Inform*. In the latter case, the message should contain the contact data for the specialist's agent, such that the personal agent can continue negotiating with this one.

In the example above, suppose the hospital reception was a service instead of an agent. In this case the web method of the reception agent would probably look like this:
`SpecialistInfo alertSpecialist(List<Symptom> symptoms),`
where the return type *SpecialistInfo* and the symptoms list parameter would have to be described in the WSDL file of the web service. Based on the list of symptoms received from the caller, the service will alert a corresponding specialist and return the contact information (the name/address of the specialist's agent) back to the caller.
As explained in section 2.4.1, we'd use SAWSDL in order to semantically annotate the service return type and parameters. Namely, we'd refer to the SNOMED terminology and to the SNOMED codes for all symptoms accepted by the web service. The caller would send its list of symptoms encoded in SNOMED as the parameter of the web method shown above.
Thus, this scenario can be implemented with agents as well as with web services. What's important to notice is that the building blocks are the same: ontologies and ontology matching.

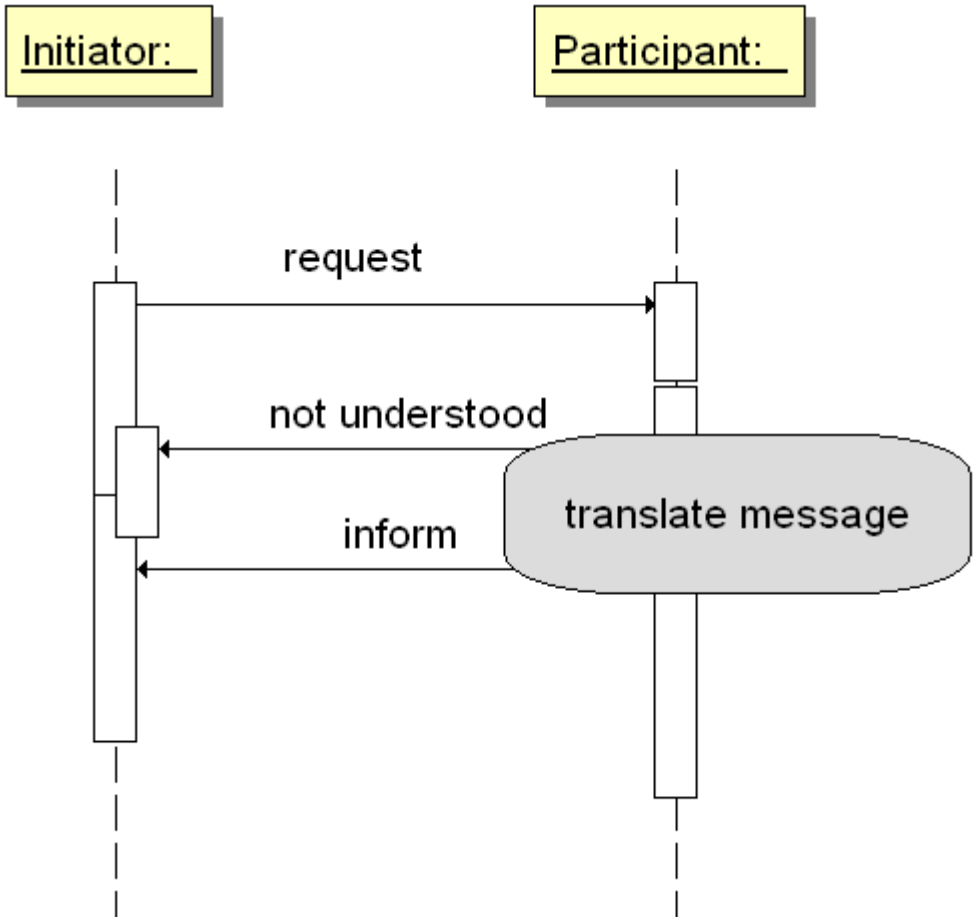


Figure 8: Interaction Protocol (IP) showing the communication personal agent – hospital reception agent; *Initiator* is here the patient's agent and *Participant* is the hospital reception agent

2.5 Semantic Registration and Discovery

2.5.1 Semantic Registration of Services

The UDDI initiative [26] has been the de facto standard for publishing web services. The idea, as shown in Figure 12, is to be able to register a web service, so that it can be discovered by a client. Once it is discovered, the WSDL is retrieved and a proxy for client-service communication could be generated dynamically. This is how the UDDI has been used so far (if at all).
Moreover, mapping SAWSDL to UDDI has been researched ([27], [28]) and it's relatively straightforward:

- either directly annotate UDDI concepts with SAWSDL annotations; for example, the semantic annotations of SAWSDL are translated to UDDI categoryBags [27]; this is a simple approach, i.e. no semantic reasoning is needed, but some of the flexibility is lost (e.g. the search in UDDI is still syntactic)
- or semantically-enabled processing modules are layered on top of UDDI; they perform the actual semantic search – OWL ontology processing and DL reasoning

However, Microsoft, IBM and SAP dropped their support for UDDI in 2007; they were in fact the major supporters of this standard. In effect, it is time for something new, something that addresses the main limitations of UDDI:

1. Overwhelming complexity
2. Ignoring security
3. Lack of good tooling – it took me a lot of time and energy to be able to finally install locally a working UDDI implementation: jUDDI for Tomcat 3.0.1 [33]

In the authors' opinion, the future of semantic web service registration and discovery lays in semantic search engines, e.g. a semantic Google-like service [32]. The essential characteristics of such a semantic search engine should be:
1. simplicity – it should offer a simple, clear to use interface through which either a user or a program can search and find the desired services
2. it should offer a secure way to access a service
3. it should offer APIs to be able to register and discover (semantic) services
4. it should be scalable to millions of simultaneous users and yet answer within a few hundred milliseconds – like the Google search engine behaves nowadays

The principles shouldn't be different from the existing, non-semantic, search engines. The new semantic search engine shall be able to index semantic services. This is in fact the registration step. What will such an engine index after all ? In case of web services, most likely the SAWSDL, i.e. the semantically annotated WSDL. Or in the case of RESTful web services, the semantically annotated WADL.
The semantic discovery will be based on searching the given profile within the repository of indexed services and providing the most appropriate ones in decreasing order of matching value. Of course, this means defining a measure for the matching value. Next section reveals some of the current research where such metrics are defined.
In chapter 5 we cover the non-functional requirements of such a new repository for services.

2.5.2 Semantic Discovery of Services

The discovery of traditional web services happens by textual search in UDDI, as shown in Figure 6. For semantic web services, we need semantic rather than textual matching.
Most research in this area proposes semantically-enabled processing modules layered on top of UDDI. These processing modules perform *ontology processing* (using OWL-S [19] or DAML-S [29]) and *logic reasoning*. The idea is to describe the service ontology using OWL or DAML and add this description in UDDI.
The main issue with these approaches is that UDDI is no longer supported by the major companies, as stated before.
Other research [30] focuses on behavioral matchmaking of web services, defining the conversation protocol of a service as a graph and then employing a graph matching algorithm. This offers indeed very accurate semantic matching (they also define a measure for matching in order to quantify web services), but the complexity of the algorithm is exponential with the conversation steps.

In the authors' opinion, the semantic discovery of web services should comprise:
- an ontology matching step
- a semantic look up step

The first step is necessary when the advertised service uses a different ontology than the service profile we're looking for. If there is a translation among the ontologies, we can find a good matching service, even if it uses a different ontology than our service profile.
The second step implies the actual semantic matching of service profiles. For this we could use current research (see above) as a source of inspiration. For example, we could describe the IOPE (inputs, outputs, preconditions and effects) of services using an ontology language such as OWL. The WSDL of the advertised services will be annotated with such semantic information, as described in section 2.4.1. A profile describing a

service we are looking for will be defined in terms of IOPE, again using ontologies. Then the semantic search engine described in the previous section will perform the semantic matching between the service profile and the registered/indexed services. As stated before, the semantic matching itself can benefit from the current research [30].

2.5.3 Semantic Registration of Agents

Semantic discovery is of course strongly related to semantic annotation: a service is discovered based on the annotations it was given.

The FIPA specification of DF [14] says that an agent that registers with the DF may specify its ontologies. However, unlike the case of web services (see SAWSDL), the description of an agent can't be annotated. Thus, besides mentioning the ontologies used to communicate, the semantics of the services offered by an agent aren't rich enough for most of the use cases. Look for example at the agent description shown in section 2.4.2: the properties don't necessarily say something about what services the agent offers; they are typically used to (hopefully) uniquely identify the agent.

The situation with web services is very different. The WSDL elements map to UDDI (registry) concepts. Moreover, SAWSDL annotations can also be mapped on UDDI items, as explained in the previous section. Using SAWSDL we annotate service's operations, inputs and outputs. Thus, these annotations are not just meant for unique identification of the services, but also for giving meaning to their operations.

We would like to offer the same capabilities to agents, resulting in real semantic search of agents based on their offered services, instead of the current syntactic search, based on properties not related to their services.

For example we've seen in section 2.4.1 that we can annotate the web service description (WSDL) with labels that say something about the life cycle of their operations: pre-conditions, effects, results, etc. We'd like to have the same richness for agents, e.g. allow agents to declare a service by specifying its expected *inputs, outputs, pre-conditions and effects (IOPE)*. Similarly, we need to be able to add these annotations to DF so that other agents looking up functionality offered by our agent can have a better match.

FIPA standard [14] allows the following parameters to be specified to an agent registered with the DF:

Parameter	Description	Presence	Type	Reserved Values
`Name`	The name of the service.	Optional	`String`	
`Type`	The type of the service.	Optional	`String`	`fipa-df` `fipa-ams`
`Protocol`	A list of interaction protocols supported by the service.	Optional	Set of `String`	
`Ontology`	A list of ontologies supported by the service.	Optional	Set of `String`	`FIPA-Agent-Management`
`Language`	A list of content languages supported by the service.	Optional	Set of `String`	
`Ownership`	The owner of the service	Optional	`String`	
`properties`	A list of properties that discriminate the service.	Optional	Set of `property`	

Table 2: The DF parameters used for registration

As you can see, apart from the standard properties such as Name, Type, etc., the "free form" properties that can be specified are the list of key-value pairs that discriminate the service (see section 2.4.2 for an example of how to use these properties). Matching of services is thus reduced to syntactically matching the property values. These values are what FIPA standard [25] calls Terms:

```
Term                    = Variable
                        | FunctionalTerm
                        | ActionExpression
                        | Constant
                        | Sequence
                        | Set.

ActionExpression        = "(" "action" Agent TermOrIE ")"
                        | "(" "|" ActionExpression ActionExpression ")"
                        | "(" ";" ActionExpression ActionExpression ")".

FunctionalTerm          = "(" FunctionSymbol TermOrIE* ")"
                        | "(" FunctionSymbol Parameter* ")".

TermOrIE                = Term
                        | IdentifyingExpression.

Parameter               = ParameterName ParameterValue.

ParameterValue          = TermOrIE.
```

As can be seen, matching two terms means syntactically matching variables, constants or – the most complex cases – matching functional terms. The latter matching means evaluating the functional term in the receiving agent's context.
Nevertheless, the matching process in DF as standardized by FIPA is essentially syntactic.

What we really need is semantic matching, as is the case with semantic web services. Our proposal is to extend the FIPA standard such that it allows that some new parameters be specified for an agent registered with DF. These new parameters should be optional and should include at least the aforementioned IOPE (inputs, outputs, preconditions, effects).
Using this idea, the example in section 2.4.2 could be rewritten as:

```
(service-description
      :name reception
      :type reception
      :ontology (set SNOMED)
      :properties (set
        (property
          :name "hospital reception area"
          :value 224890005)
        (property
          :name "temperature symptoms"
          :value 271399003)
      ....
      )
      :inputs (set
        (input
          :name "symptoms"
          :value "http://www.ihtsdo.org/snomed/ontology#Symptom"
        )
      )
      :otputs (set
        (output
          :name "specialist"
          :value "http://www.fipa.org/agents/ontology#Specialist"
        )
      )
)
```

We've added here the *inputs* and *outputs* – two of the proposed extensions to the FIPA Service Description standard. The example in section 2.4.2 was used to perform the following query: "I'm interested in the agent that represents the code 224890005 in the SNOMED ontology" (such an agent would be the hospital reception agent). With this extension, a more interesting query can be made: "I'm interested in an agent which accepts Symptom inputs as described by the SNOMED ontology and returns a specialist

agent's address, as described by FIPA". As you can see, the latter query is more abstract than the former, since it's not constrained to a specific SNOMED code. We could even make a more abstract query, by not specifying the target ontology: "I'm interested in an agent which accepts Symptom inputs as described by my medical ontology and returns a specialist agent's contact data". In order to properly find the service described above with such a query, an ontology mapping must exists between "my medical ontology" and SNOMED (see sections 2.6 and 6.3 for more details). Also there must exist an ontology mapping between the specialist agent's contact data (as specified in my query) and FIPA Specialist ontology term.

Note that DF itself needs no extension to support the proposed service description annotated with IOPE. The IOPE have to be simply recorded in the knowledge base (KB) used by the DF. Typically this means keeping key-value pairs like *agent ID – IOPE* (here IOPE can be expressed as a Java Object or simply a string like this:

```
(output
      :name "specialist"
      :value "http://www.fipa.org/agents/ontology#Specialist"
)
)
```

However, the discovery process needs to be extended with semantic matching of IOPE, as described in the next section.

2.5.4 Semantic Discovery of Agents

In the previous section we proposed an extension to the FIPA standard such that registration of agents becomes similar to registration of web services. This is based on IOPE parameters, meant to tell more about the service behavior.
Following this idea, discovery of agents becomes similar to discovery of web services, just like registration is similar. Namely, the ideas presented in section 2.5.2 could be employed for semantic discovery of agents. Thus, the discovery of agents could employ:
1. an ontology matching step; when the ontologies used by the advertised agent and the profile we're looking for are different, an ontology service would need to translate one ontology to the other (if possible); *note that since such an ontology service is technology independent, it can be used both by web services and software agents*
2. a semantic lookup step; this can benefit from current research, as described in section 2.5.2; for example, a behavioral matchmaking can be employed or one just based on IOPE

As mentioned above, semantic discovery of agents whose FIPA service description has been annotated with IOPE means in fact a semantic match between the IOPE annotations of the advertised (registered) agents with the required ones.
In order to do this, one of the following approaches can be taken:
- either the KB (knowledge base) used by the DF needs to be extended with semantic matching capabilities for IOPE; this might include ontology matching when the ontologies of the advertised and required agent descriptions are different
- or another layer of semantic matching needs to be added on top of the DF; thus, the DF would remain unchanged, but the semantic matching of IOPE would happen outside the DF

In chapter 5 we propose a semantic search engine to be used both for registration and discovery of web services. In fact we could extend this search engine to agents, since the search engine only depends on matching the IOPEs, not on the technology used here. Thus, the search engine can be seen as an extension of the second proposal above, namely the matching of IOPEs (and thus of agents) would happen inside this semantic search engine, outside the DF. Later in this thesis we will present the details of such a semantic search engine.

2.6 Ontology Matching

This is a large topic, far beyond the purpose of this document.

We've seen that ontology is the cornerstone of semantic services and agents. Without ontologies our services/agents wouldn't be capable to semantically discover each other and semantic interoperability among them wouldn't be possible.

Ontology matching is needed to bridge the gap between various ontologies used by our services and agents.

Ideally, each domain (e.g. transportation, healthcare, etc.) would have a standardized ontology and all agents and services that want to communicate in a certain domain would use the domain ontology. This would alleviate the need for ontology matching, but it's far from the current reality. Not only we don't have unique, standardized domain ontologies, but each domain might employ ad-hoc ontologies, making it difficult for agents/services to understand each other. Thus, realistically speaking, for the foreseeable future, ontology matching is going to be needed.

In [41] the authors describe several techniques of ontology matching and present several existing systems that implement these techniques.

Thus, we can have:

1. element-level techniques:
    a. string-based: they typically look at the common prefixes and/or suffixes between the concept strings or at the number of editing operations (e.g., insertions, deletions, substitutions) needed to transform a concept into the other
    b. language-based: they typically analyze the morphology of the concepts, e.g. by recognizing punctuation, plural vs. singular, etc.
2. structure-level techniques:
    a. taxonomy-based: they view ontologies as graphs containing concepts and their interrelationships; then they compare paths with links defined by hierarchical relations, compare the concepts along these paths and their relative positions to identify similar concepts
    b. tree-based: they apply rules relative to concepts as viewed in graphs, e.g. two non-leaf concepts are similar if their immediate children sets are similar
    c. model-based: they typically involve description logic (DL) to infer the similarity between concepts

One of the most popular matching systems is Cupid [43], developed by Microsoft together with the University of Washington and University of Leipzig. Cupid performs element-level matching followed by structure-level matching, using powerful thesauri for the language-based part of matching.

2.7 The Role of Logic

An ontology language such as OWL defines concepts that can be used by a processor which involves logic to infer new facts about these concepts. Such inferences are essential in discovery of services (see section 2.5.2) and in ontology matching (see section 2.6).

There is another very important usage of logic: in the way autonomous agents decide to take certain actions. In the scenario presented in section 1.1, the patient's personal agent decides to contact the emergency service because it notices deterioration in the patient's state of health. More concretely, the agent queries periodically the sensors John is wearing; the agent uses a simple inference engine and a set of built in rules to determine if it should inform the emergency service or not.
We have implemented a prototype (section 2.3) that uses the JADE framework [1] for developing software agents and the inference support in Jena [35].
In its knowledge base, the patient's software agent has a simple ontological description of the symptoms and of other entities that are part of the medical domain and needed here. The file *patient.n3* shown below is a simplification of this description, used here for illustration purposes.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
@prefix : <http://www.example.com/#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix owl:  <http://www.w3.org/2002/07/owl#>

:Symptom a     owl:Class .
:BloodPressure a        owl:Class;
   rdfs:subClassOf :Symptom .
:Pulse a          owl:Class;
   rdfs:subClassOf :Symptom .
:Service112 a   owl:Class .
:val a   owl:ObjectProperty .
:alert a owl:ObjectProperty .
```

Table 3: The simplified ontology – patient.n3 file excerpt

Basically this file defines BloodPressure and Pulse as a Symptom, while *val* and *alert* are properties of the symptoms and of the emergency service, respectively. Also, a list of RDF-based (w3.org) rules used by the Jena inference engine is part of the agent's knowledge base. An excerpt from the file *alert.rules* is shown below.

```
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix :     <http://www.example.com/#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
[bradycardia: (:Pulse :val ?x), lessThan(?x, 60) -> (:Bradycardia :val "true")]
[tachycardia: (:Pulse :val ?x), greaterThan(?x, 100) -> (:Tachycardia :val "true")]
[call112: (:BloodPressure :val ?x), greaterThan(?x 140), (:Tachycardia :val "true") ->
(:Service112 :alert "true")]
[call112_2: (:BloodPressure :val ?x), lessThan(?x 90), (:Bradycardia :val "true") ->
(:Service112 :alert "true")]
```

Table 4: The RDF rules – alert.rules file excerpt

The first rule says *"if the pulse is less than 60, then the patient has bradycardia"* and the last rule *"if the blood pressure is less than 90 and the patient has bradycardia, then you have to alert the emergency service 112"*.

The agent permanently monitors the patient's condition, by receiving the parameters from the sensors connected to the patient's body. Each parameter is translated by the agent into a statement, e.g.:

:Pulse :val 110

:BloodPressure :val 145

Having these two statements and the rules above, plus the ontological description of the symptoms in Table 2, the agent infers the statement *Service112 :alert "true"*. This is how the agent autonomously determines that it should alert the emergency service when the patient's condition deteriorates.

This example also highlights some issues related to agents' communication. First, note that we use the N3 notation [64] to describe the ontology. N3, also known as Notation3 is a non-XML serialization of RDF models. It has the advantage that it is human-readable, so any human operator can easily update it. This makes it easy to install and configure the agents on users' devices. Also, the N3 format can be used to describe the ontologies that are referenced by the web service descriptions in their semantic annotations. It offers support for RDF-based rules, which makes it suitable for logic inferences. More details about how to use N3 rules and Jena are shown in the Appendix.

In the example above (Table 3), we use a simple ontology for the sake of simplicity. In a real life scenario, a good candidate for a medical ontology is one based on HL7-RIM model ([2], [3]). The great advantage is the worldwide adoption of this standard and thus the possibility to interoperate with medical institutions all over the world.

The example in this section highlights the capabilities the agents can leverage with minimum knowledge and little computing power. These very characteristics, together

with the autonomy, make software agents suitable for the scenario described at the beginning of this section. Namely, software agents are typically used in embedded/mobile systems, having thus limited resources and intermittent Internet connection with little bandwidth.

The actions inferred by the agents can be much more complex than in this example. For instance, an agent may be interested in finding the diagnoses of all patients with similar symptoms as John. For that, an agent needs a knowledge base and an inference engine.

CHAPTER 3

COMMUNICATION

3.1 Service-to-Service Communication

Wrapping RDF with SOAP messages is described in [23]. The idea is to communicate a payload that can provide ontological reasoning, such that we get semantic interoperability. The underlying ontology in our case is based on HL7.
This nicely complements the service's annotated WSDL (with SAWSDL, see section 2.4.1) used to semantically discover the service. Now it can be semantically consumed too.

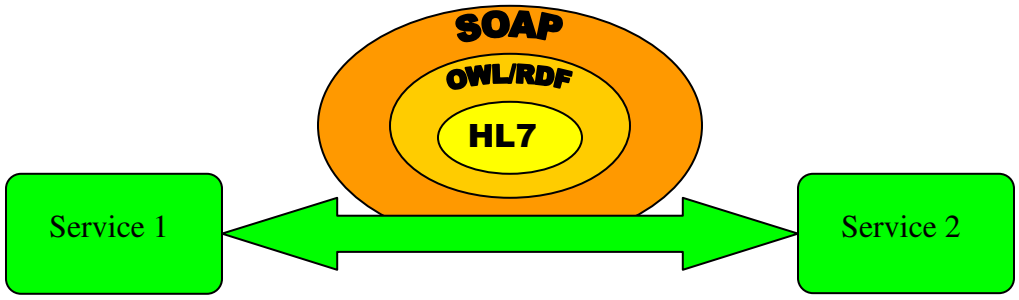


Figure 9: Service-to-service communication: an HL7 message encoded in RDF, wrapped in a SOAP message

Below we give an example of a parameter to a service, encoded as RDF embedded in SOAP:

```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
 <env:Body>
<rdf:RDF env:encodingStyle="http://rdfinference.org/rdfws/soap-
encoding"
         xmlns:rim="http://hl7.org/rim/"
        xmlns:patient="http://hl7.org/rim/patient#"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
  <rdf:Description rdf:about=" http://hl7.org/rim/patient/John Dalton">
    <patient:ssn>0123456789</patient:ssn>
    <patient:address>5 Insomnia Lane, Port Louis, CA</patient:address>
    <patient:age>71</patient:age>
    ….
```

```
  </rdf:Description>
</rdf:RDF>
</env:Body>
</env:Envelope>
```

In other words, the parameter that is passed to the service is just a string, but it carries semantic information, since it represents the concept Patient defined in a certain ontology. The service parses and interprets this concept, applies its algorithm and returns the result also as a string encoding a semantic description of a concept.
We could also pass the concept of "patient" as a type Patient defined in a certain language, as we'd typically do with any regular web service, and then bind the parameter to a SOAP/XML message as usual. This means though loosing the ontology (i.e., semantic) information, but it is still applicable if the called service annotates its WSDL description with semantic information. Then we'd know that the concept Patient needs to belong to a certain ontology. In that case, we'd have a two-step process:
1. look up a service with a parameter that is the concept Patient expressed in a certain ontology
2. after the service is found, invoke it by passing it the type Patient in any language that has a SOAP/XML binding and can thus be transported to the called service and deserialized to the Patient type expected by the service

3.2 Agent-to-Agent Communication

Similar to how services can communicate using OWL/RDF payloads for SOAP messages, agents can communicate to each other using OWL/RDF payloads with FIPA ACL [62] messages. The official document [12] introduces the standard for RDF contents in FIPA and ongoing research [24] proposes OWL (whose syntax is in fact RDF) as FIPA content language.

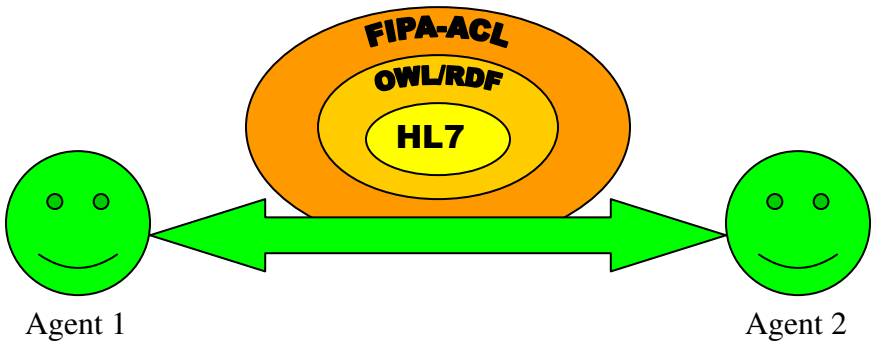


Figure 10: Agent-to-agent communication: an HL7 message encoded in OWL/RDF, wrapped in a FIPA-ACL message

Each layer of the message has a precisely defined purpose. HL7 is used for semantic interoperability – the parties use the same terminology when communicating in the medical domain. Note that HL7-RIM properties can refer to concepts coming from other ontologies – for example SNOMED [16], LOINC [15] or ICD-10 [55]. The HL7 layer however is self-contained, meaning that an HL7 v3 message can be sent from source to destination and interpreted as-is, i.e. without the need of OWL, FIPA-ACL or SOAP.
OWL/RDF is used to transform the HL7 concepts into a real ontology, with expressivity that can't be achieved by just using HL7 alone. For example, one can express conditions, such as if a certain symptom has a certain value, then other symptom might be also present. One can argue that HL7-RIM already describes an ontology. Indeed, there are entities described by HL7-RIM and the relations between them – but they are expressed as UML diagrams. We need to translate these diagrams into a "real" language meant for describing ontologies, and that would be OWL.
A message described as OWL/RDF has to be communicated either to a web service or to an agent – the two entities we are dealing with. In consequence, the message has to be wrapped either as a SOAP message or as a FIPA-ACL one.

Challenges
Our prototype (section 2.3) designed to demonstrate the concepts presented in this document uses the JADE platform. As mentioned in section 2.2, HL7 ontologies have been produced for the JADE platform [1]. This enables agents written using this platform to exchange messages containing HL7 data. The first JADE versions however would only accept communicating binary data, thus both agents involved in a conversation needed to know the exact ontology classes, making the agents very inflexible, as the following example shows:

```
ContentElement ce = manager.extractContent(message);
if (ce instanceof Predicate) {
    if (ce instanceof Abnormal) {
        // do task
    }
    // etc.
}
```

Here "Abnormal" is a user-defined ontological class which implements JADE-defined interface jade.content.Predicate. Both communication parties (agents) need this class in their runtimes/virtual machines.
Luckily, JADE version 3 introduced the XMLCodec add-on with which messages can be XML-encoded/decoded. Thus, the HL7 ontological classes can be converted to XML and parsers can be used to interpret the message contents. This is similar to using SOAP for

web services (as opposed to a binary protocol) and in fact there is research [21] aiming at integrating JADE with SOAP.
However, just parsing any XML is not good enough when dealing with ontologies. We need a more powerful way of expressing relationships among ontological concepts than just simple XML. OWL comes to the rescue. AgentOWL [22] is a research effort aimed at agent communication based on OWL messages. It allows describing ontologies in OWL and performing SPARQL [63] queries on them, as well as sending ACL messages encoded in OWL and SPARQL between agents.

3.3 Agent-to-Service Communication

A more interesting type of communication takes place between an agent and a service. In this case, the FIPA-ACL layer needs to be removed and replaced by a SOAP layer in each message sent from the agent to the service and opposite in the reverse case (see Figure 11). Who does this message conversion ?

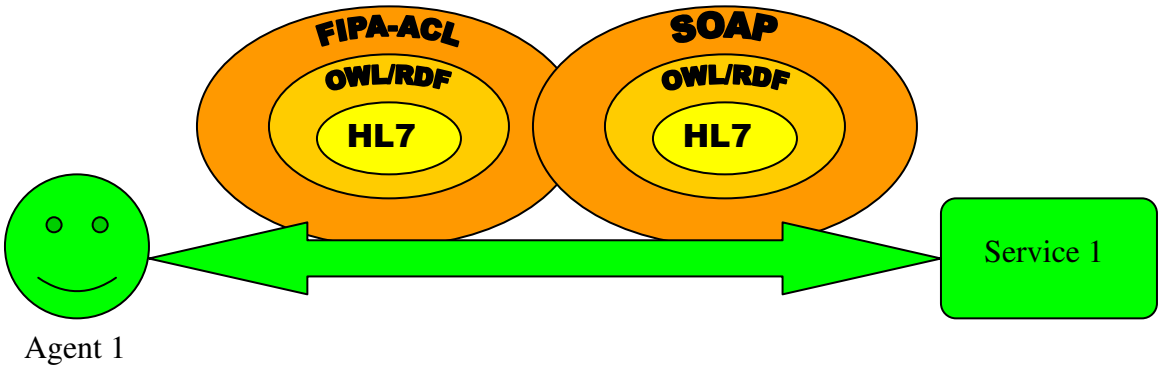


Figure 11: Agent-to-service communication: the FIPA-ACL layer is replaced by a SOAP layer and the other way around

First let's observe that in a typical scenario it doesn't make sense for a service to invoke an agent. Agents are perceived as dynamic, changing and migrating entities, while services are more stable from the point of view of the interface (contract) to be invoked and from the point of view of the location – they don't migrate from host to host like mobile agents. So it makes more sense for the agent to invoke a service rather than the other way around. However, as explained in future sections, we are dealing with services and agents indexed by a semantic search engine. So a service will be looking for another service using this search engine and the result might be a web service or an agent. Why limit ourselves to web services, when a similar functionality can be offered by an agent ?
A web service is looking for a specific functionality. Who implements that – another web service or an agent – is of no relevance to the client and it should be transparent. The client has to be able to find the functionality and invoke it.

To answer the question "who does the message conversion between the agent and the service ?", here are some considerations:

1. we'd like to keep the agent unaware of the communication party; it should be transparent for the agent that it talks to another agent or to a service
2. the same is true for the service: the service shouldn't have the knowledge of the communication party
3. the service is registered in a UDDI-like repository
4. the agent is registered in a DF repository

Considering these forces, the best would be to have a special entity registered together with the service that knows how to translate from the agent to the service world and back. Whenever the agent looks up a service in the repository, it finds this entity (let's call it service wrapper) and it uses it as if it were the service itself – see Figure 12 for details.
Of course, the agent is unaware that it talks to the wrapper instead of the service. It simply sends its FIPA-ACL message as it would when communicating with another agent. The wrapper's task is to convert this message to a SOAP message and invoke the web service. When the result from the service is ready, it wraps it as a FIPA-ACL message and it sends it back to the agent.
Certainly, such a conversion between messages will only decrease the overall performance of the communication. This is the price to be paid for semantic interoperability between agents and services.

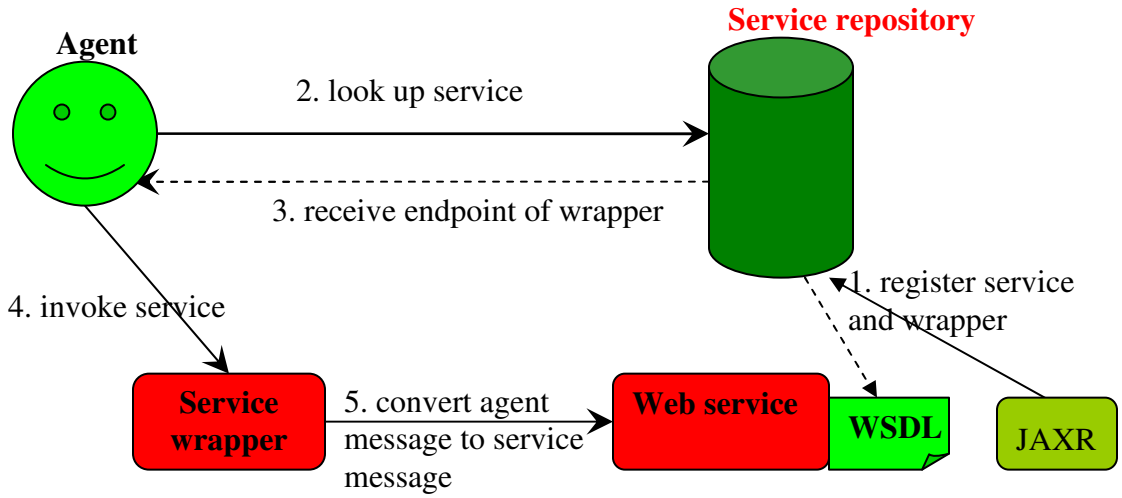


Figure 12: Agent invoking a service through a service wrapper

Note: the web service might be also a RESTful one. In such a case, no SOAP layer is needed, which enhances the overall communication performance.

In fact, we don't need to have such a purist approach when an agent invokes a service. Since a proxy to the service is generated from WSDL in a standard scenario, why not invoking the service directly through that proxy ? An agent is a piece of code that, besides sending FIPA-ACL messages to other agents, it can call other pieces of software, such as a web service, as shown in Figure 13. The client calls the local proxy and this one worries about SOAP and the remote service to call. This is what happens in practice during web service invocation, so no surprises here.

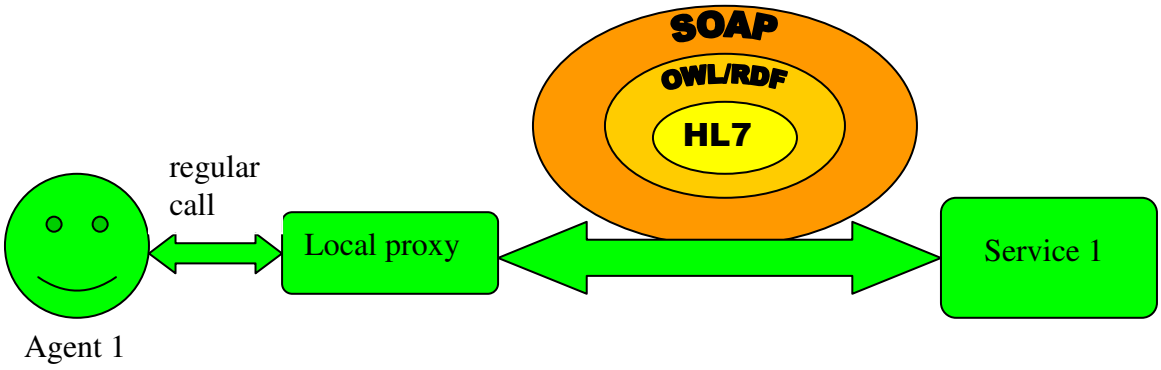


Figure 13: Simple invocation of a service

A more interesting scenario is when a service invokes an agent. Let's delve deeper into this. First of all, in section 2.5.1 we proposed a semantic search engine as a registry for web services (instead of UDDI), as well as for the semantic discovery of web services. We'll discuss this proposal in more details in the second half of this document.
Why not use the same search engine to register agents ? Why would we limit the search process only to services ? If an agent offers the operation desired by a client (i.e. it semantically matches the profile looked up by the client), then it would be a pity not to use it just because the search engine doesn't index agents. The proposal is to update the WSIG mechanism [1] offered by JADE, which allows agents to be exposed as web services. This is suggested in Figure 14. When an agent registers to the Directory Facilitator (DF), the WSIG Agent, which subscribes to such events, creates a WSDL from the agent's description and publishes it to the search engine repository. Note that the search engine typically indexes web services (thus their WSDLs) found by crawlers. Here the WSIG Agent actively adds the WSDL to the search engine. With the proposal to extend the agent descriptions with IOPE (see section 2.5.3), an agent with the following description:

```
:inputs (set
  (input
    :name "symptoms"
    :value "http://www.ihtsdo.org/snomed/ontology#Symptom"
```

```
        )
      )
      :outputs (set
        (output
          :name "specialist"
          :value "http://www.fipa.org/agents/ontology#Specialist"
        )
      )
```

…. is transformed to a WSDL with the following annotations:

```
<wsdl:input>
  <xsd:element name="symptoms">
      <xsd:complexType
    sawsdl:modelReference="http://www.ihtsdo.org/snomed/ontology#Symptom">
        <xsd:sequence>
          ….
        </xsd:sequence>
      </xsd:complexType>
  </xsd:element>
</wsdl:input>
<wsdl:output>
  <xsd:element name="specialist">
      <xsd:complexType
    sawsdl:modelReference="http://www.fipa.org/agents/ontology#Specialist">
        <xsd:sequence>
          ….
        </xsd:sequence>
      </xsd:complexType>
  </xsd:element>
</wsdl:output>
```

Once the registration of the annotated WSDL is done, Service 1 in the figure below can look up the desired service with our semantic search engine, unaware that the description found comes from an agent. Then Service 1 invokes the agent by calling the WSIG Servlet from the WSIG framework via the local proxy.

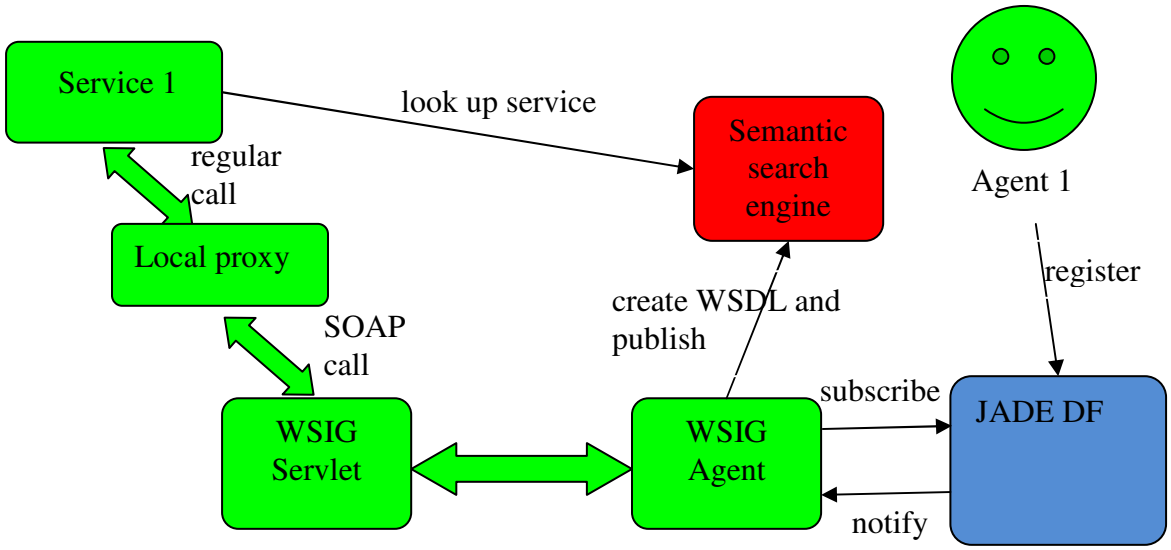


Figure 14: Invocation of an agent from a service

Ontologies and meta-ontologies

Unlike services, agents standardized by FIPA include the notion of ontology in the message, as an optional part. According to the FIPA architecture, a message may refer to zero or more ontologies. In the case of an open system, with agents communicating with each other, the ontologies need to be specified in the message. The contents of a message can only be interpreted using the ontologies specified in the message.

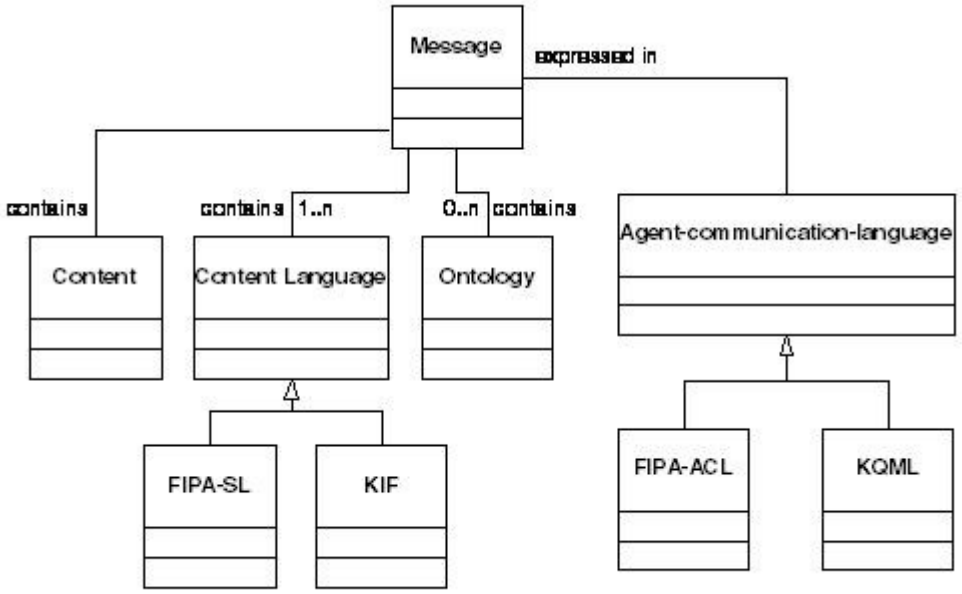


Figure 15: The structure of a FIPA message

Suppose two agents want to communicate, but they use different ontologies. How do they understand each other ? According to [13], a FIPA-compliant platform may include *ontology agents*. Their role is, among others, to translate terms from an ontology to another. So an agent B, which has received a message from agent A, can ask the ontology agent to translate the received message to its own ontology. But how does an agent communicate about terms in an ontology ? For example, the agent B might ask the ontology agent "please give me all subclasses of the class BloodPressure; translate them from ontology A to ontology B". In order for an agent to be able to talk about ontologies, a meta-ontology is defined [13], `FIPA-meta-ontology`. Another ontology, `FIPA-ontol-service-ontology`, must be used when requesting services of an ontology agent. Example of using the `FIPA-Ontol-Service-Ontology`:

```
(query-ref
 :sender
   (agent-identifier
     :name client-agent@example.com
     :addresses (sequence iiop://example.com/acc))
 :receiver (set
   (agent-identifier
     :name ontology-agent@example.com
     :addresses (sequence iiop://example.com/acc)))
 :language FIPA-SL2
 :ontology (set FIPA-Ontol-Service-Ontology
                http://www.hl7.org/spec/ontology/rim)
 :content
     (iota ?x (subclass-of ?x BloodPressure))
 :reply-with symptoms-query)
```

In this example, an agent asks the ontology agent for the subclasses of the BloodPressure class in HL7 RIM ontology.
The following example shows how an agent requests the ontology agent to translate a certain predicate from HL7 RIM ontology to SNOMED.

```
(request
  :sender
    (agent-identifier
      :name client-agent@example.com
      :addresses (sequence iiop://example.com/acc))
  :receiver (set
    (agent-identifier
      :name ontology-agent@example.com
      :addresses (sequence iiop://example.com/acc)))
  :protocol FIPA-Request
  :language FIPA-SL2
  :ontology FIPA-Ontol-Service-Ontology
  :content
    (action
      (agent-identifier
        :name ontology-agent@example.com
```

```
        :addresses (sequence iiop://example.com/acc))
    (translate (BloodPressure 135)
      (translation-description
        :from http://www.hl7.org/spec/ontology/rim
        :to http://www.ihtsdo.org/snomed/ontology)))
  :reply-with translation-query-rim-snomed)
```

Note that the content of a message can be specified using the content language FIPA-RDF. This allows for an easier integration with semantic web services.

In conclusion, our vision of semantic agents refers to:

- FIPA messages specifying one or more ontologies (but at least one); this makes it possible for agents to communicate in an open environment
- The content language to be used should be FIPA-RDF in order to facilitate easier integration with the semantic web

3.4 Rest Services

In the sections above (2.4.1, 2.5.1 and 2.5.2) the services are regarded as classic, SOAP based web services. This is the most common scenario, since legacy services are typically of this kind. To leverage the semantic interoperability, they need to be upgraded according to this paper (their WSDL description annotated according to the SAWSDL standard and their message payloads enhanced to OWL/RDF).

However, the diagrams and discussions in those sections remain valid also for RESTful web services, save for the SOAP layer, which doesn't exist in REST. Why would we consider the REST approach ?
First of all, note that REST clients are appropriate for mobile devices, since we avoid the overhead of SOAP marshalling/unmarshalling and thus the size of the message reduces. Mobile/embedded devices, with low bandwidth and intermittent connectivity are exactly the typical ones to be used in an agent environment – after all this is how our scenario started off in the first section of this document.
Secondly, invoking a web service (even a semantic one) means knowing *the signature of the method you are invoking*; there is no general mechanism of invoking a web service without knowing its interface; moreover, it's difficult and error prone to dynamically generate a proxy which is able to communicate to the service using the right method signature, marshalling/unmarshalling parameters, etc.
To keep this more concrete, think of the case of semantically discovering a web service, understanding its IOPE (inputs, outputs, preconditions, effects) from the registry, but having to go through the *performance hit* of dynamically parsing the WSDL and generating the proxy, just to be able to understand how to invoke the web service. Once you have the inputs/outputs (which you've discovered in the registry), you should be able to invoke a web method, provided you know how it's called. Of course, we could have a convention that all web methods have the same name, say "*process*", but this is artificial and error-prone. Besides, that means you'd only have one method overloaded many times, so having potentially two different methods with the same parameters and return types won't be possible. This limitation is unacceptable.

With REST one could keep it simpler and avoid the performance hit mentioned above, because the verbs are standard (GET, PUT, DELETE, etc.) and thus you don't need to know a method signature; instead, you could form a request like this:

> www.somesite.org/patients/123456 (this would be the patient ID) and it would retrieve the patient's medical record, for example.

A bit more complex request would include the ontology information – SNOMED in this case:

> www.somesite.org/patients/snomed/123456

This example tells that the ontology used is SNOMED and thus the code “123456” has a meaning in that ontology.

Also, because each traditional web service (RPC-style) has its own parameters and expected results, protection (e.g. firewalls) and routing software need more intimate knowledge of the service than in the case of RESTful commands, whose semantics is clearly defined.
Considering the advantages of RESTful web services, we’d like to add them to the mix: not just traditional web services descriptions should be annotated (SAWSDL), but also RESTful web services. As we’ll see in Part 2, these services can be also indexed by a search engine and thus made available to the world.
Paper [31] proposed an extension to WADL which is very similar to SAWSDL. An example is shown below (source: paper [31]):

```
<resources base="">
<resource path="shelftalkers">
<method name="POST" id="consumer">
<request>
<param name='friendfbId' maxOccurs="1" type='xsd:string'
sawadl:modelReference="Ontology1#Person"/>
<param name='apiKey' type='xsd:string' style="query"
sawadl:modelReference="Ontology1#BestBuyAPIKey"/>
<param name='access_token' type='xsd:string' style="query"
sawadl:modelReference="Ontology1#FacebookAPIKey"/>
<param name='api_key' type='xsd:string' style="query"
sawadl:modelReference="Ontology1#LastFMAPIKey"/>
<param name='i' type='xsd:string' style="query"
sawadl:modelReference="Ontology1#LyricsFlyAPIKey"/>
</request>
<sawadl:precondition expression="selectedFBFriend"/>
<sawadl:effect expression="retrievedLyricsforSoundTrack
^ retrievedAlbumDetails ^ retrievedASIN"/>
<response status="200">
<param name='artist' maxOccurs="1" type='xsd:string'
sawadl:modelReference="Ontology1#SoloMusicArtist"/>
<param name='albumname' maxOccurs="1" type='xsd:string'
sawadl:modelReference="Ontology1#AlbumName"/>
<param name='price' maxOccurs="1" type='xsd:string'
sawadl:modelReference="Ontology1#AlbumPrice"/>
<param name='asin' maxOccurs="1" type='xsd:string'
sawadl:modelReference="Ontology1#ASIN"/>
<param name='trackname' maxOccurs="unbounded"
sawadl:modelReference="Ontology1#TrackName"/>
<param name='lyrics' maxOccurs="unbounded"
sawadl:modelReference="Ontology1#Lyrics"/>
</response></method></resource></resources>
```

Part 2: *Semmed* – A Semantic Search Engine Proposal

CHAPTER 4

CURRENT RESEARCH

The scenario presented in section 1.1 highlighted an architecture based on intercommunicating software agents and web services.

Two of the most complex issues mentioned in the previous sections are related to publishing and discovering semantic web services/agents:

- the capability to semantically look up the semantic web services/agents
- ontology matching between an input profile and an advertised service/agent

We argued in the first part of this paper that most of the research dealing with semantic registration and discovery of web services is focused around UDDI. However, UDDI lost support from the major companies – the ones that created the standard in the first place.
In the authors' opinion, the future of semantic web service registration and discovery lays in *semantic search engines*, e.g. semantic Google-like services [32]. The essential characteristics of such a semantic search engine should be:
1. simplicity – it should offer a simple, clear to use interface through which either a user or a program can search and find the desired services
2. it should offer a secure way to access a web service
3. it should offer APIs to be able to register and discover (semantic) web services
4. it should be scalable to millions of simultaneous users and yet answer within a few hundred milliseconds – like the Google search engine behaves nowadays

Ontology matching has to do with semantic equivalence between concepts. It basically means being able to map two concepts from different ontologies. For example suppose there is a concept of *Patient* defined in *Ontology1* with the following attributes:

- *name: string*
- *address: string*
- *medicalRecord: EPR* (from another ontology, called *Ontology3*)
- etc.

Now let's take another concept, called *Stakeholder*, defined in *Ontology2* and having the following fields:

- *name: string*
- *medicalHistory: EPR* (from another ontology, called *Ontology3*)
- *currentInsurer: Insurer* (from *Ontology4*)
- etc.

An ontology matcher might infer the two concepts are equivalent, based on the common field "name" (having the same type, *string*) and based on the context, i.e. the relationship to EPR from Ontology3.
It's not always possible to deduce a relationship between two concepts from different ontologies. It might even be misleading, e.g. two entities with identical fields might represent totally different concepts. In such a case, a mapping between concepts from ontologies needs to be registered manually in some repository.
Semantic search means looking for registered concepts (the *"advertised profile"*) that can be matched semantically to some given concepts (the *"requested profile"*). Typically we want more than just matching within the same ontology. We are looking for matching across ontologies, i.e. ontology mapping. Restricting ourselves to the same ontology is too drastic and it means rejecting many potential matches.

In practice, the semantic search has two flavors:

- one that is in fact an extension of a traditional search engine; such a service is meant to be used by humans; they would simply use a text box to insert the terms and the semantic search engine would look up the terms based on their meanings; the basis for such an approach is the semantic annotation of web pages; we'll call this scenario *Business-to-Consumers*
- another one that is meant to be used by automated tools which are looking for semantic web services; such a scenario is described in section 1.1, where we present both software agents and other services that look up and invoke semantic web services; in this case a semantic search engine would index the descriptions of semantic web services – SAWSDL (or annotated WADL) documents – and the ontology files; we'll call this scenario *Business-to-Business*

In line with the research presented in the first part of this paper, the latter flavor is the one we are interested in. Nevertheless, it's worth investigating the former too, since it brings valuable feedback for the design of the latter.
Among the first category, Business-to-Consumers, we give below some popular examples.

Open Graph (Facebook)

The Open Graph protocol enables your web page to become a rich object in a *social graph*. You can do this by annotating your web page with standard Facebook metadata. This allows users to connect to each other in ways closer to the semantic web, i.e. based on common interests. A social graph, also known as a *sociogram*, is a graph that describes personal relations.

The initial version of the protocol is based on RDFa, or Resource Description Framework-in-attributes [65], as illustrated by the following example (source *ogp.me*):

```
<html prefix="og: http://ogp.me/ns#">
<head>
<title>The Rock (1996)</title>
<meta property="og:title" content="The Rock" />
<meta property="og:type" content="video.movie" />
<meta property="og:url"
content="http://www.imdb.com/title/tt0117500/" />
<meta property="og:image"
content="http://ia.media-imdb.com/images/rock.jpg" />
...
</head>
...
</html>
```

The implications of the Open Graph protocol are major, since by using it Facebook drives more traffic for social searches than Google.

Ontotext

The company called Ontotext provided what they call a “dynamic semantic publishing” mechanism to BBC in order to create the World Cup 2010 website.
Web pages are dynamically aggregated, based on metadata described with a rich ontological domain model.
For more details:
http://www.bbc.co.uk/blogs/bbcinternet/2010/07/bbc_world_cup_2010_dynamic_sem.html
The company produced KIM, a platform for semantic annotation and multi-paradigm search over documents, data and knowledge in general. In a nutshell, the platform:

- crawls the web or imports content from file storage
- organizes and uploads knowledge through mapping of ontologies, taxonomies, linked data
- extracts facts from content
- creates semantic index linking content and knowledge
- provides multi-paradigm semantic search over data, documents and facts

In order to extract the facts, the company developed their own text analysis tool, based on semantics, not on pure syntax.
The products offered by Ontotext are used in domains as diverse as life sciences, media, publishing, telecom, defense, financial intelligence, etc.

Semantic Matching

In the area of semantic matching for web services, most of the research focuses on matching based on a semantic distance between the concepts. There are many flavors of semantic matching algorithms, i.e. including only the web service *inputs* and *outputs* or taking into account *preconditions* and *effects* as well, logic-based matching, non-logic based, hybrid, etc.

In the first part of this paper we propose the matching of IOPEs (inputs/outputs/ preconditions/effects) for web service discovery, so from now on we focus on algorithms that match IOPEs. Such an algorithm is described in [34]. The algorithm is hybrid and is based on the *semantic distance* for each of I, O, P and E, computed from the ontology tree representation of each concept.

Even though the algorithm doesn't depend on the registration of web services, reference [34] and all similar papers mention UDDI as a means to register the web services.

In [36], the authors argue that UDDI lacks two important features:

1. the supported search mechanism is limited to keyword matching and does not support any inference in order to extract more meaningful information; for example, searching for *Doctor* wouldn't return any *Surgeon*, because the inheritance relationship between these concepts is not understood by UDDI
2. the search is based on a certain UDDI field, e.g. search by Category; for more accurate searches, we should take into account at least the inputs and outputs of web services; this powerful information is not exploited by UDDI

To solve these issues, the authors of [19] propose to create a plugin to UDDI that uses OWL-S in order to add semantic capabilities and inference power to the standard UDDI.

However, as mentioned before, UDDI lost support from the major vendors for several good reasons.

In [36], the authors propose matchmaking between web services based on the so-called *Tversky model*, in which similarity is based on the properties of concepts (expressed in their given ontologies). They argue that this method offers more accurate results than the traditional methods based on semantic distance. Moreover, their method can be used with concepts from different ontologies, thus the ontology matching is also taken into account.

Simply put, the Tversky distance between two concepts X and Y is defined as:

Tversky (X, Y) = NumberOfCommonProperties(X, Y) / NumberOfProperties(X)

The closer to 1 this value is, the better the semantic match. A value of zero means no match whatsoever.

In this paper, using the research already done, we focus on a realistic replacement for UDDI. We envision that as a semantic search engine, which we call *Semmed*. It employs the Tversky model for semantic matching of semantically annotated IOPEs.

CHAPTER 5

SEMANTIC SEARCH ENGINE DESIGN

In this chapter we give an overview of the requirements of a semantic search engine and detail its design.

5.1 Requirements

As mentioned in chapter 4, in this document we focus on the scenario Business-to-Business, where software components (e.g. agents) search semantic web services in order to consume them. These services are deployed on their respective websites, as usual. We consider only the semantically annotated services though, namely those that follow the SAWSDL standard [6].

Functional Requirements

- A semantic search engine needs to discover the semantic web services and index them.
- The semantic search engine should answer queries with a list of matching services, in the decreasing order of matching degree. This implies that a measure for matching has to be defined, so that the results are sorted properly.
- Once the matching services are found, the search engine plays no role in invoking those services. This implies that the client of a web service needs to be provided the URL of the service's WSDL (or the WSDL itself) by the search engine.
- Matching of a requested service to advertised services shall be done even if they refer to concepts from different ontologies. This brings us to the second topic of this paper, *ontology matching*. This new capability – the ontology matching – should be kept orthogonal to the search engine, i.e. they can work independently from each other.
- When the search engine returns an advertised service that matches a requested one, but they have different ontologies, the *ontology matcher* should map the request to the matched service so that the service can be invoked.
- The search engine is not involved in the process of matching ontologies while invoking the service.

In a standard search engine we are looking for both high precision and high recall in matching. However, in the Business-to-Business scenario described in chapter 4, we are rather interested in one good result, hopefully the best matching one among all possible matches.

The same requirements as described in this section apply for semantic RESTful services too, as long as their WADL descriptions [18] are semantically annotated.

Non-functional Requirements

- The semantic search engine proposed in this paper needs to be scalable, fault-tolerant, available, reliable, maintainable, etc. In short, it needs to have all the abilities of a traditional, openly used search engine.
- Also, the speed of reaction (the performance measure of the search engine) is expected to be comparable to the current non-semantic search engines.
- The security is also an important characteristic of our search engine. The searcher itself is implemented as a web service, so we expect all characteristics of a typical web services, including security.

The crawlers also need to be secure, so that they are not asked to crawl corrupt data, which could compromise the state of the repository. The repository itself needs to be scalable, as the amount of data stored increases with each new crawling. It also needs to provide high performance in terms of the time to read data from. The searcher needs to read data from the repository and, as we've seen, since the searcher is the component exposed to the world (i.e. it is the entry point to the search process), it needs to have a high speed of reaction. In the next sections we'll highlight the data structures used to enhance the performance of the repository.

In a world where agents and services seamlessly communicate, searching for a service or an agent needs to be a very efficient process. In a scenario like the one described at the beginning of this paper, where human lives are at stake, waiting indefinitely to find a service/agent is not an option.

Unfortunately, the current UDDI-based registries for web services and the federated DF for agents employ linear complexity operations, just like a sequential search algorithm, thus suboptimal and unacceptable for real world systems involving millions of agents/services that interoperate frequently.
We need other data structures, more efficient for search algorithms. A well balanced binary tree would offer binary complexity of search operations and a B-tree would do even better. However, using different data structures instead of those offered by UDDI-based registries would mean simply not using the latter any more. That's why we propose the semantic search engine also as a UDDI replacement.

In such a complex system, non-functional requirements play a very important role. Besides the search performance briefly mentioned above, many other abilities should be considered:
- reliability: what happens in case of failures of agents and/or services ?
- availability: how long can the services/agents be guaranteed to function ?
- scalability: how scalable (in terms of simultaneous requests) are the services/agents ?

- security: how safe are the data exchanged by the agents ? Don't forget that in our scenario, the agents exchange sensitive patient information, so the security demands are very high.
- juridical aspects: in which measure one can rely on agents/services to replace human judgment ?

It is not the purpose of this document to answer these questions. However, the design of any realistic system based on agents and web services needs to take these aspects into account.

5.2 Detailed Design

Figure 16 shows the building blocks of the proposed semantic search engine, *Semmed*. The three main functions of a search engine are covered by the blocks Crawler, Indexer and Searcher.

- Our crawlers retrieve the annotated WSDL descriptions, they parse them in order to find the references to ontologies and thus they also retrieve the ontology files. Then they add both the SAWSDL annotated descriptions and the ontologies to the repository. Since they already parse the descriptions, the crawlers can create the forward indices, mapping the WSDL IDs to the IOPEs they contain.
- The indexer transforms the forward indices to the reverted indices: one for the Inputs, one for the Outputs, one for the Preconditions and one for the Effects. Each such index maps each I, O, P or E respectively to the document IDs where they are found. The indexer also pre-calculates the Tversky semantic distances between each two concepts in each ontology found in the repository and persists these distances to the repository.
- The searcher matches the required profile – given by the *Inputs, Outputs* and optionally *Preconditions* and *Effects* – to the ones registered in the inverted indices. Whenever there is no match in the index kept in memory, the searcher needs to access the repository and read some disk blocks.

Next sections describe these blocks and the organization of the repository.
Further we describe the data structures used in order to minimize the disk accesses, which become the bottleneck of the search algorithm. Also, we detail the search algorithm, which is based on the concept of semantic distance between concepts.

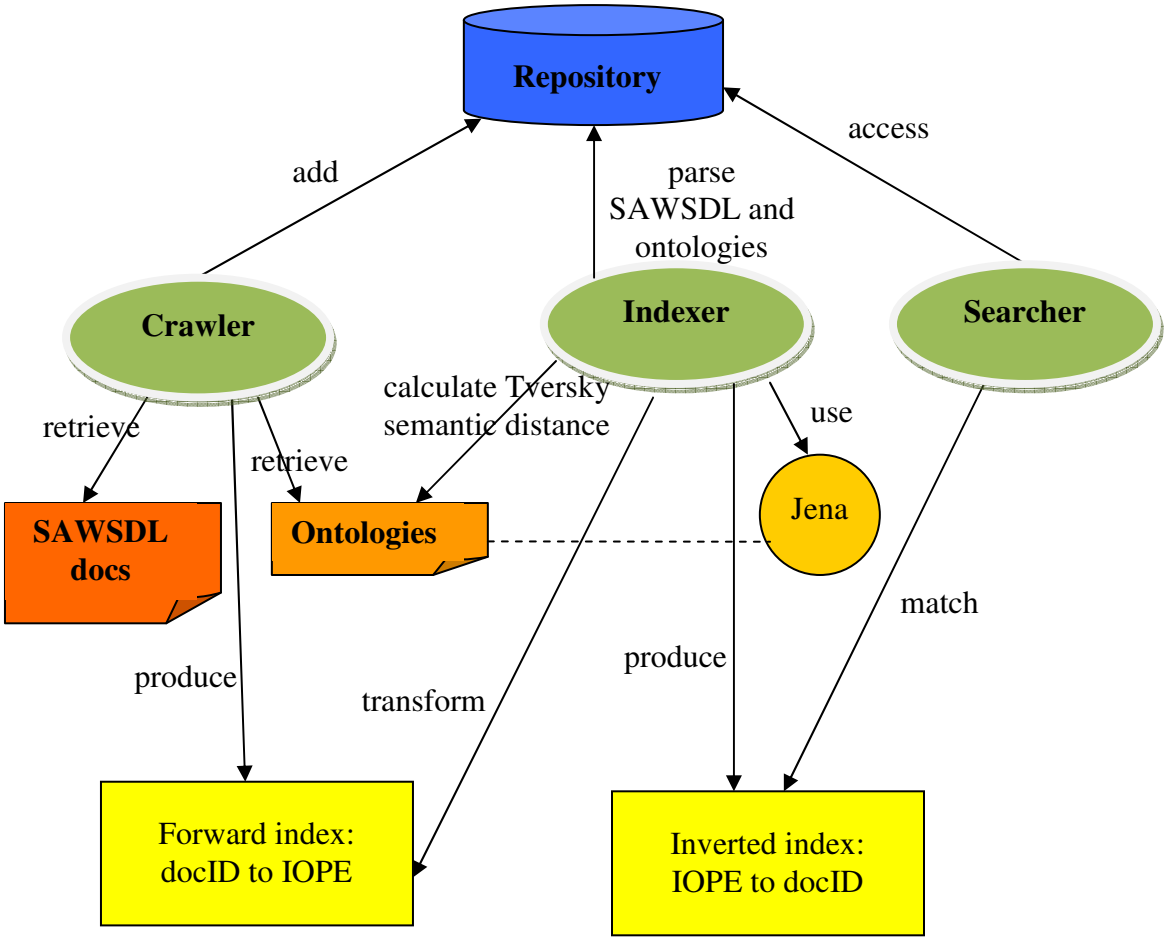


Figure 16: The building blocks of the semantic search engine

Tools Investigated

A search engine such as Google [32] uses a custom-made storage system for indexing its data, called *Bigtable*. It is a distributed storage system, designed to scale to petabytes of data across thousands of servers. For our prototype, we originally used *Apache Hadoop Distributed File System* [38], a Java open source implementation of a distributed file system designed to hold huge amounts of information and provide fast access to it. However, due to poor tooling and support, we dropped it in favor of *InfinityDB* [56]. However, we couldn't get the license for *InifinityDB* from the company that markets it. Finally we chose *BerkeleyDB* [57] (an Oracle product) to implement our indexes.

Apache Nutch [58] is an open source web crawler written in Java, based on Hadoop. It is straightforward to integrate it with *Apache Solr* [58], an open source full text search framework. Solr can also be used to index data. Its documentation states it is fast and scalable. Having crawling, indexing and searching in *Nutch + Solr* as well as high

scalability and good performance make these products together a good enterprise-level search engine.
Another open source tool, *Terrier IR* is a highly flexible, efficient, and effective open source search engine, readily deployable on large-scale collections of documents. (http://terrier.org/) It is integrated with the crawler called *Labrador*, also created at the Glasgow University.
In principle we could use either of these tools to implement our crawler, indexer or searcher, but since they are made for simple text processing, it's hard to extend them for our semantic web search engine. In a semantic web search engine, all three phases are radically different from the standard search engine:

- the crawler needs to fetch WSDL and ontology files, not HTML or other pages written in natural language
- the indexer needs to index web service data, which again, is machine-readable as opposed to the natural language terms indexed by a standard search engine
- the searcher needs to search the indexed data, i.e. the machine-readable ontological terms or XML data types

A good comparison of open source search engines is given here:
http://zooie.wordpress.com/2009/07/06/a-comparison-of-open-source-search-engines-and-indexing-twitter/

Because we need to crawl WSDL documents, we need good parsers for this format. In fact we are using semantically-annotated WSDL, known as SAWSDL, for which there are two main parsers in the open world community:

- Apache Woden + extension for SAWSDL: can parse WSDL 2.0
- SAWSDL4J: can parse WSDL 1.1

However, both have issues with parsing wsdl:types and they lack good documentation. Finally, after many debug sessions, we've come to understand how Apache Woden works and thus we can use it to extract the semantic annotations from the web services inputs, outputs, types, etc.

5.2.1 Crawlers

We've implemented our prototype using Crawler4j [39], a Java open source implementation of crawlers. We've programmed our crawlers to retrieve WSDLs from various websites, such as http://www.service-repository.com. However, note that most of the service repositories online don't contain annotated WSDL (such as SAWSDL).
Thus, we feed the crawlers manually with SAWSDL files. Note also that the SAWSDL pages typically don't have links to other SAWSDL pages (as opposed to the regular web pages).

However, the crawlers parse the SAWSDL pages looking for ontologies, pointed to by *sawsdl:modelReference* attributes. Then they also retrieve the ontology pages, e.g.: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder. Every file retrieved by a crawler is brought to a repository.

For parsing SAWSDL, originally we used SAWSDL4J [60]; it can parse WSDL 1.1 files, but it doesn't seem to parse *wsdl:type*s, so we dropped it for Apache Woden [61] combined with some extensions for SAWSDL. With this we can parse WSDL 2.0 files, which is good enough for demonstration purposes. However, in the real world we need to be able to parse both WSDL 1.1 and 2.0. Luckily, one of the latest Woden versions comes with a convertor from WSDL 1.1 to 2.0, which we use in our prototype.
The alternative is to write an own parser from scratch, potentially using C with *yacc* and *flex* for performance reasons. This is one of the future steps.
Notably, the crawlers perform an important step that is in fact typical for indexers: since they need to parse the WSDL documents, they also extract the inputs, outputs, preconditions and effects (IOPEs) of the services and store them in the repository. The next section describes the organization of data in the repository.

5.2.2 Repository

For our prototype implementation, our initial intention was to use Apache Hadoop – in fact the so-called HDFS [38] (Hadoop Distributed File System) – as repository. Among the remarkable features of HDFS, *data distribution and replication* on nodes in a cluster would be very helpful for a full-blown search engine, as mentioned in section 5.4 ("Non-functional design"). Namely, using replication in a cluster of distributed machines increases the availability and the scalability of the data store, two of the most important characteristics of a search engine repository.

However, because of immature tooling and too little support, we've dropped Hadoop in favor of InfinityDB [56]. The engine of the InfinityDB is based on an advanced B-tree data structure, which allows fast, reliable, memory-efficient data storage. In fact the very purpose of a B-tree is to implement indexes in a repository, such as to *minimize the disk access*, basically the bottleneck in read/write operations with the repository.
The issue with InfinityDB was that we couldn't even get a trial version, since we've got no answer to our request from the InfinityDB team. However, the B-tree data structure stays as the basis for our design too.

Finally, we've chosen BerkeleyDB for our prototype implementation (section 5.4) of the repository and indexes. BerkeleyDB is an Oracle product that can be used for free for non-commercial purposes. Its architecture is based on storing *(key, value)* pairs in

efficient data structures (e.g. *B-trees*), providing high concurrency and speed along with simplicity in programming. It supports ACID transactions and offers support for *replication*, enhancing thus the availability and scalability of the data store. These characteristics make it a good candidate for implementing the repository and the indexes in a search engine system.

In a real-world implementation of the repository (i.e. not the prototype which uses BerkeleyDB) we need to optimize the data structures so that we minimize the disk access – the single most time consuming operation during a search.
The WSDL files and the ontology files they point to (either RDF or OWL files) are all persisted to repository in a compressed format (*zlib*, which offers a good tradeoff between performance and compressed size).
In our prototype using BerkeleyDB, we persists the mapping between the docID (simply the URL of the WSDL file) and the compressed WSDL file.

In the real-world implementation, the repository shall contain the following data, written one after another in dedicated file(s):

- the docID (computed as a checksum of the WSDL file)
- the number of inputs
- the length (in bytes) of each input followed by the input annotation (as a string) – this is repeated for all inputs
- similar for outputs, preconditions and effects (the length followed by the list of each)
- the length of the URL of the WSDL file
- the URL itself
- the length of the compressed WSDL file
- the corresponding compressed WSDL file
- the length of the list of document IDs (refDocID) of ontology files referred to by this WSDL file
- the document IDs of these ontology files
- the address of the disk block where the entry in the ontology file repository starts (see Figure 18); at that address we find a data structure that describes the referred ontology files (see Figure 18)

This in fact is the forward index, since it contains the mapping between the WSDL files and the IOPEs.

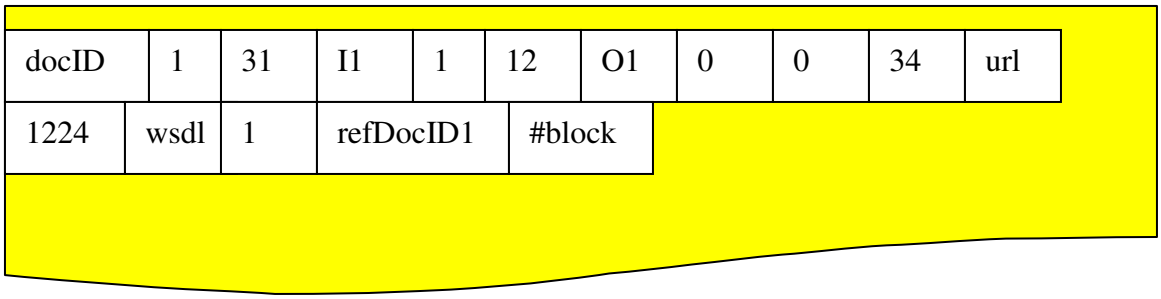


Figure 17: An entry in the repository file containing the compressed WSDL files together with their IOPEs

Figure 17 shows an example of an entry into the file containing the WSDL descriptions. The entry starts with the unique docID of the WSDL file. Then, in case we only have one input and one output, each of them is prefixed by a byte containing 1 and another byte containing the full length of the Input/Output. Note that we use *prefix compression* to save space: if a namespace is missing from any of the IOPEs, it means the namespace is identical to the previous one, e.g.:
http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#Identifier,
#Order
#Contract

The list above indicates that the concepts Order and Contract have the same namespace as Identifier, thus they belong to the same ontology.
This optimization saves a tremendous amount of space, since in practice virtually all IOPEs in a certain WSDL belong to the same ontology.
In the entry above we have zeros indicating there are no preconditions and effects (a pretty common case). Then we have 34 bytes allocated for the URL of the WSDL file, then 1224 bytes for the compressed version of the WSDL and then the compressed WSDL itself.
We close the entry with 1 reference to an ontology file, marked as *refDocID1*, which serves as entry to another repository file, as described below. Note that because a docID (also *refDocID*) is a checksum that always takes 8 bytes (by design), we don't need to specify its length in bytes.

The repository contains also ontology files, fetched by the crawlers. This data shall be kept in file(s) structured as follows:

- the docID of the ontology file
- the length (in bytes) of the compressed ontology file
- the compressed file itself

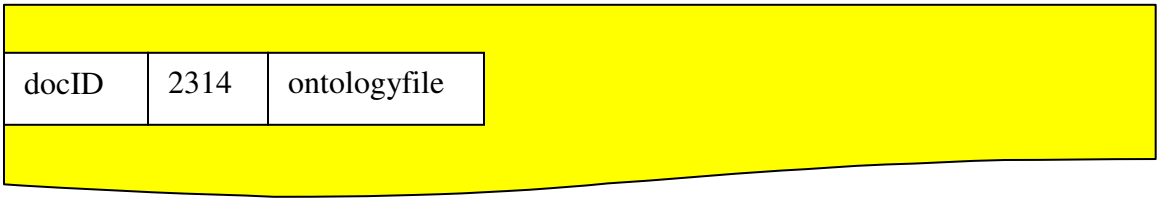


Figure 18: An entry in the repository file containing the compressed ontology files

The design decision regarding the repository structure aims at minimizing the number of accesses to disk during the search phase.

A crawler that parses a new WSDL file shall create an entry as in Figure 17 and add it to the end of the corresponding repository file. The hyperlinks to ontology files will lead the crawlers to retrieve those files, create the entries as in Figure 18 and append them to the end of the corresponding repository file.

5.2.3 Indexers

The WSDL pages are already parsed by the crawler and their IOPEs stored in the repository, so the work of the indexer becomes substantially easier.
The indexer needs to build the following indexes:

- one for the ontologies: we need to find each ontology file fast
- one for the inputs of the services: the IOPEs need to be retrieved efficiently
- similarly, we need one index for the outputs, one for the preconditions and one for the effects

The indexer takes the forward index (whose structure is shown in Figure 17) and creates the 4 inverted indexes for inputs, outputs, preconditions and effects.
Let's analyze the structure of the Inputs index, since the others will be identical.
The Inputs index is persisted in the repository. In our prototype using BerkeleyDB, we persist the mapping between each input and the list of URLs of the WSDL files where the input is found.

In a real-world implementation, each entry shall have the following structure (see Figure 19):

- the number of bytes taken by the Input text
- the Input itself
  (e.g. http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#Identifier)
- the number of documents where this Input is found
- the docIDs of the WSDL documents that contain the Input

- the block number where the entry for that docID starts in the file with WSDLs (see Figure 17)
- the block number in the file with Tversky distances where this Input starts

The indexer, besides converting the forward index into a reverse index, performs a task that speeds up the searcher: it creates a list with the concepts from the same ontology as the current Input/Output ordered by the Tversky distance to this I/O (see Figure 21). The block number where this list starts in its file is recorded into the structure mentioned above. In Figure 21 the concepts are ordered in decreasing order of the Tversky distance.

As with all the other file structures described so far, each such entry starts at a new block on disk, such that we can read it starting with that disk address. Naturally, since all entries have variable length some space is lost, because it is very unlikely that each entry will completely fill a number of blocks on disk. So each entry it's likely to waste some space in the last block where the entry is recorded.
We can minimize the wasted space by using a fixed-size wordID for each Input.

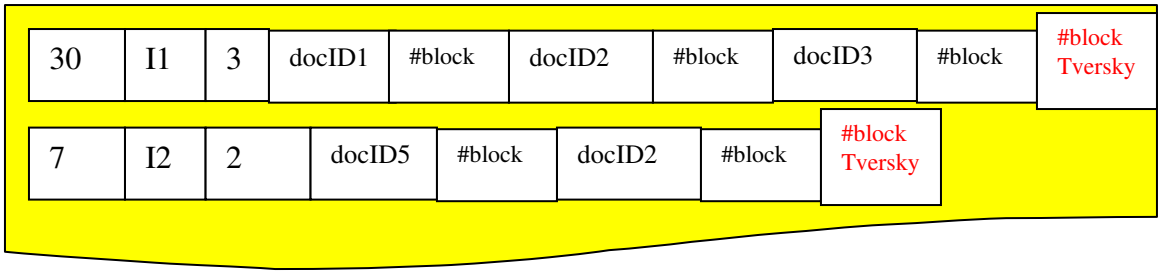


Figure 19: The structure of the Inputs inverted index (persisted in repository)

This index is mirrored in memory as a B-tree kept in sync with the persisted version. In our prototype using BerkeleyDB, we rely on the underlying B-tree implementation, without any control of how it's built.
In the real-world implementation, the in-memory index - shown in Figure 20 – shall keep the Inputs as keys while the values are ranges of disk block addresses of the corresponding entries in the persisted inverted index (see Figure 19).

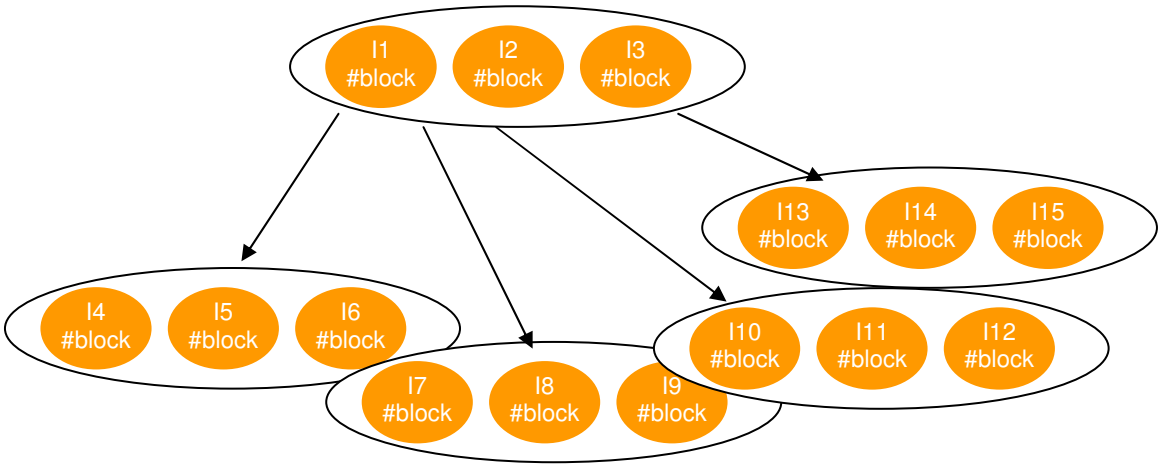


Figure 20: The in-memory Inputs index as a B-tree

A "#block" represents in fact a list of blocks: typically it is a range - the address of the start block and the address of the end block where data associated with that Input resides in the persisted index (Figure 19). However, sometimes the data of the same input *Ix* can be fragmented on disk (i.e. a new WSDL is discovered with a reference to an existing Input). In that case, the *#block* is extended to a list of ranges, e.g.:
Block1234:block1236, block1349:block1349, block1442:block1443
Note that for simplification we always have a list of ranges, even though one range may represent a single block, as is the case with *block1349:block1349*.

The keys (Inputs) are sorted alphabetically in the B-tree. Most of the operations of the B-tree are searches. Searching for a certain Input, *Ix*, is very efficient in the B-tree kept in memory. Once *Ix* is found in the tree, only the disk blocks in the retrieved range are read from the persisted index (the one in Figure 19) in order to find the corresponding docIDs (referring to WSDL files). Typically a single block will contain the docIDs where input *Ix* is found. This means one disk access for now. Then for each docID (WSDL) the disk block address in the repository file is retrieved and we seek in the file in Figure 17 at the disk block address. Thus we only read the disk blocks corresponding to the entry of that docID. Once we read the entry (see Figure 17), we have the compressed WSDL and the docIDs of the referenced ontology files. Finally, from the block address that comes with the docID, we read the compressed ontology files (see Figure 18). Also, the concepts from the same ontology in descending order of the Tversky distance from *Ix* can be retrieved fast (see Figure 21).
A simple schema that shows the order of accesses to memory and disk is as follows:

Index -> docIDs of WSDLs -> compressed WSDLs and docIDs of the ontology files -> compressed ontology files

The index is typically in memory (if it fits, read below for the case it doesn't), so hopefully we only need 3 disk accesses to get the WSDLs and the ontology files corresponding to a certain concept (input, output, precondition, effect).

Thus we minimize the disk readings – the most expensive operation – and access the WSDL and the ontology files for a given Input.
On system crash or other events, the B-tree in Figure 20 is built from the inverted index persisted in the repository (see Figure 19).

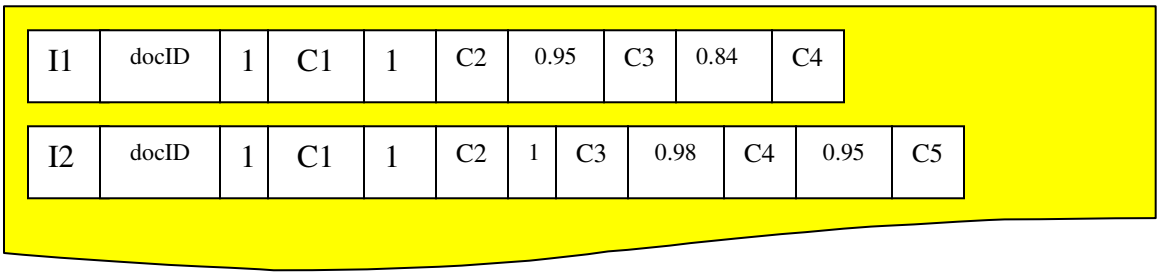


Figure 21: File with the Inputs and their Tversky distance to the concepts from the same ontology

What happens when the crawlers retrieve new WSDL with existing Inputs after the indexers have been built ? This implies the new *Input – docID* mapping needs to be inserted in the corresponding index.
The Indexer simply adds the new *Input – docID* mapping to the end of the current persisted index. Then it adds the new disk block address to the in-memory index (a B-tree). This way the persisted and the in-memory versions are kept in synchronization. However, this results in fragmentation: the docIDs for that Input are now spread in the repository and the in-memory index contains an ever-expanding list of blocks for that particular Input. In order to optimize the data structures, a special program running once at a dedicated interval (typically once a week) blocks the crawlers for a while and rearranges the repository such that the docIDs for an Input are all clustered together. It also fixes the in-memory representation by creating a new, optimal one and replacing the old one, this process being transparent to the searcher.

As explained, the data structures described so far refer to a home-grown repository, having as aim to maximize the performance of the search process. However, in our prototype implementation (section 5.4) we use *BerkeleyDB*, as explained above, which implicitly uses B-trees to represent the indexes.

Memory Constraints

Another question is: what happens if the in-memory B-tree becomes so large that it doesn't fit the memory any more ? In such a case we persist the B-tree in a repository by writing all the information in the nodes (the Input and its block references) to blocks on the disk. A new B-tree (known as an "*aux index*") is constructed containing only a fraction of the information contained previously in nodes: each node consists of more sparse inputs, i.e. I1, I6, I13, etc. The value corresponding to each Input is the block number where the input is to be found in the newly persisted data structure (from the previous B-tree). Thus, finding an input such as I12 means searching first in the aux index and finding out it is between I6 and I13, which means the search must continue at the disk block indicated by I6 (and stop before the block indicated by I13). Starting with that block, all blocks are searched sequentially until I12 is found.

Thus, only a small portion of the original in-memory B-tree is now kept in memory and the number of disk accesses is still minimized (hopefully just one extra block needs to be read, i.e. if I13 starts in the block immediately following the block that hosts I6 to I12).

This technique could continue, i.e. by creating an aux-aux index and so on, until the data structure fits in memory.

Summary of the repository data structures and indexes

In order to have a simple and illustrative example, here we give a schematic representation of the data structures (used in a real-world implementation of the search engine) presented in the previous two sections with the following interpretation: let's assume we implement each of the four persisted indexes (I, O, P and E) as four large files, i.e. one file per index (i.e. the structures in Figure 19 are stored in a file for inputs); assume the data structures used for the WSDL files (see Figure 17) are also stored in a large file and similarly the ontologies (see Figure 18) take a new large file; finally, the Tversky distance (see Figure 21) are stored in their own file.

The "#blocks" in the structures presented above are in this case file pointers into the corresponding files. A *seek* operation with such a file pointer takes us to the corresponding data structure.

The mapping from the logical file pointer to the physical disk block address is left to the operating system. Most modern operating systems use efficient B-tree structures for this mapping, so we can safely assume the seek operations in our files are efficient.

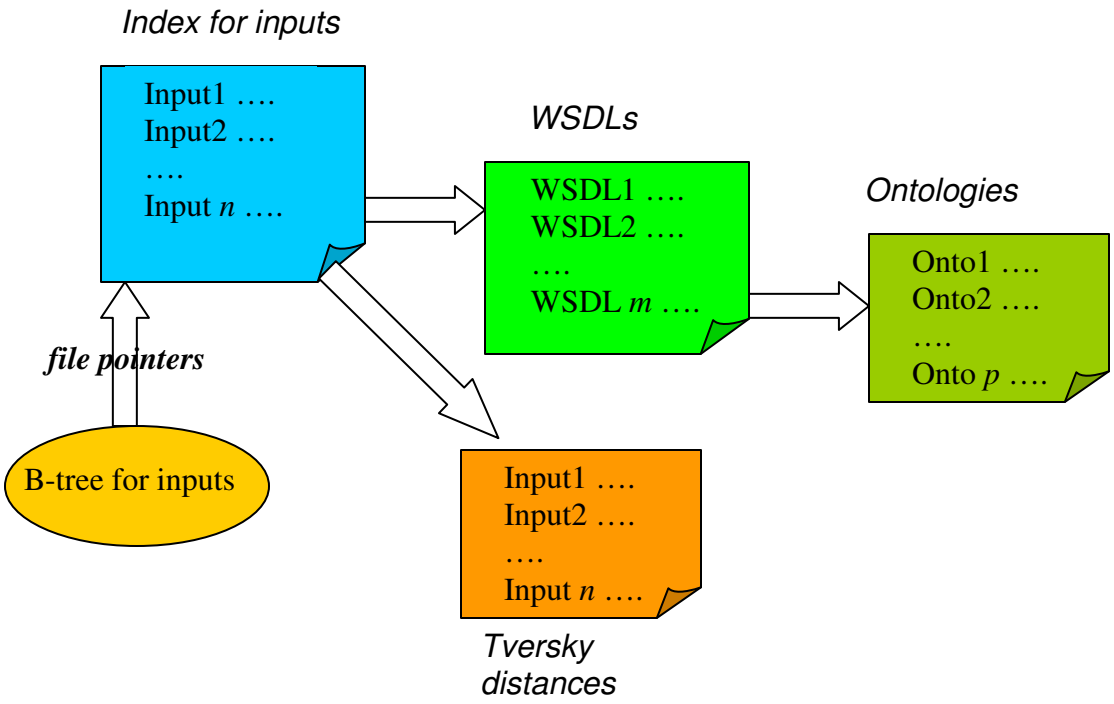


Figure 22: A possible implementation of the repository with large files

Figure 22 shows a schema of such an implementation. The B-tree is the in-memory inverted index for inputs, in whose nodes there are references to file pointers in the file dedicated to the inputs. From the latter there are references (also by file pointers) to the file that stores the compressed WSDL files, which in turn points to the files storing the compressed ontologies. The inputs index also refers to the file with the Tversky distances for inputs.

Searching a semantic web service means accessing the B-trees of the inputs and outputs, from which in a few accesses to disk we get to the actual inputs/outputs descriptions, to the WSDL files that contain them and to the ontology files referred by the latter.

5.2.4 Searcher

Given a certain profile (IOPEs), the searcher tries to find the best match among the advertised services. The matching algorithm is described in section 6.1. The whole searcher algorithm is given in section 6.2. In order to perform its algorithm, the searcher first accesses the in-memory indexes, which are stored as B-trees.

By using the in-memory B-trees, the disk accesses are minimized. No matter how large are our repositories and the corresponding B-trees, we can create auxiliary indexes (aux B-trees, as described above) until they fit the memory and starting with them we explore the persisted data structures while still minimizing the number of disk accesses.

This is utterly important for the searcher, the component for which performance is the single most important characteristic. Note also that the searcher uses the pre-computed Tversky distance to optimize the selection of the best-matching service. To further optimize the speed, the disk blocks with the corresponding Tversky distances are only loaded in memory when steps 1 and 2 of the algorithm presented in chapter 6 don't give any matches.

The performance of the search algorithm is exceptionally important for any real-world application. While the crawling or indexing steps are running behind the scenes and are not accessed real-time by clients, the search step is the "face to the client", thus its complexity is extremely important.

The search algorithm shown in section 6.2 has a few driving forces:

- the lookup steps: looking up outputs and inputs in the indexes; since the indexes are B-trees, the complexity of a lookup step is $O(\log_M(N))$, where M is the degree of the tree and N is the number of indexed inputs/outputs
- there are two nested lookup steps: one where we test all subclasses/superclasses of outputs/inputs and the other where we test all other classes in the same ontology (using the Tversky distance); this means the total complexity of the nested lookups is $O(P * \log_M(N))$, where P is the average ontology (where the inputs/outputs are found) size in terms of number of concepts

Thus, in the case we only performed a textual match among required and advertised inputs/outputs, the complexity of the search algorithm would be logarithmic in terms of number of indexed (advertised) inputs/outputs. However, in case of real semantic match complexity can increase by a factor given by the maximum number of concepts (P above) in all ontologies of the required inputs/outputs.

Remember that we use B-trees for indexes in order to minimize the number of accesses to disk, which is the bottleneck in any search operation. For a required service with one input and one output, the best case scenario – textual matching – is when the input contains its entire list of referred docIDs in one disk block, the same relation holds for the output and the match produces a single docID (the intersection of the lists of matches for inputs and outputs). We need then a minimum of 3 disk accesses: 2 to read the input and the output structures and one to read the WSDL corresponding to the matched docID, assuming the compressed WSDL fits in one disk block.

Caching

When involving caching we could avoid disk accesses altogether or at least reduce their number.

In the case of a semantic search engine, caching plays a far more important role than for a standard search engine. This is the consequence of the fact that in case of semantic matching we are typically interested only in the best match, thus we expect a single result for a match, as opposed to many in the case of alphanumeric matching.

In the cache we could keep the matched (compressed) WSDLs for the required services, while the key could be derived from the inputs and outputs, i.e.:

I1+I2+….+In+O1+O2+….+Om, where "+" represents the string concatenation operation and the inputs/outputs are ordered alphanumerically so that we make sure we treat uniformly the services with the same parameters, but different ordering.

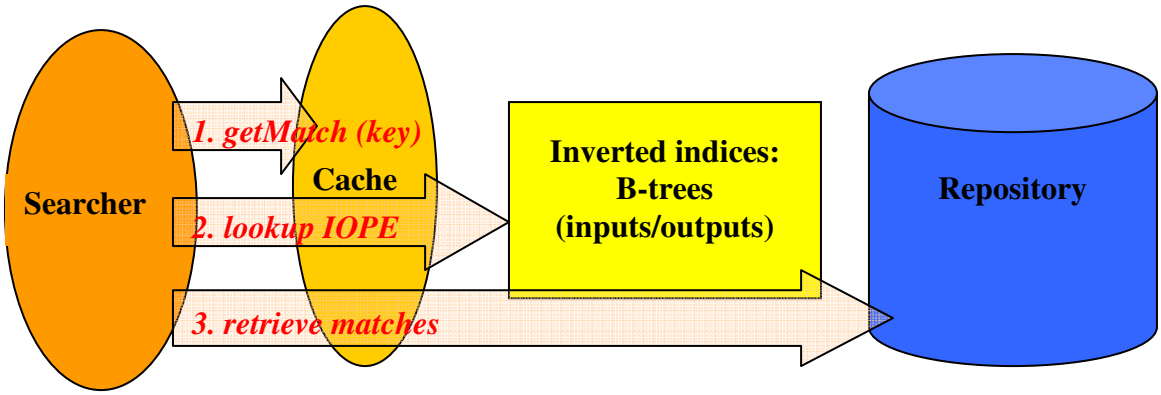


Figure 23: Searching involves the cache, the in-memory B-trees and the persistent storage

5.3 Non-functional Design – Further constraints

Just like a traditional search engine, a semantic one must fulfill certain performance criteria. Naturally, we can base our work on the experience of traditional search engines.

Among the non-functional requirements, besides performance about which we've talked above, availability and scalability score high. Typically they are dealt with redundancy, where data is replicated and distributed across the nodes, e.g. of a distributed hash table (DHT). The concept of DHT refers to systems that are both reliable and scalable, allowing distributed nodes to store and retrieve *(key, value)* pairs in a way that is transparent to the user, i.e. the user doesn't know which node stores the pair and how the nodes communicate to store and retrieve data. Many types of distributed applications, among which search engines, can be built on top of DHT data structures.

By design, disk seeks need to be avoided as much as possible, as they are expensive operations. Indexes are typically implemented as B+trees, which helps in minimizing the number of disk seeks needed to find an item.
Compression techniques should be chosen as a tradeoff between compression/ decompression speed and compression ratio. The original Google implementation chose *zlib* to compress the crawled HTML pages. Our prototype also uses zlib to compress the ontology files, which tend to be large.

5.4 The Second Prototype – a Semantic Search Engine

We have implemented a prototype for our semantic search engine, Semmed. The building blocks of this prototype are shown in Figure 16. We used open source software to implement it, as described in the next section. However, we also detail the data structures to be used in a real-world implementation. These data structures are meant to optimize the search process, i.e. to minimize the number of disk accesses. The building blocks remain the same even in the real-world implementation.
This prototype is used by the first prototype, described in section 2.3. Namely, the agents present in that prototype don't contact each other anymore by having their addresses hardcoded. Instead, they use the semantic search engine Semmed to find each other. Similarly, the "Ambulance dispatcher service", implemented as a web service, is also found by using Semmed.

We've compared the search accuracy and performance of our prototype in two cases:
- for non-annotated WSDLs
- for WSDLs annotated with IOPE according to the SAWSDL standard

In the first case, the possible matches are based on comparing the name attribute in <wsdl:input> and <wsdl:output>, e.g.:

```
<wsdl:input name = "Bank" />
```

However, the name attribute is optional here, in which case we could only compare based on the type of the respective input or output. The type comparison is more accurate, but less efficient, as e.g. the type Bank could be defined as an xsd:complexType and thus all corresponding fields need to be compared one by one. Another issue is that even if the name attribute is present, matching simply the string "Bank" could result in false positives, as "bank" could mean a financial institution in one case and a river bank in another.
After introducing IOPEs based on ontologies, the search process becomes accurate. Namely, we employ the search algorithm described in section 6.2, based on the

covariance/contravariance principle and on the Tversky distance. The result is that we get all possible matches, starting with the closest match. The results are also accurate, since now the input "Bank" is annotated with a concept from a certain ontology, so the meaning is clear. If the search process wouldn't use the semantic annotations but just do a string comparison of the "name" attribute as in the previous case, we'd miss all covariance/contravariance matches and all other matches based on the Tversky semantic distance. Besides, as explained above, we might get false positives, so we can't be sure of the accuracy of the result, if any.

Matching the input concept "Bank" with an input index of 1 million items stored in a 2-way B-tree (fitting entirely in memory) produced the following results:

- for the textual match: 16 ms, 50% false positives
- for the semantic match (involving only the covariance): 80 ms, no false positives
- for the semantic match (involving both the covariance and the Tversky distance): 149 ms, no false positives

The experiment was performed on a machine with an Intel Core i7 processor, quad-core, 2.3 GHz, 4 GB RAM.
With 10 million items in the input index we've got:

- for the textual match: 22 ms, 50% false positives
- for the semantic match (involving only the covariance): 102 ms, no false positives
- for the semantic match (involving both the covariance and the Tversky distance): 170 ms, no false positives

The accuracy gained with semantic annotations comes with a performance price. Namely, in the case without annotations the search process has $O(\log_M(N))$ complexity, where N is the number of indexed inputs/outputs and M is the degree of the B-tree used to implement the indexes. This is explained in sections 5.2.4 and 6.2. However, with semantic annotations, the search algorithm in section 6.2 has the worst case scenario complexity of $O(P * \log_M(N))$, where P is the average number of concepts across all ontologies of the inputs/outputs annotations. In practice though the indexes are fairly large, so $P << N$, thus $\log_M(N)$ becomes the dominant factor in the complexity formula, while P tends to be a constant. Besides, for very large N the indexes don't fit in memory any longer, so auxiliary indexes need to be implemented. Each auxiliary index needs an extra access to the disk, which becomes the bottleneck.

CHAPTER 6

ALGORITHM FOR MATCHING SEMANTIC SERVICES

6.1 Semantic Matching

As mentioned before, a good matching of web services is achieved if certain elements of WSDL are annotated. Namely, we refer in the first part of this paper to IOPE, i.e. inputs, outputs, preconditions and effects. In this paper we deal with services whose descriptions contain IOPEs annotated with ontological concepts.

The first important requirement to the data our search engine works with is that the WSDL descriptions of the services be annotated with semantic information. Our semantic search engine deals with services whose descriptions (WSDL files) contain IOPEs annotated with ontological concepts using SAWSDL. Our crawlers are fed with URLs pointing to semantically annotated WSDL documents. They fetch the documents and store them compressed in a repository. Each document is assigned a unique docID, derived from its URL. Importantly, the crawlers also fetch the ontology files used to annotate the WSDLs.

The indexer parses each WSDL document and creates a forward index, which keeps the relation between each docID and the hits, i.e. I, O, P, E concepts. At the same time, for each ontology referred by the IOPE concepts, the indexer involves an inference engine [35] to calculate the Tversky's matching values between each two concepts of that ontology. This list of matches is sorted in descending order by the matching values (the best matches first) and is stored together with the corresponding ontology file. As we'll see later, this greatly improves the performance of the search.
Then the forward index is converted into four inverted indexes, each corresponding to I, O, P and E respectively. For each concept from IOPE, a unique wordID is generated. The indexes are sorted by this wordID.
With these preparations in place, the search process needs to return the best matching web service for a given profile. By profile we mean a set of IOPEs representing the desired characteristics of a service. The preconditions and effects are not mandatory, but the inputs and outputs are.

The search algorithm is structured on three levels:

1. First a syntactic (textual) matching is done with the IOPEs of the given profile; this is the same as what a traditional search engine would do, but it only returns the most relevant match (preferably one that covers all IOPEs)

2. If there was no result, a first kind of semantic matching is done: for each *output* concept in the given (requested) profile, we take each *subclass* in its ontology in increasing order of the semantic distance, as defined in [34] (below we explain why we consider the subclasses for outputs and superclasses for inputs); we compute the wordID for the current subclass and look it up in the corresponding index; thus, we try to find the best matches for all outputs according to the semantic distance; if at least one output can't be matched, the algorithm moves to step 3; then we do the same for *inputs*, but now we take the *superclasses* of each given (requested) input; if we have at least one input matched, we find the matches common for the outputs and the inputs; finally, we check which of these also match the preconditions and effects of the given profile, if they exist

3. If there was no result from the previous step, more complex semantic matching is performed: for each output concept in the given profile, we take all concepts from the same ontology (except the subclasses covered at step 2) in decreasing order of Tversky's model value pre-calculated for the current output; we compute the wordID for each such concept and look it up in the corresponding index; if at least one output can't be matched, the algorithm stops and no match is returned; then we do the same for inputs; if we have at least one input matched, we find the matches (services) common to the outputs and the inputs; finally, we check which of these also match the preconditions and effects of the given profile, if they exist

Note that in step 1 we do a standard, textual match. This helps when an advertised web service is annotated with a concept such as:
http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#OrderRequest,
which is also part of the required profile and where the namespace and the name of the concept match textually. This alleviates the need for a semantic match, since we can assume both the required profile and the advertised service mean the same thing.

Step 2 does a first semantic match. Note that we need all outputs of a required profile to be matched. If any output is not matched by an advertised web service, we'd remain with a partial answer, which is not acceptable, since the outputs are used to compose and further invoke web services. As for the inputs, the requirement is less strict: we need at least an input to match, not necessarily all. In order to match an input, we use the *covariance* principle: if the requested profile's input is a subclass of the advertised service's input, they match. Intuitively this means that if an advertised service can deal

with a certain concept, it can also deal with its derived concepts, since they are just restrictions of the original concept. That's why in our algorithm we look for the superclasses of required inputs among the advertised inputs.
However, for the outputs we have the dual principle (*contravariance*): if an advertised service outputs a certain concept, we can never require a more restrictive version (i.e. a subclass) of that. Only a superclass of that concept will match.
Finally, if there are no matches so far, Tversky model is used, which provides a matching value for any two concepts of an ontology. We start with the best Tversky match (e.g. two concepts which differ by just a property) and then, if no match is found, we proceed with lower matches, until a certain threshold. The threshold is necessary to eliminate really bad matches and it is a parameter of the search process.

Example

Let's take an example to clarify the algorithm. Below we show a part of the PurchaseOrder ontology, typically used in examples of w3.org documents.

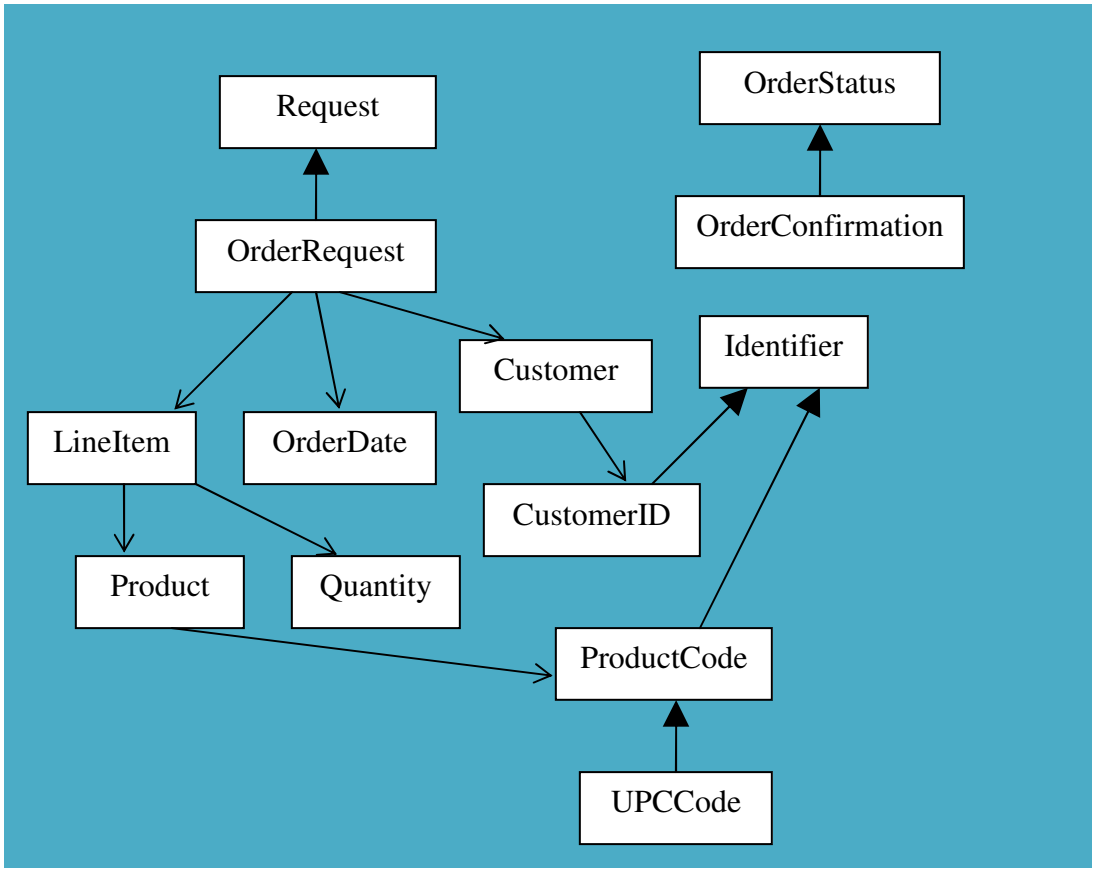


Figure 24: An example ontology

Suppose the required profile has three inputs and one output:

RI1: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#CustomerID

RI2: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#ProductCode
RI3: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#OrderDate
RO1: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#OrderStatus

Basically, the client requires a service which returns the status of an order, based on the customer ID, the ordered product code and the order date.

Let's assume there is an advertised service with the profile given by two inputs and one output:
AI1: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#Identifier
AI2: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#Identifier
AO1:
http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#OrderConfirmation

Step 1 of the described algorithm doesn't produce a match, since there is no textual match between the requested inputs/outputs and the advertised ones.
In step 2, we first try to match the requested output, RO1. Our inference engine, Jena [35], retrieves all its specialized concepts from the given ontology and we look them up in the appropriate index. Thus, we find OrderConfirmation, which exists in our index since it is an output produced by an advertised web service. This is logical, because OrderStatus is a generalization of the advertised service's output, OrderConfirmation (AO1), so, applying the contravariance principle, their services match.
Then we take the inputs of the required profile in order. For RI1, our Jena inference engine retrieves all generalizations of CustomerID, in this case just Identifier. This concept is found in the appropriate index, since it is an input of an advertised service. Similarly, the second input, RI2 is also matched with "Identifier". The third input is not matched, but overall this doesn't matter, since all outputs match and at least an input.
The conclusion is that the two profiles match. We also know how to invoke the advertised profile: since there is no match for the third input, we pass only two identifiers as parameters (the first is a customer ID and the second is a product code) and we expect one result, namely an OrderStatus.

Now suppose the advertised service has as the second output:
AI2: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#UPCCode, instead of "Identifier".
Step 1 still doesn't result in a match. At step 2, when trying to match the second input we won't get a match, since the generalization Identifier of RI2 doesn't match any more. Thus we go to step 3, where we take the closest concept from the ontology with respect to the Tversky distance [36] to our RI2 input, ProductCode. This concept is UPCCode, since the two concepts only differ by a property. Thus, we have again a match between the

required and the advertised input, though the match is suboptimal, since the Tversky semantic distance is less than 1.
In order to invoke the matching service, the client knows it has to pass an instance of UPCCode as the second parameter. If however it passes an instance of ProductCode as originally intended, invoking the service might fail (it fails in those cases where the instance is not an UPCCode, but this is the price to pay for not having an exact match).

Finally, let's suppose the advertised service has the following profile:

AI1: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#Identifier
AI2: http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#CustomerID
AO1:
http://www.w3.org/2002/ws/sawsdl/spec/ontology/purchaseorder#OrderConfirmation

Just like before, at step 2 we match both the outputs and the first respective inputs. When trying to match the second input, ProductCode, we take all ontology concepts in decreasing order of Tversky distance from ProductCode. UPCCode, which only differs by a single property, is the closest but it's not found in the index, so we move to CustomerID, with which ProductCode has 4 properties in common (in the given ontology). Thus the semantic distance is 4 / NumberOfProperties(ProductCode) = 4/6 = 0.667. We find CustomerID in the index, since it is the second input of the advertised service. We have again a suboptimal match, since the Tversky semantic distance is less than 1.
So in order to invoke the advertised service, the client can pass a CustomerID as the first parameter, but it can't pass a ProductCode as the second, since it is not accepted by the advertised service. Instead, the second parameter needs to be also a CustomerID. This means that the ProductCode that was originally intended as the second parameter needs to be "disguised" as a CustomerID, i.e. only the common properties can be used for the communication between the client and the web service. In practice this makes only limited sense and the acceptance of the web service result can only be decided by the client. For example, if the client has means to convert a ProductCode to a valid CustomerID, then the result of the web service can be considered reliable.
Another option here would be to ignore the order of the required service parameters. In other words, instead of considering the input parameters in the order CustomerID, ProductCode, we can reverse them, in which case we have a perfect match: RI1 (ProductCode) matches AI1 (Identifier) by covariance and RI2 (CustomerID) matches AI2 (CustomerID) by identity.

6.2 Semantic Search Algorithm

In chapter 5 we describe the design of the proposed semantic search engine. We also detail the data structures and the persistent structures needed to support our data, i.e. the WSDL and the ontology files, the inputs, outputs, preconditions and effects.
The first part of this chapter highlights the semantic matching algorithm. In section 2.6 we present ontology matching, needed to match services described in different ontologies, yet using similar concepts.

Here we present the algorithm employed by the semantic search engine, which makes use of the matching algorithm and of the data structures mentioned above. Simply put, given a requested profile with the IOPE description *(RI*, RO+, RP*, RE*)*, the searcher needs to find the best matching advertised service, i.e. its WSDL description and the ontology files referred to by this one.
Note that RI* means there can be any number of requested Inputs, including 0, but RO+ shows that there should be at least one requested Output.

1. For each output, ROi
- listMatchedOutputs(ROi) <- nil
- lookup ROi in the Outputs index
- if found
    - foreach matched docID // add all docIDs of the WSDL files referred to by this output
        - listMatchedOutputs(ROi).Add(docID)
        - listMatchedOutputs(ROi)[docID] = 0 // mark the semantic distance as 0, the best match
- for each subclass RSubOi of ROi in its ontology
    - lookup RSubOi in the Output index
    - if found
        - foreach matched docID
            - listMatchedOutputs(ROi).Add(docID)
            - listMatchedOutputs(ROi)[docID] = semantic_distance(RSubOi, ROi)

// try the Tversky distance
- foreach class RAnyOi <> RSubOi in the same ontology
    - lookup RAnyOi in the Output index
    - if found
        - foreach matched docID
            - listMatchedOutputs(ROi).Add(docID)
            - listMatchedOutputs(ROi)[docID] = Tversky_distance(RAnyOi, ROi)

- if listMatchedOutputs(ROi) is nil, return; //the algorithm doesn't find a match; all outputs need to be matched by the algorithm above

2 seek in all listMatchedOutputs(ROi) lists and find docIDs common to all lists // all outputs need to be matched
- make a list containing all common docIDs, let's call it listMatchedOutputs

3 if listMatchedOutputs is nil, return; // no match found

4 for each input, RIi

- listMatchedInputs(RIi) <- nil
- lookup RIi in the Input index
- if found
    - foreach matched docID // add all docIDs of the WSDL files referred to by this input
        - listMatchedInputs (RIi).Add(docID)
        - listMatchedInputs (RIi)[docID] = 0 // mark the semantic distance as 0, the best match

- for each superclass RSuperIi of RIi in its ontology
    - lookup RSuperIi in the Input index
    - if found
        - foreach matched docID
            - listMatchedInputs (RIi).Add(docID)
            - listMatchedInputs(RIi)[docID] = semantic_distance(RSuperIi, RIi)

// try the Tversky distance
- foreach class RAnyIi <> RSuperIi from the same ontology
    - lookup RAnyIi in the Input index
    - if found
        - foreach matched docID
            - listMatchedInputs (RIi).Add(docID)
            - listMatchedInputs(RIi)[docID]= Tversky_distance(RAnyIi, RIi)

5 seek in all listMatchedInputs(RIi) lists and find docIDs common to all lists
- make a list containing all common docIDs, let's call it listMatchedInputs

6 seek in listMatchedOutputs and listMatchedInputs all common docIDs

7 for each docID in the final list, calculate the match factor as the average over the semantic distances of all matched Inputs and Outputs

8 order the final list by the matching factor, increasingly (the lower the better)
- in case of equality the docID that also matches the preconditions and/or effects takes priority

9 return the WSDL file corresponding to the first docID in the final list

The algorithm tries initially a textual (syntactic) match, then a first semantic match by considering the direct subclasses/superclasses and finally a Tversky match. This order also gives the matches in their decreasing order, i.e. the best matches first.
However, note that the Tversky match is pretty costly, since it involves all classes from the ontology. An optimization of the algorithm takes into account the fact that the Tversky match would result anyway in a worse match than a textual one or a superclass/subclass one.

Let's take a look at the following example, where we look for a service with one input and one output (I1, O1). Below we give the *listMatchedOutputs(O1)* and *listMatchedInputs(I1)* as they result from the algorithm after the steps of textual match and superclass/subclass match. Note that in the brackets we give the docID of the SAWSDL file where a match for input/output appears and the semantic distance to I1/O1:

listMatchedOutputs(O1): *(docID5, 0), (docID2, 0.1), (docID12, 0.2)*
listMatchedInputs(I1): *(docID4, 0), (docID2, 0.1), (docID10, 0.15), (docID12, 0.3)*

We haven't done any Tversky match so far. Semantic distance 0 is the first in each list and it means a textual match, then semantic distances are based on the subclass/superclass match and they are ordered increasingly (the greater the semantic distance, the lower the match).
Note that applying step 6 from the algorithm we already find two matches for (I1, O1):
(docID2, 0.1), (docID12, 0.25)
The resulting semantic distance is the average over the I1 and O1 matches. That is, docID2 contains the best match for the service with (I1, O1), with the average semantic distance of 0.1

Since we are looking for the best match we know that it would only make sense to apply the step of Tversky match if that resulted in a better match than (docID2, 0.1). We could for example find a new match for I1: *(docID5, 0.1)*, which, combined with the match *(docID5, 0)* for O1 would result in *(docID5, 0.05)*, which is better than *(docID2, 0.1)*. However, we know this new potential match, *(docID5, 0.1)*, is impossible, since the list listMatchedInputs(I1) is ordered, so any Tversky match would be certainly worse than 0.3.

The conclusion is that we could filter out the expensive Tversky distance match and apply it only if really needed, i.e. only if by applying it a potentially better match could result. For example, if the lists above were:

listMatchedOutputs(O1): *(docID5, 0), (docID2, 0.1), (docID12, 0.2)*

listMatchedInputs(I1): *(docID4, 0), (docID10, 0.15), (docID12, 0.3),*

the best match so far would be *(docID12, 0.25)*; a Tversky match could find e.g. *(docID4, 0.3)* for the O1 match and then the best overall match would be (docID4, (0 + 0.3)/2) = *(docID4, 0.15)*, which is better than *(docID12, 0.25)*. In this case the step of Tversky match would make sense.

With this optimization in place, the algorithm above becomes:

1. For each output, ROi
- listMatchedOutputs(ROi) <- nil
- lookup ROi in the Outputs index
- if found
    - foreach matched docID // add all docIDs of the WSDL files referred to by this output
        - listMatchedOutputs(ROi).Add(docID)
        - listMatchedOutputs(ROi)[docID] = 0 // mark the semantic distance as 0, the best match
- for each subclass RSubOi of ROi in its ontology
    - lookup RSubOi in the Output index
    - if found
        - foreach matched docID
            - listMatchedOutputs(ROi).Add(docID)
            - listMatchedOutputs(ROi)[docID] = semantic_distance(RSubOi, ROi)

- if listMatchedOutputs(ROi) is nil, return; //the algorithm doesn't find a match; all outputs need to be matched by the algorithm above

2. seek in all listMatchedOutputs(ROi) lists and find docIDs common to all lists // all outputs need to be matched
- make a list containing all common docIDs, let's call it listMatchedOutputs

3. if listMatchedOutputs is nil, return; // no match found

4. for each input, RIi

- listMatchedInputs(RIi) <- nil
- lookup RIi in the Input index
- if found

- foreach matched docID // add all docIDs of the WSDL files referred to by this input
  - listMatchedInputs (RIi).Add(docID)
  - listMatchedInputs (RIi)[docID] = 0 // mark the semantic distance as 0, the best match

- for each superclass RSuperIi of RIi in its ontology
  - lookup RSuperIi in the Input index
  - if found
    - foreach matched docID
      - listMatchedInputs (RIi).Add(docID)
      - listMatchedInputs(RIi)[docID] = semantic_distance(RSuperIi, RIi)

5. seek in all listMatchedInputs(RIi) lists and find docIDs common to all lists

- make a list containing all common docIDs, let's call it listMatchedInputs

6. seek in listMatchedOutputs and listMatchedInputs all common docIDs

7. for each docID in the final list, calculate the match factor as the average over the semantic distances of all matched Inputs and Outputs

8. order the final list by the matching factor, increasingly (the lower the better)

- in case of equality the docID that also matches the preconditions and/or effects takes priority

9. if the Tversky match makes sense

- foreach output, ROi
  - foreach class RAnyOi <> RSubOi in the same ontology
    - lookup RAnyOi in the Output index
    - if found
      - foreach matched docID
        - listMatchedOutputs(ROi).Add(docID)
        - listMatchedOutputs(ROi)[docID]= Tversky_distance(RAnyOi, ROi)

- foreach input, RIi
  - foreach class RAnyIi <> RSuperIi from the same ontology
    - lookup RAnyIi in the Input index
    - if found
      - foreach matched docID
        - listMatchedInputs (RIi).Add(docID)
        - listMatchedInputs(RIi)[docID]= Tversky_distance(RAnyIi, RIi)

- go to step 5

10. return the WSDL file corresponding to the first docID in the final list

Note that initially we find the textual matches and those based on the superclass/subclass semantic distance. We then pair the lists of matches for all outputs and inputs in order to find the common docIDs for all inputs and outputs of the service.

At step 9 we check if applying the Tversky match still makes sense, as described above. For example, if the list at step 8 contains a docID with the resulting semantic distance 0 (a purely textual match) we definitely know we can't do better, so step 9 is skipped. This improves the speed of the search, since the Tversky step means trying out each concept from the ontology.
Thus, an update to the section 5.2.4 is needed here: the complexity of the search algorithm is:

- in the best case: $O(\log_M(N))$, where N is the number of indexed concepts (inputs/outputs) and M is the degree of the B-trees used to implement the indexes
- for semantic matching involving the subclasses and superclasses: $O(S * \log_M(N))$, where S is the average number of subclasses/superclasses across all concepts from all ontologies of the inputs and outputs
- for complex semantic matching (involving the Tversky distance): $O(P * \log_M(N))$, where P is the average size (in number of concepts) over all ontologies of the inputs and outputs (clearly, $S < P$ or even $S << P$, which results in a performance gain when the Tversky match is not needed)

Invoking a Service after the Search

Once the service is found, note that we can invoke it with a signature that differs from the actual signature of the service. This is due to the semantic match, based on covariance/contravariance and on the Tversky similarity, which means e.g. we can invoke the following service method:

```
EPR findMedicalHistory (Person p)
```

with:

`EPR findMedicalHistory (Patient p)`, because *Patient* is a subtype of *Person* in the ontology that describes the annotated input of the given service and we can apply the covariance principle.
In this example Patient and Person are types in certain programming languages. Obviously, in a regular web service invocation this would result in a compile-time error, since the signatures don't match. In order to make this work, the proxy to the service needs to perform the dynamic cast from a Patient to a Person and thus call the service

with the expected Person type. In general, the proxy needs to be able to convert from any type to another in the same ontology, provided there is a non-null semantic similarity between the two concepts. Naturally, during such conversion some information is lost (e.g. just some properties will go from Patient to Person, the other properties of the patient get lost), but this is the price to pay for less accurate matches.
The example given so far is appropriate for legacy web services, whose descriptions are semantically annotated, but their signatures don't change.

Another possibility is to have services which accept as inputs/outputs just strings, but each such string is an OWL/RDF description of the input/output concept. This is briefly described in section 3.1. In such a case the proxy doesn't need to perform any conversion, but it's the responsibility of the service to map the inputs/outputs. For example, the service would need to parse the input *Patient* shown as example in section 3.1 and convert it to a *Person* concept, losing some of the patient's information.

Ontology Matching and Searching

Note that the algorithm doesn't mention anything about ontology matching.
The issue can be formulated as follows: given a required service RS with an output defined based on ontology O1, how can we find (and include in the algorithm above) an advertised service, AS, with an output defined based on ontology O2, knowing that there exists a transformation T between O1 and O2 ? (Note that the transformation T from O1 to O2 defines an equivalence relationship between concepts, but it might be that some concepts from either O1 or O2 are not defined in the other ontology.)

We don't want to miss potential services just because they have the inputs/outputs defined in another ontology than the required service. The inputs/outputs of the required and advertised services may represent the same concept, albeit in different ontologies.
In the search algorithm described above, the ontology matching would be needed in the step where the Tversky distance is computed. So, in the step "foreach class RAnyOi <> RSubOi in the same ontology" in the algorithm above we would need to explore all classes from all ontologies referred to by the advertised services and not just the classes from the same ontology. This is of course prohibitive in terms of speed and thus it would render the algorithm useless.
However, in the medical domain there are a handful well defined ontologies, thus we can assume the transformations between them (T as described above) are known and registered. With this knowledge in mind, the step above becomes:
foreach class RAnyOi <> RSubOi in the same ontology and in each mapped ontology
By "mapped ontology" we understand any ontology for which the transformation T to the given ontology is defined and registered.
As we know, a level of indirection solves every problem in computer sciences. Consequently, the searcher asks another component – *ontology matcher* – for all mapped

ontologies to a given one in order to perform the step mentioned above. This is shown in Figure 25.

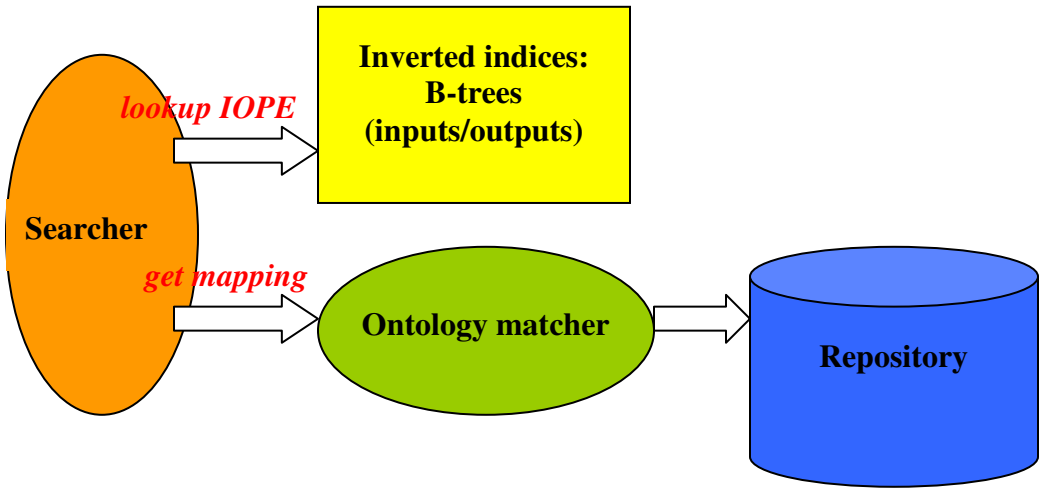


Figure 25: The ontology matcher in the context of searching

The figure shows how the searcher implements the step mentioned above. After first looking up the inputs/outputs textually in the inverted indices (B-trees), if there are no matches, the searcher proceeds with semantic matching: first the simple semantic distance is used and the direct hierarchy is considered (the sub/superclasses in the same ontology), but if that fails too, the searcher looks up all concepts in each mapped ontology in the indices.

The ontology matcher is a component that given an ontology (by ID), retrieves all ontologies for which there are transformations T defined from the given ontology. Also, given a concept and the ontology where the concept is defined, it returns all equivalent concepts in the mapped ontologies.
The ontologies are mapped manually by domain experts. Thus we expect that the medical ontologies are mapped and registered to the repository.

6.3 Ontology Matching and the Tversky Distance

In the previous sections we referred to matching concepts from the same ontology. However, the algorithm presented previously regarding the Tversky distance is also applicable to concepts from different ontologies, as described in [36].

In order to calculate the Tversky distance between concepts from different ontologies, we need to find the semantic distance between two words (the words being the concept names, but also their respective properties). This can be done with various algorithms, among which the most common are:

- Q-grams
- Jaro-Winkler distance
- Levenshtein distance
- synonyms

In our algorithm we involve Jaro-Winkler distance and synonyms.
Q-grams refer to groups of "q" letters within the analyzed words, e.g.: in the word PATIENT, the 2-grams are: PA, AT, TI, IE, EN, and NT. A simple method of assessing the semantic distance between two words is to count the number of common q-grams. However, how can we say that two concepts are similar depending just on the number of common q-grams ? We need a normalized measure (between 0 and 1) and, for example, all measurements above 0.9 lead to the conclusion of similar concepts. So, given two strings, s1 and s2, we could define the measure:
((common q-grams / |s1|) + (common q-grams / |s2|)) / 2

Thus, we take the average of the number of common q-grams split by the total number of q-grams in each string.
Another question is what is the optimal "q" ? Should we try 2-grams, 3-grams or a combination of several q-grams ? Our experiments show satisfactory results with q = 2.

However, the same experiments show even better matching using the Jaro-Winkler distance [42]. This is based on the Jaro distance, which has the formula:

$Dj = 1/3 * (m / |s1| + m / |s2| + (m - t) / m)$,
where m is the number of matching characters and t is half the number of transpositions. A transposition is a wrong ordering of two letters, e.g. in the words "John" and "Jhon", OH and HO are inversed, so they count as 2 transpositions.

A very interesting feature of the Jaro distance is that the result is between 0 and 1, which makes it easy to use to classify ontological concepts. For example, we can decide that two concepts are the same if their Jaro distance is above 0.9.

The Jaro-Winkler distance gives higher scores to strings that have common prefixes (the first few letters are common):

$Dw = Dj + L * P * (1 – Dj)$,
where L is the length of the common prefix (but maximum 4) and P is a scaling factor with the maximum value 0.25, such that the result is still kept under 1. Typically, P is chosen 0.1.

Naturally, we need to eliminate false positives, e.g. notions such as Patient and Patent have totally different meanings even though their Jaro-Winkler distance is very close to 1:
Dj = 1/3 * (6/7 + 6/6 + (6 – 0)/6) = 0.952 and
Dw = 0.952 + 0.1 * 3 * (1 – 0.952) = 0.966

In order to filter out these concepts we compare not only the concepts' names, but also their properties, e.g.:

Patient
- *name*
- *dateOfBirth*
- *address*
- *medicalHistory*

Patent
- *name*
- *domain*
- *submitDate*
- *publishDate*

etc.

Calculating the semantic distance between the properties shows that the concepts are in fact different.

On the other hand, we can also have false negatives, e.g. PatientRecord and MedicalHistory. The Jaro-Winkler distance shows that the concepts are totally different, while in fact they are synonyms. We can conclude this by either:

- employing a dictionary of synonyms or
- calculating the semantic distances between their properties (which gives a result opposite to the above example)

In section 2.6 we discussed about higher-level issues involved in ontology matching, but also about concrete tools (such as Cupid [42]) that employ various algorithms, such as the ones described in this section. In the context of the proposed semantic search algorithm we showed how ontology matching can be used – namely the block "Ontology matching" in Figure 25 could employ Cupid or another existing tool.

CONCLUSIONS

C.1. General Conclusions

- In the first part of the thesis we've presented the building blocks of a system based on software agents and web services, which is designed to assist in improving the quality of the medical act.

The system is based on semantic capabilities, which are supposed to enhance the interoperability among medical systems. Why is interoperability important ? Because in our modern, fast-paced society it is essential to collect, aggregate medical data from various sources. For example a doctor is always interested in the patient's medical history, which needs to be collected from all hospitals/clinics where the patient has been treated or diagnosed.

Semantics play a crucial role here because:

- in the process of interoperability, various parties use different terminologies/ontologies; they won't be able to communicate without semantic inference;
- semantics dramatically enrich communication, the expressivity being far better than in simple text communication.

We've tried to unify the view on agents and services by showing that both present the same challenges:

- they need to be semantically annotated; in order to do that, we need to employ ontologies;
- they need to be registered in certain repositories;
- they need to be semantically discovered;
- they need to be consumed, i.e. they communicate.

- We've proposed the extensions of FIPA service description to allow annotation with IOPE (inputs, outputs, preconditions and effects), just like the research with the same proposal for SAWSDL in the world of web services suggests. Our prototype (section 5.4) shows that just by adding IOPE semantic annotations, the search process is dramatically enhanced in terms of accuracy, while at the same time it is still scalable and efficient (i.e. the IOPE annotations don't affect significantly the search efficiency). We've compared the search process before and after adding the IOPE annotations, as described in section 5.4.

- We've also considered RESTful services with their advantages for mobile devices (where typical agents are likely to reside) and the semantic annotation of WADL – the service description.

The discussion about integrating the service-oriented architectures with agent-oriented ones started with a real-world scenario, presented in section 1.1. We've shown that some services needed by various stakeholders in the medical domain (patients, doctors, specialists, etc.) can be more logically implemented as web services, while others as software agents. Logic inference plays an important role in this scenario and in the issue of semantic interoperability in general.

The second part of the thesis delves into the issues which remained unsolved in the first part. Namely, we present a system based on software agents and semantic web services that discover each other and communicate seamlessly. In order for this to take place, two important issues need to be solved:

- the registration and discovery of services based on their semantics rather than on syntax;
- the possibility to interoperate across ontologies (a service/agent searching concepts from a certain ontology and being offered compatible concepts from another – ontology matching needs to take place).

- In this thesis we argue that instead of using UDDI for service registration and extending it such that it is able to work with semantic information, we propose the design and implementation of a search engine able to register and retrieve semantic web services.

As with a regular search engine, a semantic one involves crawlers for retrieving the semantically annotated services, an efficient repository for storing them, an indexer to index the services and a searcher which retrieves the matching services using the indexes.
There are a few assumptions made by the design of our semantic search engine:
- we only consider SAWSDL files that describe web services; SAWSDL is a standard that extends the WSDL descriptions with semantic information
- the match of web services is the match of their inputs, outputs, preconditions and effects (IOPEs)

- We also argue that the service descriptions of software agents could also be semantically annotated using SAWSDL and thus indexed by the same search engine.

- In this paper we present the design details of an efficient way to store and retrieve data, based on B-trees. The data structures we use aim at minimizing the disk access, which is the bottleneck of the tasks performed by the searcher.

- On the other hand, we present a prototype of a semantic search engine (section 5.4), whose implementation is based on open source tools, with its components: crawlers, repository, indexer and searcher. The single most important bottleneck is the disk access necessary for the searcher to match the required profile. However, with the highly efficient BerkeleyDB implementation of the repository, retrieving a record out of 100000 is a matter of 20-30 milliseconds. On the same machine, a classic SQL select operation from a relational database of similar size takes an order of magnitude longer.

C.2. Original Contributions

The main contributions of this thesis are as follows:

1. We propose a platform for the medical domain based on software agents and web services, that is, a combination of what we could call Agent-Oriented Architectures (AOA) and Service-Oriented Architectures (SOA). The agents and services interoperate with each other by using ontologies and existing standards, such as HL7, DICOM, FIPA, SOAP, SAWSDL, OWL, etc. In this way we hope to align the medical institutions and their software departments to this common platform. The main efforts for an institution to embrace such a system are dedicated to annotating their agents and services and potentially creating mappings between ontologies and registering these mappings.

2. We propose annotating the software agents' descriptions with inputs, outputs, preconditions and effects (IOPEs), much the way web services can be annotated using the SAWSDL standard. For this to take place, an extension of the FIPA standard [11] for agents is needed, namely the agent service description (registered in the Directory Facilitator (DF)) needs to be extended with semantically-annotated IOPEs. This is detailed in section 2.5.3.

3. Once the semantic annotations for both services and agents are in place, we propose as a backbone for our system a semantic search engine, which serves both to index the services and agents as well as to allow their semantic discovery. We call the search engine *Semmed*, which suggests it is based on semantics and it is meant for the medical domain (in fact it is more general, but because of the complexities of ontology mapping, it is more realistic to apply it in a certain domain, with known ontologies). The semantic search engine proposal is covered in Part 2. We also describe the implementation of a prototype (section 5.4) using open source tools.

The semantic search engine has multiple roles:
- serve as a repository for the annotated services;
- replace UDDI-based repositories, since UDDI is not supported any more by its proponents;
- add semantic capabilities to the search process.

4. The search algorithm is also an original contribution, even though it is based on the known concept of semantic distance. Namely, the algorithm probes three semantic matching steps with increasing complexity, with the purpose of minimizing the overall complexity. This is detailed in chapter 6.

5. In section 5.2 we propose certain data structures to be used by the components of the semantic search engine, so that minimize the complexity of the search algorithm. Specifically, the bottleneck in such algorithms is the disk access, which we intend to avoid until it is absolutely necessary.

6. We also propose to extend our platform to RESTful web services and we explain the advantages of doing so. This also implies annotating the WADL description of the RESTful service.

One could argue that a scenario such as the one described in section 1.1 can be implemented in other ways, without semantic concepts behind the scenes. While this is true, let's consider what the semantic interoperability brings us. First note that a doctor analyzing a patient typically describes the symptoms, the findings and the diagnosis in his language, using the local terminology and moreover, using names of medicines that are found in that particular country/region. Typically the doctor dictates all these and a software program records them in a standard document, which the doctor is able to correct later if needed. Since a patient is treated by various doctors in various hospitals, potentially in different countries, he/she would have a fragmented medical history, consisting of several documents written in various languages with various terminologies.
It is the job of the doctor's agent to translate such a document to HL7-RIM concepts and to create messages in HL7 v3. Moreover, the agent translates the medicine names to the active substance, which is the same for a particular medicine, regardless the country where it is produced/used. Thus, another doctor's agent, in another country, is able to translate such an HL7 v3 message back to regular text/voice in the language/terminology the doctor understands. With this in place, collecting the patient's medical history in a consistent way is not an issue any more.
Another interesting scenario is when a specialist doctor is helped by his agent: suppose the agent needs a service which accepts as input the patient's EPR and his/her symptoms and outputs the diagnostic; such a service can be used by anybody at any time; without it

the agent would need to keep a huge database with symptoms and diagnostics and make logical inferences (potentially using artificial intelligence concepts, etc.). It is thus important to have such services and be able to invoke them.

C.3. Perspectives for Future Developments

So far we have implemented two prototypes – one showing how JADE agents interact with each other and with web services (section 2.3) and a second one demonstrating a semantic search engine (section 5.4). For the latter we've also envisioned a real-world solution, describing the most optimal organization of the data structures in the repository and in memory, so that the number of disk accesses is minimized. Our prototype uses BerkeleyDB as a repository, but the data structures kept are not optimized in any way.
We need to implement the proposed data structures and measure the performance of the search engine. We'll also have to ensure the high scalability of the search engine. Since the searcher is implemented as a web service, this implies using clustering, load balancing and caching strategies. Also security is an important aspect, not dealt with by our prototype.
We've proposed to also consider the WADL descriptions of RESTful web services and annotate them semantically, so we also leverage the full semantic search capabilities for RESTful web services, just as we do with SAWSDL annotations for regular web services. The research in the area of RESTful web services is very limited, so a lot of work is expected here too.

While prototyping our proposed semantic search engine, we've noticed that very few SAWSDL annotated services exist in various online repositories. Consequently, we needed to feed our crawlers manually with SAWSDL descriptions. If the search engine is to be used in the near future, many SAWSDL descriptions are expected to exist in dedicated repositories. A lot of effort is needed here:

- either to create new SAWSDL repositories;
- or to extend existing WSDL repositories by annotating the descriptions according to the SAWSDL standard (and thus empowering legacy systems with semantic capabilities).

ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation, Durability – a set of properties that guarantee that database transactions are processed reliably
DF	Directory Facilitator (see also FIPA) - a mandatory component of an AP that provides a yellow pages directory service to agents
DICOM	Digital Imaging and Communications in Medicine – a standard for handling, storing, printing and transmitting medical images
DL	Description Logic
EPR	Electronic Patient Record
FIPA	Foundation for Intelligent Physical Agents
FIPA ACL	FIPA Agent Communication Language (see also FIPA)
HL7	Health Level 7 – the global authority on standards for interoperability of health information technology
HL7-RIM	Health Level 7 – Reference Information Model
ICD-10	International Classification of Diseases and Related Health Problems, 10th Revision
IOPE	Inputs, Outputs, Preconditions and Effects
LOINC	Logical Observation Identifiers Names and Codes – a standardization of terms for all kinds of observations and measurements in the medical domain
MAS	Multiagent systems
OWL	Web Ontology Language
OWL-S	An ontology for describing semantic web services; OWL-S is built on top of OWL
RDF	Resource Description Framework
RDFS	RDF Schema
REST	Representational State Transfer – an architecture style for distributed systems, such as the World Wide Web
SAWSDL	Semantic Annotations for WSDL (see also WSDL)
SNOMED-CT	Systematized Nomenclature of Medicine - Clinical Terms: a comprehensive clinical terminology
SOAP	Simple Object Access Protocol – the protocol used for exchanging structured information among web services
SPARQL	SPARQL Protocol and RDF Query Language
UDDI	Universal Description, Discovery and Integration
WADL	Web Application Description Language
WSDL	Web Services Description Language

APPENDIX

Crawler

The code snippet below shows almost the entire prototype crawler, written with Crawler4J version 1.5. Our crawler implements Crawler4J's WebCrawler class. Note the main methods: shouldVisit(), which decides what pages should be visited by the crawler and visit(), which performs the needed actions when visiting a certain page.
This version of Crawler4J has some bugs. Note for example that the following lines shouldn't be needed, however, without them, the crawling process doesn't seem to kick off.

```
if (href.contains("service")) {
    return true;
}
```

In the visit() method we call persistWSDL() in order to parse the current WSDL file, extract its Inputs and Outputs (the SAWSDL annotations) and persist them to the repository. Basically this is our simple indexer, since with this method we create the input and output indexes. The indexes for preconditions and effects are omitted for brevity.

```
public class WsdlCrawler extends WebCrawler {

  private SortedSet<String> visitedLinks = new TreeSet<String>();

  private static ServiceRepository repository;
  private static IOPEViews sv;

  static {
    try {
      repository = new ServiceRepository(
          "C://PhD//WSDLrepository//ServiceRepository");
      sv = new IOPEViews(repository);
    } catch (Exception e) {
      // ….
    }
  }

  private Pattern filters = Pattern.compile
              (".*(\\.(css|js|bmp|gif|jpe?g" +
              "|png|tiff?|mid|mp2|mp3|mp4" +
```

```
            "|wav|avi|mov|mpeg|ram|m4v|pdf" +
            "|rm|smil|wmv|swf|wma|zip|rar|gz))$");

    public boolean shouldVisit(WebURL url) {
      String href = url.getURL().toLowerCase();
      if (filters.matcher(href).matches()) {
        return false;
      }

      if (href.contains("service")) {
        return true;
      }

      if (href.endsWith("?wsdl")) {
        return true;
      }

      return false;
    }

    public void visit(Page page) {

      int docid = page.getWebURL().getDocid();
      String url = page.getWebURL().getURL();
      String text = page.getText();
      String html = page.getHTML();
      // get the links from the current page
      List<WebURL> links = page.getURLs();

      if (url.toLowerCase().endsWith("?wsdl")) {
          visitedLinks.add(url);
          persistWSDL(url, html);
      }

      // explore each link from the current page
      if (links != null)
        for (WebURL link : links) {
          if (link.getURL().toLowerCase().endsWith("?wsdl")) {
            try {
              if(
                !visitedLinks.contains(link.getURL())
              ) {
                String htmlLink = URLReader.getHTML(link.getURL());
                visitedLinks.add(link.getURL());
                persistWSDL(link.getURL(), htmlLink);
              }
            } catch (Exception e) {

            }
```

```
        }
      }
    }

    /**
     * With this method the crawler plays the role of an indexer too.
     * Here we create the mapping between Input/Output and the list
     * of the documents (WSDLs) where it appears.
     *
     * @param url
     * @param html
     */
    private void persistWSDL(String url, String html) {
      try {
        try {
          ParserSAWSDL parser = new ParserSAWSDL(url);
          parser.extractInputsAndOutputs();
          // add the inputs and outputs with the
          // corresponding URL to the repository
          List<Input> inputs = parser.getInputs();
          for (Input input : inputs) {
            sv.addToInput(input.getKey(), url);
          }

          List<Output> outputs = parser.getOutputs();
          for (Output output : outputs) {
            sv.addToOutput(output.getKey(), url);
          }

        } catch (Exception e) {
          //….
        }


        // add the WSDL file to the repository
        //(even if it's not WSDL 2.0)
        repository.addFileToDB(url, html.getBytes());
      } catch (Exception e) {
        // ….
      }
    }
  }
```

Repository

In the snippet above we show how the crawler writes the inputs and outputs to the database, together with the WSDL documents where they are found. The *ServiceRepository* above is a BerkeleyDB repository and we provide a wrapper to it, *IOPEViews*, to facilitate the persistence of the Inputs, Outputs, Preconditions and Effects (IOPEs).
The snippet below shows an excerpt of the IOPEViews class. The crawler calls the methods `addToInput` and `addToOutput` in order to store the mappings between an input/output and the docIDs of the document where it occurs.

```
package repository;

import java.util.LinkedList;
import java.util.List;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.ClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.collections.StoredEntrySet;
import com.sleepycat.collections.StoredMap;

/**
 * This is the wrapper around the databases used to store
 * Inputs/Outputs/Preconditions/Effects.
 * It offers a simple API to store/retrieve data.
 *
 */
public class IOPEViews {

    private StoredMap inputsMap, outputsMap;

    public IOPEViews(ServiceRepository db) {
      // open the database used to store class information
      ClassCatalog catalog = db.getClassCatalog();

      // the entry bindings are used to store key-value pairs
      EntryBinding documentkeyBinding =
            new SerialBinding(catalog, String.class);
        EntryBinding documentdataBinding =
            new SerialBinding(catalog, DocumentData.class);

        // the map used to store Inputs
        inputsMap =
```

```
            new StoredMap(db.getInputsDatabase(), documentkeyBinding,
            documentdataBinding, true);
        // the map used to store Outputs
        outputsMap =
            new StoredMap(db.getOutputsDatabase(), documentkeyBinding,
            documentdataBinding, true);

    }

    public final StoredMap getInputsMap() {
      return inputsMap;
    }

    public final StoredMap getOutputsMap() {
      return outputsMap;
    }

    public final StoredEntrySet getInputsEntrySet() {
      return (StoredEntrySet)inputsMap.entrySet();
    }

    /**
     * Add a key-value pair to the Inputs database.
     *
     * The key is a composition of several fields:
     * the concatenation of SAWSDL annotations, if they exists,
     * followed by the type; in case the type
     * is "xsd:simpleType" or "xsd:complexType", a serialization of the
     * whole type is used (the sub-types are ordered alphabetically, to
     * make sure the same type can be used with different order of its
     * fields).
     *
     * The value is the list of the docIDs that contain this input.
     *
     * @param input
     * @param docs
     */
    public void addInput(String input, List<String> docs) {
      getInputsMap().put(input, new DocumentData(docs));
    }

    /**
     * Add a docID to an input. If the input is in the repository, add
     * the docID to the existing list.
     * If not, add the input with the docID to the repository.
     *
     * @param input
     * @param docID
     */
```

```
public void addToInput(String input, String docID) {
  DocumentData data = (DocumentData)getInputsMap().get(input);
  if (data != null) {
        if (!data.getDocIDs().contains(docID)) {
              // add the docID if not already there
              data.addDocID(docID);
              getInputsMap().put(input, data);
        }
  } else {
        // input not added yet, add it
        List<String> docs = new LinkedList<String>();
        docs.add(docID);
        DocumentData newData = new DocumentData(docs);
        getInputsMap().put(input, newData);
  }
}

/**
 * Get the value for the given key from the Inputs database.
 *
 * @param input
 * @return
 */
public List<String> getInput(String input) {
  if (
    (getInputsMap() != null) &&
    ((DocumentData)getInputsMap().get(input) != null)
  ) {
        return
          ((DocumentData)getInputsMap().get(input)).getDocIDs();
  }
  return null;
}

/**
 * Add a key-value pair to the Outputs database.
 * Take a look above for the discussion related to the key and
 * value.
 *
 * @param output
 * @param docs
 */
public void addOutput(String output, List<String> docs) {
  getOutputsMap().put(output, new DocumentData(docs));
}

/**
 * Add a docID to an existing output.
 *
```

```
     * @param output
     * @param docID
     */
    public void addToOutput(String output, String docID) {
      DocumentData data = (DocumentData)getOutputsMap().get(output);
      if (data != null) {
            data.addDocID(docID);
            getOutputsMap().put(output, data);
      }
    }

    /**
     * Get the value for the given key from the Outputs database.
     *
     * @param output
     * @return
     */
    public List<String> getOutput(String output) {
      return ((DocumentData)getOutputsMap().get(output)).getDocIDs();
    }

}
```

SAWSDL Parser

The SAWSDL parser is one of the most complex components in the implementation of the search engine. The complexity stems from the intricate format of the WSDL file and the desire of Apache Woden tool (we use version 1.0M9 for parsing) to support all possible elements in a generic way. Below we show an excerpt from a method used to extract the “xsd:element” tags from the given SAWSDL file.

```
/**
       * Extract all element types from the WSDL, i.e. "xsd:element".
It fills in the hashmap "elements".
       */
      private void extractElements() {
            ElementDeclaration[] decls =
                  descElem.getElementDeclarations();
            for (ElementDeclaration ed : decls) {
                  ElementType elementType = new ElementType();

                  elementType.setName(
                  ((ElementDeclarationImpl)ed).getName().getLocalPart()
                  );

                  elementType.setNamespace(
```

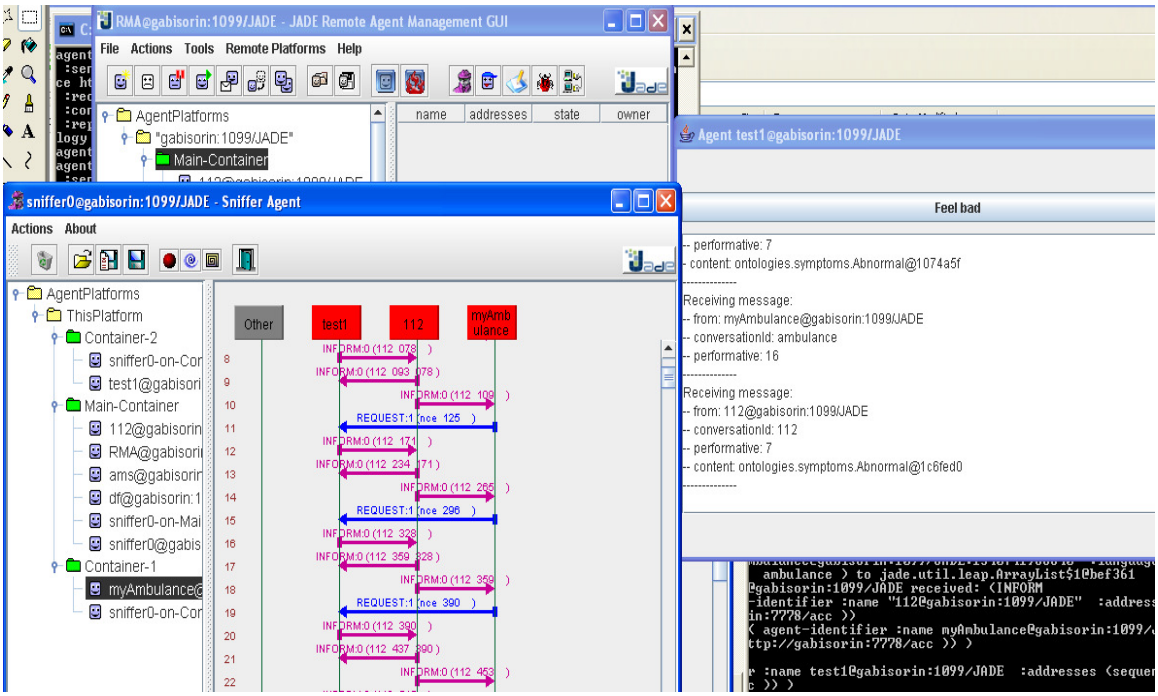
```
      ((ElementDeclarationImpl)ed).getName().
      getNamespaceURI());
if((XmlSchemaElement)((ElementDeclarationImpl)ed).
      getContent() != null) {
      if (((XmlSchemaElement)
      ((ElementDeclarationImpl)ed).getContent()).
      getSchemaTypeName() != null) {
      // the type of the element, e.g. "string" or
      // any other primitive type; the type can also
      // be complexType or simpleType, see below
          elementType.setType(
            ((XmlSchemaElement)(
            (ElementDeclarationImpl)ed).getContent())
            .getSchemaTypeName().getLocalPart()
            );
      }
}
// complex type inside the element
// e.g.:
// <xs:element name="OrderRequest">
//          <xs:complexType> ....
//          </xs:complexType>
// </xs:element>
XmlSchemaType schema =
      ((XmlSchemaElement)((ElementDeclarationImpl)ed)
      .getContent()).getSchemaType();
if (schema instanceof XmlSchemaComplexType) {
      XmlSchemaObjectCollection items =
            ((XmlSchemaSequence)
            ((XmlSchemaComplexType)schema).getParticl
            e()).getItems();
      Iterator iter = items.getIterator();
      ComplexType complexType = new ComplexType();
      HashMap<String, ElementType> elements =
            new HashMap<String, ElementType>();
      while (iter.hasNext()) {
            XmlSchemaElement elem =
                  (XmlSchemaElement)iter.next();
            String type =
            elem.getSchemaTypeName().getLocalPart();
            ElementType element = new ElementType();
            element.setName(elem.getName());
            element.setType(type);

            // check the eventual model references of
            // this sub-element
            ….
      }
}
```

```
            }
        }
```

Agents with JADE

We use the JADE platform [1] to implement our prototype software agents. The screenshot below shows three of the agents communicating among each other: “test1” is the patient’s personal agent, “112” is the emergency service agent and “myAmbulance” is the ambulance agent. The *sniffer agent* provided by JADE shows how the agents are communicating: the patient’s agent triggers the whole process (also shown in the window on the right, where the “Feel bad” button simulates the patient’s condition), informing the emergency agent about the patient’s state; “112” agent acknowledges the receipt and informs the ambulance agent to contact the patient’s agent. The blue arrow from the “myAmbulance” agent to “112” shows the request for information, based on which the ambulance agent gets the patient’s physical location, his symptoms and his medical record.

This basically follows the scenario described in section 1.1.



The snippet below shows how a personal agent can be implemented in JADE. A JADE agent implements so-called “behaviors”. In our case, the personal agent implements Inform112Behaviour and SendEPRBehaviour – see method `setup()`. The former alerts the emergency service when the patient’s condition deteriorates and the latter sends the patient’s medical record to whoever needs it. For the sake of simplicity we don’t show

the implementations of these behaviors here, but note that all communication with other agents and services happens inside these behaviors.
We also show the method `changeHealthState()`, where we show how the inference Jena can be employed: when the blood pressure becomes 90 and the pulse 110, based on the rules in the file *myrules.rules*, the inference engine decides the patient's condition is serious enough to alert the emergency service.

```
public class PersonalAgent extends Agent {

	private ContentManager manager =
		(ContentManager)getContentManager();
	private Codec codec = new SLCodec();
	private Ontology symptomsOntology =
		SymptomsOntology.getInstance();
	private ThreadedBehaviourFactory tbf =
		new ThreadedBehaviourFactory();
	private Patient patient;
	private final static boolean HEALTH_STATE_OK = true;
	private boolean healthState = HEALTH_STATE_OK;
	static final String artNS = "http://www.whatever.com/#";
	private PersonalAgentUI ui;

	protected void setup() {
		// the patient it represents
		patient = new Patient();
		patient.setName("John Johnson");
		patient.setId("1234567890");
		patient.setAge(78);
		// location is determined on the fly

		manager.registerLanguage(codec);
		manager.registerOntology(symptomsOntology);

		Behaviour b1 = new Inform112Behaviour();
		Behaviour wrapped1 = tbf.wrap(b1);
		Behaviour b2 = new SendEPRBehaviour();
		ParallelBehaviour pb = new ParallelBehaviour(this,
			ParallelBehaviour.WHEN_ALL);
		pb.addSubBehaviour(wrapped1);
		pb.addSubBehaviour(b2);
		addBehaviour(pb);

		ui = new PersonalAgentUI(this);
	}

	public void changeHealthState() {
		Model data =
```

```
                FileManager.get().loadModel("file:patient.n3");

            // I add a new statement programatically, which overrides
            // the statement :BloodPressure :val 70 in patient.n3
            Resource bloodPressure =
                  data.createResource(artNS + "BloodPressure");
            Property value = data.createProperty(artNS, "val");
            bloodPressure.addLiteral(value, 90);
            Resource pulse = data.createResource(artNS + "Pulse");
            Property value2 = data.createProperty(artNS, "val");
            pulse.addLiteral(value2, 110);
            List<Rule> rules = Rule.rulesFromURL("myrules.rules");
            GenericRuleReasoner reasoner =
                  new GenericRuleReasoner(rules);
            reasoner.setOWLTranslation(true);// not needed in RDFS case
            reasoner.setTransitiveClosureCaching(true);
            List rules2 = reasoner.getRules();
            InfModel inf = ModelFactory.createInfModel(reasoner, data);
            Resource r = data.getResource(artNS + "Service112");

            QueryExecution qe2 = QueryExecutionFactory.create(
                  "SELECT ?service " +
                  "WHERE { ?service <http://www.whatever.com/#alert>
                  'true' }",
                  inf
            );
            ResultSet rs = qe2.execSelect();
            for (; rs.hasNext() ; ) {
                  QuerySolution binding = rs.nextSolution();
                  System.out.println("Alert service: " +
                  binding.get("service"));
                  // if there is a rule, the agent should alert the
                  // emergency service
                  healthState = !healthState;
            }
      }
}
```

The rules file (*myrules.rules*) used above:

```
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix :     <http://www.whatever.com/#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

[bradycardia: (:Pulse :val ?x), lessThan(?x, 60) -> (:Bradycardia :val
"true")]
```

```
[tachycardia: (:Pulse :val ?x), greaterThan(?x, 100) -> (:Tachycardia
:val "true")]
[call112: (:BloodPressure :val ?x), greaterThan(?x 70), (:Tachycardia
:val "true") -> (:Service112 :alert "true")]
[call112_2: (:BloodPressure :val ?x), lessThan(?x 40), (:Bradycardia
:val "true") -> (:Service112 :alert "true")]
```

REFERENCES

[1] *Fabio Bellifemine, Giovanni Caire, Dominic Greenwood* Developing Multi-Agent Systems with JADE, John Wiley & Sons Ltd, 2007

[2] HL7 – Health Level 7, http://www.hl7.org/

[3] *Orgun, B.; Vu, J.* HL7 ontology and mobile agents for interoperability in heterogeneous medical information systems. Computers in Biology and Medicine 36, 2006, pp 817–836

[4] Kim Platform, http://www.ontotext.com/kim/semanticannotation.html

[5] GoNTogle, http://web.imis.athena-innovation.gr/projects/gontogle/

[6] SAWSDL – Semantic Annotations for WSDL and XML Schema, http://www.w3.org/TR/sawsdl/

[7] *van den Bosch, A.T., Menken M.R., van Breukelen, M., van Katwijk, R.*, A Test Bed for Multi-Agent Systems and Road Traffic Management, 15th Belgian-Netherlands Conference on Artificial Intelligence, 2003, pp. 43–50

[8] WSML, http://www.wsmo.org/wsml/

[9] WSDL2OWL-S, http://www.semwebcentral.org/frs/?group_id=30

[10] Protégé Ontology Editor, http://protege.stanford.edu/

[11] FIPA – Foundation for Intelligent Physical Agents, http://www.fipa.org/

[12] FIPA-RDF, http://www.fipa.org/specs/fipa00011/XC00011B.html

[13] FIPA Ontology Service Specification, http://www.fipa.org/specs/fipa00086/XC00086D.html

[14] FIPA DF, http://www.fipa.org/specs/fipa00023/XC00023H.html

[15] LOINC browser: http://search.loinc.org

[16] SNOMED browser: http://terminology.vetmed.vt.edu/SCT/menu.cfm

[17] FIPA Interaction protocols: http://www.fipa.org/repository/ips.php3

[18] WADL, http://www.w3.org/Submission/wadl/

[19] *Srinivasan N., Paolucci M., Sycara K. P.* An Efficient Algorithm for OWL-S based Semantic Search in UDDI, Semantic Web Services and Web Processing Composition, Lecture Notes in Computer Science, 2005, Volume 3387/2005, pp. 96-110

[20] *Yassin Chabeb, Samir Tata, Alain Ozanne* YASA-M: A Semantic Web Service Matchmaker: AINA '10 Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications, pp. 966-973

[21] SOAP based Message Transport for the JADE Multiagent Platform, http://www.aamas-conference.org/Proceedings/aamas09/pdf/03_Industrial_Track/ 19_108_IT.pdf

[22] AgentOWL - Agents with OWL ontology models using JADE agent system and Jena, http://agentowl.sf.net/

[23] *Uche Ogbuji* Using RDF with SOAP – Beyond Remote Procedure Calls, http://www.ibm.com/developerworks/webservices/library/ws-soaprdf/

[24] *Bernhard Schiemann, Ulf Schreiber* OWL DL as a FIPA ACL content language: Proceedings of the Workshop on Formal Ontology for Communicating Agents (FOCA), 18th European Summer School of Language, Logic and Information, 2006, pages 73–80

[25] FIPA SL Content Language Specification, http://www.fipa.org/specs/fipa00008/ SC00008I.html#_Toc26668858

[26] UDDI Specifications, http://www.oasis-open.org/committees/uddi-spec/doc/ tcspecs.htm

[27] Lucas API = SAWSDL to UDDI mapping, Thales technical note, 29 May 2006

[28] *Dimitrios Kourtesis, Iraklis Paraskakis* Combining SAWSDL, OWL-DL and UDDI for Semantically Enhanced Web Service Discovery: ESWC'08 Proceedings of the 5th European semantic web conference on the semantic web: research and applications, pp. 614-628

[29] *Massimo Paolucci, Takahiro Kawamura, Terry Payne, Katia Sycara* Importing the Semantic Web in UDDI, Lecture Notes in Computer Science, 2002, Volume 2512/2002, pp. 815-821

[30] Daniela *Grigori, Juan Carlos Corrales, Mokrane Bouzeghoub* Behavioral matchmaking for service retrieval: ICWS '06 Proceedings of the IEEE International Conference on Web Services

[31] *Naga Krishna Gollapudi* Semcloud: Using Semantics to Improve Automation on a Cloud, Master Thesis, Acharya Nagarjuna University, 2006

[32] *Sergey Brin, Lawrence Page* The Anatomy of a Large-Scale Hypertextual Web Search Engine, Seventh International World-Wide Web Conference (WWW 1998), April 14-18, 1998, Brisbane, Australia

[33] jUDDI, http://juddi.apache.org/

[34] *Pensri Pukkasenung, Peraphon Sophatsathit, Chidchanok Lursinsap* An Efficient Semantic Web Service Discovery Using Hybrid Matching, Lecture Notes in Computer Science, 2010, Volume 6162/2010, 110-119, DOI: 10.1007/978-3-642-14415-8_8

[35] Jena, Java framework for building Semantic Web applications: http://incubator.apache.org/jena/

[36] *Jorge Cardoso, John A. Miller, Savitha Emani* Web Services Discovery Utilizing Semantically Annotated WSDL, Reasoning Web, Springer-Verlag Berlin, Heidelberg, 2008

[37] Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl

[38] HDFS, Hadoop Distributed File System: http://hadoop.apache.org/hdfs/

[39] Crawler4J, an open source crawler: http://code.google.com/p/crawler4j/

[40] *Nash J.* The Essential John Nash, Princeton University Press, 2002

[41] *Euzenat J., Shvaiko P.* Ontology Matching, Springer-Verlag, Berlin, Heidelberg, 2007

[42] *Winkler, W. E.* String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage, Proceedings of the Section on Survey Research Methods (American Statistical Association), pp. 354–359

[43] *Madhavan J., Bernstein P., Rahm E.* Generic Schema Matching with Cupid, Proceedings of the 27th International Conference on Very Large Databases, Rome, Italy, 2001

[44] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/

[45] DAML+OIL, http://www.daml.org/2001/03/daml+oil-index

[46] OWL: Web Ontology Language, http://www.w3.org/TR/owl-features/

[47] RDF: Resource Description Framework, http://www.w3.org/RDF/

[48] RDF Schema, http://www.w3.org/TR/rdf-schema/

[49] Web Service Semantics – WSDL-S, http://www.w3.org/Submission/WSDL-S/

[50] SOAP, http://www.w3.org/TR/soap/

[51] *Fielding R.T.* Architectural Styles and the Design of Network-based Software Architectures, Doctoral Dissertation, University of California, Irvine, 2000

[52] Apple's Siri, http://www.apple.com/iphone/features/siri.html

[53] *Brian Roemmele* Why is Siri important ? http://www.quora.com/Apple-Products-and-Services/Why-is-Siri-important?q=Why+sir

[54] *Alexander Pokahr, Lars Braubach, Winfried Lamersdorf* Jadex: Implementing a BDI-Infrastructure for JADE Agents, Published in: EXP – in search of innovation (Special Issue on JADE), 2003, pp. 76-85

[55] ICD-10, International Classification of Diseases and Related Health Problems, 10th Revision, http://apps.who.int/classifications/icd10/browse/2010/en

[56] InfinityDB, Infinity Database Engine, http://boilerbay.com/infinitydb/

[57] *Himanshu Yadava* The Berkeley DB Book, Apress, 2007

[58] *Rafal Ku* Apache Solr 3.1 Cookbook, Packt Publishing, 2011

[59] *Michael McCandless, Erik Hatcher, Otis Gospodnetic* Lucene in Action, Second Edition, Manning Publications, 2010

[60] SAWSDL4J, http://lsdis.cs.uga.edu/projects/meteor-s/opensource/sawsdl4j/

[61] Apache Woden, http://ws.apache.org/woden/

[62] FIPA-ACL Specification, http://www.fipa.org/specs/fipa00061/index.html

[63] SPARQL Query Language for RDF, W3C Recommendation, 15 January 2008, http://www.w3.org/TR/rdf-sparql-query/

[64] Notation3 (N3): A readable RDF syntax, W3C Team Submission, 28 March 2011, http://www.w3.org/TeamSubmission/n3/

[65] RDFa Core 1.1, W3C Recommendation, 7 June 2012, http://www.w3.org/TR/rdfa-syntax/