ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

TITOLO DELLA TESI

Semantic coordination tuple centres for eHealth Systems

Tesi in

Sistemi Multi-Agente LS

Relatore
Omicini Andrea

Presentata da
Olivieri Alex Carmine

Sessione terza

Anno Accademico 2010/2011

Contents

List of Figures

List of Tables

Abstract

In this thesis starting from the syntactic limits present in the exchange of Electronic Patient Records (EHRs), we aim to create a framework that provides the exchange of semantic information.

The framework created is called Semantic TuCSoN and it is an extension of TuCSoN (Tuple Centres Spread over the Network) with the additional feature of allowing to exchange semantic information and by reasoning about them.

Semanitc TuCSoN is afterwards modelled on the eHealth context by defining the Agents and coordination policies aimed at the exchange of EHRs, by providing to it of the data persistence, required to work in a real environment.

Finally we do some test, based on the scenarios of interest, in order to evaluate its subsequent use in this area.

This thesis is dedicated to all those people who,
despite having no special skills,
thanks to the perseverance and sacrifice
can overcome their limitations and difficulties that life reserves,
succeding to achieve goals that seemed unattainable.

Chapter 1

Introduction

In the context of Multi-agent Systems (MAS), coordination models and languages based on syntactic mechanisms pose serious limitations on interoperability, which is required in open and dynamic applications (e.g. pervasive applications), where knowledge descriptions often involve different representations of conceptually equivalent concepts.

To overcome the limitations caused by syntactic matching mechanisms, this research aims using a coordination environment based on tuples centres. Tuple Centres provide a coordination mechanism where agents can exchange information, in totally spacial and temporaly decoupling, providing it of ability to work using semantic matching.

Additional Tuple Centre's features are created by exploiting the Description Logics, and their reasoning abilities, which allows us to infer new knowledge.

The resulting framework is applied to the scenario for exchange of electronic health record beetwen health organisations. This scenario has a high heterogeneity of data, which needs a semantic representation to enable the data exchange.

In this real world scenario, we meet challenges such as the scalability, dinamicity and flexibility typical of large environment.

1.1 Motivation

There are two main motivations in this thesis, the first is to extend the agent coordination framework semantic TuCSoN, as presented in [30], to the context of Swiss eHealth, to exchange Electronic Health Record (EHR). The second is to analise and model the concept of Community as defined in IHE [1] and insert it inside the context of tuple centre. In particular we focus on enabling EHR exchange using the semantic component of TuCSoN. Currently SemanticTuCSoN is a model where is possible to insert and retrieve semantic information from a model based on an Ontology, but it has some limits; firstly it does not allow to create real semantic tuple centres, with all properties necessary for the proposed scope; secondly it has no mechanisms for making the Knowledge Base persistent.

1.2 Aims & Objectives

In this thesis we aim to:

- modelling the concept of Community, and use the tuple centres as container of its informations;
- propose a framework that manages the concept of community as described on IHE and the exchange of information among them;
- studying a model that permits to have the informations contained in the community persistent over time;
- evaluate the proposed solution within the motivating case study;

Given this set of general aims, we want to achieve them by means of the following specific objectives:

- permit the creation of semantic tuple centres leaving unchanged the possibility to work on TuCSoN with the classic syntactic tuple centres;

[1] `http://www.ihe.net/`

- define the persistence model for the semantic knowledge base and implementing it;
- extend the framework so that it permits to manage the concept of community as described on IHE;
- extend Semantic TuCSoN equipping it of the possibility to exchange informations among communities, following the guidelines of IHE;
- testing the performance of the resulting system on the basis of sceneries identified in the motivating case study;

Specifically, we will provide an extension of semantic TuCSoN which can totally work in the eHealth context that we are modelling.

1.3 Contribution, Scope and Significance

The contribution of this thesis can be summarized by the following points:

- provide the possibility to create semantic tuple centres, which contain an ontology, so the agents can ask and decide if this is the tuple centre searched or not;
- model the community concept applied to a tuple centre concept in a way that respects all guide lines dictated by IHE;
- identify and create the agents nedeed to manage all the concepts regarding a community, the relationships between them and their exchange of informations;
- finally test the model created and analise the performance.

The significance of this work is to use the semantic techniques already existing in order to provide a tuple centre system of the possibility to contain and to exchange semantic informations. This framework, called Semantic TuCSoN is applied in the context of eHealth, and it must follow the constraints and the directives imposed by it.

Finally, we do a evaluation of the resulting framework in order to decide if it is the ideal solution to reach the main objective proposed.

1.4 Overview of the Thesis

In this thesis, we assume that the reader is familiar with the concept of agent and MAS. We also assume that the reader has a background in object oriented programming. The thesis consists of three parts. In the **Background** part we discuss about the relevant concepts that are necessary to create the Semantic TuCSoN framework for eHealth context. In the **Framework** part we model the Semantic Health Coordination Framework and implement Semantic TuCSoN so as to adapt to it. Finally the **Evaluation & Conclusions** part we evaluate the system and chart out directions for future work.

Part I: Background
This part is composed of Chapter 2, Chapter 3 and Chapter 4. Chapter 2 introduces to the eHealth context, focussing the interest on the exchange of information, precisely on Electronic Health Records (EHRs), by showing the strategies already existing to the exchange, and focussing on IHE initiative. After are showed the limits of the IHE profiles regarding the exchange of EHR, and is proposed a solution to resolve these limitations. Chapter 3 shows what is the Knowledge Representation, which are its origins, and introduces the Description Logics as formalism to overcome their lack of formal semantic representation. Finally is introduced the Ontoloy Web Language and the Description Logic $\mathcal{SHOIN}(D)$ on which it is based. Chapter 4 introduces TuCSoN as framework to information exchange, and explains the peculiarities for which it is chosen. Are also explained the changes that TuCSoN needs in order to be adapted to the eHealth context.

Part II: Framework
This part is composed of Chapter 5 and Chapter 6. Chapter 5 models the architecture of the system, by identifying the entities and the interactions need for the exchange of EHR among communities. Chapter 6 implements the architecture resulting by Chapter 5, by creating a version of Semantic TuCSoN framework provided of agents and coordination primitives, in order to represent in each Tuple Centre a community, that can exchange information with other communities.

Part III: Evaluation & Conclusions
This part is composed of Chapter 7 and Chapter 8. In Chapter 7 are performed two tests based on Motivating Scenario, to verify the performance

of the framework created. In Chapter 8 the results of the evaluation performed in Chapter 7 are used in order to draw conclusions on Semantic TuCSoN. Finally are proposed some interesting future work.

Part I

Background

Chapter 2

eHealth

"eHealth is an emerging field in the intersection of medical informatics, public health and business, referring to health services and information delivered or enhanced through the Internet and related technologies. In a broader sense, the term characterizes not only a technical development, but also a state-of-mind, a way of thinking, an attitude, and a commitment for networked, global thinking, to improve health care locally, regionally, and worldwide by using information and communication technology." [17]. If used in an appropriate manner, the tools and services which work on the eHealth field provide efficient healthcare services for all. eHealth works on the interactions between patients and health-service providers, institution-to-institution transmission of data, or peer-to-peer communication between patients and/or health professionals. This includes various systems like health information networks, electronic health records, health portals, wearable and portable systems which communicate, telemedicine services, and many other ICT-based tools assisting disease prevention, diagnosis, treatment, health monitoring and lifestyle management.

In this chapter we introduce the eHealth context, with particular regard to the exchange of particular informations. The chapter is organized as follows: in the section 2.1 we introduce a motivating scenario which acts as a base for our work; in section 2.2 we explain what is a Electronic Health Record (EHR), the informations that it can contain and what are the benefits that could lead; in the section 2.3 we show the already existing EHR exchange strategies adapted in the field; in the section 2.4 we focus on the IHE initiative on informations sharing in eHealth

context, and we show its principal profiles; in section 2.5 we introduce the current limitations of IHE regarding data exchange; finally in section 2.6 we summarise how this work relates to this thesis.

2.1 Motivating Scenario

As described in [34], our scenario is based in Switzerland, a federal country divided into 26 counties called cantons. The health system of Switzerland is a combination of public (i.e. hospitals) and private systems (i.e. doctors in private clinics) and health conditions can be treated in any of the competent healthcare providers. The Swiss Government has recently recommended the adoption of IHE profiles to achieve interoperability. The first pilot deployments have just been released, such as the eToile project [19] in Geneva.

In this scenario, Mrs Roux who lives in Lausanne, canton Vaud, is spending her holidays in Sierre, canton Valais. She suddenly needs urgent hospital care due to a strong chest pain. She explains to the receiving nurse that she had a heart surgery in the Hospital of Lausanne, which is also her home community and keeps all the updates of Mrs Roux health records. Such a community, does not necessarily have a copy of all the generated documents for Mrs Roux, but it knows where every document is stored.

Unfortunately Ms Roux has not with her the insurance card, which is needed to identify her home community. So, the personel of the hospital of Sierre must search the home community of Ms Roux, based to the informations that she can provide. After finding the informations of her home community (Lausanne), the personel can search for her data. The query returns all the meta-data information held on Mrs Roux (a list describing every document generated for Mrs Roux but not the documents themselves). The doctor who visits Mrs Roux is provided with the discovered information and can consult the documents of interest by retrieving the content from the community where the documents are stored. This is possible because Mrs Roux, through a web application, gave to medical doctors the right to access her medical data. Also, the rights to access Mrs Roux data can be overwritten in case of an emergency, provided that logs are created to monitor doctor's activities.

After such a consultation, the doctor asks for further investigation tests to be carried out in the hospital of Sierre. After Mrs Roux' agreement, the tests together with the doctor's diagnosis are notified to the hospital of Lausanne. The general practitioner (GP) and the cardiologist curing Mrs Roux are both subscribed with the hospital of Lausanne to receive notifications of new generated data on Mrs Roux. Not only the hospital of Lausanne is now aware of this emergency case, and her new treatment, but also her two doctors.

After her return from vacation, the information has been already notified to the hospital of Lausanne, which in turn has notified it to the two interested private clinics where the two doctors work. Next time, when Mrs Roux visits such facilities, her doctor can view the relevant new information generated on Mrs Roux.

2.2 Electronic Health Records (EHRs)

Electronic Health Records (EHRs) refer to the systematic electronic collection of health information data about individual patients or populations [21]. EHRs may include a wide range of data such as demographics, medical history, medication, allergy list, lab results or radiology[1]. An EHR is a record in digital format that is theoretically capable of being shared across different health care settings. In some cases this sharing can occur by way of network-connected enterprise-wide information systems and other information networks or exchanges.

The terms EHR, EPR (electronic patient record) and EMR (electronic medical record) are often used for describing the same concept, although there are some differences among them. In fact an EMR is a patient record created in hospitals and ambulatory environments, which can be a data source for the EHR. An EHR is generated and maintained within an institution, such as a hospital, integrated delivery network, clinic, or physician office, to facility the access of interesting medical informations by the institutions. In this thesis we focur only on EHR, but our considerations remain valid also for EPR and EMR.

The benefits of EHR integration are strongly dependent on the ability of the health care systems to share data among each other. In some cases,

[1] `http://en.wikipedia.org/wiki/Electronic_health_record`

sharing EHR han been enabled through manual configuration of different information systems. However, automatic exchange of EHR among any health institution has yet to come.

2.3 EHR Exchange Strategies

The research activity concerning the information exchange strategies of EHR is one of most important eHealth research activities. Some solutions to EHR exchange have already been developed, but only for local and national contexts. The medical data belonging to an EHR are called fragments, and they can be distributed over different EHR systems. These systems should ensure the interoperability of the EHR fragments for obtaining the benefits proposed. In EHR systems, interoperability should satisfy the requirements of distribution, openness and security [30]. In order to reach this goal, the first approach was the definition of standards for EHR-fragment format and communication. The two most important approaches that ensure the interoperability of EHR fragments are:

- Health Level Seven (HL7[2]):is the global authority on standards for interoperability of health information, which also includes the exchange, management and integration of EHR fragments [14].

- Digital Imaging and Communications in Medicine (DICOM[3]): a standard for handling and transmitting information in medical imaging.

However, these standards are not enough to achieve interoperable health systems. In fact, the result is that EHR systems use different set of format and communication standards, often incompatible, incomplete or involving overlapping scopes, thus breaking the interoperability requirement. As a response to these problems, and, as a complementary step towards the requirements of interoperability among EHR fragments the following standards and initiatives were proposed:

[2] `http://www.hl7.org/`
[3] `http://medical.nema.org/`

- openEHR[4] and CEN EN 13606[5]: standards aiming at facing interoperability among EHR fragments.
- Integrating the Healthcare Enterprise (IHE)[6]: a non-profit initiative founded in 1998 led by professionals of the e-Health industry. The initiative goal is not to develop standards as such, but to select and recommend an appropriate usage of existing standards in order to improve the sharing of information among EHR systems.

We focus on the IHE initiative because its makes a major contribution to the integration the health-care information systems with the purpose to facilitate the exchange of patient information between health-care professionals [24] and enjoys high acceptance due to its practical complement to existing standards such as HL7 CDA [23, 13].

2.4 Integrate the Healtcare Enterprise

IHE is an initiative by healthcare professionals and industry to improve the way computer systems in healthcare share information. In 1997, a consortium of radiologists and information technology experts formed IHE, how an international organisation that focuses on the development of open and global Integration Profiles. IHE creates and operates a process through which interoperability of health care IT systems can be improved. The group gathers case requirements, identifies available standards, and develops technical guidelines that manufacturers can implement. Of all issues are made interoperability showcases which serve to vendors to demonstrate that their products satisfy the interoperability constraints imposed. Because of its limited resources, IHE concentrates on specific projects. It solicits proposals and after surveying its members to better understand their priorities, it chooses areas to focus on[7].
The IHE work is focused in specifying the integration of different clinical and organizational domanis. For each domain, IHE maintains technical frameworks which contain all of the relevant information with regard to a specific domain. The most important part of the technical frameworks

[4] http://www.openehr.org/
[5] http://www.en13606.org/
[6] http://www.ihe.net/
[7] http://en.wikipedia.org/wiki/Integrating_the_Healthcare_Enterprise

is the integration profiles. These profile create specific use cases and communication scenarios based on standards, in order to provide some functionalies.

2.4.1 IHE Profiles

IHE Profiles are defined in terms of IHE Actors and Transactions. They describe the solution to a specific integration problem, and document the system roles, standards and design details to develop systems that cooperate to address that problem. Actors are components that act on informations associated with clinical and operational activities in the enterprise. Transactions are interactions between actors that communicate the required information through standards-based messages. IHE Integration Profiles are a convenient way for implementers and users to be sure they're talking about the same solution without having to restate the many technical details that ensure actual interoperability.

Below we show a list of the main Integration Profiles defined from IHE, which are involved in this thesis, with particular attention to two more important for our context:

- Cross Enterprise Document Sharing (XDS): in the profile that dictates the guidelines to exchange of clinic documentation among companies and sanitary structures of various type (hospitals, medical clinics, privates etc). This profile assumes that each organization belongs to one or more Affinity Domain. Each Affinity Domain consists of sanitary organizations, which subscribe policies and shared an technological infrastructure in order to exchange among them clinical documents. The policies object of the agreements regard aspects of patients identification, access control, obtaining consent for data treatment. They regard also the format, the content, the structure, the organization and the representation of clinical informations. So, it enables a number of health-care delivery organizations belonging to an XDS Affinity Domain to share clinical records in the form of documents as they proceed with their patients care delivery activities; Figure 2.1 shows the XDS profile. At the core of XDS there is the document repository and document registry actors which respectively deal with storing health

documents and storing meta-data about these documents to facilitate their discovery. The data are produced by a document source actor, typically a medical doctor in a hospital. A community may rely on more than one repository to store the produced documents, however, all the meta-data must be stored and submitted within one registry. A document consumer actor can use the meta-data to know which repository contains the documents of interest. A patient identity source actor feeds patient identities to the registry. Since XDS does not resolve document sharing among multiple affinity domains, the Cross-Community Access (XCA) profile specifies how medical data held by other communities can be queried and retrieved.[34]

- Cross Community Access (XCA): this profile defines the concrete guidelines for exchange of documents. In it is defined the concept of Community as set of sanitary structures, which subscribe policies and adopt shared communication protocols in order to shared clinical documentation. Each community is identified by a unique code (Home Community Id). In this profile are added two new actors, Initialing Gateway (it takes in charge of all transaction in exit from the local community and forwarding them towards the other communities) and Responsing Gateway (it manages all entering flows from other communities and it forwards it to actors which belong to local community). In other words, it supports the means to query and retrieve relevant patient medical data held by other community. The big advantage deriving by the XCA use is the interoperability that it permits. The operability is both between communities structured internally second the XDS profile, both between communities that don't adhere to XDS, or developed according legacy principles;

- Patient Identifier Cross-referencing (PIX): supports the cross referencing of patients identifiers from multiple Patients Identifier Domains;

- Patient Demography Query (PDQ): provides ways for multiple distribuited applications to query a patient information server for a list of patients, based on user-defined search criteria, and retrieve a patients demographics information directly into the application.

- Audit Trail and Node Authentication (ATNA): establishes secrity measures which, together with the Security Policy and Procedures, provide patient information confidentiality, data integraty and user accountability;
- Basic Patient Privacy Consents (BPPC): provides a mechanism to record the patient privacy consent(s), and a method for Content Consumers to use to enforce the privacy consent appropriate to the use.

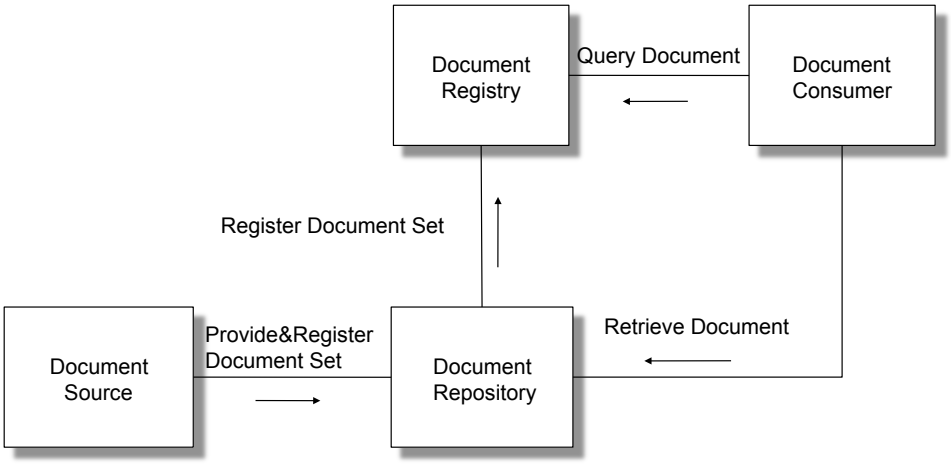

Figure 2.1: The XDS Profiles

These profiles are building blocks upon which one can develop a number of architectures, but they have also some limitations in the EHR exchange that are explained in the next section.

2.5 Current Limitation of IHE for EHR exchange

IHE based systems supports interoperability on interface level so that EHR applications of different vendors can be integrated. The IHE Integration Profiles leave details open for implementation, thus different vendors can create their own systems. This means that the interpretation may lead to slyghtly different systems. IHE addresses this issue

defining a set of IHE compliant test process. The process culminates in a week long interoperability-testing event known as "Conectathon"[25]. Apart from the commercial IHE compliant software, there also exists many open source solutions that define one or more of the above specified integration profiles.

Semantic interoperability is an important aspect to guarantee that data are interpreted identically among EHR applications. IHE has recognized the importance of semantic interoperability and aligned their Integration Profiles with internationally accepted standards for semantic interoperability such as the Clinical Document Architecture (CDA) [23] and common medical dictionaries such as LOINC [27] and SNOMED [9]. The common data format increases the likelihood that identical meaning of data can be preserved during exchange between EHR systems. With currently available technology such as CDA, a very finegrained structure can be applied to clinical data which allows even a languageindependent identical interpretation [36]. CDA however comes with some limitations such as has no ontological basis and lacks cognitive structure [33] (i.e. all CDA based records are acts).[35]

The XDS profile defines a coupling of facilities/enterprises for the purpose of patient-relevant document sharing.

Within the XDS profile, the health-care enterprises that agree to work together for clinical document sharing is called the Affinity Domain. Within an affinity domain, there can be more than one independent repositories. The assumption is that there exists only one registry where the meta-data regarding documents are stored, such data will indicate to the interested document consumer, in which repository reside the documents of interest.

Another assumption is that XDS is not concerned with the management of dynamic information that is not document-oriented, such as allergy lists, medication lists, problem lists, etc [26]. A means to access this information in a structured form and to manage updates to such dynamic clinical information is still missing.

XDS defines document sharing within an XDS affinity domain. As addressed in [24], XDS does not resolve the sharing of patient-relevant health-care information among multiple IHE environments. This would mean to define how a request for EHR would identify other IHEs which have data about a patient, how to identify the patient in the other IHE

and how to request patient information from the IHE.

The Cross-Community Access (XCA) profile already defines a means to query and retrieve EHR held by other communities by specifying a gateway that encapsulates all the incoming and outgoing cross-community communications. The XCA profile defines community a grouping of facilities/enterprises that have agreed to work together using a common set of policies for the purpose of sharing clinical information via an established mechanism. A community is identifiable by a globally unique id called the homeCommunityId. Membership of a facility/enterprise in one community does not preclude it from being a member in another community. Such communities may be XDS Affinity Domains which define document sharing using the XDS profile or any other communities, no matter what their internal sharing structure. Communities can be composed into hierarchical collections of communities which are called meta-communities.

XCA contains a gap in the communication of patient identities as it requires the community who initiates a query towards another community to determine the correct patient identifier of the patient under the authority of the receiving community [24]. It is also assumed that the communities have a pre-established agreements and knowledge of one another.

2.6 Summary

In this chapter we have introduced the eHealth context, and the advantages that it can lead. We started by explaining our motivating scenario. We will carry this throughout the thesis. We focus on the part of eHealth regarding the data exchange, We defined what are the Electronic Health Records (EHRs), and illustrated the existing solutions to EHR exchange that are proposed by IHE. We focused on IHE due to the high worldwide adoption of such standards. We explained the IHE profiles that are involved in the data exchange and pointed out some of the limitations of such profiles. We also describe how these profiles limit the realisation of our motivating scenario.

To Overcome such limitations we eill focus on semantic-agent based solutions as they are regarded as a modular solution on the highly distribuited and heterogeneous scenarios.

Chapter 3

Knowledge Representation

Knowledge Representation is a branch of artificial intelligence that studies the way in which the human reasoning is done, defining symbolisms and languages which permit to formalize the knowledge, to make it understandable to the machines, allowing automatic reasoning (through inference of already present knowledge) in order to get new knowledge. [29]

The research in the field of Knowledge Representation is usually focused on the method to provide a high-level description of the world, that can be actually used to build intelligent applications.[29]

Intelligence: *system's ability to find, based on knowledge acquired and represented explicitly, some implicit consequences.*

This chapter is organized as follows: in the Section 3.1 we briefly describe the origins of Knowledge Representation; in the Section 3.2 we give an explaination of the basis of Description Logics, by introducing the Description Logics $\mathcal{SHOIN}$(D) by describing its characteristics. We also present the Ontology Web Language, the standard of W3C[1] that implements the $\mathcal{SHOIN}$(D). Finally in Section 3.3 we conclude and summarise this Chapter.

3.1 History of Knowledge Representation

The first approaches in the field of Knowledge Representation were in the years seventy, and they were divided in two different tipologies [4]:

[1] http://www.w3.org/

- based on the logic: their origin was the idea that the predicate calculus could had been used to catch the world's facts;
- not based on the logic: the idea was based on structures like networks, and they derive from experiments regarding the capacity of human mind to remember concepts and performs tasks ;

The second approach [29] was the one pursued, but there was a need to equip it with a semantic representation, seeing that each component present in the Knowledge had a different behaviour, despite, it was identical to other. Therefore, to these specifications was given a semantics for representing structures, so that hierarchical structures could be exploited. This could have led to a gain, both in terms of easily of representation, and efficency of reasoning. The basic elements identified were:

- unary predicates: representing sets of individuals;
- binary predicates: representing sets of binary relation between individuals;

Later, it was discovered that this representation did not captured the constraints of interested structures with respect to the logic, but it could only be considered as part of them.

3.2 Description Logics

These were the directly evolutions of structures based on networks , where an additional element called Role was specified in addition to the classical IS-A relation. A Role is a characteristic which permits to represent an other type of relationship. The Role is a link which starts from a node, arrives in another node, and holds a label.

The remaining problem was to define a precise characterization of the present elements and which kinds of relationships could be present (ie define the complete syntax and semantic for all elements involved).

Sintax: was introduced a new abstract language resembling to other formalisms already present. So, first, were introduced two new disjoint alphabets of symbols, used for denoting:

- Atomic Concept: represented by predicates of unary symbols;

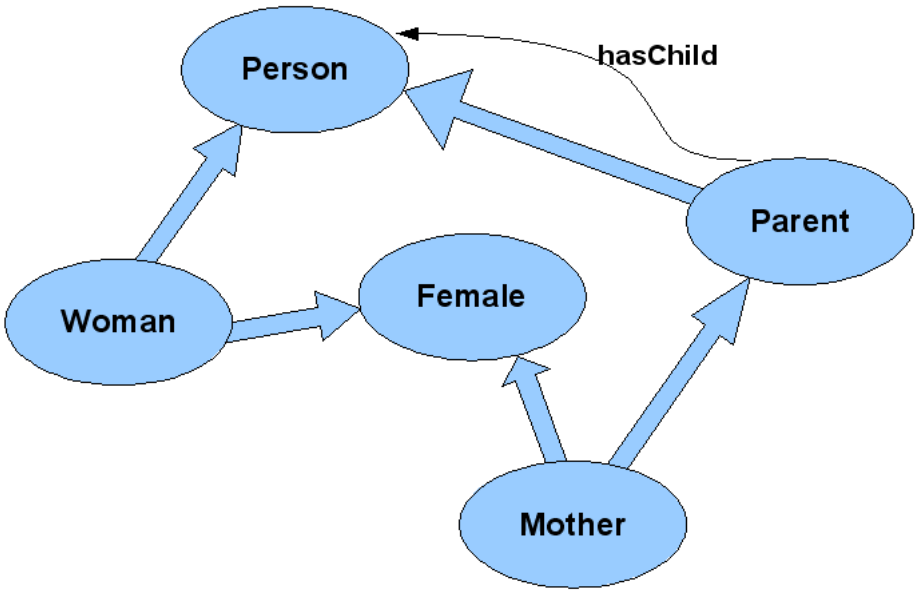

Figure 3.1: Example of a network with Role relation

- Atomic Role:represented by predicates of binary symbols, whom express links between nodes:

Semantic: was given a concrete meaning to these two alphabets, to provide an unique interpretation;

- Concept: is a sets of individuals;
- Role: is a set of relations betweens of individuals:

The union of Concepts and Roles present in a context defines its domain, that is the scheme of the Knowledge Base.

Figure 3.1 shows a network extended with the Role Relationship. In this figure are shown some IS-A relations, which model the hierarchy of the network, and a Role called hasChild. This meaning that a element of set Patient can be connect to a element of the set Person through the relationship hasChild.

3.2.1 Knowledge Base

The main idea was give a precise specification of the functionality provided to the Knowledge Base, particulary to the inference that it could be capable to perform, and indipendently from the single implementation. So an interface *Tell&Ask* was created with the purpose of be able to create the Knowledge Base (Tell), and provide a deduction service (Ask).

A knowledge representation system based on the Description Logics provides the means for install the Knowledge Base, to make reasoning over their contents and to modify them. As showed in figure 3.2 a Knowledge Representation System contains three fundamental elements:

- **Terminological Box (TBox)** [29]: is the structure of the application domain, which defines its vocabolary. It represents the syntax of the domain, and it contains concepts and relationships between them. The TBox is the fixed for a specific domain.
- **Assertional Box (ABox)** [29]: is the population of the application domain, that represents the extensional knowledge. Here are inserted the individuals, each of them with the concept to which belongs and the relationships in which is involved. The ABox is in continued changing.
- **Reasoning** [6], [15]: is the service thank to which is possible to deduce new knowledge

The reasoning as central service is an important characteristic. It allows to infer explicitely new knowledge from that implicitely contained in the Knowledge Base. Some inference patterns already present in other applications are used to understand and structure the world, in order to classify concepts and individuals to permit reasoning.

- concept classification: determines the relation between concepts of a given terminology, so that a subsumption's hierarchy can be build;
- individual classification: determines if a given individual is always an instance of a determinate concept;

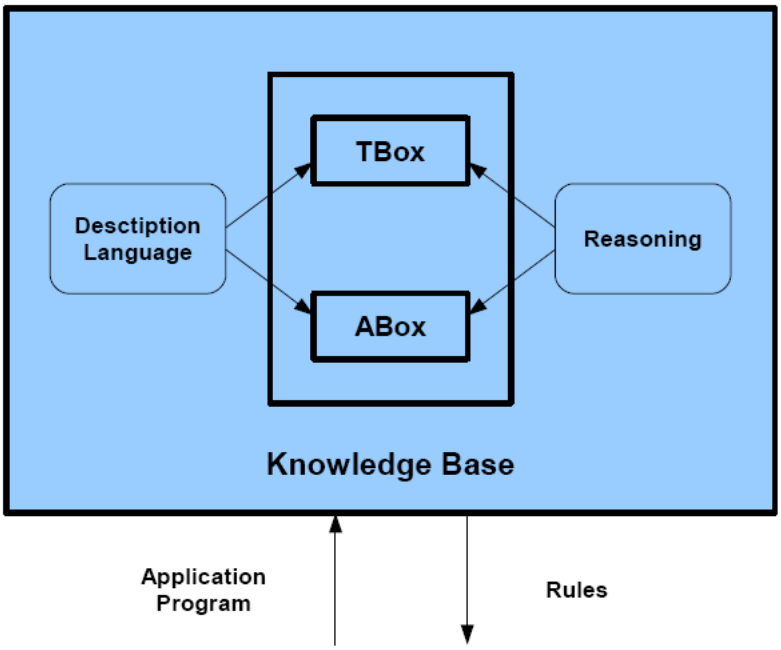

Figure 3.2: Knowledge Representation System based on Description Logics

In order to providing formal reasoning on the Knowledge Base were introduced the Description Logics, which are explained in follow section. The Description Logics should answer to queries in a reasonable time, and primarily they should ensure that the response provided is certain. Unfortunately the response time is not always ensured; decidability and complexity of the inference problems are dependent by the expressive power of the Description Language. In fact, Description Languages too expressive are useful to resolve inference problem with high complexity, but they can be undecidable, while Description Languages with efficent reasoning procedures may not be sufficiently expressive to represent important concepts of an application [1].

The Description Logics descend from network infrastructures, and there are based on three ideas on which have largely shaped their development:

1. the sintactic base blocks are:
    a) atomic concepts (belong to the TBox);
    b) atomic roles (belong to the TBox);

c) individuals (belong to the ABox);

2. implicit knowledge on the concepts and the individuals can be automatically inferred with the help of inference's procedures, of which the main are:

   a) subsumption relations between concepts;

   b) instance relations between individuals and concepts;

3. the expressive power of the language is limited to a little set of atomics constructors, starting by them is possible build complex concepts and roles descriptions;

Above we have said that starting from the elementary descriptions of atomic concepts and atomic roles, is possible build complex descriptions, exploiting concept constructors. The set of the constructors used, identifies the complexity of the language. In Table 3.1 we show the set of concept's constructors for the less expressive language $\mathcal{AL}$. Starting with $\mathcal{AL}$, and adding new constructors is possible obtain languages of any expressivity [5].

Symbol	Description
A	Atomic Concept
$\top$	Universal Concept
$\bot$	Empty Concept
$\neg A$	Atomic Negation
$C \sqcap D$	Intersection
$\forall R.C$	Universal Restriction
$\exists R.C$	Existential Restriction

Table 3.1: $\mathcal{AL}$: Conventional Notation

Abstract notation: A & B for atomic concepts, R for atomic role, C & D for concept descriptions.

The explanation of the Conventional Notation showed in table 3.1 is the following:

- the Atomic Concept is a set of individuals;
- the Universal Concept is the set of all individuals of the universe;

- the Empty Concept is an empty set;
- the Atomic Negation is the set of individuals of the universe that are not present in the Atomic Concept A;
- the Intersection between C and D is the set of individuals present either in C that in D;
- figure 3.3 shows the Universal Restriction, and, it is a set of individuals that participate or not in the relationship called rel1. But if an individual participates in this relationship, this one can be directed solely towards individuals of Class1.
- figure 3.3 shows the Existential Restriction, and, it is a set of individuals that participate in the relationship called rel1. Unlike the Universal Restriction if an individual participates in this relationship, this one can be directed towards individuals of different Classes.

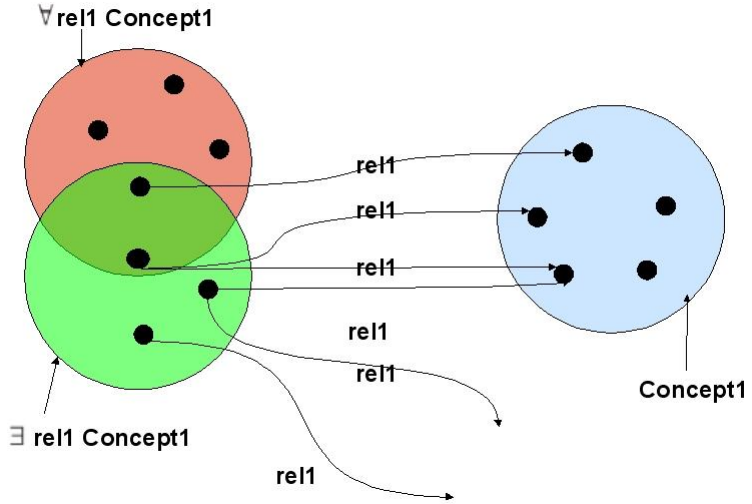

Figure 3.3: Class Restrictions

As introduced previously the TBox is fixed, and it defines the domain to be modeled. It can be compared with the schema of a Database. The ABox, which changes continuously, is the actual instance contained inside the Database. There is a substantial semantic difference between

ABox and a instance of Database, because an instance of a Database represent exactly one interpretation, precisely that are the classes and relations present in the schema that are interpreted from the object in the instance, while an ABox represents some different interpretations, exactly all various domains modeled. This means that absence of information, contrary what happens in the Database where is interpreted as negative information, in the ABox, it is interpreted only as lack of knowledge. In other words the semantics of the ABox is an open-world semantics [28].

3.2.2 $\mathcal{SHOIN}$(D)

This is the Description Logic used in the semantic extension of TuCSoN [30]. It extends the expressive capabilities of the $\mathcal{AL}$ with this characterization:

- $\mathcal{S}$: indicates the possibility of write expressions of logic equivalence and of subsumption using base terms (true & false), terms compounded from operators (and, or & not), quantificators role simple ($\forall R.C$ & $\exists R.C$) and the roles transitivity axiom;
- $\mathcal{H}$: indicates the possibility to make subsumption between roles, in order to establish their hierarchies.
  - subsumption ($R \sqsubseteq S$): the role S subsume the role R;
  - equivalence ($R \equiv S$): the role R is equivalent to the role S. This is not directly expressible, but obtainable by the double subsumption $R \sqsubseteq S$ and $S \sqsubseteq R$;
- $\mathcal{O}$: indicates the possibility to define terms for enumeration;
- $\mathcal{I}$: indicates the possibility to define inverse role, which permits to invert domain and range of a role;
- $\mathcal{N}$: indicates the possibility to define cardinality not quantified;
- D: indicates the possibility to use concrete domains, denoted by atomic terms, like naturals, floats, characters, strings;

For more detail about the Description Logics refer to [6]

3.2.3 Ontology Web Language (OWL)

Since 2044 OWL is a W3C recommended standard, it is a practical realization of a Description Logic known as $\mathcal{SHOIN}(\mathcal{D})$ [22]. Using OWL it is possible to define *classes* (also called *concepts* in the DL literature), *properties*, and *individuals*. An OWL ontology consists of a set of class axioms that specify logical relationships between classes, which constitutes a *TBox* (Terminological Box); a set of property axioms to specify logical relationships between properties, which constitutes a *RBox* (Role Box); and a collection of assertions that describe individuals, which constitutes an *ABox* (Assertional Box). Classes are formal descriptions of sets of objects. Class axioms allow one to specify that subclass ($\sqsubseteq$) or equivalence ($\equiv$) relationships hold between certain classes and the domain and range of a property. Assertions allow one to specify that an individual belongs to a class ($\mathsf{C(a)}$ means that the object denoted by a belong to the class C), and that an individual is (or is not) related to another individual through an object property ($\mathsf{R(b,c)}$ means the object denoted by b is related to the object denoted by c through the property R). As said in section 3.2.1 complex classes can be specified by using Boolean operations on classes: $\mathsf{C} \sqcup \mathsf{D}$ is the union of classes, $\mathsf{C} \sqcap \mathsf{D}$ is the intersection of classes, and $\neg\, \mathsf{C}$ is the complement of class C. Classes can be specified through property restrictions: $\exists\, \mathsf{R.C}$ denotes the set of all objects that are related through property R to some objects belonging to class C at least one; if we want to specify to how many objects an object is related we should write: $\leq \mathsf{n}\mathsf{R}$, $\geq \mathsf{n}\mathsf{R}$, $= \mathsf{n}\mathsf{R}$ where n is any natural number; $\forall\, \mathsf{R.C}$ denotes the set of all objects that are related through R only to objects belonging to class C. To realise the framework, we need to express some preconditions for the $\mathsf{reaction}$ part of the reaction rules. Every precondition can be a class assignment as defined by OWL DL, a query executed thanks to the reasoning services of a reasoning tool or, a Prolog predicate used to construct some specific function[2]. In order to execute a reaction, all its preconditions must be satisfied[3].

Given that there is not an official standard query formalism for OWL DL, in this paper we decide to adopt this one that is inspired from [8]

[2] In what follows we do not give the specification details of such predicates as they are intuitive, and, with a straightforward specification.

[3] As in Prolog, it is possible to specify preconditions to be executed in or by surrounding those with round brackets followed by a semicolon. Section 5.4 makes use of such specification.

and allows to express the queries that are available in the DL Query tab of Protègè[4].

?-C $\sqsubseteq$ D $\Rightarrow$ true/false checks the subclass relationship;
?-C $\equiv$ D $\Rightarrow$ true/false checks class equivalence;
?-C $\Rightarrow$ true/false checks if the class is satisfiable;
?-C(a) $\Rightarrow$ true/false instance checking;
?-C(*) $\Rightarrow$ {a1,....an} retrieval, C can be a complex class.

In our implementation those queries are executed using the Java Jena API [5].

3.3 Summary

In this chapter we introduce the concepts behind the description logics. We explain the $\mathcal{SHOIN}(D)$ which is an extension of $\mathcal{AL}$. In this thesis each tuple centres contains a ontology which model the eHealth context. We show how we model these issues in Chapter 5. In the next chapter we explain how Semantic TuCSoN uses $\mathcal{SHOIN}(D)$ to reason about tuples generated in the system framework.

[4]http://protege.stanford.edu/
[5]http://incubator.apache.org/jena/

Chapter 4

From tuple spaces to tuple centers

In this chapter we introduce TuCSoN, a framework to share information that we use in order to modelling the exchange of EHR among communities. TuCSON is a framework that permits to exchange information on form of Tuple, in a distribuited environment.

This chapter is organized in this mode: in the section 4.1 we explain the context of information's dimension introducing the coodination amond entities; in the section 4.2 we explain the tuple space argument, speaking briefly of the most important, the Linda tuple space, introducing its limitations; in section 4.3 we speak of the tuple centres and how they can overcome the limitation of tuple spaces, and we focus our interest on TuCSoN, explaining its characteristics and introducing its semantic extension. Finally in Section 4.4 we conclude and summarise this Chapter.

4.1 Coordination among entities

In any meta-model of software engineering there are two different dimensions:

- computation: general elaboration of data, that consists, given an algorithm and some input data, obtain some data in output;
- coordination: how the space of the interactions is constrained;

In this thesis we are interested to the argument of data exchange, so, we focus our attention only on the coordination. In the coordination's dimension the first class entities are [7]:

- coordination model: is the glue that binds separate activities into an ensemble;
- coordination language: is the linguistic embodiment of a coordination model;
- coordination middleware: is the execution environment for deployment, execution and manage of the coordination abstractions.

and the meta-model of coordination proposed is:

- coordination entities: the entities which interactions are governed by the model;
- coordination media: the abstractions that allow and regulate the interaction among coordinables;
- coordination laws: the laws that define how the coordination media behaves in response to interaction events;

There are two main types of coordination models. In the first, called *data-driven*, the coordinables cooperate and/or compete among them producing and using information that are presents in some shared space. And the second, called *control-driven*, where the coordinables work generating and reacting to events or signals exchanged in well defined channels. Both approaches have some positive and some negative features. In this thesis, we are interested in the data-driven approach. The context of our work has the following characteristics, ideals to work with a data-driven approach:

- it involves open system environments, where the set of coordinables is not known;
- coordinables are in a environment with a high level of autonomy, and they encapsulate their behaviour;
- the focus is on the exchange of information among coodinables;

- the environment is a concurrent and distributed system;

The most famous type of data-driven model is the Tuple Space model. In the next section we describe this type of model, and we introduce its main features.

4.2 Tuple Space

Linda is the language for the most important data-driven model called Tuple Space model [20]. It is a coordination language which extends the traditional languages, allowing the construction of application on distribuited environment. It has the feature to be indipendent from architecture underlying and by the programming language used. Linda implements an associative memory (Tuple Space) logically divided among all processes present in the application. Linda is characterized from the following entities individuated below:

- tuple space as coordination media;
- tuple as communication language;
- primitive (in, read, out, eval) as coordination language;

In this model the coordination is always left to the coordinables, and this is a big limit, because they have to know the coordination policies of the system, and implement them. This solution is not elegant and it is in conflict with the ortogonality between computation and coordination, in fact, in this model the coordinables have these two behaviours mixed between their.

4.3 Tuple Centre

In order to overcome such a limitation, a number of tuple-based coordination models and languages were proposed. The intent was extend the original tuple-space model, by allowing to its behaviour to be programmable so as to embed coordination policies within the coordination media. The concept of tuple centre was developed starting from [10],

where the notion of programmable coordination medium was explicited for the first time. Tuple centres are tuple spaces whose behaviour can be determined through a specification language. This language defines how a tuple centre should react to incoming/outgoing communication events [31]. Unlike tuple spaces, the behaviour of tuple centres can be programmed with reactions, in order to encapsulate coordination policies within the coordination medium.

4.3.1 TuCSoN

TuCSoN (TupleCentres Spread over the Network) [32] is an infrastructure based on the Tuple Centre concept. It permits communication and coordination among agents, which work by inserting, reading and consuming tuples in this Tuple Centre. The operations needed to work on the tuple centre can be either blocking or non-blocking. TuCSoN has all the features present in a classic Tuple Centre, and it provides a ulterior one, which has already been discussed as extension of Tuple Space. This feature is the possibility of create multiple Tuple Centres in the same node, allowing the subdivision of the work among more Tuple Centres. TuCSoN implements the idea of programmability of Tuple Centres through ReSpecT [2]. ReSpecT is a logic-based language that allow to specification the tuple centre behavior through a set of first-order logic tuples.

4.3.2 ReSpecT

Since ReSpecT is Turing-equivalent any computable coordination policy required by specific application scenario can be in principle embedded inside a ReSpecT Tuple Centre. A specific ReSpecT can associate to any primitive performed on the Tuple Centre, a specific reaction implemented to manage the coordination of the system.

An reaction can be inserted into the tuple centre with a special primitive called `set_spec`, which must contain the ReSpecT code (or must specify the file where the ReSpecT code is). A reaction ReSpecT has this form **reaction (Event, Guard, Body)**, which has the following meaning:

- Event: is the event (primitive in the Tuple Centre) that triggers the reaction;

- Guard: is a couple of parameters whom defining the proprieties of the triggering event;
- Body: is a list of instructions that the reaction must perform;

In the body can be writing a set of elementary non-blocking operations, but not only, in fact TuCSoN is build over tuProlog, and it is integrated with JAVA [12], so is possible use Java and Prolog code inside the reaction. ReSpecT being a coordination language, must check that all is well done for the correct execution, otherwise it must go back. To satisfy this requirement it has a rool-back mechanism, which deletes all actions already done if something go wrong before the end of the reaction.

4.3.3 Semantic TuCSoN

Semantic TuCSoN is an extension of TuCSoN which aims to overcome the syntactic limits of the classical Tuple Centre, and provide a semantic support. The idea has been developed in [30], and the work done had two main targets, strictly dependent to one to another. The first target was the necessity to insert in this framework the possibilty of working with semantics, in order to overcome the syntactic limitations. For this, the ontologies and well known tecnologies for work over them, have been used. In fig 4.1 are showed all components created to permit semantic in TuCSoN.

The correlation between the components and the technologies is the following, as showed in figure 4.2 :

- Ontology Manager: uses the OWL Ontology to obtain and manage an ontology;
- Reasoner: uses Pellet Java Library [1] to permit reasoning on the ontologies;
- Query Generator: uses Jena Library [2] to create Query SPARQL [3], in order to query the ontology;

[1] `clarkparsia.com/pellet/`
[2] `incubator.apache.org/jena/`
[3] `www.w3.org/TR/rdf-sparql-query`

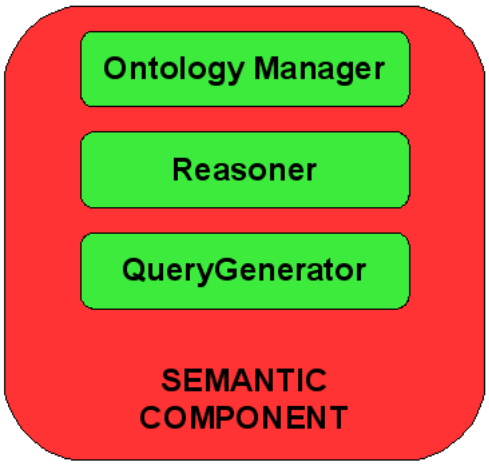

Figure 4.1: Semantic Component

- Semantic Component: is the component, that using the other components, permits to coordinate all work regarding the new semantic part of TuCSoN;

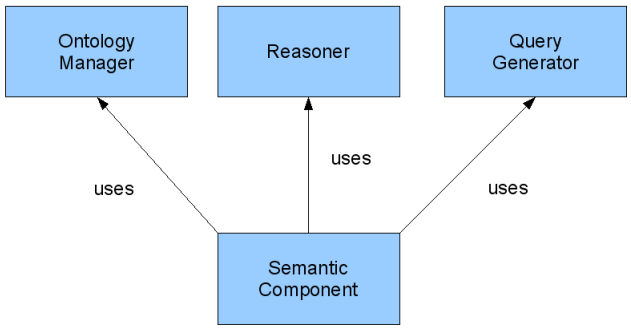

Figure 4.2: Block Scheme

In the second extension, a mechanism was provided to insert and retrieve information exploiting not only the Tuples seen as up at this time, but also in a semantic way. To realize this, the Java class LogicTuple, has been extended in two directions, as showed in fig 4.3. One to permit the inserting semantic information through a new definition of Semantic

Logic Tuple, and the second one to permit to get semantic information through a new definition of Semantic Logic Tuple Template.

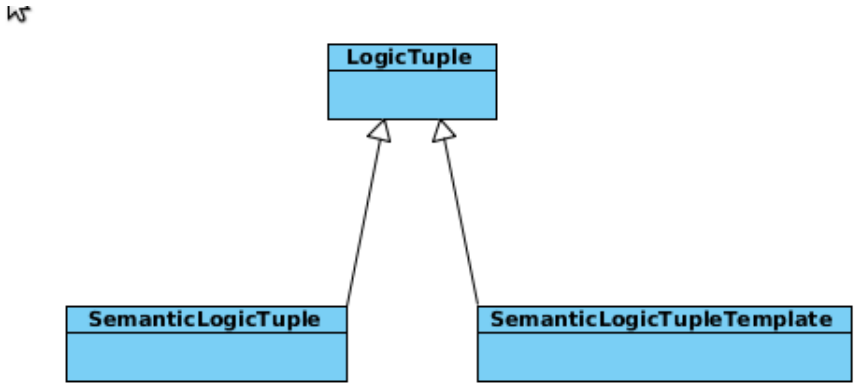

Figure 4.3: Extension of LogicTuple Class

These extensions use the theory of Description Logic $\mathcal{SHOIN}$(D). In fact the SemanticLogicTuple is modelled in a way that it permits to express sentences whose describing the name of individuals, their data and their relationship on this Description Logic. While the SemanticLogicTupleTemplate permits to express sentences whose describing all concept descriptions allowed by $\mathcal{SHOIN}$(D), in order to retrieve the information required.

At the present this extension permits to insert and retrieve knowledge throught Semantic Tuple and Semantic Template, but it does not permit to use a Tuple Centre as very Semantic Tuple Centre, which should contain a Knowledge Base, and can not works in persistent manner as the objective of this thesis requires. In fact a Semantic Tuple Centre should be a container of persistent knowledge, where should be possible insert information (Tell), retrieve information (Ask) and reasoning about it.

4.4 Summary

In this chapter we have introduced the theory behind the TuCSoN framework, starting from the concept of coordination, arriving to the concept

of Tuple Centre, via Tuple Space. We have also described the extension of the semantic component which enables TuCSoN with the capability to work on Knowledge Base. Finally we have explained that this extension is still not complete in order apply it to our motivation scenario a real working framework. In the implementation chapter we will show all the mechanisms that it needs in order to work in a complete mode. We also show how we apply it to Semantic Health Coordination Framework.

Part II

Framework

Chapter 5

The Semantic Health Coordination Framework

Motivated by the need to have a system for EHR exchange in an distribuited environment, in this chapter we describe how model the concept of distributed communities. We connect them in a tree structure, and define coordination primitives to support the EHR exchange. We specify such framework using Semantic TuCSoN and a set of agent that coordinate over the platform.

We define a semantic knowledge base and assume that every community has the same schema. The difference among them is not their semantic structure, but is the data that this structure contains. In fact every community can be different from the others in terms of its organisation, the services it offers, the policies it uses, and others characteristics as we show later in this chapter.

What it follows has been previously discussed in [34].

The chapter is organized as it follows: in section 5.1 we model the system, explaining how each community is structured, what type of tuple centres are present inside the community, and how is built the topology of the system; in section 5.2 we explain the ontology community present in each tuple centre of to the system, we then describe all present entities in it (Classes and Properties); in section 5.3 we introduce the concept of Policy Tuple Centre, explaining all tasks that it performs, and all primitives of coordination defined inside this type of tuple centre; in section 5.4 we introduce the concept of Community Tuple Centre, which is the tuple

centre that takes in charge the task to perform the more expensive queries in the knowledge base (research of patients and communities). Finally in Section 5.5 we conclude and summarise this Chapter.

5.1 The Architecture of the System

Every community containt a set of data which are semantically described with an ontology. The ontology describes the concepts of the community, using the TBox. The actual data, represented in the ABox, will be different in the various communities. The union of ABox and TBox builds the Knowledge Base of a single community, on which the system can reasoner to obtain the information wished.

An Affinity Domain as defined and described in Chapter 2 can be seen as a Community. The Community will encapsulate the EHR system and the mechanisms for modeling, modifying and exchanging EHR data.

Figure 5.1 shows the logical architecture of a Community.

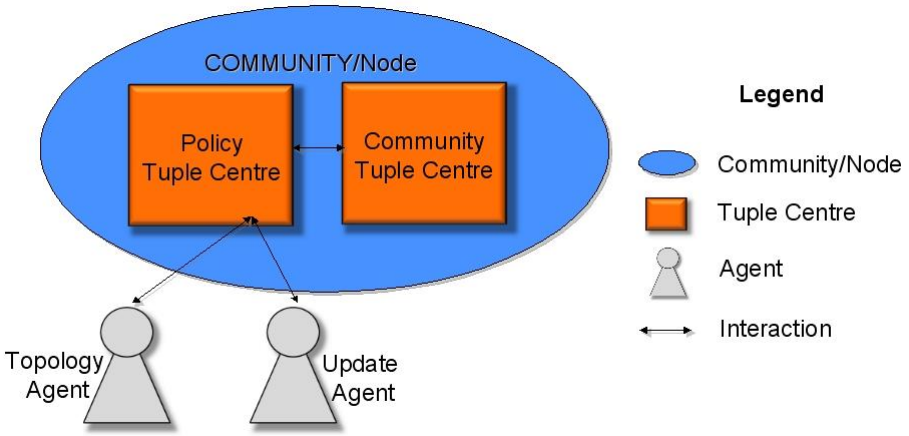

Figure 5.1: Logic Architecture of a Community.

In this figure is shown that each community has inside it two tuple centre, the Policy Tuple Centre and the Community Tuple Centre. The Policy Tuple Centre (PTC) deals with incoming requests from other communities to either connect into a tree structure or to subscribe to notification of events. For every Community there can be a Father Commu- nity and

many Child Communities (See Fig). PTC specifies the coordination primitives for adding, and removing a child or a father to a Community and the primitives for allowing subscription and unsubscriptions from other Communities. These tuple centres are semantics tuple centres of Semantic TuCSoN framework. At the moment we use a soft model of agency where agents simply react to specific messages exchanged in the tuple centres as opposed to a hard model of agency where the agents have complex cognitive models to perform complex reasoning. TuCSoN thanks to the ReSpecT language [2] can enable the coordinations of the different communities. In fact, in each Policy Tuple Centre and the Community Tuple Centre are specified the ReSpecT reactions that coordinate the interested communities. The Community Tuple Centre (CTC) is an additional coordination module used to evaluate search queries that are generated in the system. Communities generate queries to search a new community or to search data related to a patient. Such search queries are computationally expensive thereby we evaluate them outside the PTC. In fact the PTC receives also requests for search queries, but it forwards them to the CTC which may either find the result of a query or propagate it to the PTC-s of the father and the child communities. In this way we evaluate expensive queries in parallel to the normal functionalities offeered in PTC, and do not directly expose the CTC to the whole system.

For each community we have identified two main agents needed for this infrastructure :

- Topology Agent: is the agent responsible of sending and catching messages regarding the context of manage the topology of the entire community system;
- Update Agent: is the responsible of sending and catching messages regarding the manage of the knowledge base;

After a careful analysis we have decided to structure our topology in a tree structure as showed in figure 5.2. There are two reasons for our choise. The first is that the structure will be a dinamic one, where the communities can be added or removed in a self-organising manner. In this way, only the direct interested have to be involved. The second is, because there could be present some queries, where the addressee community is not known. The search of the community could be computationally

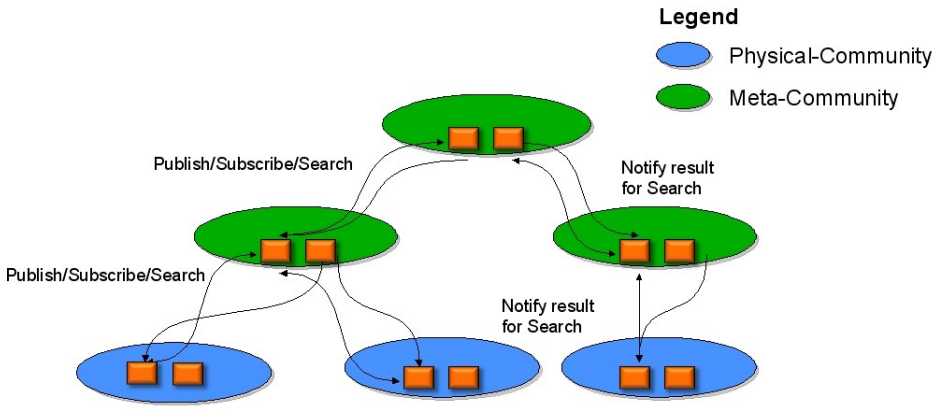

Figure 5.2: High view of the tree structure.

expensive for each node. With a tree structure, the queries propagation is enabled only in the branches connected with the requester, and possibly forwarded in their connections. The research will continue up to which the information will not be found. In this way, can be avoided the send of search message in a broadcast manner. In this tree structure, you can see two type of community, the meta-community and the community. The difference between them is that in the meta-community there is only the infrastructure shown in figure 5.1, while in the community there is also the physical healthcare system inside them. Furthermore, real world communities are usually organised following a tree structure due to their geographical disposition within a region and a state. Therefore, keeping a tree structure, we simplify the representation of real communities within our system.

Figure 5.2 shows also the presence of some interaction. We model the system in a way such as all communications must pass through the PTC. In this way, it can check the policies and decides if the communication is allowed or not. Instead, the CTC can only send information to PCT.

5.2 Ontology Community

As we explained above, every community has its own knowledge base. The schema, represented by the TBox and RBox, is fixed and invariable for all the Communities. The TBox represents all the classes present in the knowledge base, and their IS-Relationships, while the RBox represents all object properties and datatype properties present in the knowledge base. Each property has its domain and range. Figure 5.3 shows the complete schema of the OWL Community Ontology [1] present in each node.

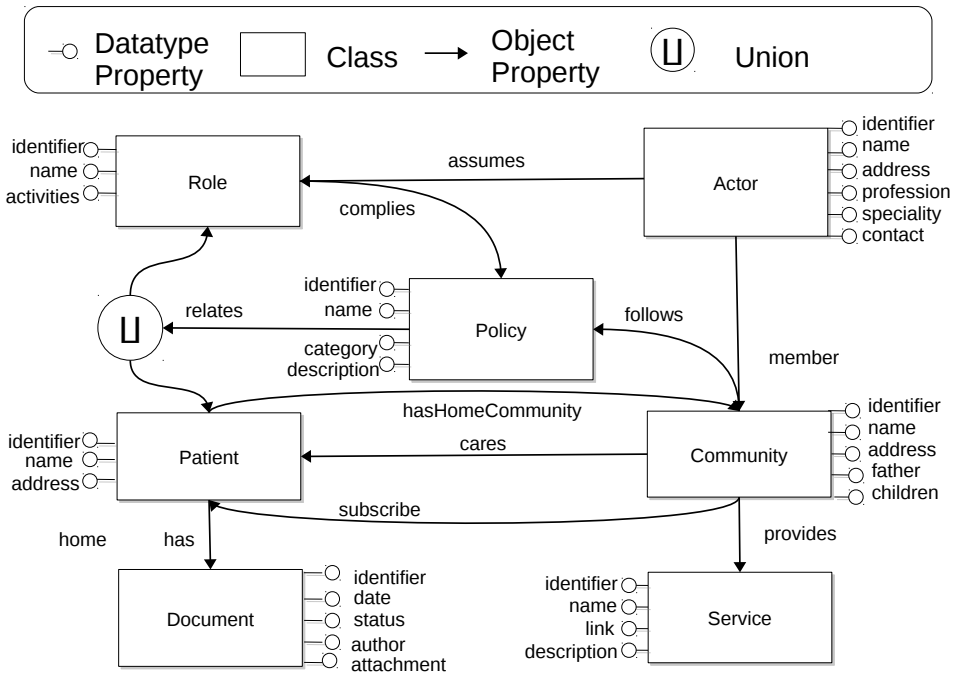

Figure 5.3: The OWL Community Ontology.

Based on [8], we formilise the schema presented in Figure 5.3 The *RBox* of the Community Ontology contains the following object properties, where their name is followed by the *domain* and the *range* for everyone.

[1] The full ontology can be found in http://aislab.hevs.ch/assets/OntologyCommunity.xml

has : Patient $\rightarrow$ Document;
cares : Community $\rightarrow$ Patient;
subscribe : Community $\rightarrow$ Patient;
member : Actor $\rightarrow$ Community;
provides : Community $\rightarrow$ Service;
follows : Community $\rightarrow$ Policy;
assumes : Actor $\rightarrow$ Role;
complies : Role $\rightarrow$ Policy;
relates : Policy $\rightarrow$ (Patient $\sqcup$ Role);
hasHomeCommunity : Patient $\rightarrow$ Community;

We also specify if any property has got the attribute of functionality (or inverse functionality), as follows:

InvFun(has);
InvFun(cares);
InvFun(provides);
InvFun(follow);
Fun(relates);
Fun(hasHomeCommunity);

The *TBox* contains the subsequent axioms that defines cardinality restrictions for the defined properties:

Document $\sqsubseteq$ =1 has^{-};
Service $\sqsubseteq$ =1 provides^{-};
Patient $\sqsubseteq$ =1 cares^{-};
Actor $\sqsubseteq$ =1 member;
Policy $\sqsubseteq$ =1 follows;
Policy $\sqsubseteq$ =1 complies;

We provide an example in order to clarify the meaning of the formalism above used: *the property* relates *has how domain an individual of* Policy *class and how range an individual of union of* Patient *class with* Role *class, and has the attribute of functionality*

This means that an individual belonging to Policy class can partecipate to an only one property relates. And the property has how range or an individual of a Patient class or a individual of Role class.

In the following section we give a description of any definition introduced above, explaining their meaning in our domain.

5.2.1 Entity Description

Community: represents a community, which can be a meta-community or a community

- each Community can provide a set of Services and a service have to be provided always and only to a community;
- each Community can follow a set of Policies and a policy have to be followed always and only to a community;
- each Community can cares about Patients and a patient have to be cared always and only from the home community of this community;
- each Community can subscribe to a Patient of another Community in order to receive notification in case of somewhere regarded this patient happened.

Service: represents a set of services provided from a community

Are the service that each community makes available to patients and involving the possible content of EHRs.

Policy: represents a set of policies that a community has to follow

The Policies are aimed to aid the integration of the Communities by making explicit under which rules the data are shared. In particular, they state which policies are applied to other communities which require patient data or subscribe to patient data in a community, or merge or delete

themselves from the community structure. Such policies are enforced in the Policy Tuple Centre.

- each Policy can be related to a Role or to a Patient, but not at same time;
- each Patient has one and only one Community as home community;

Patient: represents a patient of a community

- each Patient has got a set of Documents, and a document has to belong always and only to a patient;
- each Patient has one and only one Community as home community;

Document: represents part of a patient health record

Are generated and stored within a community. Every document relates to a specific patient. When a document is generated, it has an author, which is an actor in the community and has a set of properties which indicates the content of the document. The community that generates such documents can also update their status by making documents obsolete or deleting them.

Actor: represents an actor of the community

Such actors must perform their activities in complying way with the Policies of the Community.

- each Actor must be member of Community, exactly the home community of this community;
- each Actor can assume a set of Role;

Role: represents a role that an Actor can assume

- each Actor have to be comply to one and only one Policy;

The thing that changes in every Community Ontology is how the knowledge base is filled, and this represents the extensional knowledge (ABox). Each community can interrogate for information or subscribe to updates happening in the knowledge bases of other communities.

5.3 Policy Tuple Centre

The Policy Tuple Centre (PTC) deals with incoming requests from other communities to either connect into a tree structure or to subscribe to notification of events. All the communications to a community are made to the PTC. For every Community there can be a Father Community and many Child Communities (See Figure 5.2). PTC specifies the coordination primitives for adding, and removing a child or a father to a Community and the primitives for allowing subscription and unsubscriptions from other Communities.

5.3.1 Adding and Removing a Community

As said above, is possible add or remove dynamically the Community in the tree structure shown in Fig. 5.2. Topology Agent is the entity recognised to execute this job, and it has a twofold function. The first, is react to commands from administrator users, and ,the second is, to listen to the communications of other communities. The behaviour the Topology Agent can be described as follows:

- It interfaces with the users, which decide how a community connects to the rest of the topology structure. The user can also decide if the community must disconnect from the topology structure. The Topology Agent writes the request message (in case of connect request) or a delete message (in case of disconnect request) in the PTC of the community from which the user specifies it wants connect or disconnect;

- The Topology Agent also listens to accept, reject, add and remove messages generated in its own Policy Tuple Centre;
  - In case of an accept message, it adds the father community to the knowledge base;
  - In case of an add message, adds the requester community as child community in the knowledge base;
  - In case of a remove message it deletes the community form the knowledge base. If the removed community was the father, then it must perform a request to connect to an other community;
  - In case of a reject message, it must perform a request connection to another community;

The PTC specifies the following coordination primitive for a request message:

```
reaction(out(request(id, name, addr, policies),
  (operation, invocation),
  ?-Community(id) ⇒ false ,
  ?-Policy(*) ⇒ {p1...pn} ,
  subset(policies,{p1..pn}),
  out(add(id, name, addr),
  out(id, accept(myid, myname, myaddr) ).
```

The above reaction specifies that the communities accept as their children only those communities that are not already connected to the community (?-Community(id) ⇒ false) and whose policies are a subset of its own policies (?-Policy(*) ⇒ {p1...pn}), and in this case, it sends an add message to its PTC, and an accept message to PTC of requester community.

In a similar manner a community can be deleted from the tree structure. In order to delete a community, the Topology Agent sends a delete message to the community father of the requester community and, if there are any, to its children community . The coordination primitive for such operation is defined as follows:

```
reaction(out(delete(id)),
```

```
(operation,invocation),
?-Community(id) ⇒ true ,
out(remove(id))).
```

The reaction above specifies that the community deletes the community requester only if it is present in the knowledge base. For this this scope it sends a remove message to its PTC.

5.3.2 Subscribing to Community Events

A community can subscribe to events generated by other communities. We are envisaging three types of subscriptions: subscriptions to events regarding a patient, subscriptions to changes on the services a community offers and subscriptions to the changes of the policies that a community offers.

In this thesis we treat only a simplified subscription mechanism for receiving updates regarding patients from other communities. The two others subscriptions have similar considerations to the ones presented here.

The Update Agent is used to subscribe its own community to patients of other communities for the purpose of receiving patients updates, and also manages the information received from the updates. Its persistent behaviour is the following:

- If the home community and the community interested on a patient differ, the Update Agent generates a subscribe (when the patient is recorded)/ unsubscribe (when the patient is deleted) message in the PTC of the home community of the patient. In case of a subscribtion of a patient without a home community, then it before generates a searchCommunity message containing the criteria that it knows about the new patient. The query must produce a complete description of the patient's home community.
- The Update Agent listens to add, remove, and reply messages generated in its own Policy Tuple Centre;
  - In case of an add message, it performs this two action: firstly, if the community that wants subscribe a patient is not present

in the knowledge base of the patient's Home Community, the Update Agent adds this community to this knowledge base; secondly, the Update Agent adds the subscription relationship between the subscriber community and the patient.

- In case of a remove message, it removes in the home community the subscribe relationship between the community subscribed and the interested patient from the knowledge base;
- In case of a reply message, it delivers the data to an human actor, which can decide if the received information should be added to the knowledge base.

The coordination primitive for subscribing to patient updates is specified as follows:

```
reaction(out(subscribe(community, patient)),
  (operation, invocation),
  ?- Patient(patient) ⇒ true ,
  ?- Policy ⊓ (∃relates.{patient}) ⊓
  (∃category.{"filesharing"}) ⊓ (∃description.{"consent"})⇒ true,
  out(add(community, patient))).
```

The primitive is activated when another community requests a subscription to the PTC of a given `community` regarding the information of a given `patient`. In this case, the PTC checks if the identified patient has already contained in the knowledge base and if exists a policy describing the patient consent into sharing its own files.

In a similar way, communities may choose to unsubscribe to communities. This may happen because the patient is not anymore in care at the community and such updates are no longer necessary.

```
reaction(out(unsubscribe(community, patient)),
  (operation, invocation),
  ?- (∃subscribes.{patient})(community) ⇒ true,
  out(remove(community, patient))).
```

The above primitive specifies what happens in case of an unsubscribe request performed by an Update Agent. The PTC which receives the request checks first if the unsubscribing community has previously subscribed the patient, then it sends the remove message to its PTC.

5.3.3 Notification Update

When a new document regarding a patient is generated, modified or removed anywhere in the tree structure, the home community of the patient is notified by default, as defined by the first of the two following coordination primitives. If the change happens in the home community or an update about a patient arrives into home community, such update is propagated to all interested subscribers, as defined by the second coordination primitives. The following coordination primitives dealing of such discrimination:

Updates for a patient with a different home community

```
reaction(out(update(patient, document)),
  (operation, invocation),
  ?- Document(document) ⇒ true ,
  ?- Patient(patient) ⇒ true ,
  (∃homeCommunity⁻.{patient})(*) ⇒ {home},
  home ≠ myid,
  out(home,update(patient, document))).
```

Updates for a patient within the home community

```
reaction(out(update(patient, document)),
  (operation, invocation),
  ?- Document(document) ⇒ true ,
  ?- Patient(patient) ⇒ true ,
  (∃homeCommunity⁻.{patient})(*) ⇒ {home},
  home = myid,
  ?- Community ⊓ (∃subscribes.{patient})(*) ⇒ {c1...cn},
  out({c1...cn},update(patient, document))).
```

The above coordination primitives respectively check first if the document and the patient are presents in the knowledge base, then retrieve the home community of the patient. After there is a checking which disciminates between two different behaviours. If the patient's home community is not the community of this node, this means that an update for a patient with a different home community from the one who

generated the update have to be propagated to the home community with send of an update message to its PTC. While, if the patient's home community is the community of this node, this means that an update for a patient is generated in its home community and all subscribers to such event should be notified with an update message to the their PTC. No agents are used in this operation as the PTC can directly update other PTCs by using TuCSoN coordination primitives.

5.4 Community Tuple Centre

The Community Tuple Centre (CTC) is an additional coordination module used to evaluate search queries which are generated in the system. Communities generate queries to search a new community or to search data related to a patient. Such search queries are computationally expensive thereby we evaluate them outside the PTC. In fact the PTC receives also requests for search queries, but it forwards them to the CTC which may either find the result of a query or propagate it to the PTC of the father and the child communities. In this way we evaluate expensive queries in parallel to the normal functionalities offered in PTC, and do not directly expose the CTC to the whole system.

5.4.1 Searching a Patient

A community can search patients by generating a query to the father community and to the eventual children communities. If the community which receiving the query does not find the requested data, it will propagate the query to its father and to its eventual children (excluding the community that has sent the query). The Update Agent is in charge of generating `patientSearch` message. In the query message it indicates the sender of the message, the community that is requesting the data (in the first step these two coincide) and a list of criterias used for the search. The same query is propagated in the tree structure until the data is found following a flooding-like algorithm that stops when all the nodes are visited, and this is reason, because the sender changes during the query's propagation.

In the search of a patient some criteria are not specified. *For example, in an emergency case, the patient may not be able to produce a home*

community therefore the `homeCommunity` *of the patient may be unknown.* A community can search the data of the patient by specifying some of the patient's demographic data for example. The coordination primitive for searching a patient is defined as follows:

```
reaction(out(patientSearch(community, sender, listCriteria)),
  (operation,invocation),
  Criteria1 ≡ Criteria2 ≡ Criteria3
  ≡ Criteria4 ≡ ⊤
  (member(('identifier', id), listCriteria),
    Criteria1 ≡ (∃identifier.{id}));
  (member(('name', name), listCriteria),
    Criteria2 ≡ (∃name.{name}));
  (member(('address', addr), listCriteria),
    Criteria3 ≡ (∃address.{addr}));
  (member(('hasHomeCommunity', community), listCriteria),
    Criteria4 ≡ (∃hasHomeCommunity.{community}));
  ?-(Patient ⊓ Criteria1⊓ Criteria2 ⊓ Criteria3
   ⊓ Criteria4)(*) ⇒ {p1,...pn }
  (empty({p1,...pn }, false),
    out(community, reply(myID, {p1,....pn})));


reaction(out(patientSearch(community, sender, listCriteria)),
  (operation,invocation),
  Criteria1 ≡ Criteria2 ≡ Criteria3
  ≡ Criteria4 ≡ ⊤
  (member(('identifier', id), listCriteria),
    Criteria1 ≡ (∃identifier.{id}));
  (member(('name', name), listCriteria),
    Criteria2 ≡ (∃name.{name}));
  (member(('address', addr), listCriteria),
    Criteria3 ≡ (∃address.{addr}));
  (member(('hasHomeCommunity', community), listCriteria),
    Criteria4 ≡ (∃hasHomeCommunity.{community}));
  ?-(Patient ⊓ Criteria1⊓ Criteria2 ⊓ Criteria3
   ⊓ Criteria4)(*) ⇒ {p1,...pn }
  (empty({p1,...pn }, true),
    ?- (Community⊓(∃ father.{"true"})
```

```
 ⊔(∃ child.{"true"}))(*) ⇒ {c1,....cn},
remove(sender,{c1,....cn}, {c1,....cj}),
out({c1,....cj},
    patientSearch(myID, community, Criteria)));).
```

The above primitives specify the reactions for coordinate the retrieve operation on the basis of the submitted parameters in the `listCriteria`. To test which are the specified criteria we use `member/2` Prolog clause.

The first primitive covers the case in which the list of specified criteria identifies one or more patients that satisfy the query, while the second one covers the case in which the data which satisfy the query are not found. The `empty/2` is the control that discriminates between the two case, in fact it is used to check if the result of the query returns or not, one or more individuals. If some result are returned, a reply message containing all the found result will be sent to the requester of the query. In the second case, the message patientSearch is forwarded to the father and eventual children communities. We exclude from the propagation of the query the `sender` using the `remove/2` predicate).

5.4.2 Searching a Community

A community can search other communities by generating a query to the father community and to the eventual children communities. If the community which receives the query does not find the requested data, it will propagate the query to its father and to eventual its children if any (excluding the community that has sent the query). The Update Agent is in charge of generating `communitySearch` messages. In the query message it indicates the sender of the message, the community that is requesting the data (in the first step these two coincide) and a list of criterias used for the search. The query is similar to the ones described in the previous section. The coordination primitives for searching a community are defined as follows:

```
reaction(out(communitySearch(community, sender,listCriteria)),
  (operation,invocation),
  Criteria1 ≡ Criteria2 ≡ Criteria3
  ≡ ⊤
```

```
  (member(('identifier', id), listCriteria),
    Criteria1 ≡ (∃identifier.{id}));
  (member(('name', name), listCriteria),
    Criteria2 ≡ (∃name.{name}));
  (member(('address', addr), listCriteria),
    Criteria3 ≡ (∃address.{addr}));
  ?-(Patient ⊓ Criteria1⊓ Criteria2 ⊓ Criteria3)(*) ⇒ {p1,...pn}
  (empty({p1,...pn }, false),
    out(community, reply({p1,....pn})));


reaction(out(communitySearch(community, sender,listCriteria)),
  (operation,invocation),
  Criteria1 ≡ Criteria2 ≡ Criteria3
  ≡ ⊤
  (member(('identifier', id), listCriteria),
    Criteria1 ≡ (∃identifier.{id}));
  (member(('name', name), listCriteria),
    Criteria2 ≡ (∃name.{name}));
  (member(('address', addr), listCriteria),
    Criteria3 ≡ (∃address.{addr}));
  ?-(Patient ⊓ Criteria1⊓ Criteria2 ⊓ Criteria3)(*) ⇒ {p1,...pn}
  (empty({p1,...pn }, true),
    ?- (Community⊓(∃ father.{"true"})
     ⊔(∃ child.{"true"}))(*) ⇒ {c1,....cn},
    remove(sender,{c1,....cn}, {c1,....cj}),
    out({c1,....cj},
        communitySearch(myID, community, Criteria)));).
```

The above primitives specify the reactions for coordinate the retrieve operation on the basis of the submitted parameters in the `listCriteria`. To test which are the specified criteria we use `member/2` Prolog clause.

The first primitive covers the case in which the list of specified criteria identifies one or more community that satisfy the query, while the second one covers the case in which the data which satisfy the query are not found. The `empty/2` is the control that discriminates between the two case, in fact it is used to check if the result of the query returns or not, one or more individuals. If some result are returned, a reply message containing all the found result will be sent to the requester of the query. In

the second case, the message communitySearch is forwarded to the father and eventual children communities. We exclude from the propagation of the query the `sender` using the `remove/2` predicate).

5.5 Summary

In this chapter we presented the architecture of the Semantic Health Coordination Framework. We identified the Topology and Update Agents which are in charge respectively to connect the communities in a tree structure and to propagate updates regarding patients to interested communities.

We describe a community in terms of an Ontology, the structure of which can be shared among the communities. The Ontology serves as a basis for the interpretation of data from other communities.

We also specified two Semantic Tuple Centres. The PCT which specifies the coordination primitives for adding/removing/subscribing and insubscribing to other communities, and to forward to the subscribers the updates. The CTC which takes in charge the duty to perform queries about research of communities and patients.

In the next chapter we will implement the system modelled, adapting Semantic TuCSoN to this context.

Chapter 6

Implementation

In this chapter we aim to implement the objectives identified in the introduction chapter. Figure 6.1 shows the entire Semantic TuCSoN Health Coordination Framework implemented in this thesis, with all relations between the components which compose it.

The implementation is divided in three main parts: in the Section 6.1 we take the semantic component already created in [30], that permits to work in a semantic mode, and we integrate it inside TuCSoN in order to provide a more complete semantic framework. This framework is called Semantic TuCSoN and it contains knowledge base that permits to work on it. In the Section 6.2 we adapt Semantic TuCSoN to the context of eHealth. We create the agents useful to manage the interactions and implement the coordination primitives needed to exchange EHR. To do this, we show the main scenarios, so that it is possible identify all tasks that the agents must perform, and all the coordination primitives needed for the system's coordination. In the Section 6.3, we implement the persistence of the data in the framework, using as support the PostgreSQL Database, which has the task to contain the persistent version of the knowledge base. Finally in Section 6.4 we conclude and summarise this Chapter.

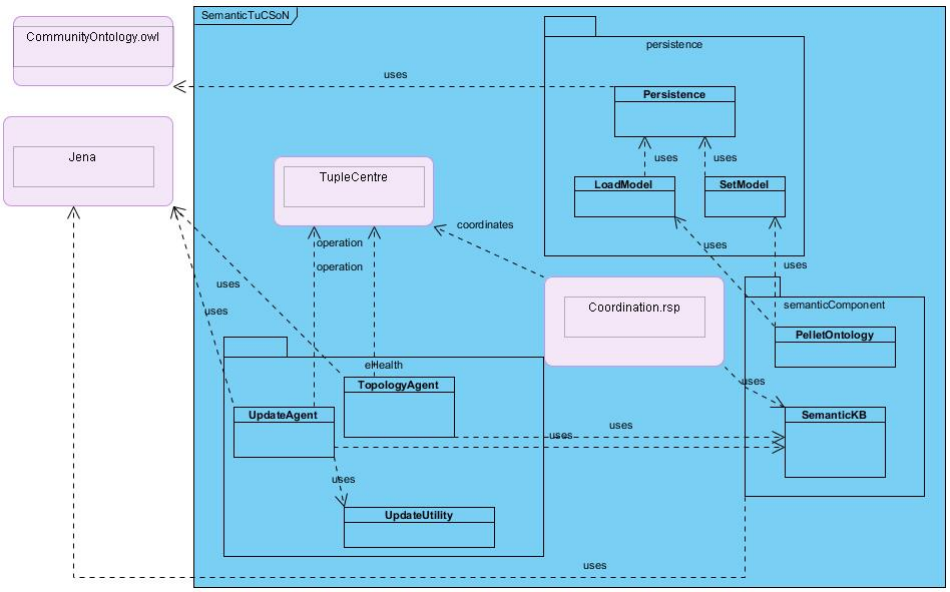

Figure 6.1: Semantic TuCSoN Health Coordination Framework

6.1 Integration of Semantic Component inside TuCSoN

Semantic TuCSoN extends the previous versions of TuCSoN in three main directions:

- possibility of contain knowledge base inside the tuple centres;
- creation of a semantic component to manage the knowledge base;
- extend the Logic Tuple concept in a way that permits to express individuals and concepts of the Description Logic $\mathcal{SHOIN}(D)$

In the following Sections we explain those extensions in more detail.

6.1.1 Semantic Component

Figure 6.2 shows the semantic component structure. The semantic component is a tool which is possible manages a knowledge base contained

inside a tuple centre. The knowledge base is composed by three parts, the TBox and the RBox, which are provided from an ontology and the Abox that is the set of the informations inserted into it. This component permits to load the ontology into the tuple centre, and thereafter to insert information into the knowledge base, and to retrieve informations from it.

An ontology is a structure like the schema of a Database. It is loaded inside the Semantic TuCSoN, through the use of the Java Jena API. During the load operation it is transformed in a OntModel. The semantic component, permits to insert, or retrieve information, transforming the sentences of the Description Logic written by agents, in operations, which, also work using the Java Jena API, in order to handle the OntModel. An OntModel is an enhanced view of a Jena model that is known to contain ontology data, under a given ontology vocabulary (such as OWL). The interface of this component towards TuCSoN is ISemanticKB, and all methods that it provides in order to work on the knowledge base are provided from its implementation, the class SemanticKB, which uses the methods presents in others three interfaces.

6.1.2 Individuals & Concepts

To represent the individuals and concepts of the Description Logics was extended the concept of Logic Tuple so that individividual and concepts could be modelled as tuples these two items.

Individuals in the description logics have the following well defined features:

- a name;
- a concept to which belongs;
- some eventual relationships in which it is involved;

The Java class *alice.semantics.tuple.SemanticLogicTuple* was created to extend the Logic Tuple class, and model the structure of the individual of description logics.
Figure 6.3 shows the features of an individual. In fact here we can see an individual called alex which belongs to Concept1 and has two object

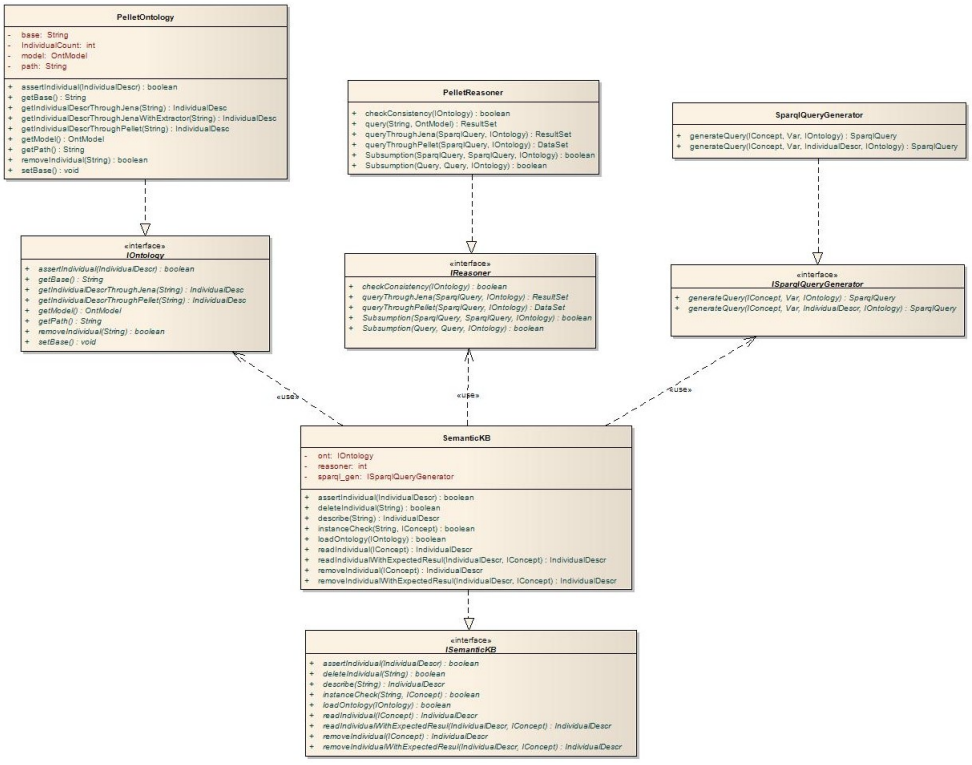

Figure 6.2: Semantic Component

relationships rel1, rel2 with other individuals, whom belong to others concepts, and a set others datatype relationships about its characteristic set of relationships. Each relationship, regardless of the type, has a domain, which is the start point, and has a range, which is the arrival point.
A SemanticLogicTuple allows agents to insert informations which represent individuals of description logics, by writing sentences (Java String) with the following form, which is based on the example in figure 6.3:
semantic alex : 'Community1'(rel1 : range1, ..., relN : rangeN)

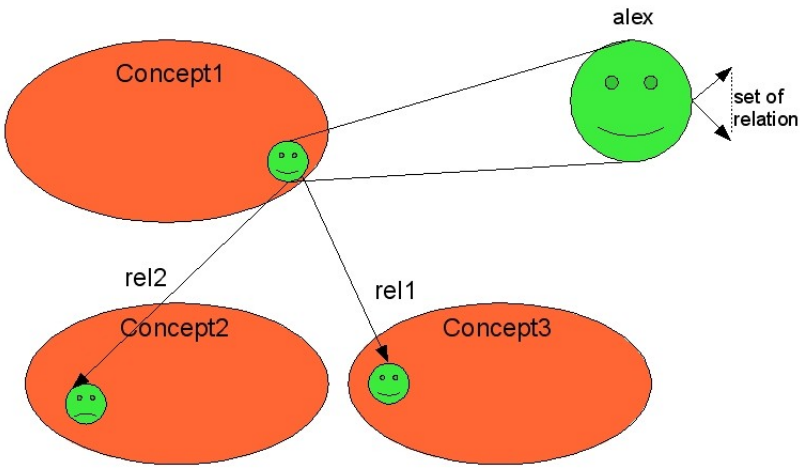

Figure 6.3: Individual

The concepts can be atomic or complex descriptions of them. Figure 6.4, shows a structure that models conceptss, and it is on this structure that are based the Semantic Tuple Template.

This structure permits to model all types of concepts existing in the description logic $\mathcal{SHOIN}(D)$, and based on it , was created a Java class *alice.semantics.tuple.SemanticLogicTupleTemplate* which extends the Logic Tuple class and models the structure of a concept description. A SemanticLogicTupleTemplate allows agents to request informations which represent a concept description of the description logics $\mathcal{SHOIN}(D)$. Sentences (Java String) with the two following form:

a) semantic alex matching *concept description*

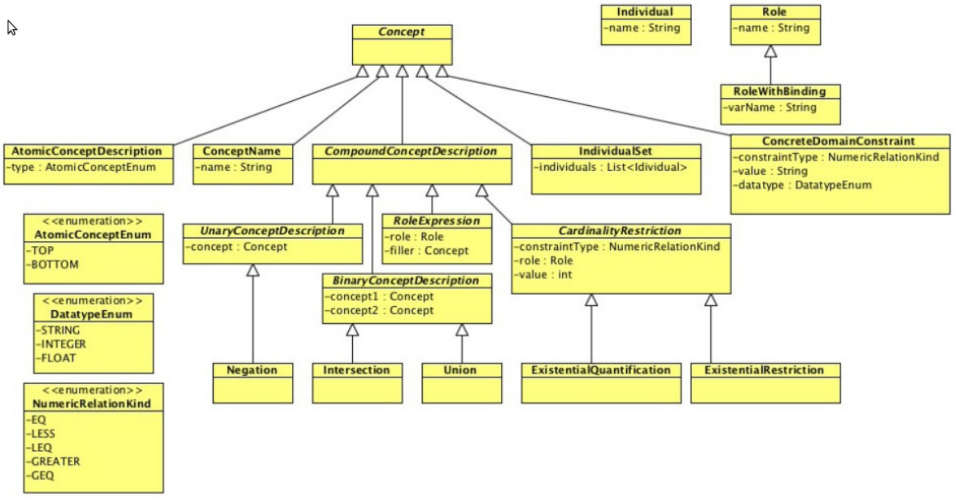

Figure 6.4: Elements Describing the Concept Related to a Semantic Tuple Template

b) semantic 'Result' matching *concept description*

describe respectively an individual called alex, which must belong to a concept described from the *concept description*, while and a Result that requires an individual without specifying its name, which must belong to a concept described from the *concept description*.

For more details concerning the construction of the semantic component, and its features, refer to [16].

The first implementing step of this thesis is to integrate the extensions made on TuCSoN to allow the construction of semantic tuple centres. We want to enable agents to insert individuals in the knowledge base and read/retrieve informations from it. The idea is to achieve this goal, without upsetting the structure of TuCSoN.

To do this, has been made a deep analisys of the TuCSoN framework, in order to understand where and how we should make the necessary changes. There main arguments were identified which we explain in the following sections.

6.1.3 Ability to create Semantic Tuple Centres with a relative ontology

The idea is to extend TuCSoN withe the ability to create semantic tuple centres, without changing the possibility to create classics tuple centres. The reason for this conclusion is that not always we have the need of a knowledge base. Also a semantic tuple centre ccan be omputationally expensive. When there is not a real need for it, the users can create a traditional tuple centre. The knowledge base inside the semantic tuple centre is defined from the ontology loaded in it. We want also that each tuple centre may contain the desidered ontology, which can be different among various tuple centres. So, we must provide a mode to define the ontology which we want to load into the tuple centre, at its startup, and this mode has to be avalaible from outside, and not dedicated inside Semantic TuCSoN.

The way to construct a tuple centre is shown in figure 6.5. The figure illustrates the changes needed in order to build semantic tuple centres inside TuCSoN. The following extension are identified

1. construction of an agent called setOntologyAgent that through an out primitive specifies which ontology has to be loaded into the tuple centre;

2. the AgentContextSkeleton must recognize the tuple inserted from the setOntologyAgent; when this special tuple is inserted, it must call a new method resolveSemanticCore, in which it specifies the ontology decided by the setOntologyAgent

3. add the method *resolveSemanticCore(String tcName, String ont)*, which wants the additional argument *ont*, in the Node class. This method invokes the method bootTupleCentre of the same class, which also has to know the chosen ontology;

4. add the method *bootTupleCentre(String tcName, String ont)*, which wants the additional argument *ont*, in the Node class. This method invokes the method createTC of the TucsonTCContainer class, which also has to know the chosen ontology;

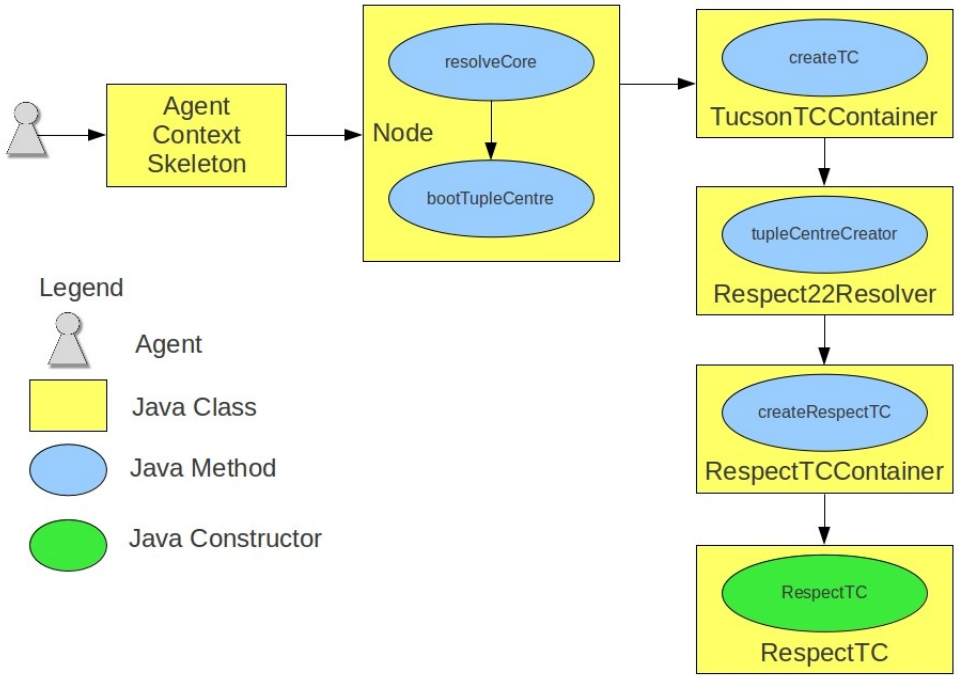

Figure 6.5: Building path for a tuple centre

5. add the method *createTC(TucsonTupleCentreId tcName, int q, String ont)*, which wants the additional argument *ont*, in the TucsonTCContainer class. This method creates a new Java Object PelletOntology which is part of the semantic component. It has as argument the ontology that the setOntologyAgent wants loaded inside the tuple centre. This method invokes the method tupleCentreCreator of the Respect22Resolver class, with in addition the PelletOntology object just created as argument;

6. modify the method tupleCentreCreator in the Respect22Resolver class, adding to it the argument PelletOntology. It invokes the method createRespectTC of the RespectTCContainer class, where in addiction it must pass the PelletOntology;

7. modify the method createRespectTC in the RespectTCContainer class, adding to it the argument PelletOntology. This method creates a new Java Object RespectTC which is the real tuple centre, passing it to the PelletOntology as argument.

8. RespectTC is the class with more changes more. We must create here the main item of semantic component, the class ISemanticKB. Once this component is created, we must load the ontology, so, the OntModel can be build and the tuple centre from this moment works in a semantic mode.

6.1.4 Inserting SemanticLogicTuple and retrieving/reading SemanticLogicTupleTemplate

In order test the functionality of the semantic component the possibility to use an out primitive to insert information through a SemanticLogicTuple and to uses an rd/in primitive to retrieve an information through SemanticLogicTupleTemplate was introduced. As said above, the information contained inside the SemanticLogicTuple and SemanticLogicTupleTemplate is a Java String. This must be transformed in an individual in case of SemanticLogicTuple or in a concept description in the other case. In order to perform this transformation two PROLOG parser are used. The assert_parser.prolog which given a string it returns an individual, and the template_parser.prolog which given a string it returns a concept description. The parsing mechanism is based on the ontology loaded inside the tuple centre, thus the agents have to create the Java Object PelletOntology to perform the parsing.

After a careful analisys, we have decided that this behaviour was not good, because we did not want that an agent manages the semantic component. So, we studied a strategy, where, at the startup of the tuple centre, a tuple containing a string representing the used ontology, has to be inserted into the tuple centre. With this modification, an agent can require the ontology used inside the tuple centre through a new special getOntology primitive. This primitive returns a string representing the ontology. An agent can also decide if the ontology contained in this tuple centre is the ontology on which it wants to work. In this mode, we can also shift the parsing of the sentences from the agents to the class AgentContextStub, which is part of the system, in such a way that the semantic component is now invisible to the agents. To permit this change, we have created, for any primitive involved in the semantic operation, a new correspondent primitive, with an additional String argument. This string represents the ontology. In a semantic tuple centre, the agents must use these new primitives, which has four arguments instead of three.

The last consideration about the semantic extension is that, the individuals and the concepts description are utilised to work on the knowledge base, but their presence is not important in the physical structure that contains the inserted tuples. So we decide that this structure, which is a *Java Dataset*, must contain only syntactic tuples, as usual. When an semantic primitive out is done in the Dataset, only a classic tuple with a special name semantic, containing as argument the name of the individual will be inserted. This tuple can not be replicated because in the semantic field is not allowed to have two individuals with the same name. This special tuple can be seen as a pointer to an individual in the knowledge base, in order to avoid the attempt to insert individuals already present in the knowledge base, or to avoid the attempt to search individuals not present in the knowledge base, that could lead to unnecessary computation. To do this, the SpeakingState class, which is the class that represents the speaking-state of the tuple centre's virtual machine, is changed to recognize if a tuple is instance of SemanticLogicTuple or instance of SemanticLogicTupleTemplate, and in this case, checks the presence or not of the tuple that points the individual required.

6.2 Modelling Semantic TuCSoN for eHealth context

The implementation of our framework is based on the semantic component for TuCSoN tuple centres as defined in [30] and integrated inside it to create a new version called Semantic TuCSoN as described in Section 6.1. Additionally, we interface with openXDS[1], an open source implementation of the XDS profile, in order to have documents stored and retrieved in an IHE compatible manner. Both of these infrastructures are JAVA based.

Figure 6.6 shows how the architecture of the system is related to TuCSoN. Every community is represented with a TuCSoN node and has its own semantic knowledge base. The semantic knowledge base is used by Jena to manage the defined ontology and from Pellet/SPARQL to perform queries and reasoning over the ontology. In order to provide the persistence of the data model, the semantic knowledge base is stored in

[1] https://www.projects.openhealthtools.org/sf/projects/openxds/

a PostgreSQL database, of which its implementing details are explained in Section 6.3.

The User Agent is an external agent which interfaces with the users of the system. After a user's request to add or retrieve informations to the knowledge base takes place, the system has to work on the knowledge base. In case of the addition or modifications of documents, IHE compatible meta-data are generated to be stored in the registry of the XDS profile and the same meta-data are stored as semantic data in the community knowledge base. Updates to the meta-data of the documents of a patient are propagated towards the home community of the patient and to the subscribed communities. We assume that the actual fetching of the documents is realised using one of the existing IHE profiles (XCA already addresses this issue).

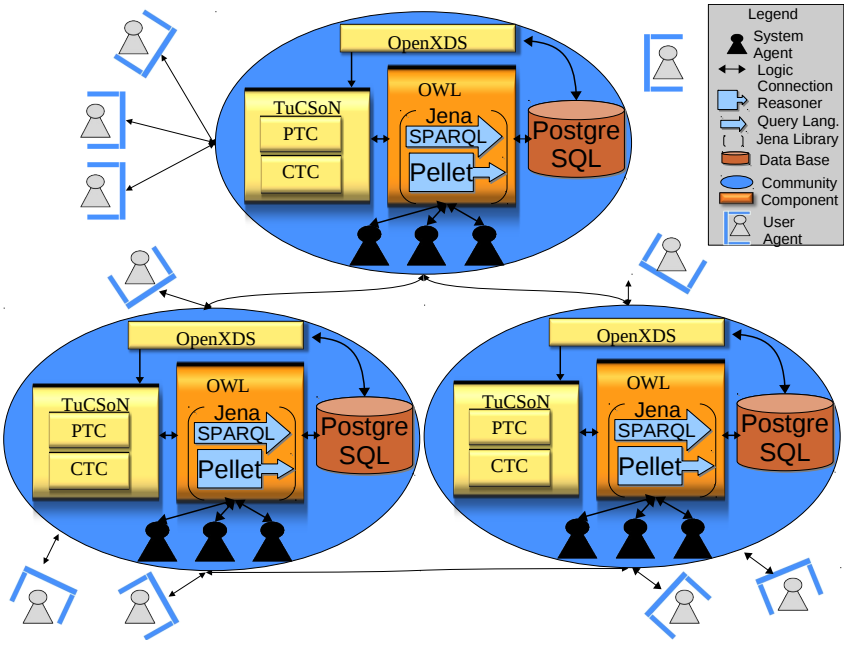

Figure 6.6: The implementation of the system.

In the Chapter 5 we introduced two agents (Topology Agent and Update Agent), which are persistent in the system. Each one of them has a particular characteristics, and has the behaviour specified before. In this section we illustrate their implementation details, accompanied by sequence diagrams, to explain better the system's behaviour.

There are three differences between what explained in the chapter about the System Architecture 5 and the implementation. The first is that, not needing for now of a mechanism to check all policies present in the community, the Policy Tuple Centre (PTC) is not considered. The coordination primitives specified for a Community are all included in the Community Tuple Centre (CTC). The second is that we want to create a prototype to test the motivating scenario introduced in Section 2.1, our attention is focused on it, so, only the Topology Agent and the Update Agent are implemented. It is also seen the need to model a new agent, called Coordination Agent, whose job is charged into the CTC the ReSpecT specification, which defines the coordination policies enabled inside it.

The three agents are created in the class RespectTC, where the out primitive related to the tuple inserted by the setOntologyAgent is taken.

6.2.1 Coordination Agent

This agent is very simple, and its only duty is specified into the tuple centre the ReSpecT specification defined. This specification is contained in a file called CommunityReaction.rsp. The reaction primitives content inside the specification calling JAVA code. This is possible because TuCSoN is based on tuProlog [11], a Java based implementation of Prolog that allows a seamless integration between Java code and Prolog predicates. For more informations about the use of TuCSoN refer to [3].

6.2.2 Topology Agent

There is a Topology Agent for each community node. The agent reacts to tuples generated by external User Agents or to tuples generated by the CTC of the community. The Topology Agent uses non-blocking in operations in the CTC and reacts to messages either by writing in the semantic knowledge base or by performing out operations in the CTC.

In the Section 5.3.1 we said that the Topology Agent listens for tuples. The tuples that this agent is interested have the following template:
topologyAgent(operationName,[args]).
Its behaviour changes in base of the type of operation specified in the

operationName argument. There are six types of operation to which it is interested:

- connect
- accept
- reject
- add
- disconnect
- remove

The connect, accept, reject, add operations are relative to Connection phase, while disconnect, remove operations are relative to Disconnection phase. These two phases are described below, with all actions performed by all entities involved.

Connection to Topology

The sequence diagram in figure 6.7 shows what happens when an users, usually an administrator user, asks to connect this community to a topology. In this figure are present two Communities, the Community Sierre, which asks to be added to a topology, precisely as child community of the Community Valais.

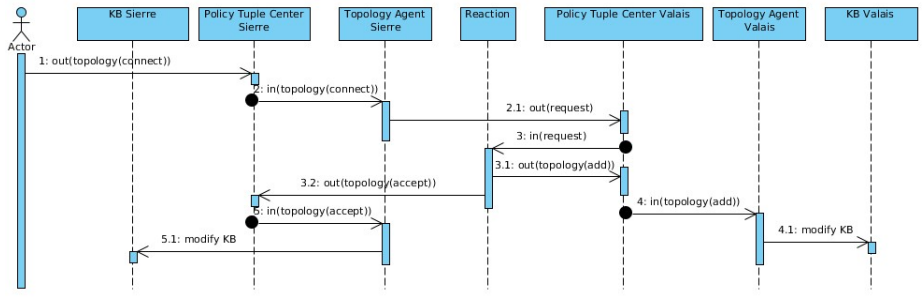

Figure 6.7: Connection to the Topology.

The behaviour is explained as follows:

1. An actor triggers the Topology Agent Sierre to require a connection to the topology, by inserting a tuple with this template topologyAgent("connect",idFather,nameFather), where it specifies the identifier and the name of the community Valais of which Sierre wants to become a child. Topology Agent Sierre creates an out primitive called request with this template request(idSierre, nameSierre, addressSierre, policiesSierre, nameValais) containing the information of the community where it lives, and sends it to the CTC of community Valais.

2. The CTC reacts automatically at this out primitive, thanks to a reaction ReSpecT, which retrieves the tuple and checks if the conditions are satisfied how specified in reaction 5.3.1. Here the policies checking is performed, associating to each community a number, that identifies the level of policies of the community, in such a way that can be done a check on the values of the two communities. If the reaction is done, at the end, it sends an out primitive called add with this template topologyAgent(add,idSierre, nameSierre, addressSierre, policiesSierre) to its CTC, and an out primitive called accept with this template topologyAgent(accept,idValais, nameValais, addressValais, policiesValais) to the Sierre CTC, otherwise it sends an out primitive called reject to the Sierre CTC. The code of this reaction is showed in A.1

3. The Topology Agent Valais when an out primitive called add is made, retrives the tuple from its CTC and create a semantic out primitive for add the community Sierre to its knowledge base specifying that is its child.

4. The Topology Agent Sierre when an out primitive called accept is made, retrieves the tuple from its CTC and create a semantic out primitive for add the community Valais to its knowledge base specifying that is its father. In case of out primitive called reject the Topology Agent Sierre, retrieves the tuple from its CTC, creates an other out primitive request, and sends it to its CTC for trying to connect itself to an another eventual father.

The java code of the Topology Agent to perform the activity of connection is showed in B.1

Disconnect from Topology

The sequence diagram in figure 6.8 shows what happens when an users, usually an administrator user, asks to disconnect this community from a topology. In this figure are present two Communities, the Community Sierre, which asks to be removed to a topology, and Community Valais, which must delete the Community Sierre from its knowledge base.

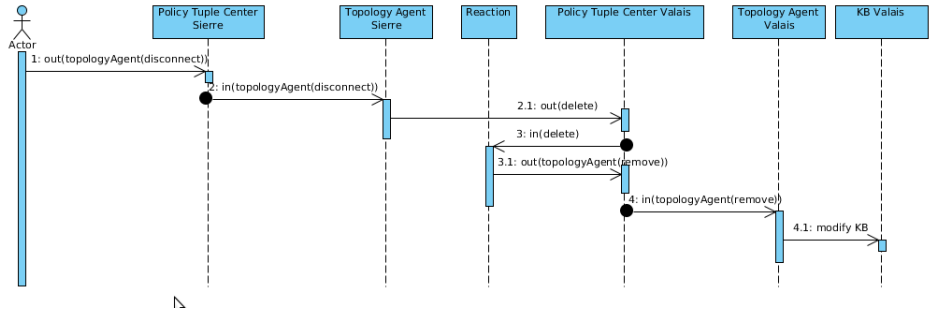

Figure 6.8: Connection to the Topology.

The behaviour is explained here:

1. An actor triggers the Topology Agent Sierre to required a disconnection from the topology, by inserting a tuple with this template topologyAgent("disconnect"). Topology Agent Sierre creates an out primitive called delete with this template delete(nameSierre), and sends it to the CTC of community Valais.

2. The CTC reacts automatically at this out primitive, thanks to a reaction ReSpecT, which retrieves the tuple and checks if the conditions are satisfied how specified in reaction 5.3.1. If the reaction is done, at the end, it sends an out primitive called remove with this template topologyAgent(remove, nameSierre) to its CTC. The code of this reaction is showed in Appendix A.1

3. The Topology Agent Valais when an out primitive called remove is made, retrives the tuple from its CTC and delete the community Sierre as its child.

Verily, the community which asks to be deleted from a topology must send the topologyAgent("disconnect") to its father and all its children.

But here we have showed the procedure only for the relation between Sierre and his father Valais. The eventual children of Sierre, in case of its deletion, lose the father, so, they must request to become children to another community.

6.2.3 Update Agent

There is a Update Agent for each community node. The agent reacts to tuples generated by external User Agents or to tuples generated by the CTC of the community. The Update Agent uses non-blocking in operations in the CTC and reacts to messages either by writing in the semantic knowledge base or by performing out operations in the CTC.

In the Section 5.3.2 we said that the Update Agent listens for tuples. The tuples of which it is interested have the follows template:
updateAgent(operationName,[args]).
Its behaviour changes in base to the type of operation specified in the operationName argument. There are nine types of operation in which it is interested:

- subscribe
- unsubscribe
- add
- remove
- update
- searchCommunity
- searchPatient
- replyCommunity
- replyPatient

These operations can be divided in three groups, the subscribe, unsubscribe, add, remove operations are relative to Subscription phase, the update operation is relative to Update Notification phase, the searchCommunity, searchPatient, replyCommunity, replyPatient operations are relative to

Research of informations phase, These three phases are described below, with all actions performed by all entities involved.

Subscription to information

The sequence diagram in figure 6.9 shows what happens when an users, usually an administrator user, asks to subscribe this community to a patient of another community. In this figure are present two Communities, the Community Sierre, which asks to subscribe a patient which has as home community, the Community Valais.

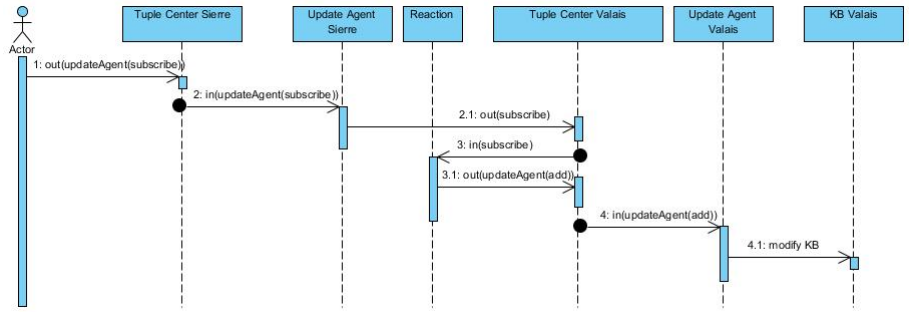

Figure 6.9: Subscribe to a Patient.

The behaviour is explained here:

1. An actor triggers the Update Agent Sierre to require a subscription, by inserting a tuple with this template updateAgent("subscribe", idValais, nameValais, homeCommunityPatientID)), where is specified the identifier and the name of the community Valais and the identifier of the patient in his home community. Update Agent Sierre creates an out primitive called subscribe with this template subscribe(idSierre, nameSierre, addressSierre, policiesSierre, homeCommunityPatientID) containing the information of the community Sierre and the patient's identifier, and sends it to the CTC of community Valais.

2. The CTC reacts automatically at this out primitive, thanks to a reaction ReSpecT, which retrieves the tuple and checks if the conditions are satisfied how specified in reaction 5.3.2. If the reaction

is done, at the end, it sends an out primitive called add with this template
updateAgent(add,idSierre, nameSierre, addressSierre, policiesSierre, homeCommunityPatientID) to its CTC.

3. The Update Agent Valais when an out primitive called add is made, retrieves the tuple from its CTC and create a semantic out primitive in order to insert in the knowledge base the information about the subscription.

The sequence diagram in figure 6.10 shows what happens when an users, usually an administrator user, asks to unsubscribe this community by a patient of another community. In this figure are present two Communities, the Community Sierre, which asks to unsubscribe a patient which has as how home community, the Community Valais.

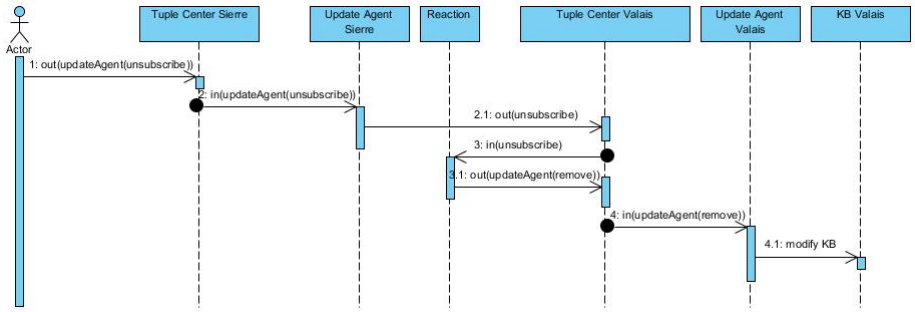

Figure 6.10: Unsubscribe by a Patient.

The behaviour is explained here:

1. An actor triggers the Update Agent Sierre to require a unsubscription, by inserting a tuple with this template
updateAgent("unsubscribe", idValais, nameValais, homeCommunityPatientID)), where is specified the identifier and the name of the community Valais and the identifier of the patient in his home community. Update Agent Sierre creates an out primitive called unsubscribe with this template
unsubscribe(nameSierre, homeCommunityPatientID) containing the name of the community Sierre and the patient's identifier, and sends it to the CTC of community Valais.

2. The CTC reacts automatically at this out primitive, thanks to a reaction ReSpecT, which retrieves the tuple and checks if the conditions are satisfied how specified in reaction 5.3.2. If the reaction is done, at the end, it sends an out primitive called remove with this template
updateAgent(remove, nameSierre, homeCommunityPatientID) to its CTC.

3. The Update Agent Valais when an out primitive called remove is made, retrieves the tuple from its CTC and it deletes the subscription object property between the community Sierre and the interested patient from the knowledge base.

Update Notification

In this part of implementation we use a functionality of Jena API that permits to catch the changes in the OntModel. It provides various types of changes captured, in form of methods that the user must implement [2]. Implementing them each one can choose what type of change recognise. In our case we want recognise the change of something regarding any Document of a subscribed Patient. The Java code of our implementation for this case is showed in Appendix B.2.

The sequence diagram in figure 6.11 shows what happens when a changement of a document of a patient subscribed happen.

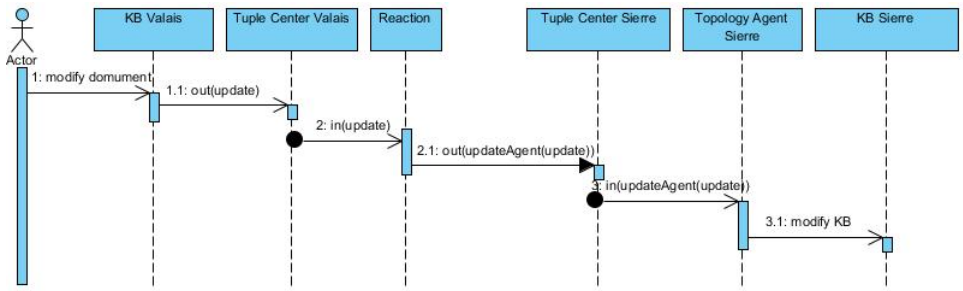

Figure 6.11: Update Notification.

The behaviour is explained here:

[2] com.hp.hpl.jena.rdf.listeners

1. The listener catch the changement, and creates an out primitive called update with this template update(patientId, documentId, homeCommunityId, caresCommunityId), where is specified the identifier of the patient, the identifier of the document, the identifier of the home community of the patient, and the identifier of the community where the change happens.

2. The CTC reacts automatically at this out primitive, thanks to a reaction ReSpecT, which retrieves the tuple and checks if the homeCommunityId is the same of caresCommunityId as defined in reactions 5.3.3 and 5.3.3.

    - in case the two identifier are equal, this means that the change happens in the home community, and it must send the change's information to all subscribers of that patient with a out primitive called update with this template updateAgent("update",changements). In the Appendix A.2 is shown the reaction that performs this behaviour;
    - otherwise, the community where the change happens, must send the information of this event to the home community of the patient.

3. The Update Agent when an out primitive called update is made, retrieves the tuple from its CTC and work on the knowledge base in order to insert the modifications received.

Research of information

The sequence diagram in figure 6.12 shows what happens when an users, usually an administrator user, asks to search informations about a community. In this figure are present two community, the Community Sierre, which performs the request, and the community which represents the set of the communities connected to Sierre in the topology.

The behaviour is explained here:

1. An actor triggers the Update Agent Sierre to require a research of a community, by inserting a tuple with this template

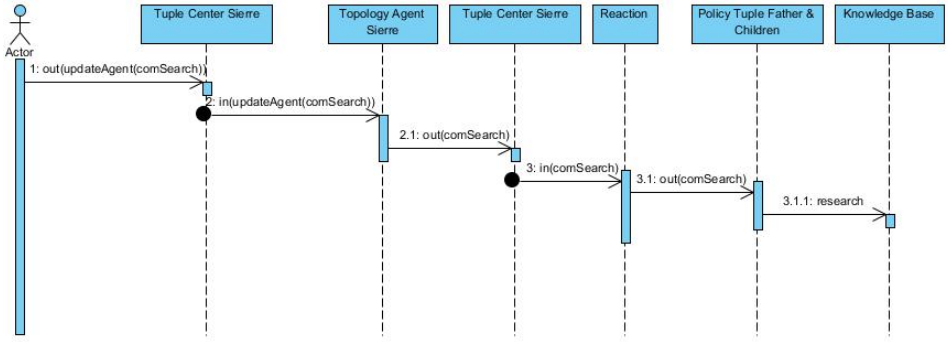

Figure 6.12: Research of a Community.

updateAgent("comSearch", criteria)), where is specified a list of arguments. The list of arguments called criteria represents the informations known by the user about the community searched. Update Agent Sierre creates an out primitive called comSearch with this template
comSearch(idSierre, nameSierre, criteria, nameSierre) containing the identifier and the name of the community Sierre, the list of arguments for the research and the name of the tuple centre sender. This tuple is send to all CTC of communities connected to Sierre.

- If a connected community finds the community searched, it sends a tuple with this template
  replyCom(criteriaSearched)), to the Community Sierre. So, the Community Sierre can insert the informations into the knowledge base as shown in the sequence diagram in figure 6.13.
- If the connected communities do not find the community searched, they forward the tuple comSearch changing the fourth argument with its name, to all connected communities except the sender community.

6.3 Persistence in Semantic TuCSoN

An important requirement for a real system that contains knowledge is the persistence. In fact is vain have a container of information if this can

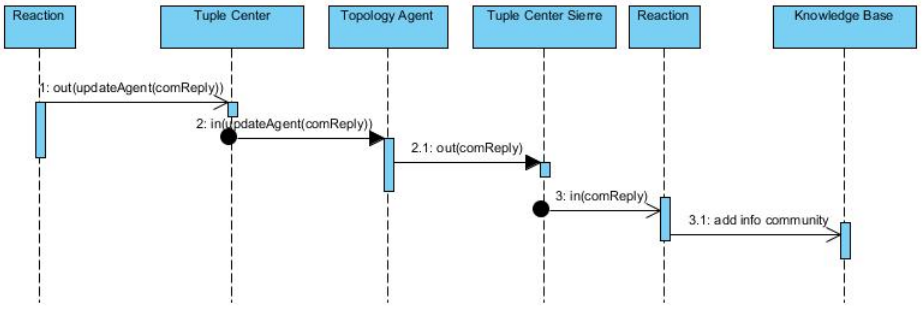

Figure 6.13: Reply about Community information.

lose some important informations. Jena API can provide the persistence to the OntModel using a database [18]. In order to reach this goal we use a PostgreSQL database and we implement the persistence in the follows manner. We create two classes, one used for the startup of the tuple centre, in order to load the ontology and transform it in a OntModel. While the other one used for reload the OntModel if the tuple centre collapses. The Java code of these two classes is showed in Appendix B.3. For the persistence is necessary use the JDBC driver for PostgreSQL and do this three operations in it:

- create a database called "persistence";
- create an user called "tucson"
- create an user called "tucson"

6.4 Summary

In this chapter we have implemented all elements needed to have a first implemented version. This version satisfies all requirements identified in the previous chapters. The resulting system will be utilized in next chapter to realize the evaluation based on the use cases identified in the motivating scenario.

Part III

Evaluation & Conclusions

Chapter 7

Evaluation

In this chapter we test the Semantic TuCSoN based on the sceneries identified in the motivation case 2.1. These sceneries are the follows:

a) notification of updates to subscribed communities;

b) research of a community based on some informations.

The chapter is organized as follows: in the section 7.1 we explain the architecture of the machine where we are making the tests, and the characteristics of the Knowledge base used for the tests; in the section 7.2 we test the performance of our framework in the scenario of notification of updates to subscribed communities; in the section 7.3 we test the performance of our framework in the scenario of research of a community based on some informations; in the section 7.4 we evaluate the perfomance of the framework, and we do some reflections of it, basing on the tests; in the section 7.5 we summarise the work done in this chapter, and we say for what we use the results obtaining.

7.1 Test Architecture

We performe tests to evaluate the proposed solution on a 24 cores Intel, 2.93 GHz and 96GB RAM machine. We create 7 semantic tuple centres, each of them represents a different community and contains the ontology showed in figure 5.3, and we install each one in separate virtual machine.

In each community we insert some informations, which consist of Individuals of description logic with their object and datatype properties. The step for make the tests are the follows:

- insertion of an individual that represent the home community of the tuple centre;
- insertion of some individual for each Class presents in the ontology;
- insertion of one thousand individuals belonging to Patient class.

Furthermore, for both of the tests, we give to the 7 tuple centres, other communities, starting by one hundred communities present in each tuple centre, up to arrive to one thousand communities present in each one. These communities are not deployed in virtual machines. We did this to understand how the computation time of the different queries increases as the semantic knowledge base increases in size, because in the context of eHealth there is the needs of contain a large number of informations.

For both test, before, we create a topology as showed in figure 7.1 in order to check all procedures to simulate a real use environment. The figure shows the tree structure considerated.

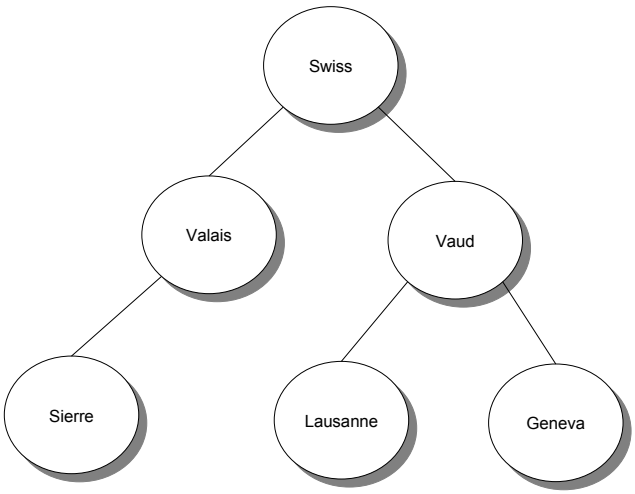

Figure 7.1: Test Topology

7.2 Notify Updates

In this test we varied the number of subscriptions for a patient from 1 to 6 and measured the average time that a community takes to send updates regarding that patient to the subscribed communities.
The interval time for the evaluation is the time that intercorre from the moment when a update is done in the home community to the moment when it is sended to all subscribers.

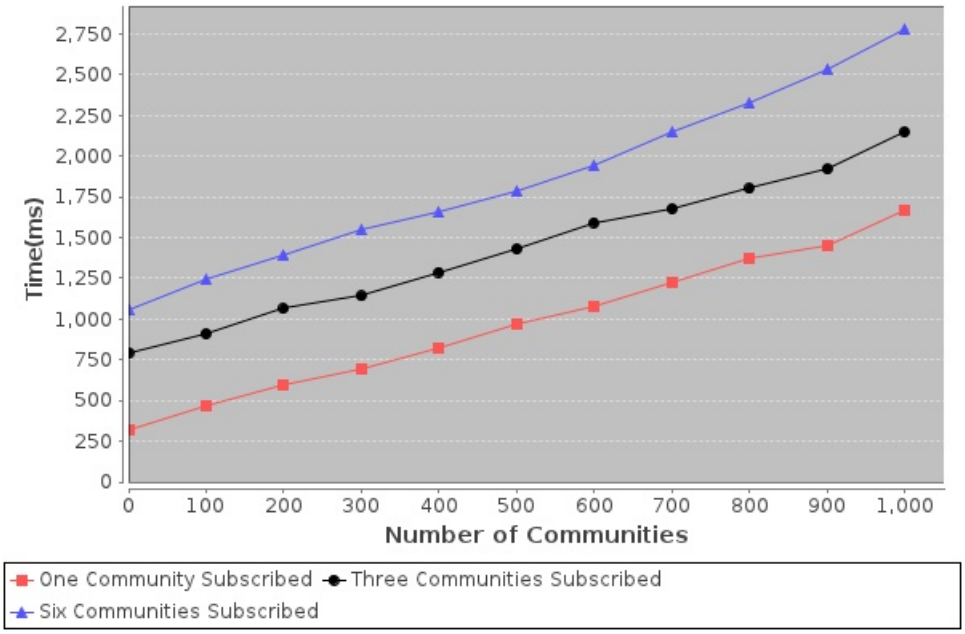

Figure 7.2: The Update Time.

Fig. 7.2 shows how the update time changes with a growing number of known communities and subscriptions to a patient. The time to update other communities grows linearly with the number of community individuals held in the knowledge base. This is due to the increase on the time to search for the communities that are subscribed to a specific patient. Also a growing number of subscriptions per patient introduces a latency as more than one update message has to be sent into other tuple centres.

7.3 Community Research

In this test we measured the time that a distributed search takes to find the results. We searched data that were at 1 then 2 and then 3 hops in the tree structure.
The interval time for the evaluation is the time that intercorre from the moment when a research of community start to the moment when the reply arrives.

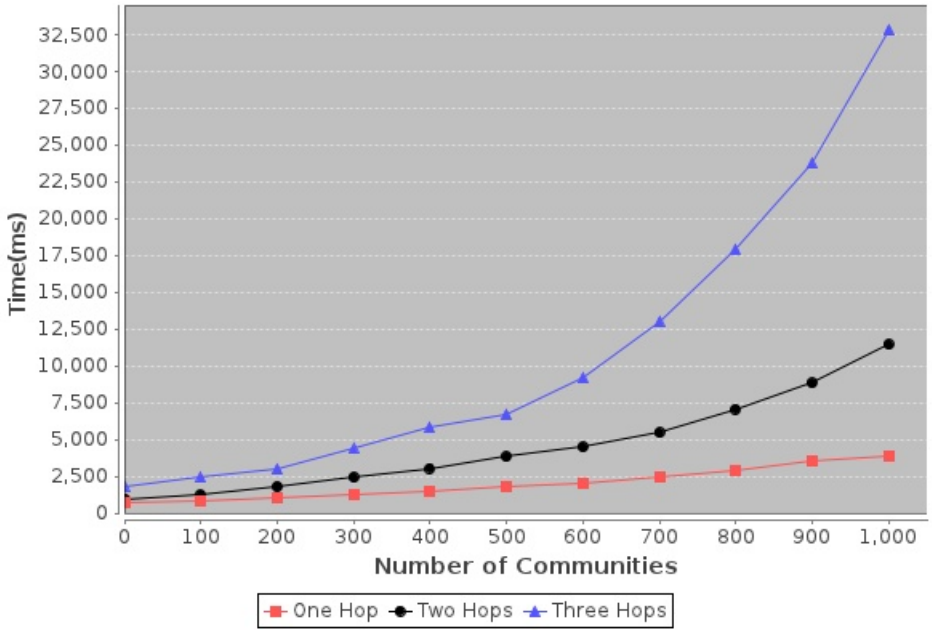

Figure 7.3: The time to search in other communities.

Fig.7.3 shows how the search time in other communities of the tree structure changes as the number of community individuals held in every knowledge base grows. The results show that the time to search in other communities grows exponentially as the search gets propagated in the tree structure. This is to be expected as the search has no prior knowledge as to where to send the query and each node has to evaluate the query before establishing that there is no data held and forward it to the neighbouring nodes.

7.4 Results Evaluation

The tests shown that for the update notification, the growth, at the increase of quantity of informations contained inside the knowledge base is linear, and it is a good result. But, for the research of community, in the same conditions, we have a exponential growth. This is not good, in a context where must be present a large number of communities. After a careful analysis, our evaluation is that the guilt of this incremental growth is to charged to the TuCSoN limitation to work with a large number of tuple, that has a decadence of performance, when this number grows too.

7.5 Summary

In this chapter we have evaluated the performance of the system for the main scenarious identified. After these evaluations we have evaluated the results obtained, which are the bases for the conclusions present in the next chapter.

Chapter 8

Concluding Remarks

8.1 Conclusions

Based on the evaluations done in Chapter 7 we can draw the following conclusions about the new framework Semantic TuCSoN applied to the eHealth context:

a) it works well to resolve the deficiencies of the IHE profiles. In fact it is perfect in order to reach the goal of information exchange among communities without syntactic constrains.

b) a controindication is that Semantic TuCSoN does not support a large number of informations inside a single tuple centre, and the eHealth is a context where a lot of informations should be managed. The solution of this, is inserted inside the tuple centre only the informations needed to drive the coordination of the system.

8.2 Summary

In this thesis we were motivated by the need to have a framework overcomes the limits of the IHE profiles about the exchange of semantic informations. We have identified the framework TuCSoN to reach the objective. This is because it permits to exchange information in distribuited environment and was provided of semantic functions in order

to work on the knowledge base. Our work has been to integrate these functions inside TuCSoN in order to have tuple centres really semantics, thus having a first functional version of Semantic TuCSoN. The second work has been apply this resulting framework to the context of eHealth, creating all entities needed to the exchange of informations and for their coordination, by following the model developed in Chapter 5. As final step we have made the test of the main sceneries identified in order to obtain some performance evaluation that will help us to decide if the work will continue with Semantic TuCSoN or not.

8.3 Future Work

As our future work we focus on the inserting of this framework in a real use for sanitary organisations. We plan to provide the framework of Graphic Interfaces dedicated for every user, based on access rights. The build of Graphic Interfaces will permit to overcome the present need to know the $\mathcal{SHOIN}(D)$ in order to use the framework. We plan also to address security issues arising from an open environment. Apart from the logging of events, for which we can use a Log Agent, the set of policies may help to check that the emerging behaviour of the actors performing the queries is that expected within the community sub-system. Finally we plan to allow communities to subscribe to different types of events (other than patient updates) and allow for filters to be applied to the exchanged information. Both of these extensions will require more complex semantic reasoning than the one presented in this thesis.

Appendix A

Reaction ReSpecT

A.1 Connection request

```
reaction(out(request(X,Y,W,Z,RequestedName)),
        (operation,completion),
        (
        get_semanticKB(KB),
        KB<-getBase returns Base,
        KB<-getModel returns Model,
        text_term(Ytext,Y),
        text_concat(Base,Ytext,IndividualName),

        Model<-getIndividual(IndividualName) returns Individual,

        Individual = _,

        text_term(RequestedNameText,RequestedName),
        text_concat(Base,RequestedNameText,MyIndividualName),
        Model<-getIndividual(MyIndividualName) returns
                IndividualThisCommunity,

        text_concat(Base,'policies',NamePoliciesProperty),
        Model<-getDatatypeProperty(NamePoliciesProperty) returns
                PoliciesProperty,
        IndividualThisCommunity<-getProperty(PoliciesProperty) returns
                MyPolicy,
        MyPolicy<-getObject returns PolicyObject,
        PolicyObject<-asLiteral returns PolicyLiteral,
        PolicyLiteral<-getInt returns PolicyValue,

        Z >= PolicyValue,

        text_concat(Base,'identifier',NameIdProperty),
        text_concat(Base,'name',NameNameProperty),
        text_concat(Base,'address',NameAddressProperty),
```

```
        Model<-getDatatypeProperty(NameIdProperty) returns
                IdProperty,
        Model<-getDatatypeProperty(NameNameProperty) returns
                NameProperty,
        Model<-getDatatypeProperty(NameAddressProperty) returns
                AddressProperty,
        IndividualThisCommunity<-getProperty(IdProperty) returns
                MyId,
        IndividualThisCommunity<-getProperty(NameProperty) returns
                MyName,
        IndividualThisCommunity<-getProperty(AddressProperty) returns
                MyAddress,
        MyId<-getObject returns IdObject,
        MyName<-getObject returns NameObject,
        MyAddress<-getObject returns AddressObject,
        IdObject<-asLiteral returns IdLiteral,
        NameObject<-asLiteral returns NameLiteral,
        AddressObject<-asLiteral returns AddressLiteral,
        IdLiteral<-getString returns IdString,
        NameLiteral<-getString returns NameString,
        AddressLiteral<-getString returns AddString,

        current_tc(TC),
        TC ? in(request(X,Y,W,Z,RequestedName)),
        out(topologyAgent(add,X,Y,W,Z,'null','null')),
        out_tc(Y @ X,topologyAgent(accept,IdString,NameString,
                AddString,PolicyValue))
        )
).
```

The reaction rule specifies that when an out of a subscribe tuple is made into the tuple centre, then the reference to the JAVA object representing the semantic knowledge base KB is used (the ← notation represent a call to a java module) to obtain the URI and the model Model of the ontology. Now we calls the Java method getIndividual(nameOfIndividual) in order to retrieve from the OntModel the Individual with name nameOfIndividual. The Individual represent the home community of this tuple centre. Of this home community we obtain the number that identifies the policy level, so, it can be compared with the policy level of the community that has submitted the request. If the result of comparison is true, the reaction retrieves from the OntModel all properties of the individual home community, in the similar mode as before has taken the policy value. At the end, the reaction picks up the tuple request from the tuple centre, and perform an out primitive topologyAgent(add,X,Y,W,Z towards the local tuple centre, and perform an out primitive topologyAgent(accept,IdString,NameString,AddressString,PolicyValue) towards the tuple centre of the requester.

A.2 Update subscribers

```
reaction(
        out(update(A,B,C,D,E,F,G)),
        (internal,completion),
        (
        current_time(Time1),

        C = D,
        text_term(Btext,B),
        text_term(Etext,E),
        text_term(Ftext,F),
        text_term(Gtext,G),

        current_tc(TC),
        TC ? in(update(A,B,C,D,E,F,G)),

        get_semanticKB(KB),
        KB<-getBase returns Base,
        KB<-getModel returns Model,

        java_object('coordination.UpdateAgentUtility',[Model,Base],
                MyUpdateUtility),

        MyUpdateUtility<-utilityUpdate(A,'subscribe') returns Iterator,

        memberAll(Iterator,ListPr), println('semReaction'(ListPr)),

        forall(member(Mem,ListPr),(
                text_concat(Base,'identifier',NameIdProperty),
                Model<-getDatatypeProperty(NameIdProperty) returns
                        IdProperty,
                Mem<-getProperty(IdProperty) returns IdStatement,
                IdStatement<-getObject returns IdObject,
                IdObject<-asLiteral returns IdLiteral,
                IdLiteral<-getString returns IdValue,

                Mem<-getLocalName returns CommunityName,

                out_tc(ComName @ IdValue,
                        updateAgent(update,Btext,Etext,Ftext,Gtext))
                )
        ),
        )
).
```

The reaction rule specifies that when an out of a subscribe tuple is made into the tuple centre, then the reference to the JAVA object representing the semantic knowledge base KB is used (the ← notation represent a call to a java module) to obtain the URI and the model Model of the ontology. We use an UpdateUtility java module to check if the policies allow us to subscribe the community Community to the patient Patient. MyUpdateUtility is a variable containing an UpdateUtility object constructed with the

model Model and URI of the ontology. Finally, the tuple add is sent to the Update Agent which inserts the new information in the knowledge base.

A.3 Search community

```
reaction(
        out(communitySearch(A,B,C,D,E)),
        (operation,completion),
        (
        current_tc(TC),
        TC ? in(communitySearch(A,B,C,D,E)),

        get_semanticKB(KB),
        KB<-getBase returns Base,
        KB<-getModel returns Model,

        java_object('coordination.UpdateAgentUtility',[Model,Base],
                MyUpdateUtility),

        MyUpdateUtility<-checkCommunityCriteria(D) returns Iterator,

        Iterator =_,

        text_term(Etext,E),
        text_concat(Base,Etext,LocalComNameIndividual),

        Model<-getIndividual(LocalComNameIndividual) returns
                IndividualLocalCom,

        MyUpdateUtility<-getCommunities(LocalIndividualCom, A) returns
                IteratorCom,

        memberAll(IteratorCom,ListPr), println('semReaction'(ListPr)),

        forall(member(Mem,ListPr),
        (
                text_concat(Base,'identifier',NameIdProperty),
                Model<-getDatatypeProperty(NameIdProperty) returns
                        IdProperty,
                Mem<-getProperty(IdProperty) returns IdStatement,
                IdStatement<-getObject returns IdObject,
                IdObject<-asLiteral returns IdLiteral,
                IdLiteral<-getString returns IdValue,

                Mem<-getLocalName returns ComName,

                out_tc(CommunityName @ IdValue,
                        communitySearch(E,B,C,D,ComName))
        )
        )
        )
).
```

The reaction rule specifies that when an out of a communitySearch tuple is made into the tuple centre, then the reference to the JAVA object representing the semantic knowledge base KB is used (the ← notation represent a call to a java module) to obtain the URI and the model

Model of the ontology. We use an UpdateUtility java module to verify if the knowledge base contains the community searched. MyUpdateUtility is a variable containing an UpdateUtility object constructed with the model Model and URI of the ontology. Now we calls the Java method chechCriteria(Criteria) in order to verify if the OntModel contains a Community Individual with the Criteria inserted. The Iterator represents the list of arguments of the Community searched. The content of Iterator discriminates the behaviour. If the content is null the communitySearch tuple (with the sender updated) is forwarded to all connected communities excluding the sender.

Appendix B

Java Code

B.1 Connection request

```
if(tupleArgumentName.equals("connect")){
  System.err.println("[Topology Agent] Connect tuple arrived");
  /*
   * a) from the tuple, the agent must retrieve the ID and the name of the community which
   *    the Actor (normally an administrator of the local community) wants as father
   *
   * b) it sends to the requested father the tuple with all the information about own community;
   * the template is the follow:
   * request(String:myIdentifier, String:myName, String:myAddress, int:myPolicies,
   *       String: fatherName)
   * P.S. the fatherName is an additional information for various reasons.
   *
   * c) the agent must wait an answer from the father;
   * The answer can be of two type
   * request accepted -> template: topologyAgent(accept,String:fatherID,
   *       String:fatherName, String:fatherAddress, int:fatherPolicy)
   * request rejected -> template: topologyAgent(reject)
   *
   * d) request accepted:
   *  1) create his father as Semantic Individual with class Community
   * 2) exit to the waitAnswer while
   */
  Individual ind = model.getIndividual(base+localTC);
  Statement idStatement = ind.getProperty(idProp);
  Statement nameStatement = ind.getProperty(nameProp);
  Statement addresstatement = ind.getProperty(addressProp);
  Statement policiesStatement = ind.getProperty(policiesProp);
  String id = idStatement.getString();
  String name = nameStatement.getString();
  String address = addresstatement.getString();
  int policies = policiesStatement.getInt();

  String fatherId = tuple.getArg(1).toString();
  String fatherName = tuple.getArg(2).toString();
```

```
    LogicTuple requestTuple = new LogicTuple("request", new Value(id),new Value(name),
                        new Value(address),new Value(policies),new Value(fatherName));

    externalId = new TucsonTupleCentreId(fatherName+"@"+fatherId);
    cnt.out(externalId, requestTuple, (Long) null);

    boolean waitAnswer = true;

    while(waitAnswer){
        LogicTuple topologyAnswer = new LogicTuple("topologyAgent",new Var("A"),
                            new Var("B"),new Var("C"),new Var("D"),new Var("E"),
                            new Var("F"),new Var("G"));
        tuple = cnt.in(tid, topologyAnswer, (Long)null);
        tupleArgumentName = tuple.getArg(0).getName();

        waitAnswer = false;

        if(tupleArgumentName.equals("accept")){

        id = tuple.getArg(1).toString();
        name = tuple.getArg(2).toString();
        address = tuple.getArg(3).toString();
        int father = 1;
        int children = 0;
        policies = tuple.getArg(4).intValue();

        String community = "semantic "+name+" : 'Community'(identifier : "+id+
                ", name : "+name+ ", address : "+address+", father : "+father+
                ", children : "+children+", policies : "+policies+")";

        LogicTuple getOntology = new LogicTuple("ontology", new Var("Ontologia"));
        LogicTuple res = cnt.getOntology(tid, getOntology, (Long)null);
        String ontology1 = res.getArg(0).getName();
        cnt.out(tid, community, (Long) null, ontology1);
    }else if(tupleArgumentName.equals("reject")){
        System.out.println("[TopologyAgent] Reject case not yet implemented");
    }
}
```

B.2 Jena Listener

```
/**
     * Listener over the statement's change
     * The interested change in this case is about documents of patients.
     *
     * @param com.hp.hpl.jena.rdf.model.Statement
     *
     */
    public void addedStatement(Statement arg0) {

        String base = tc.getSemKB().getBase();
        OntModel model = tc.getSemKB().getModel();

        Resource documentClass = model.getResource(base+"Document");
        Resource resourcePatient = model.getResource(base+"Patient");
        Resource resourceCommunity = model.getResource(base+"Community");

        OntProperty hasProp = model.getOntProperty(base+"has");
        OntProperty caresProp = model.getOntProperty(base+"cares");
        OntProperty hasHomeCommunityProp = model.getOntProperty(base+
            "hasHomeCommunity");

        Individual documentIndividual;
        Individual patientIndividual;
        Individual communityIndividual;

        boolean flag = false;

        /*
         * Retrieve all individuals present in the Class "Document"
         */
        ExtendedIterator<Individual> documentIndividualIterator =
        model.listIndividuals(documentClass);
        while(documentIndividualIterator.hasNext()&& !flag){
            documentIndividual = documentIndividualIterator.next();
            /*
             * We are interest only to a changes relates to a Document
             * So, here we check if the subject of the statement is the individual extracts
            from the iterator
             */
            if(documentIndividual.getLocalName().equals(arg0.getSubject().getLocalName())){
            /*
                 * Retrieve all individuals present in the Class "Patient"
                 */
                ExtendedIterator<Individual> itPatient = model.listIndividuals(resourcePatient);
                while(itPatient.hasNext()&& !flag){
                    patientIndividual = itPatient.next();
            /*
                     * Retrieve all Statements where a individual patient is the Subject of
            the property "hasProp"
                     */
                    StmtIterator iter = patientIndividual.listProperties(hasProp);
                    while(iter.hasNext()&& !flag){
                        /*
                         * We are interest only to a patient who has the document which changed
                         * So, here we check if the subject of the statement is an individual belonging
                                                                * to class Document
```

```
                    */
                    if(iter.next().getSubject().asResource().getLocalName()
                    .equals(patientIndividual.getLocalName())){
                        String patient = patientIndividual.getLocalName();
                        String document = documentIndividual.getLocalName();
                        String homeCommunity =
                        patientIndividual.getProperty(hasHomeCommunityProp)
                        .getObject().asNode().getLocalName();
                        ExtendedIterator<Individual> itCommunity =
                        model.listIndividuals(resourceCommunity);
                        while(itCommunity.hasNext()&& !flag){
                            communityIndividual = itCommunity.next();
                            StmtIterator iter2 = communityIndividual.listProperties(caresProp);
                            while(iter2.hasNext()&& !flag){
                                if(iter2.next().getObject().asResource().getLocalName()
                                .equals(patientIndividual.getLocalName())){
                                    flag = true;
                                    String caresCommunity = communityIndividual.getLocalName();
                                    LogicTuple lT = new LogicTuple("updateAgent"
                                    , new Value("updates"), new Value(patient),new Value(document)
                                    , new Value(homeCommunity), new Value(caresCommunity)
                                    , new Value("remove"));
                                    LogicTuple removed = (LogicTuple) tc.getVM()
                            .getRespectVMContext().readMatchingTuple(lT);
                                    if(removed == null){
                                        tc.getVM().getRespectVMContext().addTuple(lT);
                                    }
                                }
                            }
                        }

                    }
                }
            }
        }
    }
}
```

B.3 Persistence

```
public class SetModel {

    public static final String DB_URL = "jdbc:postgresql://localhost/persistence";
    public static final String DB_USER = "tucson";
    public static final String DB_PASSWD = "tucson";
    public static final String DB = "PostgreSQL";
    public static final String DB_DRIVER = "org.postgresql.Driver";

    // source URL to load data from; if null, use default
     private static String s_source;

     // if true, reload the data
     private static boolean s_reload = true;


    public static OntModel setModel(String path){

        // load the driver class
        try {
            Class.forName(DB_DRIVER);
        } catch (ClassNotFoundException e) {
             System.err.println( "Failed to load the driver for the database: " + e.getMessage() );
               System.err.println( "Have you got the CLASSPATH set correctly?" );
        }

        PersistentOntology po = new PersistentOntology();

          // we pass cleanDB=true to clear out existing models
          // NOTE: this will remove ALL Jena models from the named persistent store, so
          // use with care if you have existing data stored
          ModelMaker maker = po.getRDBMaker( DB_URL, DB_USER, DB_PASSWD, DB, true );

          // now load the source data into the newly cleaned db
          return po.loadDB( maker, path );

    }
}
```

```
public class LoadModel {

    public static final String DB_URL = "jdbc:postgresql://localhost/persistence";
    public static final String DB_USER = "tucson";
    public static final String DB_PASSWD = "tucson";
    public static final String DB = "PostgreSQL";
    public static final String DB_DRIVER = "org.postgresql.Driver";

    // source URL to load data from; if null, use default
     private static String s_source;

     // if true, reload the data
     private static boolean s_reload = true;


    public static OntModel loadModel(String path){

        // load the driver class
        try {
            Class.forName(DB_DRIVER);
        } catch (ClassNotFoundException e) {
             System.err.println( "Failed to load the driver for the database: " + e.getMessage() );
               System.err.println( "Have you got the CLASSPATH set correctly?" );
        }

        PersistentOntology po = new PersistentOntology();

          // we pass cleanDB=true to clear out existing models
          // NOTE: this will remove ALL Jena models from the named persistent store, so
          // use with care if you have existing data stored
          ModelMaker maker = po.getRDBMaker( DB_URL, DB_USER, DB_PASSWD, DB, false );

          return po.openDB(maker, path);
    }
}
```

Bibliography

[1] The description logic handbook. page 622, 2007.

[2] Andrea and Omicini. Formal respect in the a & a perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97 – 117, 2007. ¡ce:title¿Proceedings of the Fifth International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2006)¡/ce:title¿.

[3] apiCE Lab. Tucson guide. http://apice.unibo.it/xwiki/bin/download/TuCSoN/Documents/manual.pdf, 2006.

[4] F. Baader. Logic-based knowledge representation. In *Artificial Intelligence Today*, pages 13–41, 1999.

[5] F. Baader, R. Küsters, and F. Wolter. Extensions to description logics. In *Description Logic Handbook*, pages 219–261, 2003.

[6] F. Baader and W. Nutt. Basic description logics. In *Description Logic Handbook*, pages 43–95, 2003.

[7] P. Ciancarini. Coordination models and languages as software integrators. *ACM Comput. Surv.*, 28(2):300–302, June 1996.

[8] M. Colombetti. Dispense corso di Ingegneria della conoscenza: modelli semantici, Facolta di ingegneria dell informazione, Politecnico di Milano. http://home.dei.polimi.it//colombet/IC/materiale/IC2011.

[9] S. CT. Systematized Nomenclature of MEDicine Clinical Terms, 2011. `http://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html`.

[10] E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In D. Garlan and D. Le Mtayer, editors, *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 274–288. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63383-986.

[11] E. Denti, A. Omicini, and A. Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[12] E. Denti, A. Omicini, and A. Ricci. Multi-paradigm javaprolog integration in tuprolog. *Science of Computer Programming*, 57(2):217 – 250, 2005.

[13] R. H. Dolin, L. Alschuler, S. Boyer, and C. Beebe. An update on hl7s xml-based document representation standards. *Proceedings of the AMIA Symposium*, pages 190–194, 2000.

[14] R. H. Dolin, L. Alschuler, S. Boyer, C. Beebe, F. M. Behlen, P. V. Biron, and A. S. Shvo. Model formulation: Hl7 clinical document architecture, release 2. *JAMIA*, 13(1):30–39, 2006.

[15] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages (extended abstract). In *Description Logics*, pages 87–90, 1991.

[16] N. Elena. SEMANTIC COORDINATION THROUGH PROGRAMMABLE TUPLE SPACES. http://apice.unibo.it/xwiki/bin/download/Theses/NardiniPhdTheses/NardiniPhdThesis.pdf, 2011.

[17] G. Eysenbach. What is e-health? *J Med Internet Res*, 3(2):e20, Jun 2001.

[18] S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors. *Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters, WWW 2004, New York, NY, USA, May 17-20, 2004*. ACM, 2004.

[19] A. Geissbuhler, S. Spahni, A. Assimacopoulos, M. Raetzo, and G. Gobet. Design of a patient-centered, multi-institutional healthcare information network using peer-to-peer communication in a highly distributed architecture. *Medinfo*, 11(Pt 2):1048–52, 2004.

[20] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, Jan. 1985.

[21] D. T. Gunter and P. N. Terry. The emergence of national electronic health record architectures in the united states and australia: Models, costs, and questions. *J Med Internet Res*, 7(1):e3, Mar 2005.

[22] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[23] HL7, 2011. `http://www.hl7.org/`.

[24] IHE. Ihe iti cross couminity information exchange, 2008. `http://www.ihe.net/Technical_Framework/upload/IHE_ITI_TF_White_Paper_Cross_Community_2008-11-07.pdf`.

[25] IHE. Connectathon, 2011. `http://www.ihe.net/connectathon/`.

[26] IHE. Ihe integration profiles vol 1, 2011. `http://www.ihe.net/Technical_Framework/upload/IHE_ITI_TF_Rev8-0_Vol1_FT_2011-08-19.pdf`.

[27] LOINC. Logical Observation Identifiers Names and Codes, 2011. `http://loinc.org/`.

[28] S. MAZZOCCHI. Closed world vs. open world: the first semantic web battle. `http://www.betaversion.org/~stefano/linotype/news/91/`.

[29] D. Nardi and R. J. Brachman. An introduction to description logics. In *Description Logic Handbook*, pages 1–40, 2003.

[30] E. Nardini, M. Viroli, and E. Panzavolta. Coordination in open and dynamic environments with TuCSoN semantic tuple centres. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*.

[31] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277 – 294, 2001.

[32] A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2:251–269, 1999. 10.1023/A:1010060322135.

[33] R. Sojer, H. Muller, D. Aronsky, and P. Ruch. e-Heallth Semantic And Content For Switzerland, 2011.

[34] V. Urovi, A. C. Olivieri, S. Bromuri, M. I. Schumacher, and N. Fornara. An agent coordination framework for ihe based cross-community health record exchange, 2012.

[35] U. Visara. Semhealthcoord - first technical report. SemHealthCoo. HES-SO, 2012.

[36] F. Wozak, E. Ammenwerth, A. Hrbst, P. Sgner, R. Mair, and T. Schabetsberger. Ihe based interoperability - benefits and challenges. In S. K. Andersen, G. O. Klein, S. Schulz, and J. Aarts, editors, *MIE*, volume 136 of *Studies in Health Technology and Informatics*, pages 771–776. IOS Press, 2008.

Acknowledgements

Ho sempre pensato che i ringraziamenti dovessero essere rivolti solo verso chi ha contribuito in qualche maniera alla realizzazione della tesi, e resto convinto di questa idea.

Ma la tesi è la fine di un percorso, quindi, senza entrare troppo nel dettaglio, voglio spendere qualche parola anche per tutte le persone che mi hanno accompagnato a questo traguardo.

Vorrei partire ringraziando tutte le persone che ho avuto il piacere di conoscere durante questi splendidi anni universitari. Non farò nomi per paura di tralasciare qualcuno, e poi sareste in troppi, ma tutti quanti in un modo o nell'altro avete lasciato una traccia del vostro passaggio. Qualcuno se n'è andato, qualcun'altro lo farà, ma l'importante sono le persone che restano, e sono sicuro che alcuni di voi, soprattutto il gruppo UaUa, in un modo o nell'altro sarete sempre al mio fianco, e questo mi rende orgoglioso della persona che sono.

Ora vorrei fare un ringraziamento speciale alla mia famiglia, che è stata la più grande fortuna della mia vita. A partire dai miei fantastici nipoti, ho sempre sentito la presenza di tutti voi in qualsiasi fase della mia vita, grazie di cuore. Qui però quattro nomi in particolare li devo fare.
Parto ringraziando mio Babbo, che nonostante le difficoltà che ha dovuto affrontare, ha avuto il coraggio di tenere uniti e crescere sette figli, e nonostante i suoi metodi rudi e a volte scorbutici, ha fatto in modo che diventassimo sette persone fantastiche, che si supportano e si aiutano in ogni situazione. GRAZIE BABBO.
Poi voglio ringraziare mia sorella Barbara e mio cognato Leandro, che mi hanno accolto in casa loro, e mi hanno cresciuto come un figlio, anzi meglio. Penso che poche persone possano dire di avere avuto genitori al loro pari. Nel loro piccolo non mi hanno mai fatto mancare niente, a costo di rinunciare loro stessi a tante cose. Di sicuro non senza di voi la

mia vita sarebbe stata diversa, e non sarei mai diventato quello che sono ora. GRAZIE BABI E LEO.
Infine vorrei ringraziare mio fratello Giuseppe, che ho sempre visto come un esempio da seguire. Diciamo che la pacatezza e tranquillità che ha saputo trasmettermi hanno bilanciato la parte oscura presente in me. Hai sempre creduto in me, dandomi tutto il supporto possibile senza mai dubitare e senza che ti dovessi mai chiedere nulla. GRAZIE GIUSEPPE. Sono tutto tranne che un santo, ma ho un gruppo di angeli che hanno sempre rispettato la mia indipendenza, ma in punta di piedi hanno saputo rimettermi sul binario giusto ogni volta che imboccavo la strada sbagliata.

Ecco, ora che le lacrime, che non mi rigavano il viso da 12 anni, sono finite, voglio ringraziare le sei persone cha hanno avuto un ruolo attivo durante il periodo di svolgimento della mia tesi.

Parto ringraziando il mio Relatore, Andrea Omicini, una persona che ha creduto in me, dandomi la possibilità di fare questa bellissima esperienza all'estero, e con il quale è sempre bastata una semplice email di una riga per capirsi.
Poi devo fare un ringraziamento speciale a Michael I. Schumacher, che mi ha accolto a braccia aperte nel suo Team, non facendomi mai pesare il mio inglese un pò zoppicante.
Voglio fare un ringraziamento anche a Johannes per l'aiuto che mi ha dato in fase di Evaluation.
Un grande grazie a Karine, la prima persona che ho conosciuto in Svizzera, che dal primo momento mi ha fatto sentire a mio agio e mi ha aiutato in ogni ambito burocratico durante la mia permanenza.
Ora arriviamo a due persone che per un caso della vita ho ritrovato al Techno Pole dopo averci studiato insieme, e mi hanno fatto sentire come se fossi a casa. Inizio dal ringraziare Stefano, che ho scoperto essere un vero amico, con cui abbiamo condiviso tante passeggiate al lago a parlare di tante cose di cui molte volte non si parla neanche con persone che si conoscono da tanti anni. Il suo aiuto è stato fondamentale. A ogni dubbio o ostacolo bastava andare a porre a lui la questione, e come un Google umano, in pochi secondi ti forniva la soluzione. Infine ringrazio Visara, con cui ho collaborato in questo progetto. Una persona fantastica, una vera amica sempre disponibile, che mi ha aiutato fino all'ultimo istante (nel vero senso della parola) e che si prodigata per farmi fare una tesi di un livello al quale da solo non avrei mai potuto aspirare.

Anche questo capitolo della vita si è concluso, e un piccolo ringrazia-

mento lo faccio anche a me stesso, per la tenacia che ho sempre avuto nell'affrontare ogni situazione. Nella vita ho perso tante volte, e di sicuro perderò ancora, ma quando qualcosa dipende dalle mia capacità farò sempre di tutto per riuscire ad uscire da ogni situazione vittorioso.