Projecte de Final de Carrera

Enginyeria Informàtica

UNIVERSITAT
ROVIRA I VIRGILI

Agent-based execution of medical guidelines represented in the SDA* formalism

Author:
José Miguel Millan Rosa
josemiguel.millan@estudiants.urv.cat

Project Director:
Dr. Antonio Moreno Ribas

Escola Tècnica Superior d'Enginyeria (ETSE)
Universitat Rovira i Virgili (URV)
http://www.etse.urv.es

Grading Period: 2006-07

Table of contents

1. Introduction

This project is related to the design and implementation of an agent-based platform within the K4CARE project[1]. The work described in this document refers to a part of the work done by the University Rovira i Virgili in this European research project. This document is divided in 4 parts.

One of these parts is the integration of the medical workflows designed or generated for the project into the K4CARE's Multi-Agent System, also called clinical practice guidelines. This medical workflows are represented in a new formalism designed which will be explained in section 2. As it is still in development, the goal of this project is to design and implement the tools needed to use a first version of this format.
The product of this integration work is an Agent which will be in charge of the execution of all the structures represented with this formalism in the system. All the agents in the platform which need to execute one or more of these structures will invoke this agent in order to delegate to him all the tasks of interpreting and managing all the tasks related to this structure interpretation.

In the last part of this document, some tests over the generated code are presented in order to have a verification of the correct functionality of the written code. These tests are structured in some levels: the first level is centered in testing the data structure used to represent the medical guidelines, the second level tests the execution engine, and in the third level the interaction of this execution engine with the rest of the agents is evaluated.

2. Project Context Description. The K4CARE Project

[1]In eHealth it is increasingly necessary to develop tele-informatic applications to support people involved in providing basic medical care (physicians, nurses, patients, relatives, and citizens in general). The care of chronic and disabled patients involves life long treatment under continuous expert supervision. Moreover, healthcare workers and patients accept that being cared for in hospitals or residential facilities may be unnecessary and even counterproductive. From a global view, such patients may saturate national health services and increase health related costs. The debate over the crisis of financing healthcare is open and is a basic political issue for old and new EU member countries and could hinder European convergence.

To face these challenges we can differentiate medical assistance in health centres from assistance in a ubiquitous way (Home Care -HC- model); the latter can undoubtedly benefit from the introduction of ICT (Information and Communication Technologies).

2.1. What is K4CARE

The K4CARE Project is one ICT project that will develop a platform to manage the information needed to guarantee an ICT Home Care service. It will:

a) integrate information of different types and from different sources.
b) be integrated with ICT whilst ensuring private and customized data access.
c) use ontologies to define the profile of accessing subjects (e.g. physicians, patient) and objects (e.g. disease, case study).
d) have a mechanism to combine and refine the ontologies to personalize the system, taking into account the way a physician works and the individual patient characteristics.
e) incorporate 'know-how' from geriatric clinical guidelines as Intervention Plans (IP).
f) generate IPs from the healthcare centres databases if clinical guidelines do not exist or are inappropriate for a particular situation.
g) configure a knowledge-based decision support tool that can supply eServices to all subjects involved in the Home Care model.
h) extract evidence from real patients and integrate it with published evidence derived from RCTs.

The main objective of the K4CARE project is to improve the capabilities of the new EU society to manage and respond to the needs of the increasing number of senior population requiring a personalized HC assistance[1]. The project will capture and integrate the information, skills, expertises, and experiences of specialised centres and professionals of several old and new EU

1 - HC has to be properly addressed to the patients who can derive the higher benefit: the typical HC Patient (HCP) is an elderly patient, with co-morbid conditions and diseases, cognitive and/or physical impairment, functional loss from multiple disabilities, impaired self-dependency. We shall refer to this "average patient" as the HCP.

countries, and will incorporate them in an intelligent web platform in order to provide e-services to health professionals, patients, and citizens in general. To achieve this goal, the members of the project will provide the scientific and technical knowledge, develop the intelligent technologies to manage that knowledge, supply the ICT infrastructure for anticipating and hastening the medical assistance, implement a web-based platform to approach these technologies to healthcare professionals, patients, and citizens, and assess the platform services in a scenario of combined old and new EU healthcare institutions: HC services and related hospitals, rehabilitation centres, geriatric departments, and town councils.

The above main objective can be detailed in other more specific objectives. In the next lines general objectives, scientific objectives, and technological objectives are exposed separately. General objectives describe the aims of the project from a global perspective. Scientific and technological objectives concern the technical aspects of the project either from a CS, ICT, or medical perspective.

General objectives

O1. Generate a new ICT Sanitary Model (K4CARE model) for assisting HCPs in the enlarged Europe. The system will seamlessly integrate services, healthcare practices, and assistance knowledge coming from old (e.g. Italy, UK) and new (e.g. Czech Republic, Romania, Hungary) European countries.

O2. Propose a telematic and knowledge-based CS platform (K4CARE platform) that implements the above model. This platform will include all the technologies developed in the project and it will assist all the human actors involved in the care of HCPs. These actors include physicians, nurses, social workers, rehabilitative professionals, patient relatives, patients themselves, and citizens in general.

O3. The platform will be tested on west (Italy, UK) and east (Czech Rep., Romania, Hungary) EU societies through pilot tests in order to highlight their differences and also to pursue a convergence to a homogeneous way-of-doing, contributing to a unique European Healthcare ICT society in HC.

O4. The K4CARE platform can serve as a means of integrating knowledge about HCPs assistance all over the new and old EU countries. Some healthcare centres from Italy, UK, Czech Republic, Romania and Hungary will work to demonstrate that sharing this sort of knowledge across EU countries is not only possible but also beneficial and necessary for achieving an European standard HC service supported by the new technologies.

Scientific Objectives (Integrating information)

O5. The project will define a solution for Electronic Health Record (EHR) incorporating lessons learned in past experiences (e.g. I4C/TripleC, PROREC and Provenance projects), and exploiting the knowledge of the consortium about standards within this field. The defined EHR will be implemented and used to store information about HC. This EHR will integrate different data

types (e.g. text, numerical values, multimedia parts) and documents coming from different sources (e.g. hospital services, laboratories, consultations, specialists, relatives and patients at home).
O6. Within the project, the cooperating healthcare partners will pre-process information about physicians, patients, citizens and other agents involved in the K4CARE model and will fill in the EHR with it in order to have a test bench with real data. The EHR will integrate information coming from different EU member countries (homogenising the differences) and will be under continuous evaluation and adaptation deriving from the specialised partners indications.

Scientific Objectives (Knowledge representation)
O7. Define the Actor Profile Ontologies (APO) for representing the profiles of the subjects involved in the K4CARE model: healthcare professionals, patients and relatives, citizens, and social organisms. APOs contain the skills, concerns, aspirations, etc. of the people that they represent, together with the healthcare services that those people offer to or receive from the K4CARE model (e.g. medical services such as drug prescription, clinical consultations, laboratory analysis; social services such as counselling, information , advice, social support).
O8. Define the patient-Case Profile Ontologies (CPO) for representing symptoms, diseases, syndromes, case mix. APOs and CPOs describe "know-what" knowledge about agents accessing the K4CARE model, and "pure" pathologies the K4CARE model gives support to, respectively. These ontologies are based on terms related to symptoms, signs, drugs, medical and surgical procedures, dietetic conditions, physical and hygienic requirements, cognitive functions, self-dependency, etc.
O9. Define Formal Intervention Plans (FIP) for a number of disease and syndrome treatments. These FIPs will be generated from the information deriving from the available evidence-based clinical practice guidelines that represent standards of practice, particularly in the fields of geriatrics. These FIPs represent the professional worldwide existing "know-how" knowledge within the K4CARE platform, and they will guide the services the system offers to the professional users. In other words, FIPs are the explicit expressions of how HC must be provided in a growing ageing EU.

Technological Objectives (Knowledge adaptation and use)
O10. Personalise the access to the K4CARE platform. Adapt the APOs to the user requirements in order to customize the access to the EHR and the assistance provided by the K4CARE model, because not all the patients wish their care in the same way (in terms of services, treatment, explanation, terminology, etc.). The developed personalising methods will also be applied to the care-givers ontologies in order to represent the fact that not all the professionals interact with the healthcare system in the same way. The particularized ontologies and the interface technologies, together with usability and security aspects, will play a crucial role in the acceptance and socialization of the K4CARE platform.

O11. Personalise the assistance to senior citizens. CPOs as they stand are not valid in real practice since a HCP has a combination of features which makes his/her treatment different from any other treatment. Developed technologies for merging prototypic CPOs will be used to have CPOs adjusted to the individual condition of the patient.
O12. FIPs will be inductively learned from the EHR with the use of new machine learning techniques. These techniques must be developed and tested in the domain of HCPs. They are learned from the procedures regarding past patients stored in the system.

Technological Objectives (Service Supplying)
O13. Design and implement intelligent agents that allow users to access the EHR, edit, adapt, and merge ontologies, and introduce and induce FIPs. Combine these intelligent agents in a multi-agent system that provides e-services to care-givers, patients and citizens (e.g. scheduling of prolonged clinical treatments, intelligent decision support, intelligent distribution of data among users). Deliver those services through the Internet and the mobile telephony in a safe, everywhere, anytime way.
O14. Develop an application that will be integrated in the K4CARE platform for localizing patients topographically. This is particularly relevant for some sort of patients with memory or cognitive impairments and also in orderto anticipate the medical and care actions.

2.2. The K4CARE Model

In this section the whole project model[02] is described. This model contains a description of the different actors, services, etc. which conform the system. First the model will be described, and in the next sections its different elements will be explained:

- The system actors.
- The professional actions and liabilities.
- The services and procedures.
- The information documents.

2.2.1. The Model

The K4CARE model provides a paradigm easily adoptable in any of the EU countries to project an efficient model of HC[03].

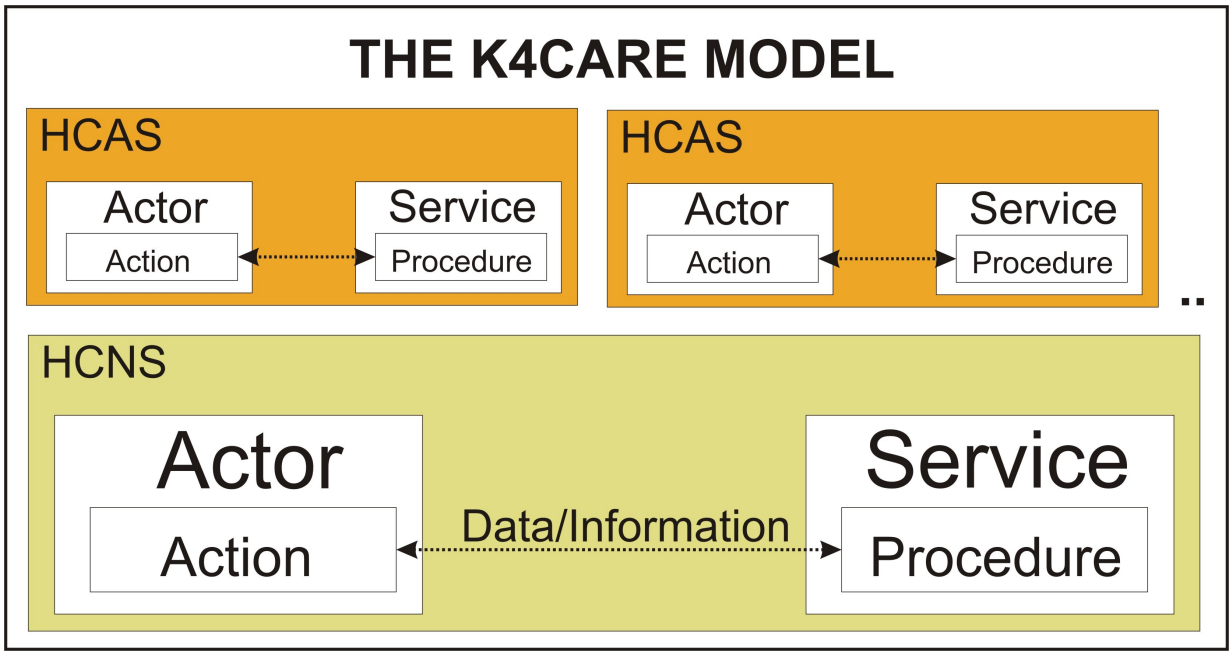


Figure 1. The K4CARE Model Architecture for HC

In the model, services are distributed by local health units and integrated with the social services of municipalities, and eventually with other organizations of care or social support. The model is aimed at providing the patient with the necessary sanitary and social support to be treated at home. To accomplish this duty, the K4CARE model gives priority to the support of the HCP, his relatives and Family Doctors (FD) as well. Because of its aim, the model is represented by a *modular structure* that can be adapted to different local opportunities and needs. The success of this model is directly related to the levels of *efficacy, effectiveness and best practice* of the health-care services the model is able to support. As shown in Figure 1, the K4CARE Model is based on a *nuclear structure (HCNS)* which comprises the minimum number of common elements needed to provide a basic HC service. The HCNS can be extended with an optional number of *accessory services (HCAS)* that can be modularly added to the nuclear structure. These services will respond to specialized cares, specific needs, opportunities, means, etc. The distinction between the HCNS and the complementary HCASs must be interpreted as a way of introducing flexibility and adaptability in the K4CARE model. Going into detail, each one of the HC structures (i.e. HCNS and HCAS's) consists of the same components: *Actors* are all the sort of human figures included in the structure of HC; *Professional Actions and Liabilities* are the actions each actor performs to provide a service within the HC structure; *Services* are all the utilities provided by the HC structure for the care of the HCP; *Procedures* are the chain of events that leads an actor in performing actions to provide services; *Information* are the *documents*

required and produced by the actors to provide services in the HC structure. As new HCASs are incorporated to the K4CARE Model, new actors, actions, services, procedures and information enter to be part of the extended model. In this way, the K4CARE model is compatible both with the current situation in the European countries where the international, national, and regional laws define different HC systems for different countries, and also with the forthcoming expected situation in which a European model for HC was decided.

2.2.2. Actors

In HC there are several people interacting: patients, relatives, physicians, social assistants, nurses, rehabilitation professionals, informal care givers, citizens, social organisms, etc. These individuals are the members of three different *groups of HC actors*: the *patient*; the *stable members* of HCNS (the family doctor, the physician in charge of HC, the head nurse, the nurse, the social worker, each of them present in the HCNS); the *additional care givers*.

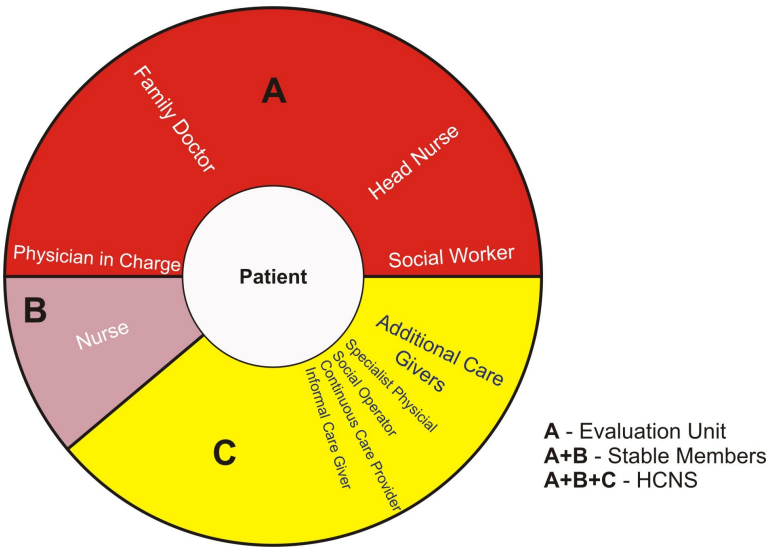


Figure 2. Actors in the Home Care Nuclear Structure (HCNS)

The family doctor, the physician in charge of HC, the head nurse, and the social worker join in a temporary structure – the *Evaluation Unit* – devoted to assess the patient's problems and needs, to decide the treatment (by constructing the *Individual Intervention Plan* – IIP – based on one or more FIPs) and to monitor its progress. The patient (i.e. the HCP) is in the centre of the HCNS of the K4CARE model (see Fig.2), and the rest of the groups are organised around it as a symbol of a patient-oriented HC model.

2.2.3. Professional Actions and Liabilities

These represent general actions that each one of the actors in the K4CARE model performs in his duties within the HCNS service. Two lists of actions are provided for each sort of actor: the list of general actions, and the list of HCNS actions. The list of general actions is intended to contain all the actions that actors are expected to perform in a general purpose Home Care System. The list of HCNS actions complements the explanation with the specific actions the K4CARE Model define for the actors involved in the HCNS. The HCNS actions are grouped following the standard below:

- P.xx – Patient Actions
- BO.xx – Back Office Actions
- EU.xx – Evaluation Unit Actions
- M.xx – Medical Actions
- M.FM.xx – Medical Actions performed by the Family Doctor
- M.SP.xx – Medical Actions performed by the Specialist Physician
- N.xx – Nursing Actions
- CM.xx – Case Management Actions
- S.xx – Social Actions

HCNS actions are those required to accomplish the procedures that implement the care services of the HCNS. Any action represents a professional activity for which the actor (or group of actors) is liable. In the next tables some action examples are presented:

P.1	confirm appointment
P.2	agree on interventions
P.3	give consent
P.4	request certification
P.5	ask intervention
BO.9	supervise HCP information
BO.1	provide information
BO.2	ask information

Table 1. HCNS Patient list of actions

BO.1	provide information
BO.2	ask information
BO.4	assign actor
EU.1	evaluate through scales
EU.2	define intervention plan
EU.3	define outcomes
EU.4	schedule controls
EU.5	schedule re-evaluations
M.1	perform Clinical Assessment
M.2	perform Physical Examination
M.3	request Diagnostic Procedures
M.4	request Laboratory Analysis
M.5	prescribe Pharmacological Treatment
M.6	prescribe non-Pharmacological Treatment

Table 2. Summarized HCNS Physician in Charge list of actions

2.3. Services and Procedures

The HCNS provides a set of services for the care of HCP. These services are classified into Access services, Patient Care services, and Information services.

Access services see the actors of the HCNS as elements of the K4CARE model and they address issues like patient's admission and discharge from the HC model. *Patient Care services* are the most complex services of the HC model by considering all the levels of care of the patient as part of the HCNS. Finally, *Information services* cover the needs of information that the HCNS actors require in the K4CARE model. Examples of very relevant services are: the *Comprehensive Assessment* (which is the service devoted to detect the whole series of HCPs diseases, conditions, and difficulties, from both the medical and social perspectives), the *Intervention Plan Definition* (which represents the course of actions to be performed in order to provide care to the HCP in terms of treatment and support) and the *Intervention Plan Performance* (which defines the execution of a previously defined IP). In the K4CARE Model a *procedure* represents *the way that the actions provided by/to the actors are combined* to accomplish one service. The following table summarizes some of the services and procedures in the K4CARE:

A. ACCESS SERVICES

1. Individual Services

i. HC Request

ii. HCP Admission

iii. HCP Discharge

iv. Professional Admission

v. Professional Discharge

vi. Edit HCP/Professional information

2. Structural Services

i. EU Constitution

ii. EU-HCP Binding

...

Table 3. Summarized K4CARE HCNS Services

HC Request

(a) The FD makes a *HC Request* demand HC for a particular HCP.

HCP Admission

(a) PC and HN consider the pertinence and the relevance of information reported in the request
(b) In case of non sufficient information the PC and HN ask the FD to integrate the request
(c) In case of non pertinent or non relevant request, the PC and HN reject the request
(d) PC and HN admit the patient to the HC, if the information is sufficient

HCP Discharge

(a) The HCP is discharged in case of: reaching of outcomes (as defined by the IIP); refusal of services; moving to other place than house; death. The discharge is done by PC and HN.
(b) The service can be suspended, in case of temporary admission to hospital. For the time of admission. Suspension is done by PC and HN.

Professional Admission

(a) Defined actors are admitted in the system at the time of first assignment to a service. The admission is done by the PC and HN.
(b) The actor accepts admission.

Professional Discharge

(a) An actor with a professional profile is discharged after a defined period subsequent the last participation to a service. It is done by the PC and HN.

Edit HCP/Professional information

(a) The change of administrative data concerning the actors is done by the PC and HN

Table 4. K4CARE Procedures in Individual Services from Access Services

2.3.1. Information Documents

The HCNS structure defines a set of information units whose main purpose is to provide information about the care processes realized by the actors to accomplish service. Different kinds of actors will be supplied with specific information that will help them to carry out their duties in the K4CARE model. All these data are considered here to be part of documents. Different types of document have been classified as follows:

- *Documents in Access Services*: the information required in each one of the K4CARE access services;
- *Documents in Patient Care Services*: the support to actors taking part of the patient care services. Since these documents may have different general purposes inside the sets of services and procedures, they can be sub-divided into request documents, authorization documents, prescription documents, and anamnestic documents;
- *Documents in Information Services*: in order to support the information services a list of documents is defined. In general, information service documents report on underlying activities or on officially recognized information, related to HC. A special service is represented by the possibility of exchanging messages among actors.

Nevertheless, a classification of the documents according to the patient care services is also provided, in order to relate documents to the patient care services in which they are used.

In the next table, some documents in the K4CARE are listed. The rights of the actors to read or write these documents are also depicted in the table:

Name	Code	EVALUATION UNIT																	
		FAMILY DOCTOR		PHYSICIAN IN CHARGE		HEAD NURSE		SOCIAL WORKER		NURSE		SPECIALIST PHYSICIAN		SOCIAL OPERATOR		CONTINUOUS CARE PROVIDER		PATIENT	
HC Request Document	AD01	W			R		R		R										
HCP Admission Document	AD02		R	W	R	W	R		R										
HCP Discharge Document	AD03		R	W	R	W	R		R		R		R		R		R		R
Professional Admission Document	AD04			W	R	W	R		R*		R*		R*		R*				
Professional Discharge Document	AD05			W	R	W	R		R*		R*		R*		R*				
EU Constitution Document	AD06		R		R	W	R		R										

* According to competencies; W: authorization to write and modify; R: authorization to read.

Table 5. Documents in K4CARE Access Services

2.4. K4CARE Partners

As one can deduce, the K4CARE Project is being developed by a group of member entities each one of them specialised on its field. The next table is presentes alist of these participants:

Participant no.	Name	Participant Organization short name
1 (coordinator)	Universitat Rovira i Virgili	URV
2	Centro Assistenza Domiciliare Azienda Sanitaria Locale RM B	CAD
3	Czech Technical University in Prague	CVUT
4	University of Perugia	UNIPG
5	Telecom Italia S.p.A.	TI
6	European Research and Project Office GmbH	EURICE
7	Ana Aslan International Academy of Aging	ANA
8	Fondazione Santa Lucia	FSL
9	Computer and Automation Research Institute of the Hungarian Academy of Sciences	MTA SZTAKI
10	The Research Institute for the Care of the Elderly	RICE
11	Amministrazione Comunale di Pollenza	COMPOL
12	General University Hospital in Prague	GUH
13	Szent Janos Hospital	SJH

Table 6. The K4CARE partners

These partners are divided in three groups: the administration group, the technical group and the medical group.

- The administration group is composed by the partners which are implied in tasks of coordination and administration of the whole project. The participants who belong to this group are CVUT, CAD, EURICE and URV.

- The technical group consists of the members which are involved in the technical elements of the project. This means the design, development and testing of the different technical components in the project, and, also the connection of these tools with the medical knowledge. The entities involved in this task are TI, MTA SZTAKI, CVUT and URV.

- The medical group is the one which is in charge of the different medical issues in the

project. This means that the partners from this group are in charge of formalising and providing their knowledge to the technical group members, and also, to supervise and feedback the new knowledge and tools. The members of this group are SJH, GUH, RICE, COMPOL, FSL, ANA, UNIPG and CAD.

2.5. URV work in the K4CARE

As it can be inferred the University Rovira i Virgili has some tasks into this project. These tasks are mainly related to the administrative and to the technical parts of the project. In the case of the administrative tasks the URV has to:

- Coordinate the work of the different partners in the project.
- Be in charge of writing the documentation related to this coordination.
- Administrate its internal works.

The list of technical tasks is, in fact, longer than the administrative one because the University's staff which is involved in this project are mainly IT Engineers and Doctors. So the tasks of the URV in the technical part ofthe project are the following:

- Develop a formalism to represent the medical guidelines which appear in the project and all the needed techniques to work with this formalism.
- Design and develop the required ontologies to represent the medical knowledge related to the HomeCare.
- Develop techniques of tailoring for each actor in the system.
- Design and implement part of the Multi-Agent System (MAS) which will be the final K4CARE platform system.
- To research about the most useful techniques and products in the market that would be useful in all the mentioned tasks.

The work reported in this document is centered in the 13th K4CARE's objective which is the design and implementation of an intelligent agent platform. Concretely this work consists in the design and implementation of the medical workflows execution engine. The following chapters will describe this work, so the next chapter describes the medical guidelines formalism, the SDA*. In chapter 4 the K4CARE's Multi-Agent Platform will be described, and finally, the chapter 5 will introduce to the reader the design and implementation of an agent-embeddable execution engine.

3. The SDA* Model

One of the K4CARE's objectives is to develop a new formalism to represent clinical guidelines. Clinical practice guidelines, [13] are collections of practical information for use by doctors and other medical professionals. Often, these are gleaned from systematic review of medical journals and other published material. They are a prime tool for evidence based medicine, and require frequent updating as new information becomes available.

There exist some formats to represent guidelines like ASBRU [04][19][21], PROforma [04][20][21], etc, but K4CARE uses a new formalism [04] designed by Dr. David Riaño which is presented here.

3.1. Introduction

In the K4CARE project, procedures, formal intervention plans and individual intervention plans are the basic structures to represent health care procedural knowledge (or know how). In this setting, a procedure is described as an implementation of a health care service by means of a combination of actions. For example, the steps that configure a blood analysis, or the health care activities involved in a comprehensive assessment.

Formal Intervention Plans (FIPs) are defined as formal structures representing the healthcare procedures to assist patients suffering form particular ailments or diseases. They contain indications to all the actors involved in the care process (i.e. healthcare professionals, patients and relatives, etc.) in order to provide the best coordinated and effective action plan. A FIP on hypertension, for instance, provides the indications of how to act with a hypertensive patient in general. FIPs are general structures that have to be adapted to the particularities of a patient before it is actionable and applicable to this patient. In the K4CARE project the structure resulting from this adaptation is called Individual Intervention Plan (IIP).

In the K4CARE Project, these three structures are used to:

- Represent the professional worldwide existing "know-how" knowledge within the K4CARE platform.
- Guide the K4CARE services the system offers to the professional users
- Make explicit the way Home Care (HC) must be provided in a growing ageing EU
- Offer a knowledge representation frame in which the new machine learning techniques developed in the project make explicit the knowledge about HC interventions implicit in the Electronic Health Care Record (these are learned from the procedures regarding past patients stored in the system);
- Offer a representation frame in which procedural knowledge about “pure” pathologies

can be integrated in complex or co-morbid pathologies;

- personalize the care to particular patients, taking into account their specific characteristics;
- develop a family of FIPs representing procedural knowledge about the treatments of the syndromes targeted in the K4CARE project;
- adapt to a common representation several clinical guidelines already published by international healthcare organizations as the National Library of Medicine and the National Guideline Clearinghouse in the USA, the New Zealand Guidelines Group, the Scottish SIGN, etc.;
- use knowledge engineering methods to create new formal representations for conditions and diseases relevant in the project that do not have any trustable treatment published or known. These FIPs will integrate the experiences in the treatment of such cases by all the healthcare partners of the K4CARE consortium.

3.2. The SDA* Model: Syntax and Semantics

SDA[04] stands for State-Decision-Action, SDA* (SDA star) represents the repetition of states, decisions and actions in order to describe health care procedural knowledge as, for example, K4CARE procedures, FIPs, or IIPs. In the SDA* model, states are used to describe patient conditions, situations, or statuses that deserve a particular course of actions which is totally or partially different from the actions to be followed when the patient is in other state. It provides a response to the fact that a disease, ailment, pathology, or syndrome can present alternative degrees of evolution whose treatment must be distinguished. Decisions in the SDA* model capture the need of procedural knowledge to represent alternative options whose selection depends on the available information about the patient. In this sense, decisions are able to unify in a single representation of the procedural knowledge alternative courses of actions that have to be applied to patients that meet different conditions. Unlike states, decisions are not intended to make the degree of evolution of a disease explicit, but to orientate a general purpose treatment to the particular characteristics of the patient; for example in the treatment of hypertension, high-blood-pressure is a patient condition that may deserve a special treatment and, therefore, if should be represented as a state. On the contrary, in the treatment of cardiac insufficiency, the patient condition high-blood-pressure provides information which is relevant to adapt the treatment, but not to decide on the treatment as a whole, which is based on other conditions as structural-heart- disease or prior-heart-problems. So in cardiac insufficiency, high-blood-pressure should be taken as a decision. Finally, actions are the proper treatment steps in the SDA* procedural knowledge that are performed according to the preceding decisions.

States, decisions, and actions are combined to form a joined representation of how to deal with a particular health care situation (e.g. a therapy). For example, Figure 2 depicts the FIP that was published by the Institute for Clinical Systems Improvement (www.icsi.org) to diagnose and treat hypertension. It is based on the following indications:

I. Patients in the FIP can be in four alternative states:

a) Screening and identification of elevated BP in patients with diabetes, chronic kidney disease, heart failure, or CAD (FIP element #1).
b) Initial assessment completed; i.e. evaluated, accurately staged, and complete risk assessed (FIP element #3).
c) Hypertension is suspected to be caused by secondary causes (FIP element #5).
d) Hypertension is under control and a continuing care must start (FIP element #12).

II. The process is based on three yes-no decisions (one of them appearing twice in the FIP):

a) Is a second cause of hypertension suspected (FIP element #4)?
b) Is the blood pressure at goal; i.e. within normality limits (FIP elements #7 and #9)?
c) Is it a resistant hypertension; i.e. have we fail to achieve a normal BP despite the use of a rational triple-drug regimen in optimal doses (FIP element #10)?

III. The actions proposed for the diagnosis and treatment are:

a) Confirm hypertension on the initial visit, plus two follow-up visits with at least two BP measures at each visit; following standardized BP measurement techniques, including out of office or home blood pressure measurements (FIP element #2).
b) Consider a thiazide-type diuretic as initial therapy in most patients with uncomplicated hypertension (FIP element #6).
c) For many patients, two or more drugs in combination may be needed to reach hypertension goals (FIP element #8).
d) Refer to hypertension consultation (FIP element #11)

In the next subsections the SDA* model is formally introduced, followed by the explaination of how sequences and cycles are made in the model, what non-determinisms the model is able to deal with, and the temporal model which is beneath the SDA* model.

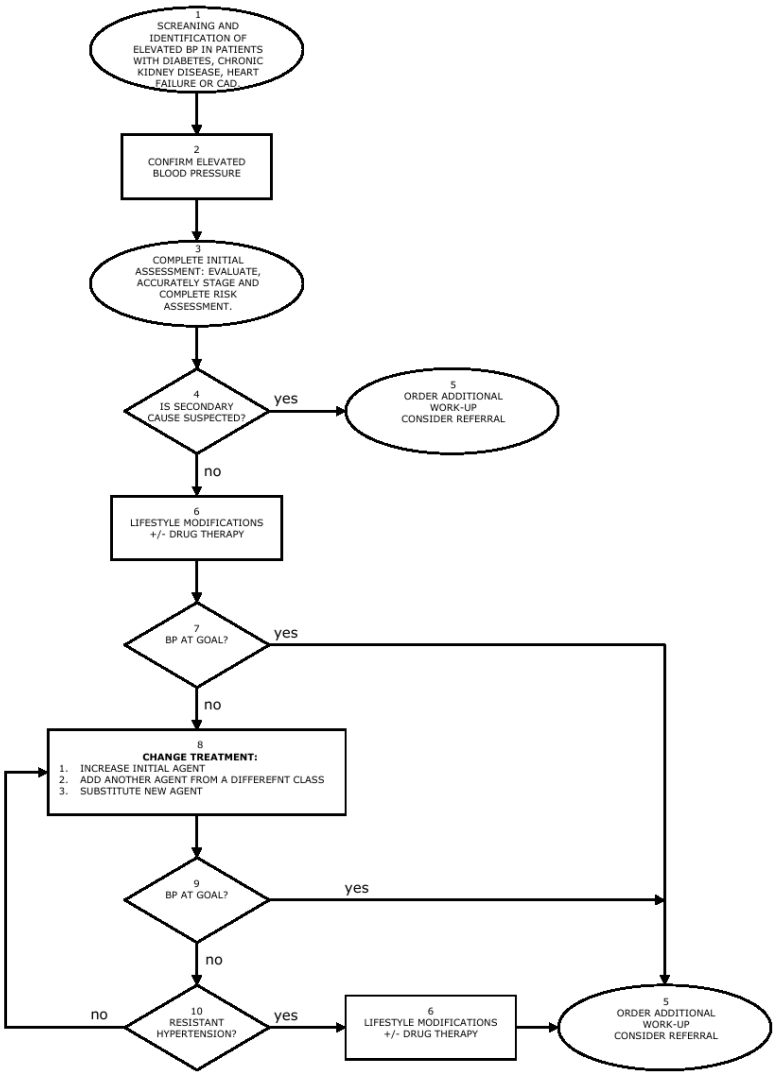


Figure 2. FIP on hypertension diagnosis and treatment

3.2.1. Formal description

The SDA* model is introduced to represent knowledge on procedural activities in health care. In the next sections, the SDA* model is described in terms of the domain terms, the elements, and the connections that describe the health care procedure that is being formalized.

3.2.1.1. The Universe of Discourse

Given D a particular disease, ailment, pathology, or syndrome, a finite set of **terms** $V_D=\{v1, ..., vn\}$ within the medical domain of D is defined to represent any descriptive or procedural health care knowledge on D. For example, the terms in the hypertension treatment contained in Figure 1

are seventeen: *screening-and-identification-of-elevated-BP, diabetes, chronic-kidney-disease, heart-failure, CAD, confirm-elevated-blood-pressure, complete-initial-assessment, secondary-cause-suspected, additional-work-up, consider-referral, life-style-modifications, drug-therapy, BP-at-goal, change-treatment, resistant-HT, HT-consult, and HT-continuing-care.*

Some of these terms are defined as **state terms** (i.e. $S_D \subseteq V_D$ is the set of state terms). State terms represent facts that are useful to determine the condition of the patient in the process the SDA* model is describing. In the SDA* model, a **patient condition** contains all the terms observed for the patient in a particular moment (i.e. signs, symptoms, antecedents, taking drugs, secondary diseases, test results, etc.), therefore it is a subset of the set of terms V_D.

The set of **decision terms** $D_D \subseteq V_D$ is the set of all the terms in V_D that may be required by medical experts to choose among alternative medical, surgical, clinical, or management actions within the treatment of the disease D that the procedure globally describes. State and decision terms may be used to define any patient condition possible in D.

The set of **action terms** $A_D \subseteq V_D$ is the set of all the terms that represent the medical, surgical, clinical, or management actions that a doctor may decide on a patient in the course of the treatment of that patient's disease or health care procedure.

Though these three sets are not necessarily mutually disjoint, they together must contain all the feasible terms in D, i.e. $V_D = S_D \cup D_D \cup A_D$. For example, in the above mentioned case of hypertension, Shypertension = {*screening-and-identification-of-elevated-BP, diabetes, chronic-kidney- disease, heart-failure, CAD, complete-initial-assessment, secondary-cause-suspected, BP-at-goal, HT-continuing-care*}, Dhypertension = {*secondary-cause-suspected, BP-at-goal, resistant-HT},* and Ahypertension = {*confirm-elevated-blood-pressure, additional-work-up, consider-referral, life-style-modifications, drug-therapy, change-treatment, HT-consult*} would be the set of state variables, decision variables, and action terms, respectively. Observe that the underlined terms are state and decision terms simultaneously.

In the first version of the SDA* model the universe of discourse is based on a set of the primitive medical terms that may be used to construct states, decisions, and actions. In forthcoming versions of the SDA* model, the universe of discourse will be extended to include Boolean variables (i.e. variables that are allowed to have two values: TRUE or FALSE), and later multi-valued variables (i.e. variables that can take one out of several possible values). This means that in this first version the health condition of a patient is defined exclusively by all the signs and symtoms this patient has.

3.2.1.2. Elements

The set of terms V_D is used to define the three basic elements of the SDA* model: states, decisions, and actions. Formally speaking, a state s is a subset of state terms (i.e. $s \in \wp(S_D)$); a decision d is based on a subset D of decision terms (i.e. $D \in \wp(D_D)$) and it is defined as a finite

list <*D; D1, D2, ..., Dk*>, such that $Di \in \wp(D)$ is a decision alternative (or branch), $D=D_1 \cup D_2 \cup \ldots \cup D_k$, and $k \geq 0$ is the branching factor of the decision. An action a is a subset of action terms (i.e. $a \in \wp(A_D)$).

From the point of view of semantics, a state (or SDA* entry point) describes an abstract patient condition in which all the terms in the state hold. For example, the state {diabetes, complete-initial-assessment} represents all the patients with both diabetes and a complete initial assessment, but which may also have other possible features. From a logical point of view, a state is a conjunction of state terms. From a functional point of view, the states of a SDA* procedure are the entry points to that procedure or, in other words, the points where the treatment described can start.

If $C \subseteq (S_D \cup D_D)$ is the current condition of a patient, we say a state s of a SDA* procedure is a feasible entry point of that patient in that procedure if and only if $s \subseteq C$. It may happen that one patient has several feasible entry points for the same SDA* under the same condition. It may also happen that one or several states are included in other states of the same SDA* (e.g. $s_1 \subseteq s_2$). In this case, every time s_1 is a feasible entry point, s_2 is also a feasible entry point. Empty states are also possible and they represent states in the SDA* procedure that any patient meets. Observe that a state s that does not contain a state term v will be a feasible entry point to both patients whose condition comprises v and patients whose condition does not comprise v (see Table 7). If we want to change this behavior we have to define two terms for the same health care concept, one being the negation of the other one (e.g. diabetes and not-diabetes). This way, a state containing the term not-diabetes (i.e. negation of diabetes) will not be a feasible entry point for diabetic patients whose condition does not comprise not-diabetes.

	$v \in$ *PATIENT CONDITION*	$v \notin$ *PATIENT CONDITION*
$v \in s$	s is a feasible entry point	s is NOT a feasible entry point
$v \notin s$	s is a feasible entry point	s is a feasible entry point

Table 7. Basic logic rule of feasible entry points

A decision (or SDA* branching point) describes a point of the SDA* where the treatment can follow alternative courses of action depending on which are the decision terms the treated patient meets. For example, the set of decision terms *D = {stage1-HT, stage2-HT, low-BP, BP-at-goal}*could be used to propose alternative treatments whether the patient is hypertensive (*D1 = {stage1-HT, stage2-HT}* $\subseteq D$), hypotensive (D_2=*{low-BP}* $\subseteq D$) or none (D_3=*{BP-at-goal}* $\subseteq D-(D_1 \cup D_2)$). From a logical point of view, a decision represents a disjunction of conjunctions on a set of decision terms. From a functional point of view, decisions allow the represented SDA* to be as general and flexible as to combine several variations on the treatment of a disease, and to make the application of these variations depend on the particularities of the patient.

If $C \subseteq (S_D \cup D_D)$ is the current condition of a patient and $d=< D; D_1, D_2, ..., D_k >$ a decision element of a SDA* procedure, we say D_i is a feasible branch for that patient if and only if $D_i \subseteq C$. One or several branches may contain none decision variable; in this case, all these branches are feasible. It may also happen that in the same decision two or more branches totally or partially overlap. In the first case (i.e. $D_i \subseteq D_j$), D_i will be a feasible branch whenever D_j is feasible, and D_j will not be a feasible branch if Di is not. In the second case (i.e. $(D_i \cap D_j) \neq \emptyset$) each situation must be studied separately. Observe that if $D_i = D_j$, both branches are evaluated the same for any possible patient. Empty conditions are always feasible.

Concerning the branching factor k of a decision, it must be zero, one, or greater than one. An SDA* decision with a branching factor of zero or one is interpreted as unfinished element, maybe because at the time of developing of the SDA* there is not health evidence on how to branch patients at that point of care. A branching factor k=0 transforms a SDA* decision into a dead end element. A branching factor k=1 acts as a filter of the patients that may proceed with the treatment at the decision point. A branching factor k=2, allows the construction of SDA* binary decisions as <D; D_1, D-D_1>. Observe that binary decisions are not equivalent to IF-THEN-ELSE structures since any patient with a condition C containing all the decision terms in D will make both branches of the above decision feasible. Like in the case of the state elements, IF-THEN-ELSE behaviors may be achieved through the definition of contrary terms (e.g. diabetes and not-diabetes), and the definition of decisions as *<{diabetes, not-diabetes}; {diabetes}, {not-diabetes}>*. In this case, diabetic patients will follow the first branch, and non diabetic patients the second one.

An alternative interpretation of a decision $d=< D; D_1, D_2, ..., D_k >$ is that it is based on a "fictitious" variable d whose domain (i.e. the values that the variable can take) is D, and each branch Di is a subset of these possible values. For example, *BP=<{stage1-HT, stage2-HT, low-BP, BP-at-goal}; {stage1-HT, stage2-HT}, {low-BP}, {BP-at-goal}>*.

Let us observe that a branch $D_i=\emptyset$ of a decision d is always feasible for any patient arriving to d. If a patient condition C includes none of the branches of a SDA* decision $< D; D_1, D_2, ..., D_k >$ (i.e. $D_i \not\subset C$ *for all i=1, 2, ..., k*), then none of the branches is feasible, and the decision becomes a dead end element of the SDA* procedure for all the patients under that condition. In order to avoid this situation a SDA* decision can contain an otherwise branch (i.e. $< D; D_1, ..., D_k$, *otherwise*>) which is feasible only if the patient condition C makes none of other branches feasible. For example, *BP= <{ BP-at-goal }; { BP-at-goal }, otherwise>* that represent the decisions #7 and #9 in the FIP of Figure 1.

An action element (or SDA* action block) describes a group of actions in the SDA* procedure. These elements do only represent action proposals whose application must be seen out of the SDA* model. So, if the SDA* suggests the physician to prescribe a beta-blocker it is up to the physician to decide whether the drug is finally prescribed or not, and it is up to the patient (or some other person) to make sure that the patient takes the drug. This means that two sequential actions in the SDA* model do not necessarily represent a sequential execution of the actions in

the real world, but consecutive action proposals within the SDA* procedure.

The SDA* model does not distinguish between instant actions (i.e. those actions with an immediate end as for example an expert recommendation) and abiding actions (i.e. those actions which extend in time as for example starting an assessment process that may last several days). The reason is that actions in the SDA* model represent the launch of the action, regardless whether this is an instant or an abiding action in the real world. Typical sorts of actions are: recommendations (e.g. stop-smoking, start-soft-exercise, avoid-salt-in-meals, etc.); prescriptions; radiographies; analyses; medical, surgical or clinical procedures; specialist consultations; application of an alternative SDA* procedures, etc. From a functional point of view, action blocks represent the core elements of the SDA* model since the final purpose of this model is to represent health care procedures as a combination of actions.

Each action term in an action element has two constraints: the first one (called the set of petitioners) is on the sort of actors that are allowed to request the action (e.g. only medical doctors are allowed to prescribe drugs). The second one (called the set of performers) is on the sort of actors that are allowed to perform the action in the real world (e.g. injecting some drugs can be restricted to nurses and to medical doctors, but some other drugs can also be injected by the own patient or some relative). These constraints on the actions permit the description of collaborative medical treatments in which several professionals may interact. Any petitioner in the set of petitioners is allowed to requests the action to be executed. Any performer in the set of performers is allowed to execute the action.

Action blocks are independent of the patient condition; therefore they use to be preceded either by a state that describes what the state of a patient should be in order to deserve that action, or by a decision that determines whether the patient meets the features required for the action to be applied. Empty action blocks have the meaning of "do nothing", which is the same as not having the action block in the SDA*.

Flowcharts are used to represent SDA* procedures in a graphical way. Figure 3 shows how states, decisions, and actions are represented in this sort of flowcharts.

Figure 3. Elements of the SDA model*

The correct combination of states, decisions, and actions allows the construction of explicit health care procedural knowledge within the SDA* model. This combination of elements is made by means of connectors.

3.2.1.3. Connectors

This section explains how the SDA* elements introduced in the previous section can be combined to form proper health care procedures.

In the SDA* model, a connector is defined as an arrow that goes from one element in the input of the connector (or in-element) to another element in the output of the connector (or out- element). From the point of view of the SDA* elements, any state is an in-element of one connector, but it may be an out-element of any number of connectors in the FIP (including none). Decisions are in-elements of as many connectors as the branching factor of the decision, and out-elements of one or several connectors in the SDA* procedure. Finally, actions are in-elements of one only connector, and out-elements of at least one connector. These restrictions are graphically shown in Figure 4.

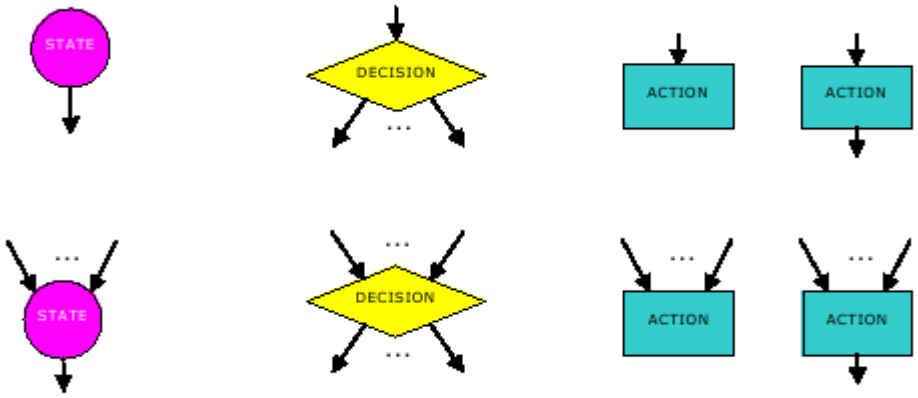


Figure 4. Feasible element connections in the SDA Model.*

In that figure, the up-left state and the bottom-left state describe a situation in which all the patients whose condition makes the state a feasible entry point evolve following the outgoing connector. The difference between them is that in the first case the SDA* procedure do not inform about when a patient can reach that state in the middle of a treatment (i.e. it is an input state of the health care procedure). In the second case, the state can be either an input state of the health care procedure for new incoming patients, or an intermediate state which is reached after the application of any of the elements in the incoming connections of the state.

A decision was defined as a list $< D; D_1, D_2, \ldots, D_k >$ of sets of decision terms; D being all the possible terms in the decision, D_i a subset of D for all i=1..k, and k the branching factor. Each alternative D_i in the decision is assigned a different outgoing connector of the decision. The meaning of a decision point is that any patient reaching the decision (by one of the incoming connectors) may follow any of the outgoing connectors whose Di is contained in the patient condition (i.e. one of the feasible branches of the decision).

An action block contains all the action terms that are to be suggested to deal with the patient reaching that element. The up-right actions in Figure 5 describe types of action blocks that are only reachable from one element in the SDA*. Within this group, the left one describes a terminal action in which the information of how to proceed after the action is not provided by the health care procedure. The actions in the bottom are general cases describing action blocks to follow after the application of any of the elements in the action incoming connectors. They also act as a joint of several courses of action of the SDA* procedure that converge to stop (action on the left) or that converge into one single action block to propose the same group of actions and then proceed in the same way through the action block outgoing connector (action on the right).

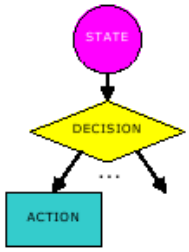


Figure 5. SDA Sequence

3.2.1.4. Sequences and cycles

The basic structure of the SDA* model is the SDA sequence that connects one state with a decision and each branch of that decision with an action. Figure 5 represents this basic structure.
The SDA sequence can be simplified with the elimination of one or several of the elements in the sequence. So, the elimination of the state must be interpreted as if there is not a health care reason to describe the state of the patient at this point of care (e.g. lack of medical meaning, medical irrelevance, cause of confusion, disagreement, etc.). Sometimes, the application of a set of actions is mandatory for all the patients arriving to the SDA sequence. In this case the decision element is eliminated and only one action block with all the common actions is connected after the state. Sometimes, a decision element is not enough to arrive to a conclusion about the sort of actions to carry on or the representation of all the possibilities with a single decision is confusing. In these cases the action block must be eliminated from the SDA sequence in order to chain several decisions. All these cases of SDA sequence reduction are depicted at the top of Figure 6.
At the bottom of Figure 6 the cases of elimination of two elements of a SDA sequence are represented. The left side case describes a situation in which two (or more) states from consecutive SDA sequences are connected. Although this is a correct sequence, there is not a clear reason that justifies it since a sequence of states is equivalent to a single state containing the state terms of all the states in the sequence. The case in the middle represents a sequence that connects two decisions. This is a common practice in the construction of health care procedures with the SDA* model. The last case in the bottom-right side represents a sequence of two (or

more) actions of consecutive SDA sequences directly connected. Like it happened with the states in the first case, this sequence is better replaced by a single action containing all the action terms of the action blocks involved in the sequence.

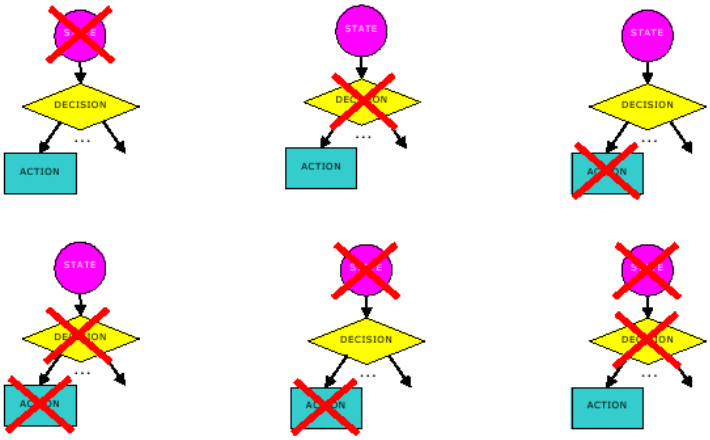


Figure 6. Simplified SDA Sequences

SDA sequences (and their reductions) can be concatenated by means of connectors. Figure 7 shows the most general case of a SDA sequence concatenation where none of the elements in the SDA sequences have been eliminated.

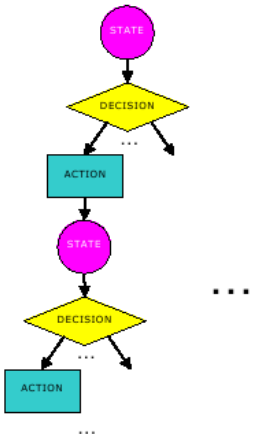


Figure 7. Concatenation of SDA Sequences

Apart of sequences, the SDA* model can represent cycles. A cycle is defined as a repeated sequence of elements in a SDA* procedure. Cycles may be used to represent repetitions in a medical process or jumps to an already previously observed situation in the course of action followed. Cycles in this model do not have explicit termination conditions; the exit of a cycle occurs when one of the decisions of the cycle drives the patient to an outgoing connection which is not part of the cycle.

3.2.1.5. Non-determinism

Determinism is the principle by which every event, act, and decision (effect) is the consequence of some antecedents (causes). In healthcare, these causes can be medical, surgical, genetic, environmental, managerial, familiar, social, etc. On the contrary, non-determinism states that there are events which do not correspond to a cause. Historically, there have been defined three types of non-determinisms: one that holds that some events are uncaused (e.g. from a practical point of view, in healthcare, uncaused events are equivalent to events with an unknown unfindable cause), another one that holds that there are nondeterministically caused events (e.g. a physician that follows alternative therapies for equivalent cases without an explicit explanation), and the third one that holds that there are agent-caused events (e.g. external events like the arrival of a patient whose health condition allows the treatment to start at different points). The SDA* model can deal with all the above types of non-determinism. As a consequence of this, for the same situation (i.e. patient condition) a non-deterministic SDA* is able to represent several different interventions with no support to decide which one should be followed. A non-deterministic SDA* procedure may propose more than one intervention and it must be the physician the final responsible of the selection.

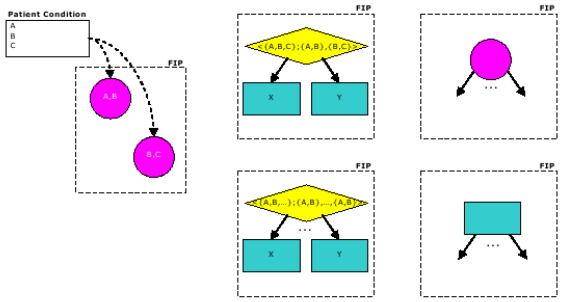


Figure 8. Non-Determinism in the SDA Model*

Figure 8 shows the three sorts of non-determinism in the SDA* model that can be observed in a SDA* FIP. From left to right, the first case *(type-0 non-determinism)* describes the situation in which a patient with a particular condition can match several states at the same time and therefore be non-deterministically recommended to start one out of several alternative interventions. In the second case *(type-1 non-determinism)*, the current condition of a patient can satisfy several branches of the same decision, and therefore be able to follow any of them. For example, if the patient condition is {high-blood-pressure, taking-drugs}, then any branch of the sort *{}, {high-blood-pressure}, {taking-drugs}, or {high-blood-pressure, taking-drugs}* is a feasible branch. The last case in the right side of Figure 8 *(type-2 non-determinism)* describes a situation in which either a state or an action in the FIP are in-elements of several connectors. Here, the SDA* procedure introduces two or more alternative paths that patients going out of these elements may (or may not) non-deterministically follow.

3.2.1.6. Time

The time model establishes two sorts of temporal constraints: those which are related to the terms in a SDA* element and those others related to the connectors. Each term and connector may optionally have one constraint or not. The time constraints of the terms are of the sort ***[start, end, frequency]*** and they mean that the term is observed from the start time, to the end time with the frequency indicated. For example, when *v = (antidepressant, [3w, 1d, 24h])* is a state term it means that the state of the patient is conditioned by the fact that "(with respect to the current moment) he has been taking one antidepressant every day since three weeks ago to one day ago". Observe that taking two antidepressant units should be said *(twoAntidepressant, [x, y, 24h])* or *(antidepressant, [x, y, 12h])* if the units are taken together or in two doses, respectively. The first case can also be represented by introducing the term v in the state two times.
If *v* is a decision term the meaning is equivalent to the question "has the patient been daily taking one antidepressant between three weeks ago and yesterday?". But if it is an action term the meaning is an order of taking that antidepressant starting in the start time and ending in the end time with the frequency indicated in the frequency value (i.e. a prescription). In this case, start must be a nearer to the current time than end.

s	Seconds
m	Minutes
h	Hours
d	Days
w	Weeks
M	Months
y	Years

Table 8. Time units in the SDA model*

In the SDA* model, this sort of terms with a time constraint are called temporal terms.
The second sort of time constraints in the SDA* model is related to the SDA* connectors and it has the form **[min, max]**. They are optional and represent delays (or durations). Both values are also optional. A connector with such time constraint indicates that the evolution from the in-element to the out-element of the connector takes between *min* and *max* times. If only the *min* value is present, it means that the connector can be crossed only if a *min* interval of time passes. If only the *max* value is in the constraint, the meaning is that the connector can be followed not later than a *max* interval of time.

The sort of temporal units of the start, end, frequency, *min*, and *max* components of the time

constraints are the ones included in Table 8 and any of these values is represented by a natural number followed by one of these temporal units (Other temporal concept as *"now"*, *"birth-time"*, and *"death-time"* are also possible). For example, *15s* for *"fifteen seconds"*, *5m* for *"five minutes"*, *3h* for *"three hours"*, *4d* for *"four days"*, *7w* for *"seven weeks"*, *10M* for *"ten months"*, and *3y* for *"three years"*.

States and decisions describe past or current aspects of the patient, and therefore the temporal constraint of start must be bigger than the temporal constraint of the end (e.g. *[3d, 2d]* or *[1y, 3w]*). On the contrary, action elements represent future actions and the start value of a temporal constraint must be smaller than the temporal constraint of the end (e.g. *[1d, 3y]* or *[1h, 6d]*).

3.2.1.7. Parallelism

Parallelism is admitted by the SDA* model but in a ***patient-oriented*** (instead of a procedure-oriented) fashion. From the point of view of the patient following a SDA* procedure, this person has a single treatment in which several evens may concur in time. In this approach parallelism does not mean that the patient is following several treatments at the same time (this will be a procedure-oriented approach), but that the actions of the treatmentoverlap in time.

This idea must be conceived together with the fact that SDA* procedures do not represent the health care procedures themselves, but the indications of what health actions have to be started now and, expectedly, in the future. Parallel to the SDA* procedure, the evolution of the real patient in the real world is what conditions how to apply the SDA* procedure in the next encounter with the patient. In other words, the SDA* procedure suggests a set of actions according to the current patient condition, and provides a farther perspective of how the treatment of this patient should be in the future, based exclusively on the limited current evidence provided by the current state of the patient and not on the real future evolution of the patient. Of course, this perspective is founded on both health care knowledge and experiences about the feasible evolutions of patients in the disease the SDA* procedure is dealing with.

In this context, all the action terms of an action block are launched in parallel, subject to their respective temporal constraints. In Figure 9 the action terms A_i and A_j, belonging to the same action block, have a parallel region where both behave simultaneously on the patient. These actions may also be in parallel to actions as Ak from other action blocks, as the figure also depicts.

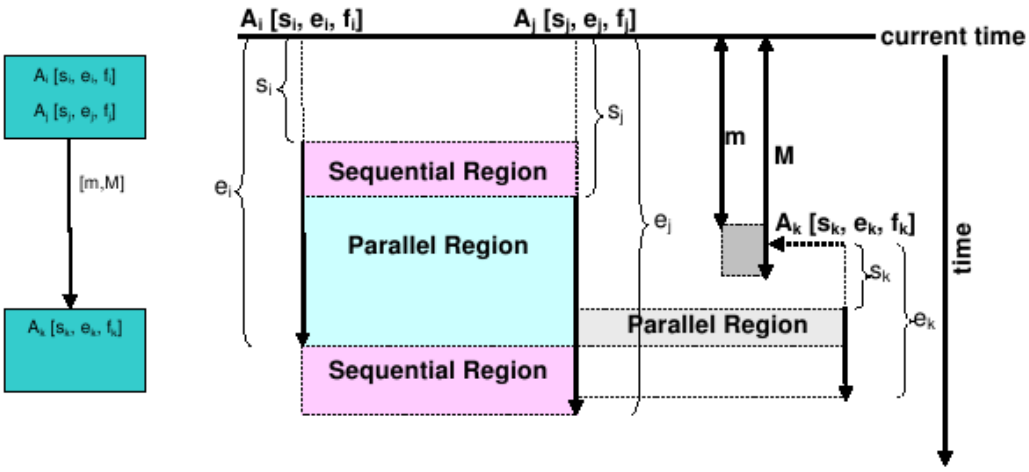


Figure 9. Parallel actions in time

3.3. Construction and execution of health procedures with the SDA* Model

This section introduces the procedures that the SDA* model is able to describe as an abstract data type (ADT), providing the specification of a basic functional interface to manage the construction of such health procedures. An XML Schema is proposed that provides the structure to represent SDA* procedures as XML documents. The ADT functions are used to describe the execution of a health procedure under the SDA* model. The section finishes with several examples of SDA* procedures in the K4CARE project.

3.3.1. Abstract data type SDA* procedure

This section aims at providing a formal proposal about the ***basic constructors*** that any system capable of defining SDA* procedures is recommended to have. This proposal follows the definitional notation of formal specification of abstract data types.

Time & Actors	TIME = λ | NUMBER{ s | m | h | d | w | M | y } PETITIONERS = Set of ACTOR PERFORMERS = Set of ACTOR
elements	EmptyState: → STATE InsertTerm: Term × [TIME]3 × STATE → STATE EmptyBranch: → BRANCH OtherwiseBranch: → BRANCH InsertTerm: Term × [TIME]3 × BRANCH → BRANCH EmptyDecision: → DECISION

	InsertBranch: BRANCH × DECISION → DECISION EmptyAction: → ACTION InsertTerm: Term × [TIME]3 × PETITIONERS × PERFORMERS × ACTION → ACTION
*SDA**	EmptySDA*: → SDA* InsetElement: Element × SDA* → SDA* InsertConnector: {STATE | ACTION}2 × [TIME]2 × Element × SDA* → SDA* InsertConnector: BRANCH × DECISION × [TIME]2 × Element × SDA* → SDA*

Table 9. SDA Abstract data type: basic constructors*

Patient states, branches, and actions are sets containing temporal terms of the form (term, [$time_1$], [$time_2$], [$time_3$]), any of the three times being optional; *decisions* are sets of *branches*, and SDA* are sets that contain either *states*, *decisions*, *actions*, or *connectors*, where connectors can be elements of the form (sa_1, sa_2, [$time_1$], [$time_2$]) or (branch, decision, [$time_1$], [$time_2$]), sa_i standing for a *state* or an *action*.

3.3.2. Textual representation of the SDA* procedures

The procedures of the SDA* model can be expressed in textual format. Table 10 shows the body of the XML Schema to define SDA* procedures as XML files.

```
...
   <xs:simpleType name="sda_time">
     <xs:restriction base="xs:string">
       <xs:pattern value="[0-9]*[smhdwMy]"/>
     </xs:restriction>
   </xs:simpleType>
   <xs:complexType name="sda_term">
     <xs:sequence>
       <xs:element name="start" type="sda_time" minOccurs="0"/>
       <xs:element name="end" type="sda_time" minOccurs="0"/>
       <xs:element name="frequency" type="sda_time" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="sda_actionterm">
     <xs:sequence>
       <xs:element name="start" type="sda_time" minOccurs="0"/>
       <xs:element name="end" type="sda_time" minOccurs="0"/>
       <xs:element name="frequency" type="sda_time" minOccurs="0"/>
       <xs:element name="petitioner" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
       <xs:element name="performer" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
     <xs:attribute name="name" type="xs:string" use="required"/>
     <xs:attribute name="type" type="xs:string"/>
   </xs:complexType>
   <xs:complexType name="sda_connector">
     <xs:sequence>
```

```
      <xs:element name="min" type="xs:time" minOccurs="0"/>
      <xs:element name="max" type="xs:time" minOccurs="0"/>
      <xs:element name="element" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="sda_branch">
    <xs:sequence>
      <xs:element name="sda_term" type="sda_term" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="sda_connector" type="sda_connector"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="sda_state">
    <xs:sequence>
      <xs:element name="sda_term" type="sda_term" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="next" type="sda_connector" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="sda_decision">
    <xs:sequence>
      <xs:element name="sda_branch" type="sda_branch" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="otherwise" type="sda_connector" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="sda_actionblock">
    <xs:sequence>
                    <xs:element   name="sda_action"   type="sda_actionterm"   minOccurs="0"
maxOccurs="unbounded"/>
      <xs:element name="next" type="sda_connector" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="sda_procedure">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="sda_state" type="sda_state"/>
      <xs:element name="sda_decision" type="sda_decision"/>
      <xs:element name="sda_action" type="sda_actionblock"/>
    </xs:choice>
  </xs:complexType>
...
```

Table 10. SDA XML Schema*

3.3.3. Execution of SDA* procedures

One of the key aspect to fully understand procedures described under the SDA* model is to know how they are executed for a particular patient. Before going on, we must recall that these procedures are formal representations of the general intervention to deal with a particular disease, ailment, pathology or syndrome. They are not representations of the evolutions of the patients under such circumstances. This means that applying the indications of a SDA* to a particular patient does not necessarily imply that this patient will evolve the way the SDA* indicates. This fact describes a parallel view of the problem at ***two levels***, the level of the course of actions indicated in the SDA* (medical knowledge) and the level of the evolution of the patients that

follow the SDA* (reality or medical data).
This duality may disturb or confuse the reader. However, this same reader must think of the SDA* as indications that a particular patient may follow, may not follow, follow in part, or follow during not enough time; or, also common in medicine, even strictly following the indications, the patient may evolve unexpectedly.
The practice of medicine is universally based on the encounters between the patients and the healthcare professionals. In a particular ***encounter*** a patient exhibits a specific *condition* within the disease he is assisted for. The role of the health care professional in the encounter is to interpret these signs, symptoms, and the rest of the information provided in order to conclude about the set of *actions* to follow (e.g. recommendation, prescription, procedure, etc.).
In the SDA* model, a ***patient condition*** is described as a set of temporal terms representing the patient current condition, including the patient health antecedents. These terms can be state, decision or action terms. For example, {(ElevatedBloodPressure), (BPAtGoal, [t_1,t_2,-]), (Antidepressant, [t_3, t_4, t_5])} is the condition of a patient that has an elevated blood pressure (i.e. BP≥140/90), but who has had the pressure at goal between the times t_1 and t_2, and who has been taking antidepressant between t_3 and t_4 with a frequency t_5.

Patient condition	EmptyCondition: → PCONDITION InsertTerm: Term × [TIME]^3 × PCONDITION → PCONDITION

Table 11. Patient condition abstract datatype: basic constructors

Given a patient condition and a SDA*, both based on the same set of terms, the execution of that SDA*procedure for that patient starts at any of the states of the SDA* that are feasible entry points. If it is required, the health care professionals in the encounter can select a subset of all the alternatives. Each feasible entry point in the selected set starts alternative feasible treatments of the patient.
A treatment consists of all the action (temporal) terms found in a SDA* **path that starts in a feasible entry point and finishes either in a state that is not a feasible entry point (i.e. The patient condition must change before evolving in this line) or in a connector with a temporal range [min, max] with min>0 (i.e. the SDA* procedure sets a temporal break before the patient treatment can continue)**. This path is a sequence of SDA* elements, each one being an out-element of a connector with in-element the pervious element in the sequence. If the in-element is a decision, all the branches that the patient condition meets (or the branch otherwise if none meets), i.e. the feasible branch of the decision, can be followed. If an element is non-deterministically connected to other elements all the non-deterministic connectors can be followed. In both cases, it is the health care professional who selects the treatment to apply among all the feasible alternatives supported by the SDA*procedure.

We define a temporal term (v,[t_1,t_2,t_3]) is equally or more restrictive than another temporal term (v,[t_1',t_2',t_3']) if the following conditions hold:

1. (t_1'≤t_1) or (t_1' is void).
2. (t_2≤t_2') or (t_2' is void) or (t_2' = 0).
3. (t_3≤t_3') or (t_3 is void)

Given a patient condition, we say a SDA* state is a feasible state if all the terms in the state can be found in the patient condition and they are equally or more restrictive in the state than in the patient condition. In a similar way, we say a branch of a SDA* decision is feasible if all the terms in the branch are in the patient condition contains and they are equally or more restrictive in the branch than in the patient condition.

For example, a SDA* state *{[beta-blocker, 1M, 1w, 1d]}* (i.e. patients taking one beta-blocker per day during the last month until last week) will be feasible for patients that meet condition c1 in Table 11 (more restrictive), but not feasible for patients meeting conditions c2 (the patient has been taking the drug since more time than one month ago), c3 (the patient has been taking beta-blockers during the last week, just in contradiction with the indication of stop taking beta-blockers one week ago pointed out by the state), or c4 (not only the frequency but the duration of medication is shorter).

Provided a decision term [*highBloodPressure, 1M, 3d,* -] (i.e. true if the patient has got high blood

pressure since one month ago till recently), all the patients meeting conditions c5 or c6 in Table 12 (more restrictive) will evaluate the decision term to true, but not c7.

Condition	**Expression**	**Meaning**
c1	*[beta-blocker, 1y, 1d, 12h]*	***Taking a beta-blocker every twelve hours since one year ago till yesterday.***
c2	*[beta-blocker, -, 2d, 8h]*	***Taking beta-blocker every eight hours till two days ago (since much time ago).***
c3	*[beta-blocker, 1w, -, 8h]*	***Taking beta-blocker every eight hours since one week ago (till now).***
c4	*[beta-blocker, -, -, 3d]*	***Taking beta-blocker every eight hours since much time ago till now.***
c5	*[highBloodPressure, 2M, 1d, -]*	***Till yesterday, Blood Pressure has been high during the last two months.***
c6	*[highBloodPressure, 2M, -, -]*	***Blood Pressure has been high during the last two months (and it is now).***
c7	*[highBloodPressure, 3w, -, -]*	***Blood Pressure has been high during the last three weeks (and it is now).***

Table 12. Some examples of temporal terms and meaning

3.3.4. Examples

This section contains partial and complete examples of SDA* procedures. The action terms have been classified into recommendations, prescriptions, radiographies, analyses, procedures, specialists, FIPs, and any, as Table 13 summarizes.

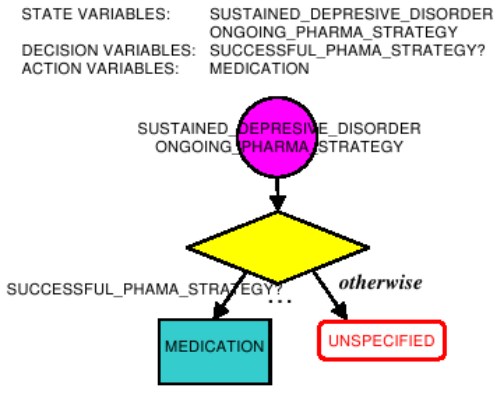
[RECOMMENDATION]	[REC]	Variable that represents a recommendation of the physician.
[PRESCRIPTION]	[PRES]	Variable that represents a drug prescription.
[RADIOGRAPHY]	[RAD]	Variable that represents an order of radiography.
[ANALYSIS]	[ANA]	Variable that represents an order of analysis.
[PROCEDURE]	[PROC]	Variable that represents the application of a procedure.
[SPECIALIST]	[SPEC]	Variable that represents the specialist the patient is derived to.
[FIP]	[FIP]	Variable that represents the execution of another FIP.
[ANY]	[ANY]	Variable that represents any sort of action.

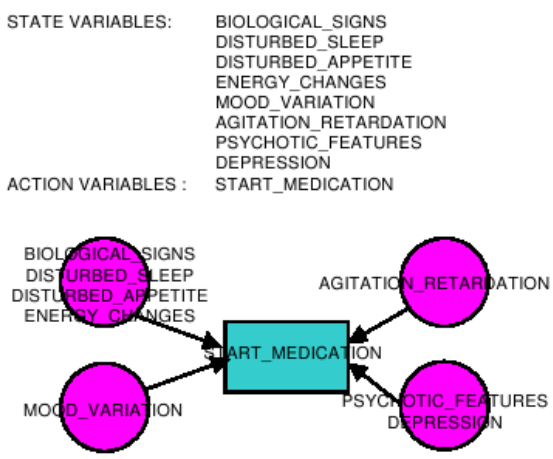
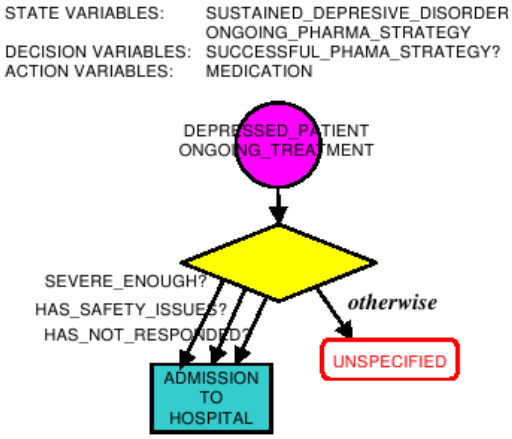
Table 13. Sorts of action variables

The treatment of hypertension in 3.3.4.2 is taken from the guideline published by the Institute for Clinical Systems Improvement (www.icsi.org) in the National Guideline Clearinghouse in the USA. The rest of procedures were taken from the "Consensus Guidelines for Assessment and Management of Depression in the Elderly" of the NSW Health Department in Australia. An exception is presented in subsection 3.3.4.3, where the K4CARE procedure for Comprehensive Assessment of homecare patients is represented following the SDA* model.

3.3.4.1. Representing partial knowledge

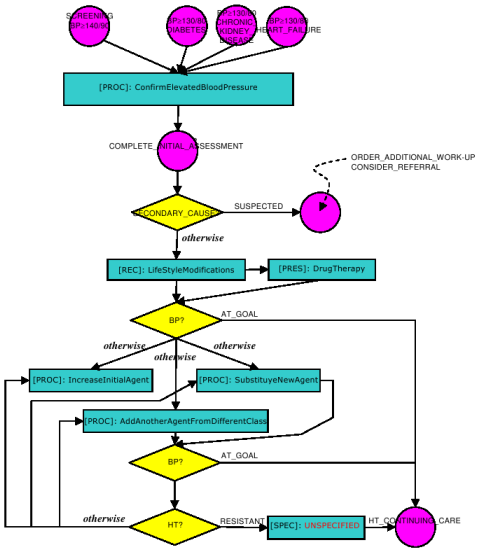
Many times clinical practice guidelines contain valuable knowledge that is apparently disconnected from other pieces of knowledge that may appear in the same guideline. This kind of knowledge is usually represented as text. This section contains some examples of textual knowledge pieces extracted from real guidelines and it shows how they could be represented in the SDA* model.

"Medication is likely to be needed where there is any sustained depressive disorder and when non-pharmacological strategies are not achieving their goals"	

"Useful signs to indicate commencing medication are: • *Presence of biological signs, disturbed sleep, appetite and energy changes* • *Diurnal variation in mood* • *Agitation or retardation* • *Depression with any psychotic features."*	
"Admission to hospital can be essential where the depression: • *Is severe enough to impair reasonable daily living function and supports cannot be put in practice* • *Has safety issues –suicidal ideas or plans, psychotic signs, severe psychomotor agitation otherwise or retardation* • *Has not responded to fair treatment"*	

3.3.4.2. CSI's Hypertension Diagnosis and Treatment

The Clinical Algorithm provided by the Institute for Clinical Systems Improvement (ICSI) that represents the processes of diagnosis and treatment of hypertension is translated to the SDA* notation. The result is:



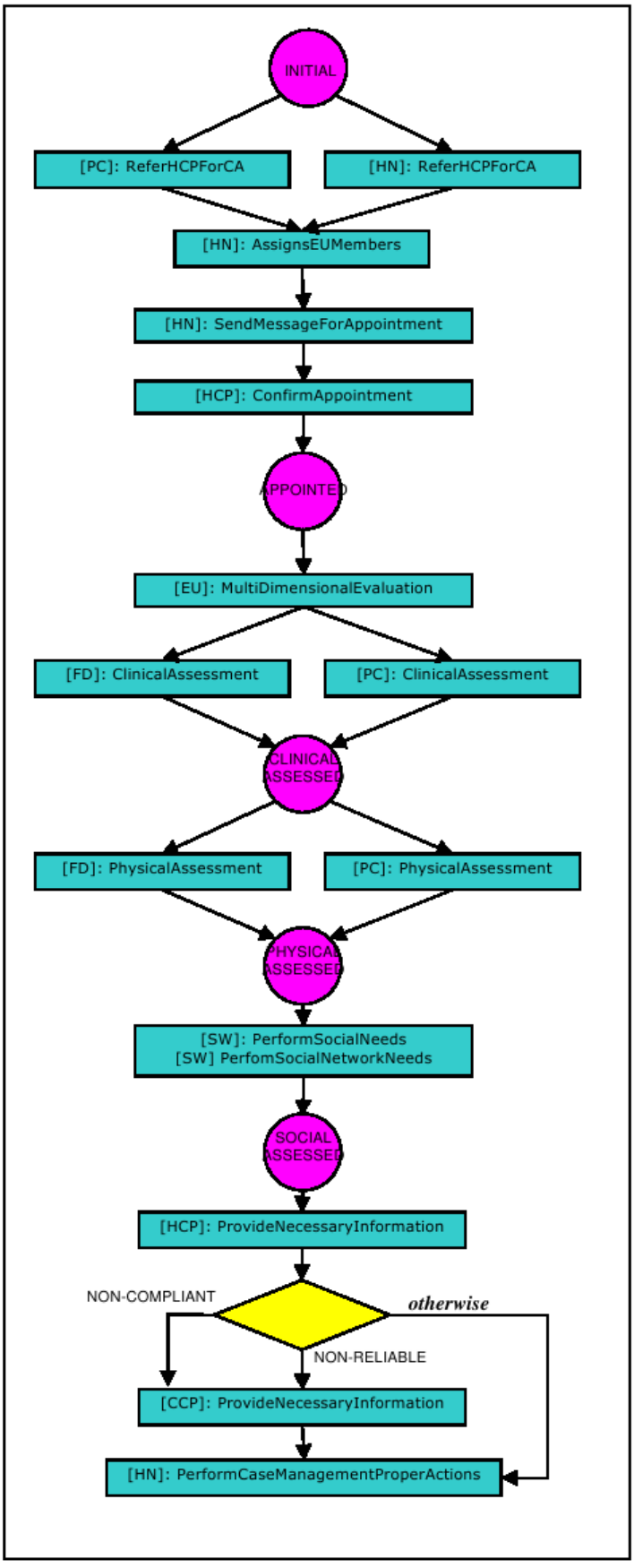
3.3.4.3. Comprehensive Assessment K4CARE Procedure

In the K4CARE healthcare model *comprehensive assessment* is a service that comprises multidimensional evaluation plus clinical assessment and physical examination (integrating the medical side) and social needs and social network assessment (integrating the social side). It is the service devoted to detect the whole series of patient diseases, conditions and difficulties, from both the medical and social perspectives. This service is implemented with a procedure that may be represented in the SDA* model as it follows.

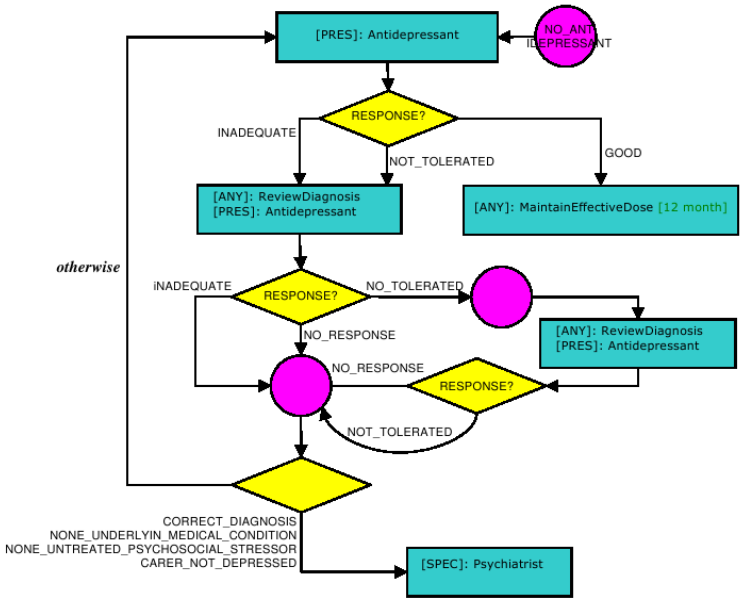
Here, the actions indicate the actor performing the action (i.e. performers) because comprehensive assessment is a collaborative process achieved with the combined actions performed by different actors.

[HCP]	Action performed by the Home Care Patient.
[FD]	Action performed by the Family Doctor.
[PC]	Action performed by the Physician in Charge of the patient.
[HN]	Action performed by the Head Nurse.
[SW]	Action performed by the Social Worker.
[EU]	Action performed by the Evaluation Unit (nuclear work team).
[CCP]	Action performed by the Continuous Care Provider.
[ANY]	Action performed by any of the K4CARE actors.

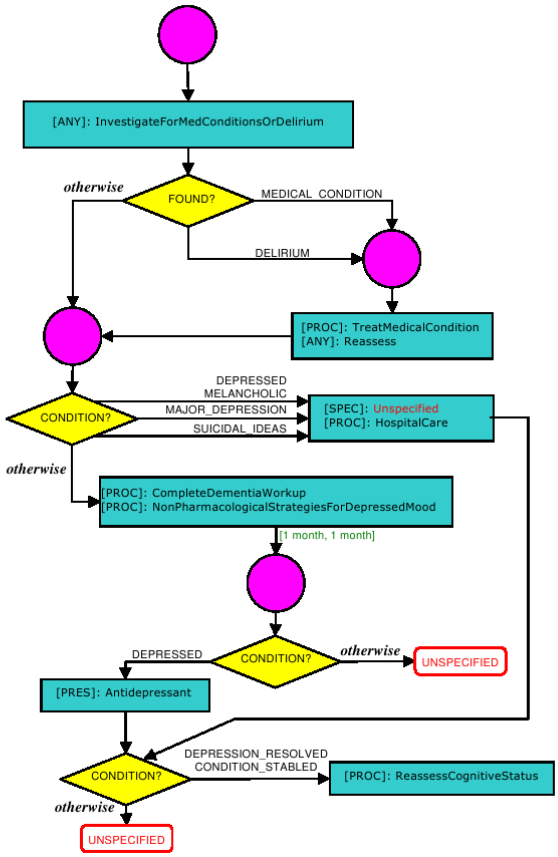
Table 14. Sorts of action variables for comprehensive assessment


INITIAL
[PC]: ReferHCPForCA
[HN]: ReferHCPForCA
[HN]: AssignsEUMembers
[HN]: SendMessageForAppointment
[HCP]: ConfirmAppointment
[EU]: MultiDimensionalEvaluation
[FD]: ClinicalAssessment
[PC]: ClinicalAssessment
[FD]: PhysicalAssessment
[PC]: PhysicalAssessment
[SW]: PerformSocialNeeds
[SW] PerfomSocialNetworkNeeds
[HCP]: ProvideNecessaryInformation
NON-COMPLIANT
otherwise
NON-RELIABLE
[CCP]: ProvideNecessaryInformation
[HN]: PerformCaseManagementProperActions

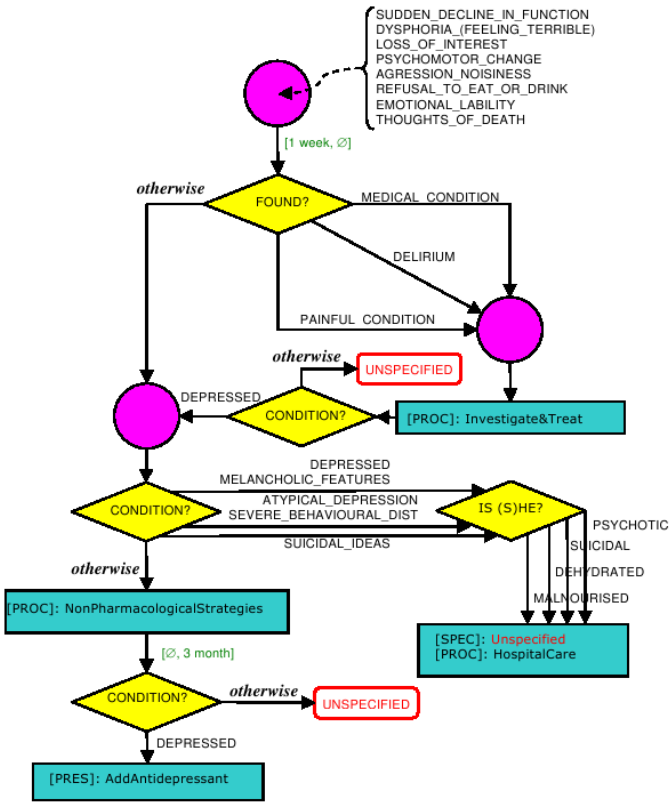
3.3.4.4. The use of Antidepressant Medication in the Elderly



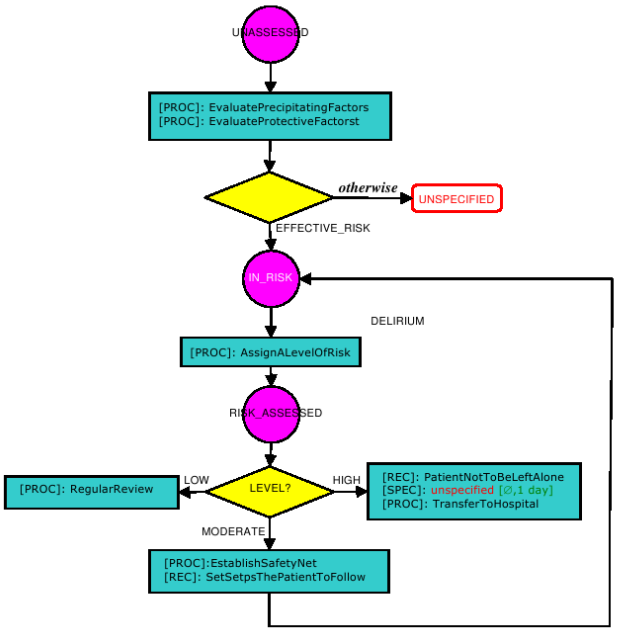
3.3.4.5. Management of Depression with Cognitive Impairment



3.3.4.6. Management of Depression with Dementia



3.3.4.7. Suicide: Risk of Assessment and Management



4. K4CARE Multi-Agent System

In this section the concepts related to the K4CARE's Multi-Agent System will be introduced. As the base of this project is founded on agents, as said in section 2.1.1, an introduction about why to use a Multi-Agent System in the K4CARE platform will be done.

If the reader has no knowledge about what Multi-Agent Systems are, a description is presented in the Annex1. This knowledge is needed to understand the concepts shown in the sections from 4.2 to 4.4 where the K4CARE MAS Architecture is described, and the work related to the architecture done in this project is explained. So section 4.3 will give to the reader an extensive description of the SDA* Agent and its common tasks within the project, and finally, in section 4.4 an SDA* ontology will be detailed. It's interesting to remark that this ontology has a crucial mission in the whole system because it's the only way to communicate the SDA* Agent with the rest of the K4CARE agents without losing the semantics associated to each event related to the execution of one SDA* structure.

These 3 last subsections will be very important to understand the rest of the document because the design and the implementation that have been done are based on this architecture, and some of the reasons why the generatedagent is as it is will be explained here.

4.2. K4CARE MAS Architecture

As described in section 1, one of the K4CARE project objectives is the creation of a web application which will give all the wished services to the final user. This application will be composed by a user interface, a control layer (the MAS) and a data layer (the information repositories). This section will introduce to the reader the internal architecture of the MAS, and its situation in the whole system.

The MAS is the part of the K4CARE which manages the functionality of the whole system. Its main goals are to connect the Web Interface (as this is am ICT project it will have a web-based user interface) with the HC Agents, to represent all the people involved in HomeCare with one intelligent agent in the K4CARE system, to consult/modify the EHCR of the patients, etc, in short its main goal or objective is to give the desired funcionality of the project to the real actors which will use it.

Firstly it's important to do a general description of the knowledge managed in the K4CARE, the ontologies and the EHCR. The *Ontologies*[03], as a set of concepts, properties and relations, constitute a feasible paradigm to represent the declarative knowledge used in the system. There are two basic ontologies in K4CARE. The first ontology, named *Actor Profile Ontology* (APO), details the basic elements of the *K4CARE HC model* (actors, actions, services, procedures, documents) and the relationships between them (e.g. which actions may be performed by each

kind of actor, or which document is associated to each action). The second one, named *Case Profile Ontology* (CPO), stores all the medical terms related to HC (diseases, syndroms, signs, symptoms, assessment tests, clinical interventions, laboratory analysis, social issues) and the relationships between them (e.g. the diseases included in a certain syndrom, or the symptoms of a disease). Agents will be able to reason using the knowledge contained in this ontology, which can be considered as a bridge between the concepts that agents are able to recognize (conditions, diseases) and how actors have to act on those situations (associated interventions). Taxonomic and non taxonomic relations between concepts have been defined in order to allow structuring the information in an appropriate way to answer high level queries about that data.
The EHCR (Electronical HealthCare Record), as is deductible from its name, is a kind of database which contains all the information related to a real Patient. This means, his personal data, his clinical history, etc, in short, all his HC personal information and case history. One of the main features of the K4CARE project is to define the internal structure of this record.
In second place, the multiagent system provides us a set of Gateway Agents used to connect the User Interface (Web browser) with the agents in the system (HealthCare agents). These last agents are in charge of executing all its activities described in the D01[02], in concrete, two of these activities are the execution of Individual Intervention Plans and the execution of Procedures. These two information models will be represented in the format proposed by Dr. David Riaño, the SDA*[04]. The next section presents an agent capable of managing these structures. It's important to remark that this is only a data structure to represent a real knowledge, so this data structure can be stored as XML, as Java objects, as text, etc. therefore we must only be interested in how to perform the interpretation and execution of this data structure and the decisions concerning its storage format will be taken later.
To achieve all these goals we can present the next schema where the connections listed above are represented. In this schema, it is possible to see how the MAS has access to the K4CARE's ontologies (where the knowledge of the actors and diseases is represented) and has also access to the EHCR of each patient.

In figure 10 this architecture is presented, where the K4CARE MAS is clearly described. In this internal architecture there are 3 diferent types of agents, the Gateway Agents, the SDA* Agents and the HealthCare Agents:

- The Gateway Agents are the ones in charge of connecting the HealthCare Actors (patients and medical staff) with their respective HealthCare Agents. In order to achieve this, this kind of agent will manage a connection between the web browser and the respective HealthCare Agent.
- The SDA* Agents are dynamically created agents which are in charge of managing, executing and storing the SDA* structure associated to one concrete HealthCare Agent. This kind of agent will be asked to execute the Individual Intervention Plans and Procedures described in the K4CARE model.

- The HealthCare Agents are a software representation of the medical staff and the patients. They are in charge of managing all the tasks that one real actor must perform and informing to this actor, and also to the system, about the state of these tasks or actions.

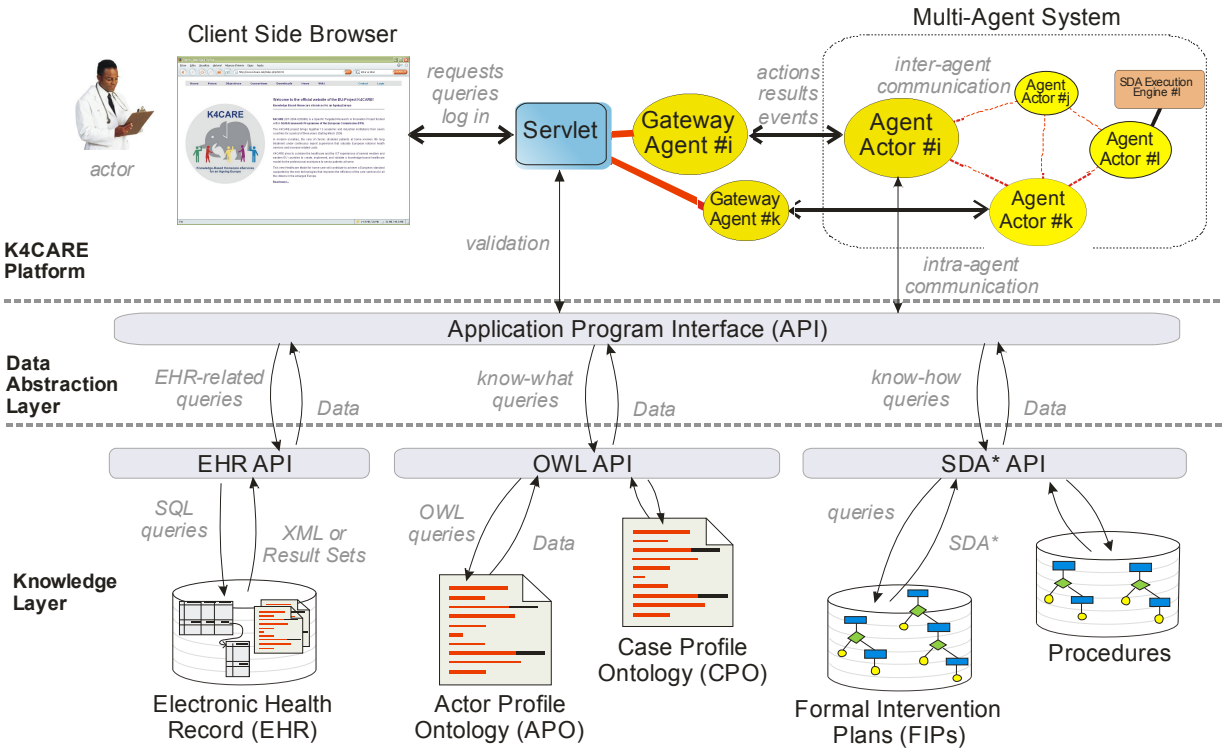


Figure 10. The K4CARE System architecture

The Data Abstraction Layer provides a Java-based API that allows the K4CARE platform agents to retrieve the data and knowledge they need to perform their tasks. That layer offers a wide set of high level queries that provide transparency between the data (knowledge) and their use (platform).

The Knowledge Layer includes all data sources required by the platform. It contains an Electronic Health Record that stores patient records (personal information, medical visits and ongoing treatments). The procedural -organisational and medical- knowledge (know-what) is represented in the APO and CPO ontologies, using OWL. Medical procedures (that implement services) and Formal Intervention Plans are coded using the flowchart-like representation SDA* and stored in specific databases.

4.3. SDA* Agent-based execution

The aim of this section is to describe the design of an agent prepared to manage and execute SDA* structures. We will start describing what it means to execute a SDA* structure, so in order to describe this execution flow, the execution of some described procedures in the D01 and of an arbitrary Individual Intervention Plan will be presented. After this some requirements of the SDA* Agent will be mentioned.

4.3.1. SDA* Agent

Basically the initial idea was to separate the complex tasks of interpreting and managing the medical guidelines from the other tasks assigned to the K4CARE agents, as initially, these agents were not prepared to interpret and execute the medical guidelines formalism used in the project, the SDA*. So, one new agent type has been designed and developed. This agent is the main objective of this work.

The designed solution is one agent capable to interpret and execute the SDA* structure, separating this task from the other tasks that the K4CARE agents perform and, using a communication ontology, communicate to these agents the tasks derived from the SDA* execution. Then all the agents interested in managing the SDA* structures (basically the agents interested in interpreting an Individual Intervention Plan or any procedure described in [02]), will have to create a new SDA* Agent and using this communicative ontology ask some defined tasks to him. This new dynamically created agent will manage and execute the SDA* structure, and perform the related tasks to this management and execution, freeing of these tasks to his invoker.

So in the next section the actions flow generated from the execution of some SDA* structures will be described. In section 4.4 the ontology designed to communicate the different actions and concepts will be presented. Finally, in chapter 5 the design and implementation of the SDA* Agent will be introduced.

4.3.2. SDA*'s actions flow

This section presents the flow of actions derived from the execution of one SDA*, in concrete one Individual Intervention Plan.

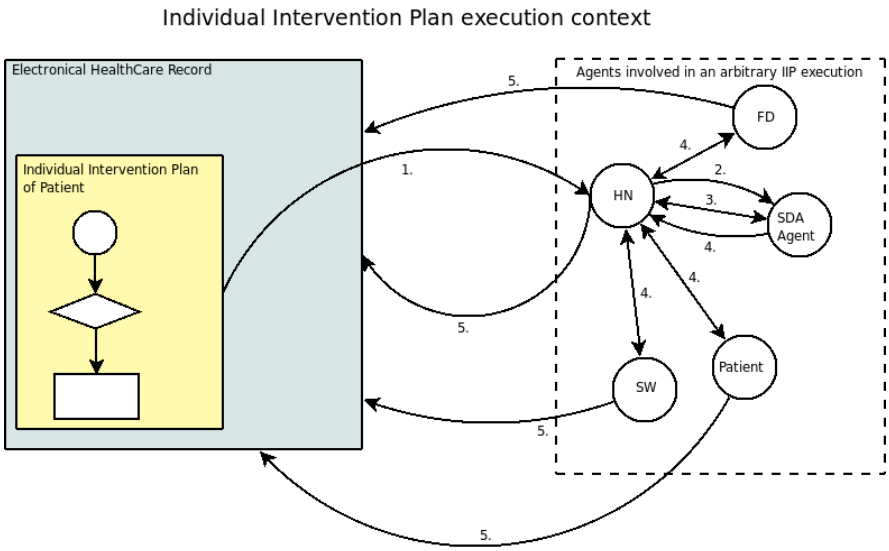


Figure 11. Individual Intervention Plan Actions flow

About figure 11, it's interesting to remark the actions flow presented. The process starts when the

Head Nurse (as the agent in charge of this execution) recovers the patient's IIP from the EHCR (step 1). After this, the execution of this IIP is started by a dynamically created SDA Agent (step 2) who is in charge of interpreting the SDA* structure, linked with the responsible HN who will send action requests (using the FIPA Request Protocol) to the implied agents in order to execute the SDA* (step 3). To finish this execution, the agents fill in their related document subsections, creating the final document which will be saved into the EHCR of the patient (step 4).
After this explanation, it's interesting to analyse the possible actors involved in the SDA* execution. There is an execution coordinator agent (the Head Nurse in the case of the IIPs) and a set of support agents which help this coordinator to complete the execution of the IIP. There is another interesting component in the system, this is the information repository, in this case the EHCR. We also note that the ontologies and other databases proposed in other documents will act as information repository too. Finally, it is also interesting to think about the results of this execution, which are the generation of one or more documents which must be saved in the EHCR of the patient.

4.3.3. IIP execution message flow

Here we present an arbitrary IIP (is important to remark that this IIP isn't real) and the message flow between the agents involved in its execution. First the IIP is presented:

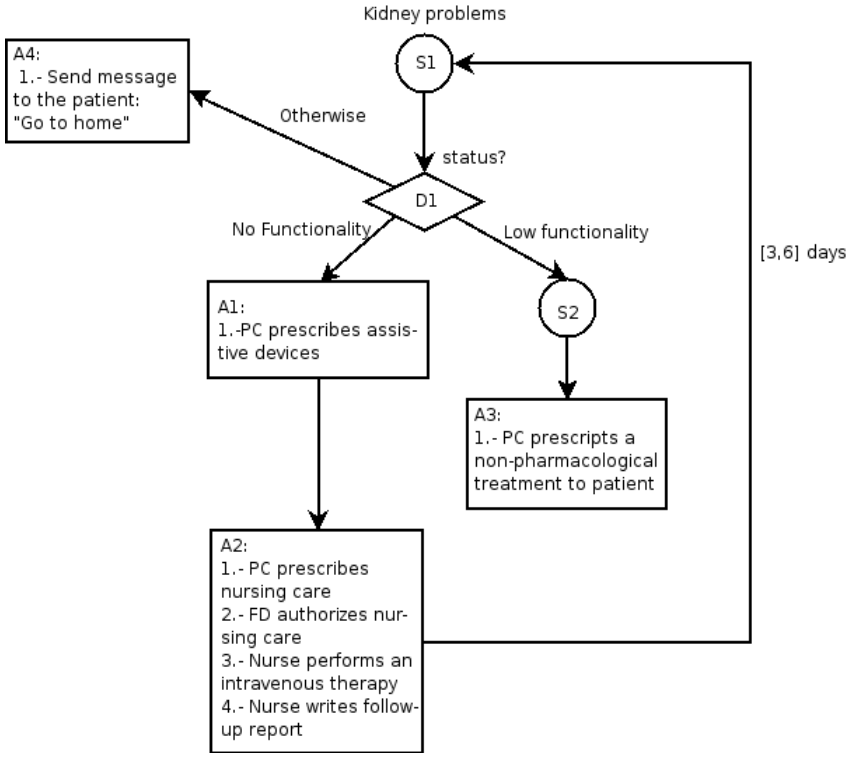


Figure 12. Kidney problems fictitious SDA diagram*

The IIP has four action blocks, and each block has its own group of subactions. Two of these

blocks are performed if the patient hasn't kidney functionality, Action 3 is performed if he has low functionality, and in the other cases Action block 4 is performed.

The next diagram represents the message passing between the agents implied in the execution of the Action blocks A1 and A2, which are the blocks with more actions to perform. Block A3 is represented in the next section, as it has a procedure invocation. Block A4 hasn't any interest as it only represents a message to the patient.

In the first step the gateway agent asks the performance of the patient's IIP, as this service must be given by the platform to the users, and the way to communicate them is through a Gateway Agent. After this is sent, the Head Nurse Agent reads from the EHCR the IIP and, given the status of the patient, arrives to Action A1. Once the execution has arrived here, the HN sends messages to the involved Agents of this Action block in order to achieve its results.

To avoid representing all the execution, in the next diagram of the presented IIP only the messages passed between agents arerepresented:

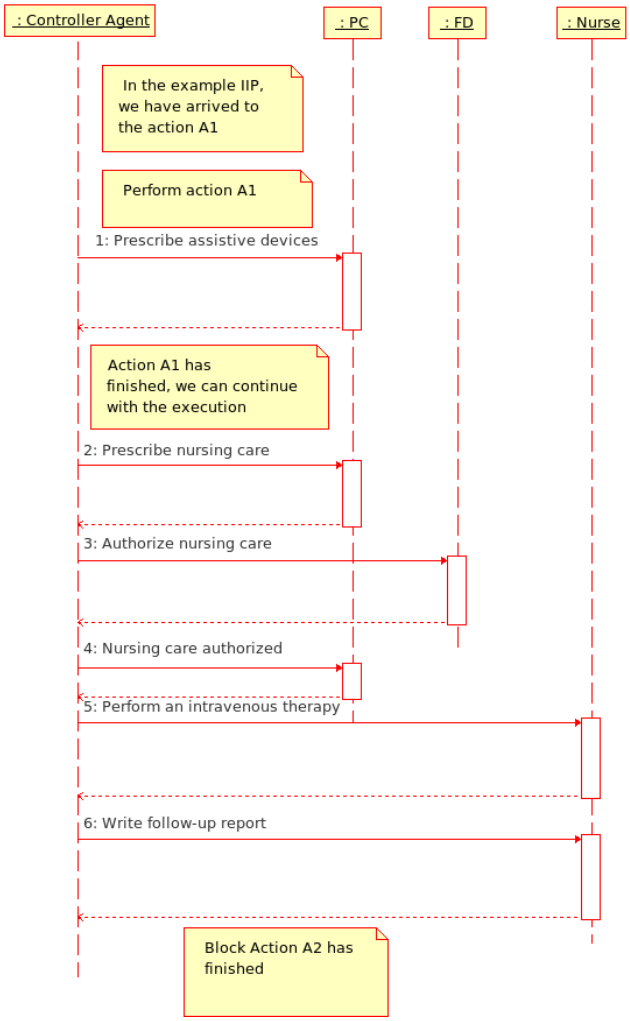


Figure 13. Agent message exchange of the IIP's Action first branch

4.3.4. Procedure execution message flow

The next diagram presents the performance of the procedure "Prescription of Non-Pharmacological Treatment" described in D01, and invoked in the IIP's action block A3. There are 3 steps in this procedure. In the first step the three actors which can invoke this procedure are presented.

Figure 14 presents this sequence diagram:

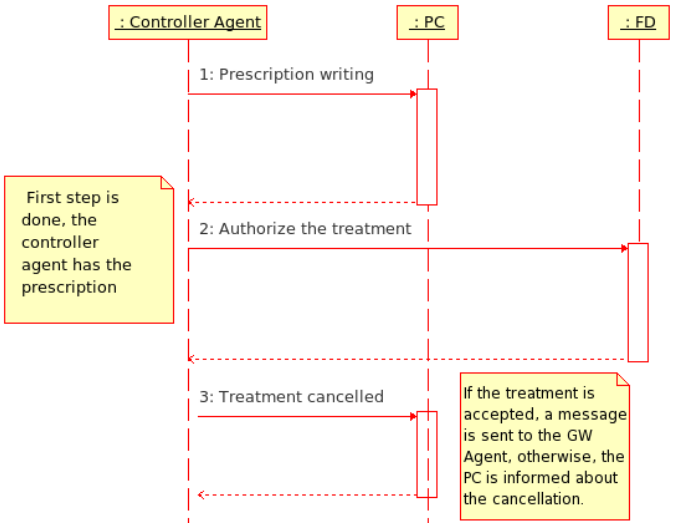


Figure 14. Agent message exchange of the IIP's Action second branch

It's important to remark the last message (number 3), where the PC is informed about the cancellation of the treatment performed by the FD; in any other case, the gateway agent is notified about the acceptance of the treatment, and it shows this fact to its owner.

Finally, the supposed message exchange of a complex procedure is presented. In this case a more complex diagram, which is described in [02] called "Comprehensive Assessment", is presented. To represent this, some assumptions are done; the first is that there isn't any agent in charge of the execution, so a controller agent is created, but as in the previous diagram, its functionality can be passed to another of the HC staff agents. The second one is the agent who represents the EU, here is it assumed that the HN is this agent because this agent is also in charge of the assignment of members to the EU. This second assumption can be discussed because it's not clear who represents the EU and, like in the SDA* execution, there are two clear options, an EU agent or an agent representing all the EU. In the diagram we have chosen the solution where one of the agents represents all the EU, in this case the FD.The actions "n.1" (where n is a number

between 0 to the maximum steps in the procedure) represent an asking alternative when a HC staff agent needs information about HCP. The “auto-arrows” refer to actions in one procedure which are done by the same agent. Figure 15 represents this complex SDA* structure execution:

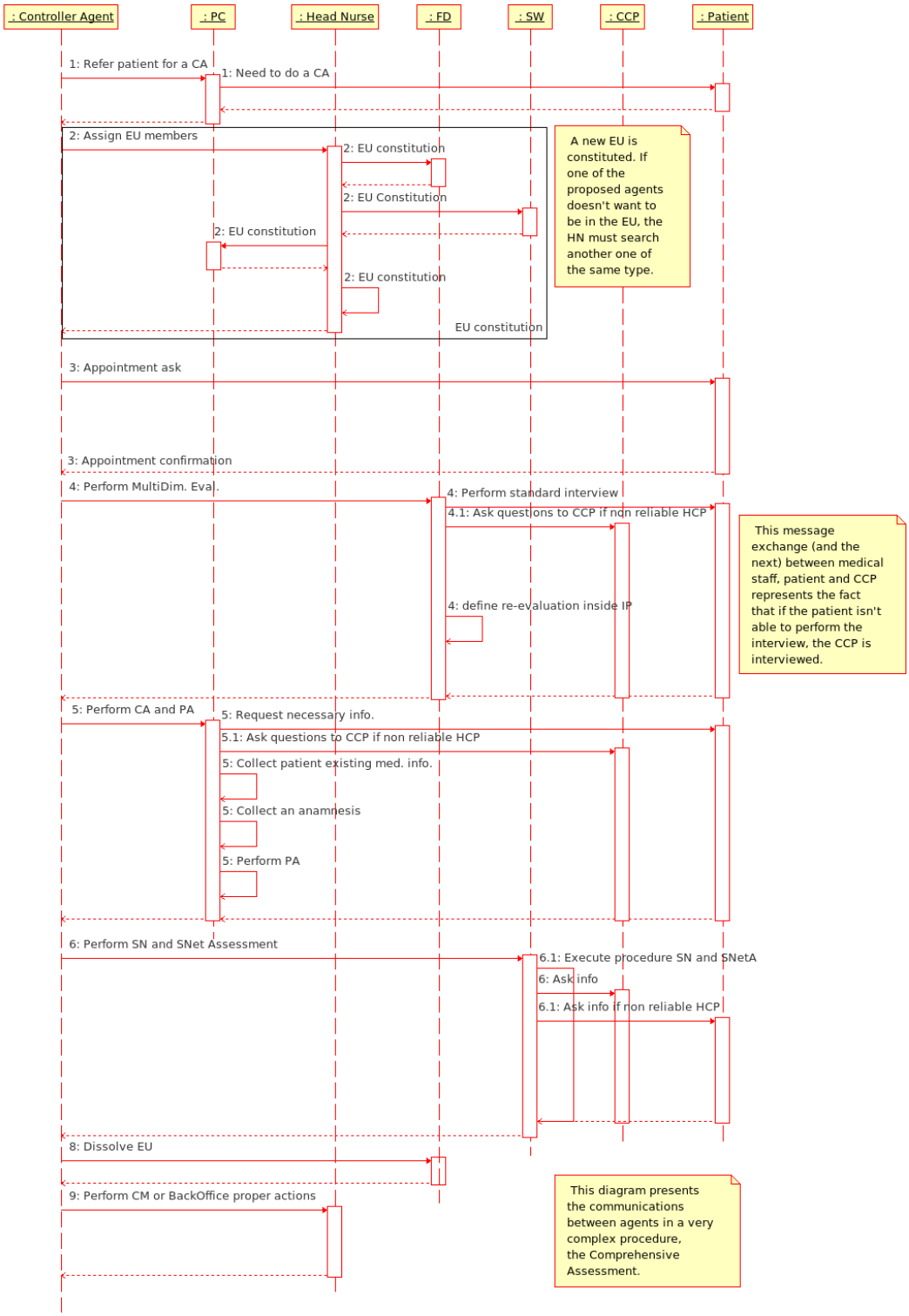


Figure 15. Message exchange between agents in the Comprehensive Assessment execution

4.4. K4CARE SDA* Ontology

As the reader can imagine, to achieve the execution of the SDA* structures the agents should have some common language in order to communicate all these messages. To accomplish this objective the agents who are interested in executing an SDA* structure must know the concepts and actions defined in the SDA* Ontology described here.

This ontology contains all the elements which are needed in order to execute the SDA* structures, like the actions that one agent must perform, or the concepts related to the execution of the concrete SDA*, etc. In figure 16 the context of this ontology within the whole system is presented:

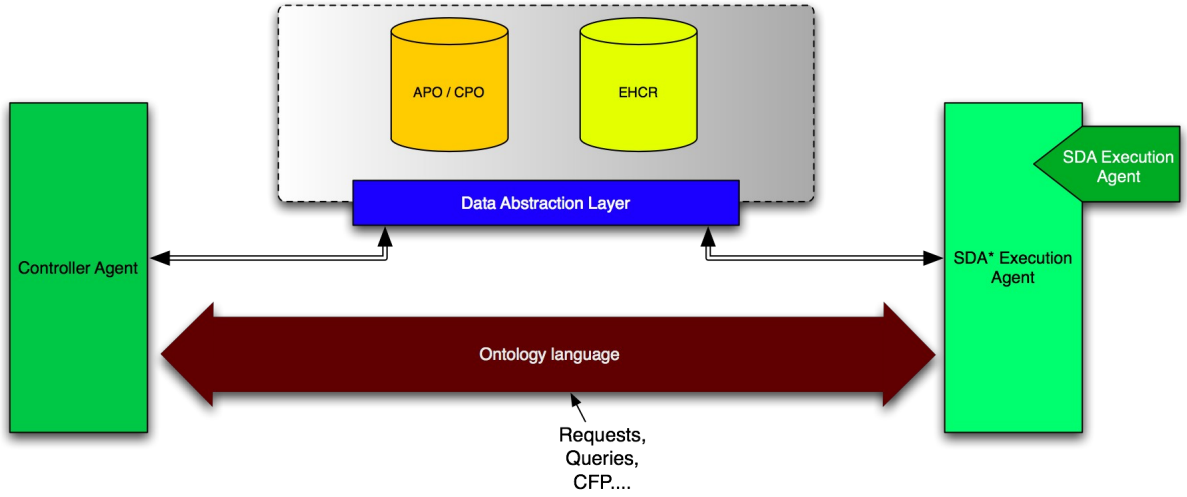


Figure 16. Context of the SDA ontology in the system*

Figure 17 presents the structure of the different concepts and agent-actions cotained in this ontology:

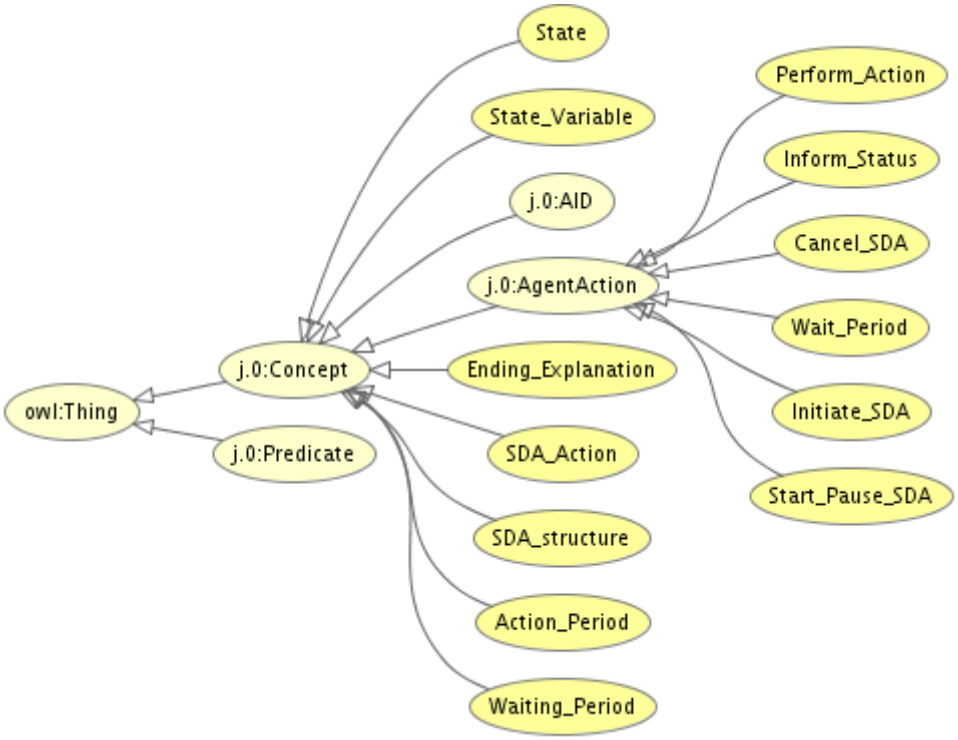


Figure 17. SDA MAS Ontology*

4.4.1. Concepts

In this section the diferent concepts in the SDA* model execution which can appear in a two agent conversation are presented. These concepts are the following:

- *The SDA* structure itself*
  This is the conceptual representation of the SDA* structure itself.

Attribute	Type	Opt.	Description
Id.	String	M	Identification of the SDA* in K4CARE terms
Procedure	Boolean	O	Refers to the fact that SDA* is one of the K4CARE defined procedures

- *The current State node information*
  This is a representation of the patient's state or the state of the execution of a procedure, this means the health status of the patient or the position in the SDA* execution, or anything which can be represented with a state node.

Attribute	Type	Opt.	Description
Id.	String	M	Identification of the SDA* current State node
Procedure	Boolean	O	Refers to the fact that SDA* is one of the K4CARE defined procedures
Status	Set of State Variables	M	A set of the status variables into this State node related to the SDA*

- *The current Action node information*
  The concept of an Action node presented in the SDA* specification, the 4-tuple and the time interval required to represent the action correctly.

Attribute	Type	Opt.	Description
Id.	String	M	Action Identification (from the K4CARE Action list)
Receiver	String	M	Which actor must receive this action
Performer	String	M	Who must perform this action
Time_Interval	SDA* 3-valued time interval	O	Indicates the interval of time and the repetitions in which this action must be performed
Document_Related	String	M	Indicates the document in which the results of this action must be written

- *3-valued time interval*
  This is the conceptual abstraction of the waiting periods which are described in the SDA* formalism.

Attribute	Type	Opt.	Description
Start	Date	M	The start of this period
End	Date	M	The end of this period
Period	Time	M	The time period which this action has to be repeated

- *A period of time*
  This is the conceptual abstraction of the waiting periods which are described in the SDA* formalism.

Attribute	Type	Opt.	Description
Min_time	Date	M	The minimum time which should be waited
Max_time	Date	M	The maximum time which has to be waited

- *Explanations about an unexpected execution stopping situation*
  This is a conceptualization of an execution error, i.e. a representation of an unexpected status of the SDA* execution.

Attribute	Type	Opt.	Description
Ending_Condition	String	M	Name or code of the ending condition
Description	String	O	A short description explaining why the execution of the current SDA* has finished

4.4.2. Actions

This section introduces the list of possible actions which the K4CARE agents may request to the SDA* agent and vice versa. This actions list could be increased in the future, depending on the possible needs of the system. These three first actions are referred to the ones which the K4CARE agent system would ask to the SDA* Agent, the last one has the inverse sense, and is requested by the SDA* Agent to the system agents.

- *Initiate a new SDA* - From controller to SDA* agent*
  Request to initialize a new SDA* structure.

Attribute	Type	Opt.	Description
SDA*	SDA* Structure	M	The SDA* structure to be initiated

- *Start/Pause the current SDA* - From controller to SDA* agent*
  Used when a controller agent wants a SDA* agent to start or pausethe current SDA*.

Attribute	Type	Opt.	Description
SDA*	SDA* Structure	M	The SDA* structure to be paused or

			started
Start	Boolean	M	If the SDA* structure has to be started or not, if not it means that it has to be paused

- *Cancel the SDA* execution -From controller to SDA* agent*
  Request the cancellation of a SDA* structure execution which is in progress.

Attribute	**Type**	**Opt.**	**Description**
SDA*	SDA* Structure	M	The SDA* structure which has to be cancelled

- *Perform these actions – From SDA* to controller agent*
  When a SDA* agent reaches an action node into the SDA* structure needs to ask to the related actor in the system to perform those actions, so this abstraction represents the performance of these actions.

Attribute	**Type**	**Opt.**	**Description**
Action_lists	Set of Actions to be performed	M	A set of actions (node information structure) which has to be performed by the asked agent

- *Wait this period – From SDA* to controller agent*
  To indicate to the receiver that he must wait the referred period of time expressed into this action.

Attribute	**Type**	**Opt.**	**Description**
Period	Time_period	M	The period that the asked agent should wait

5. Design and implementation of the SDA* Agent

This chapter of the document presents the design and structure of the SDA* Agent. As said in section 4.3.1, this agent is capable to execute the SDA* structures (as they are nowadays). To do so, this agent has an internal data structure to represent and manage this formalism.

Besides this data structure, the Agent needs some kind of engine to execute the contents of this structure, as its purpose is to execute this representation. So the SDA* agent has an execution engine which gives to him this execution capabilities.

5.1. Code Structure

The code written to implement the SDA* Agent has been structured in some packages, as shown in figure 18:

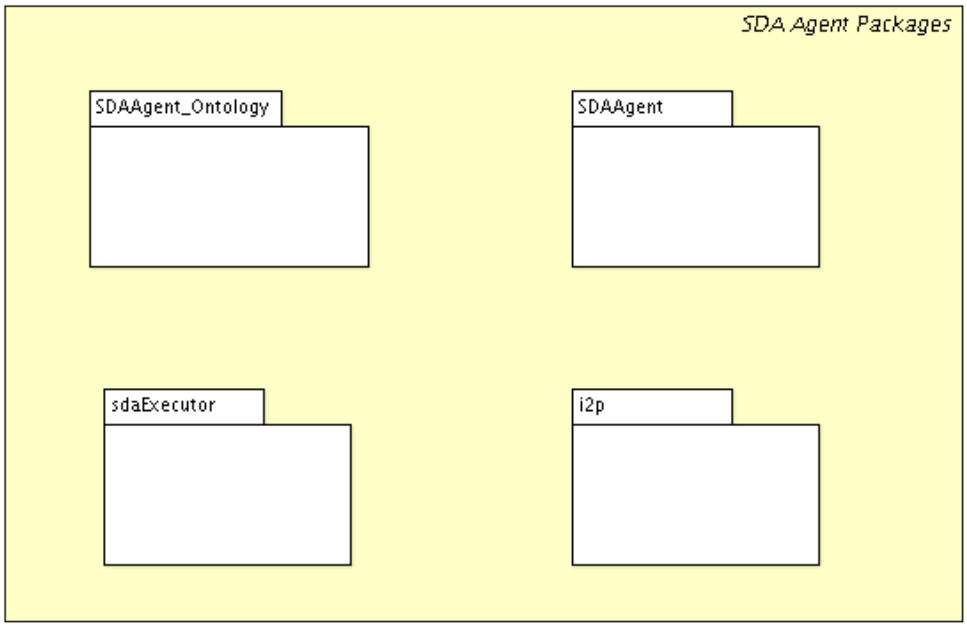


Figure 18. The SDA Agent Packages

The code is structured in 4 main packages: the `SDA_Ontology`, the `I2P`, the `sdaExecutor` and the `SDAAgent.`

- The `I2P` is the representation of the SDA* formalism in Java classes. It implements a graph-like structure with the needed modifications in order to correctly represent this formalism.
- The `sdaExecutor` is the event oriented engine capable to interpret and execute SDA* structures. As it will be shown in the next sections, this engine is the basis of the whole functionality of the SDAAgent.
- The `SDA_Ontology` is the package which contains the code related to the ontology

described in section 4.4. This package includes the concepts and the actions related to the execution of SDA* structures.

- Finally, the `SDAAgent` is the concrete implementation of the Agent described above. This agent uses the other 3 packages in order to understand and execute SDA* structures, and to communicate to the other agents the different events that occur during this execution.

5.2. SDA* graph data structure

In this section the data structure used to represent the SDA* formalism will be described. The objective of this data structure is to implement the SDA* format in a Java package, in order to be accessible by the agents in the system who would like to use it. To do so, this structure must accomplish the next points:

- It must be robust.
- It should have operations of creation, modification and reading.
- It has to provide functions to perform traversals over it, respecting the time and decisions restrictions that will appear during this process.
- It must be prepared to support concurrent access.

To achieve these objectives, the package has been structured in 3 blocks. The first one is the representation of the different SDA* node elements, which are the States, the Decisions and the Actions. The second block is the representation of the connectors between these SDA* node elements. These connectors can have different functionalities, depending on their purposes. Finally, the third block is the SDAGraph structure itself, which uses the other 2 blocks and creates the whole structure. The next figure shows a global overview of the I2P package:

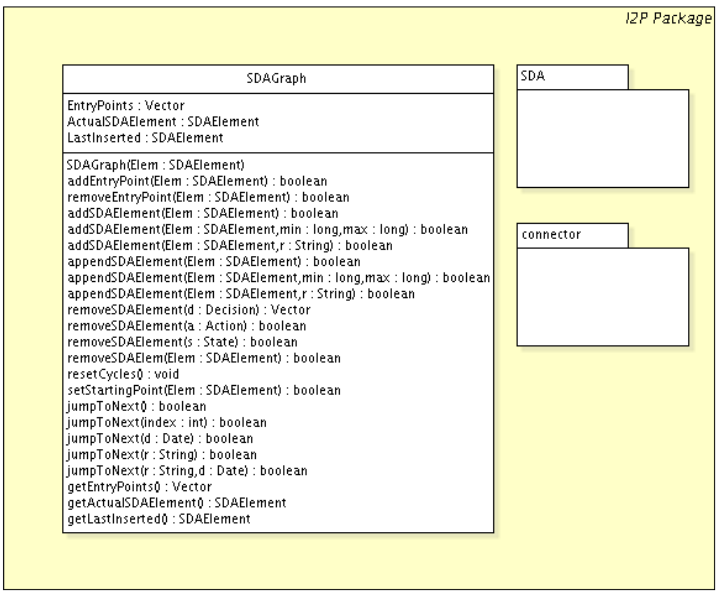


Figure 19. SDA representation package*

As it is possible to view, the package contains the 3 mentioned blocks. In the next two figures the classes contained in the SDA subpackage and in the connector subpackage, and the relations between the different classes in these subpackages are shown:

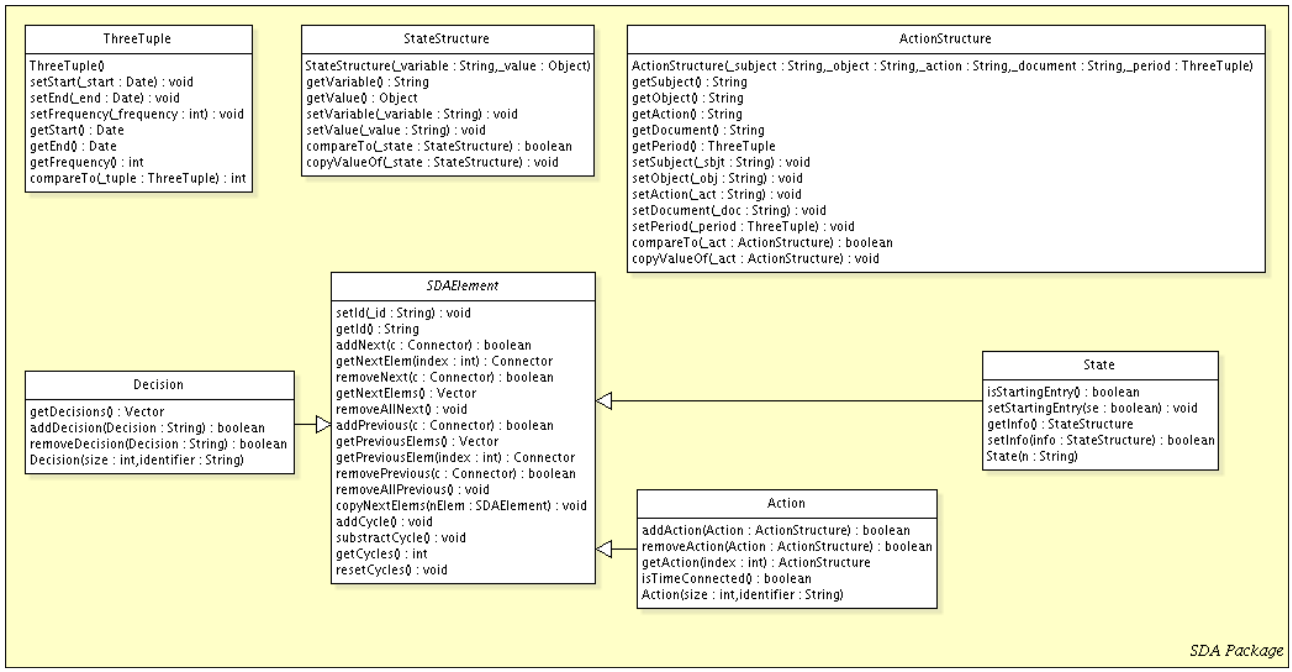


Figure 20. The SDA Package from the I2P data structure

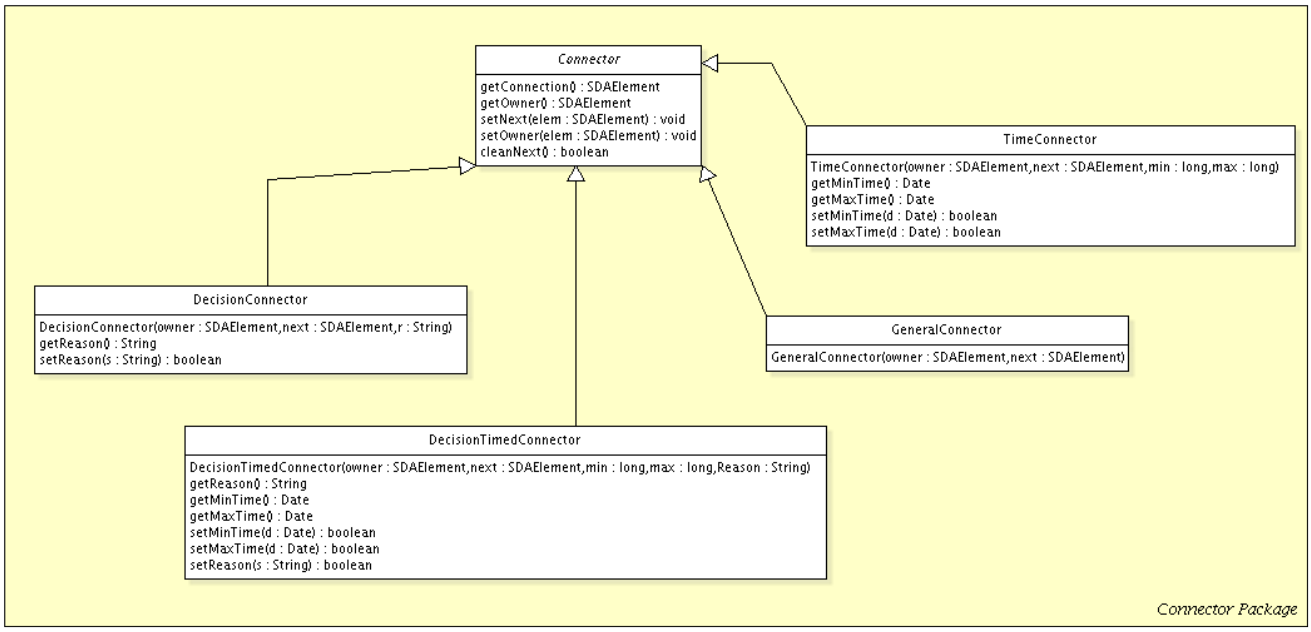


Figure 21. The connector package from the I2P data structure

In the two figures presented above it is possible to view the different fields of each class and the respective position of each class in the internal structure of the package. In the next subsections the concrete funcionality of each one of the three presented blocks will be described.

5.2.1. SDA Subpackage

As said, this subpackage contains the concrete representation of each of the possible nodes in a SDA* structure. To do so, the States, Decisions and Actions inherit or implement an abstract class which represents the concept of a SDA element, which is the main concept of this class.

The SDAElement abstract class has functions to connect itself with the connectors of the connectors package. It also has functions which permits the executor to know which node is referenced and to control the cycles of that node (see section 6.6 for more information).

There are another 2 classes which need to be analyzed deeper, which are `StateStructure` and the `ActionStructure`. The first one is used to represent the contents of the States and contains information about the variable which it represents and the value associated to this variable. The second one represents the semantic information related to an action, so it is composed by five elements:

- ➔ The subject who performs the action.
- ➔ The object which gives the action.
- ➔ The document related to this action.
- ➔ The action code (as specified in document D01).
- ➔ The interval time of repetition of this action (as described in the SDA* formalism)

With this five fields, this class can represent all the information related to a medical action, as it is needed to know who performs the action, and who receives this action. It is also needed to know which document must be filled after this action, and which concrete action must be performed during which time period.

5.2.2. Connector Subpackage

This package is used to connect the different nodes present in the SDA subpackage, respecting the different constraints present in the formalism related to the connection between these nodes. So, these connectors must give to the developer possibilities to represent the time restrictions, and the decision restrictions, too.

To do so, an inheritance model has been designed, as in the case of the SDA package. The different connectors inherit from a general class, which is called connector and has information about the two nodes which it connects.

The other subclasses are the following:

- ➔ General connector: this is the representation of a connection between two nodes without any time or decision restriction.

- ➔ Decision connector: this represents the connection between a decision node and another type of node. It has the information about why the path that it sets has been chosen.
- ➔ Time connector: represents a timed connection with a time restriction that must be respected. This time restriction is of the 2-tuple form, in other words, it contains the miminum time and maximum time that must be spent to let the execution follow through this path.
- ➔ Decision timed connector: this is a mix of the time connector and the decision connector, as it could be interesting to jump through a decision path, but at the same time, wait a concrete time before doing the next action.

5.2.3. I2P Class

This is the main class of this package. It uses the two subpackages described above to give to the system the desired functionality. To do so, it gives SDA* creation methods, as add or delete nodes and connections; and at the same time, it gives traversal methods, in order to permit the execution of the represented SDA* graph.

The additional methods don't have any intrinsic problematic, as they permit to add a node after the last inserted node, using the parameters to specify which kind of connection will connect these two nodes. The problematic resides in the removal methods, in concrete in the deletion of decision nodes, as deleting a node which connects with another node is quite easy, but the deletion of a decision node is not as trivial because normally a decision divides the graph into 2 subgraphs, so, the deletion of a decision returns the list of possible subgraphs generated from this deletion.

When the SDA* structure is formed, it is interesting to go through it, so, "jumping" methods are given. These methods have parameters which permit to jump to the next node specifiying the possible restriction, as for example, a time restriction, permiting to the execution package to transparently control the possible time constraints.

5.3. SDA* executor

After having defined the SDA* data structure, it was required to have some kind of interface or engine to interpret and execute the contents of this structure. To do so, the SDA* executor has been designed. This executor is based in an event model, and it follows the next steps:

1. Someone asks to the executor to load an SDA*
2. The executor is asked to advance (or step forward) in the passed SDA*
3. After this the executor generates an event informing about what has happened
4. These two last steps are iteratively repeated until the SDA* finishes

In this section, the event orientation, its implementation and the finalisation conditions will be described. Finally, the sdaExecutor final design will be introduced.

5.3.1. Event-based Orientation

Before starting to describe how to design an event-based package, the event orientation will be defined. *Event-driven programming* or *event-based programming* is a computer programming paradigm in which the flow of the program is determined by user actions or messages from other programs. In contrast, in batch-programming or flow-driven programming the flow is determined by the programmer. Batch programming is the style taught in beginning programming classes while event-driven programming is what is needed in any interactive program. Event-driven programs can be written in any language, although the task is easier in some languages than in others. Some programming environments make the task quite easy, others less so. This kind of paradigm gives to the execution of the SDA* an intuitive approximation which allows a clear separation of components, allowing to have a clear implementation.

To use this technique some other techniques were required in order to give to our Java code this orientation. After some researches in Software Engineering it was clear that the most useful tool to achieve this objective were the Design Patterns[07]. A design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-Oriented design patterns typically show relationships and interaction between classes or objects, without specifying the final application classes or objects that are involved.

Design patterns can be classified in terms of the underlying problem they solve. Examples of problem-based pattern classifications include:

- Fundamental patterns.
- Creational patterns, which deal with the creation of objects.
- Structural patterns, ease the design by identifying a simple way to realize relationships between entities.
- Behavioural patterns, that identify common communication patterns between objects and realize these patterns.
- Concurrency patterns.

In concrete, to solve the event-orientation a behavioural pattern was chosen, the *observer pattern*. The observer pattern is a design pattern used in computer programming to observe the state of an object in a program. It is related to the principle of implicit invocation.

The essence of this pattern is that one or more objects (called observers or listeners) are registered (or register themselves) to *observe* an event which may be raised by the observed

object (the subject). The object which may raise an event generally maintains a collection of the observers. The figure below illustrates this structure:

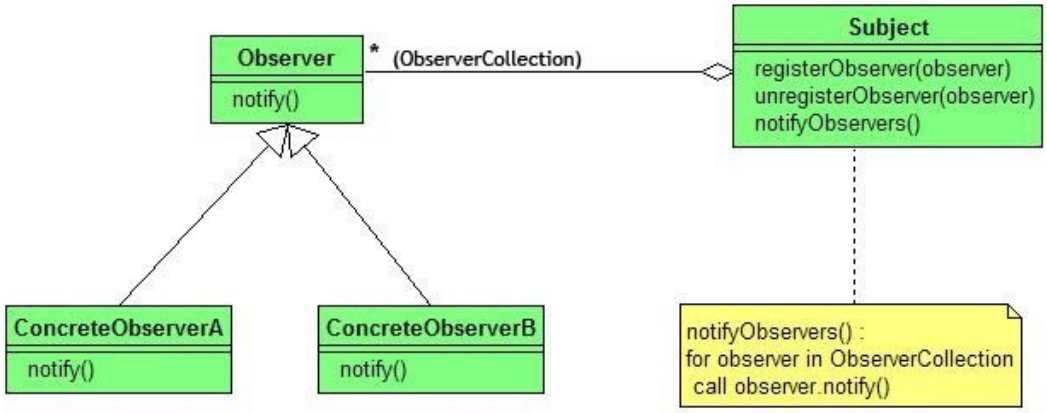


Figure 22. The Observer pattern components

The participants of the pattern are detailed below. Member functions are listed with bullets.

Subject

This class provides an interface for attaching and detaching observers. The subject class also holds a private list of observers. It contains these functions:

- *Attach* - Adds a new observer to the list of observers observing the subject.
- *Detach* - Removes an observer from the list of observers observing the subject.
- *Notify* - Notifies each observer by calling the notify() function in the observer, when a change occurs.

ConcreteSubject

This class provides the state of interest to observers. It also sends a notification to all observers, by calling the Notify function in its super class (i.e. in the Subject class). It contains this function:

- *GetState* - Returns the state of the subject.

Observer

This class defines an updating interface for all observers, to receive update notifications from the subject. The Observer class is used as an abstract class to implement concrete observers. It contains this function:

- *Update* - An abstract function, to be overriden by concrete observers.

ConcreteObserver

This class maintains a reference with the ConcreteSubject, to receive the state of the subject when a notification is received. It contains this function:

- *Update* - This is the overridden function in the concrete class. When this function is called by the subject, the ConcreteObserver calls the GetState function of the subject to update the information it has about the subject's state.

When the event is raised each observer receives a callback. This may be either a virtual function of the observer class (called 'notify()' on the diagram) or a function pointer (more generally a function object or "functor") passed as an argument to the listener registration method. The notify function may also be passed some parameters (generally information about the event that is occurring) which can be used by the observer.

Each concrete observer implements the notify function and as a consequence defines its own behaviour when the notification occurs.

5.3.2.Finish conditions

Before starting the execution of a SDA* structure there are some finishing conditions that must be detailed, as if they aren't it's possible that the execution won't ever finish. So, in the frame of this project the next SDA* finishing conditions have been taken into account:

- The interpretation of a node *n* times: as there can be cycles in the SDA* structure, and it is a medical structure, it has been decided that if some action has been performed more than *n* times it should be probable that this SDA* won't be the most appropiate for the patient.
- The arrival to a node which hasn't continuation: if the execution arrives to a node which hasn't any next node it's clear that the execution has finished.
- The execution of a decision node which hasn't the branch that is asked.
- Or, the jumping to a node with a time restriction which can't be accomplished, this means, two nodes connected with a time connector and is required to jump from the first to the second, but the time restriction is not fulfiled.

5.3.3. SDA* Executor design

This section presents the design of the execution engine. After explaining the requirements to execute in an event orientation paradigm and the ending conditions of the SDA* execution the next design has been created:

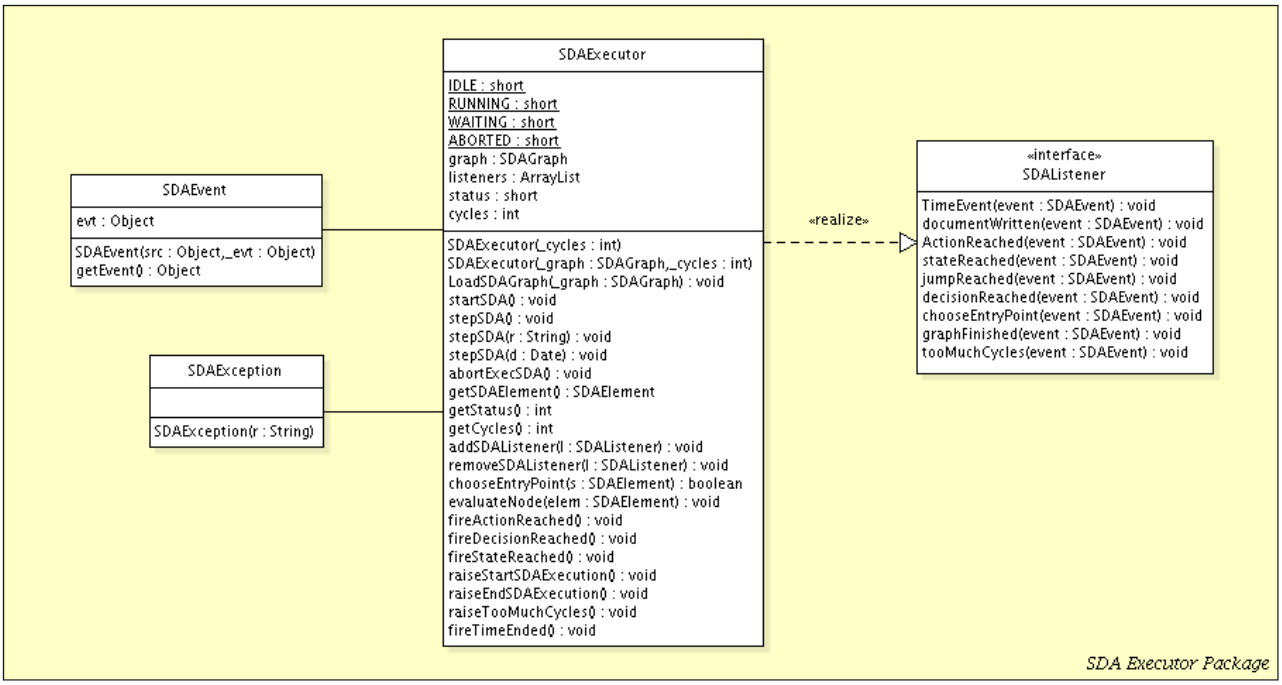


Figure 23. The SDA Executor Package

As described in the figure above, the executor is composed by four elements:

- The representation of the Events, the SDAEvent class. This class contains information about the event that has been produced, as which node has produced this event.
- A tailored exception, to be thrown in case of errors.
- The interface that must be implemented by any class which wants to listen to SDA events. In this interface the possible events that are generated from the execution of a SDA* structure are described. All these events have an associated SDAEvent class which has the information related to the concrete event. This interface represents the *Observer* class in the pattern.
- The execution engine, which interprets the contents of the SDA* structure and raises the different events to the registered listeners. It performs the *Subject* role in the observer pattern.

5.4. SDA* Agent

In this section the SDA* Agent package, and its interaction with the rest of the packages listed above will be described . As shown in figure 24 the SDA* Agent has 2 main classes:

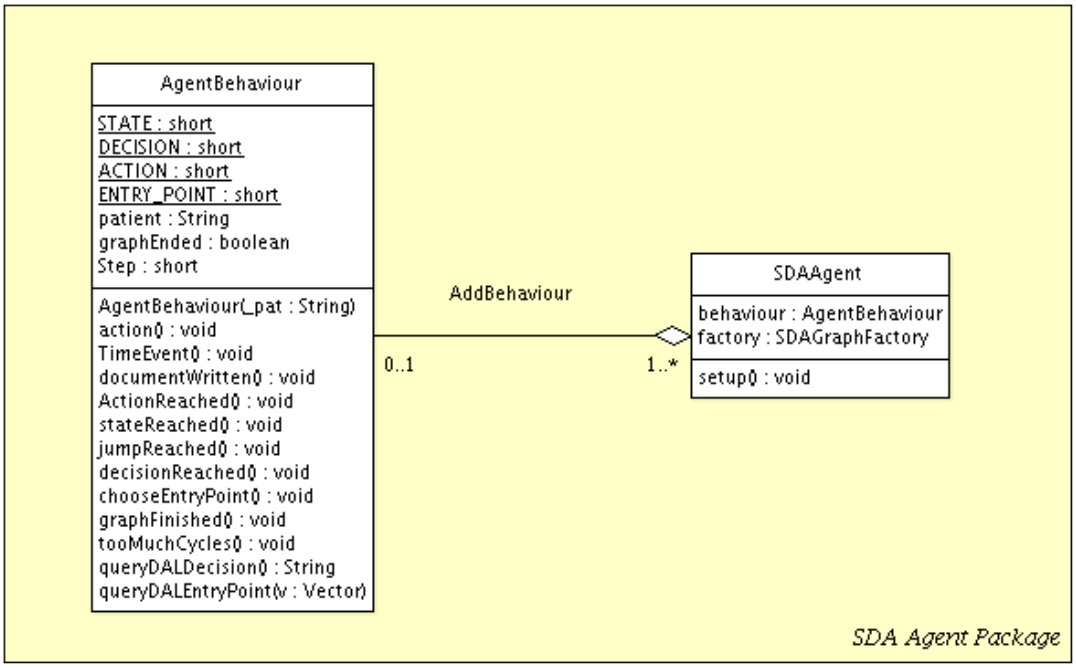


Figure 24. SDA Agent Package*

This agent is composed by the implementation of an Agent by the JADE library (see section 5.1) and the definition of its SDA* execution behaviour:

- The Agent JADE code, as said, and by references to the behaviour that it has and to a SDAGraphFactory, which is a connection with the SDA* repositories.
- The behaviour is a concrete implementation of a Cyclic Behaviour, which asks to its associated SDAExecutor (because it implements a SDAListener) to step in the SDA execution at each cycle of the behaviour. After this step this behaviour receives the generated events and treats them.

In the next diagram (figure 25) the connections of the SDA Agent with the rest of the classes are presented, in order to have a global view of the whole design:

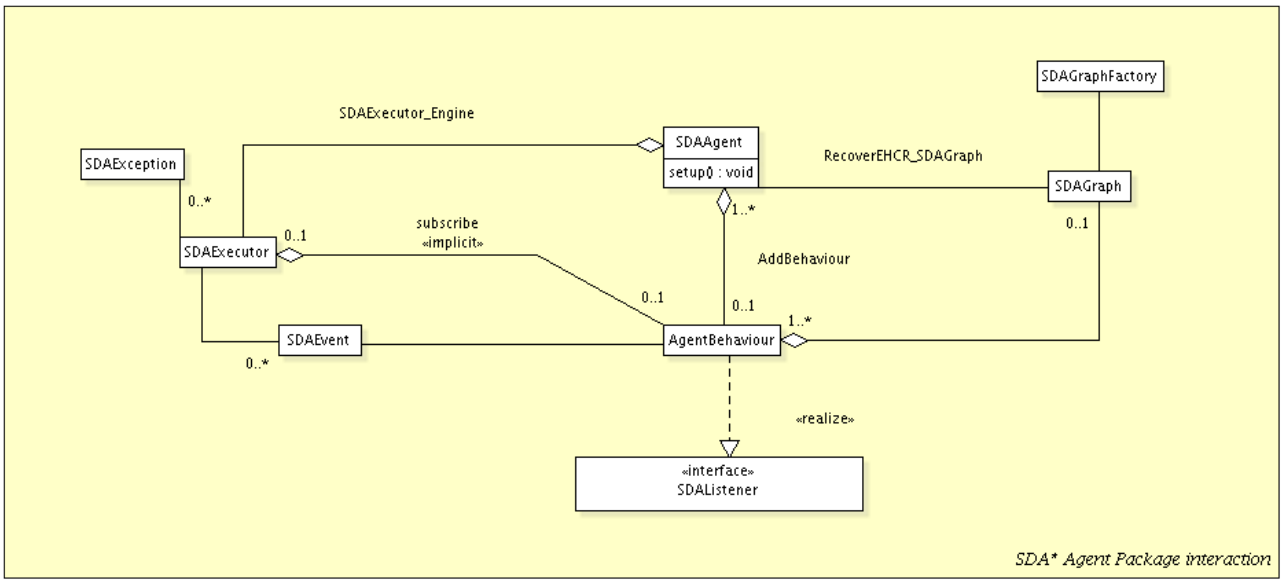


Figure 25. SDA Agent Package Interaction*

As is possible to see, the agent has connections to an executor, to a SDAGraph, and obviously to its behaviour. When this behaviour is added to the agent, it is subscribed to the executor in order to receive the events, and it's also connected to the SDA* representation, in order to have knowledge about what is being executed. These connections are clear in the next code fragment, where the finishing conditions can also be noticed:

```
import sdaExecutor.SDAExecutor;
public class SDAAgent extends Agent {
AID recv;
SDAExecutor sdaEngine;
AgentBehaviour behaviour;
SDAGraphFactory factory;
int maxCycles = 3; //Finishing condition
      protected void setup() {
            System.out.println("Agent ” + getLocalName() + " started.");
            factory = new SDAGraphFactory(); //Consult the EHCR instead
            sdaEngine = new SDAExecutor(maxCycles);
            behaviour = new
            AgentBehaviour(this,”patient_name”,sdaEngine);
            sdaEngine.addSDAListener(behaviour); //Connection Agent -
            SDAExecutor
            sdaEngine.LoadSDAGraph(factory.buildSDA());
            // Add the SDABehaviour
            addBehaviour(behaviour);
      }
}
```

5.5. SDA* Ontology

This section describes the process of generation of the Ontology used to represent the concepts present in the execution of the SDA* structures. This section explains the process of creating a JADE ontology using the Protégé tool and its plugin the Protégé Bean Generator.

5.5.1. Defining and creating the Ontology

In any JADE ontology there are 3 important elements:

- The concepts: represent the basic information in the communication between agents.
- The predicates: represent some determined conditions that must be fulfiled by the response data to an information request.
- The actions: represent the request of the performance of a concrete task by one agent, and they have all the needed information for its execution.

So, first of all we needed to codify the concepts, predicates and actions formally described in section 4.4 in an ontology. As the process of creating a JADE ontology is quite slow, an assistive tool has been used to speed up this process. To do this, the Protégé tool has been used to create an ontology and a plugin of Protégé to generate the code. To define the ontology, the template provided to create a JADE ontology using the Bean Generator plugin was used, because the Bean Generator has some restrictions about the generation of this kind of ontologies, and one of these constraints is the use of this template.

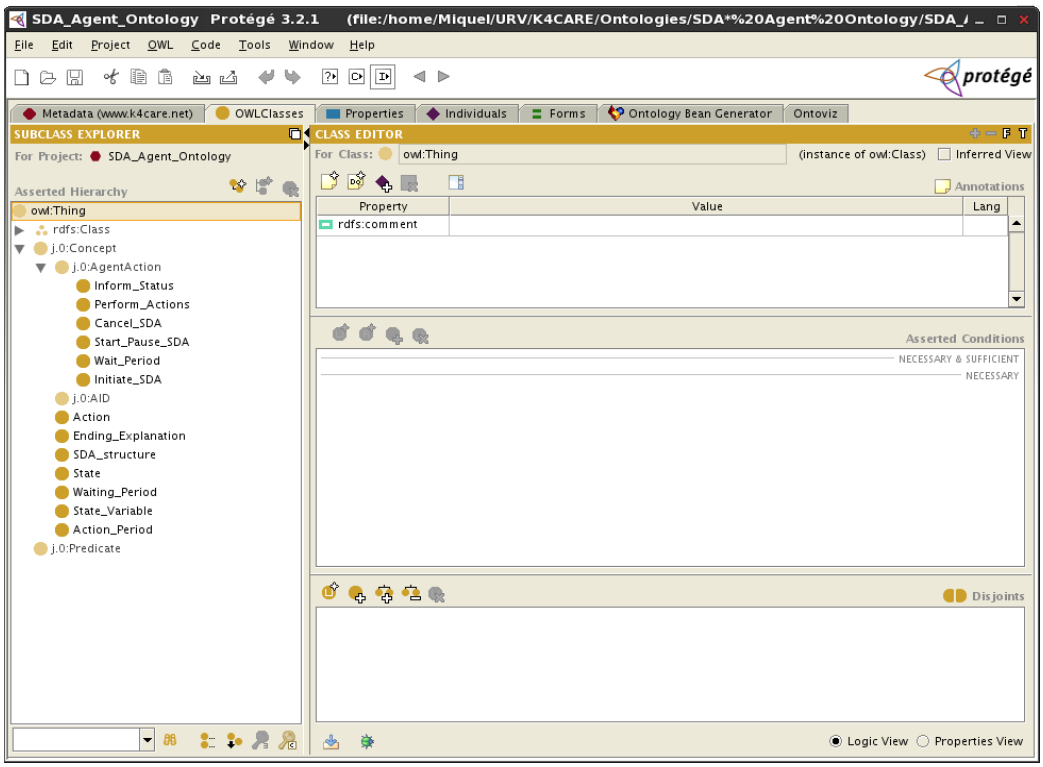


Figure 26. Ontology created with Protégé

As can be seen, the ontology has the concepts and the agent actions described in the ontology section. Once it is done, it's possible to use the Bean Generator tab, in order to create the JADE ontology. Using this tool, Protégé will create the needed classes to define a JADE ontology without having to be written by the developer, and having an easy maintenance. Figure 27 shows this tab window:

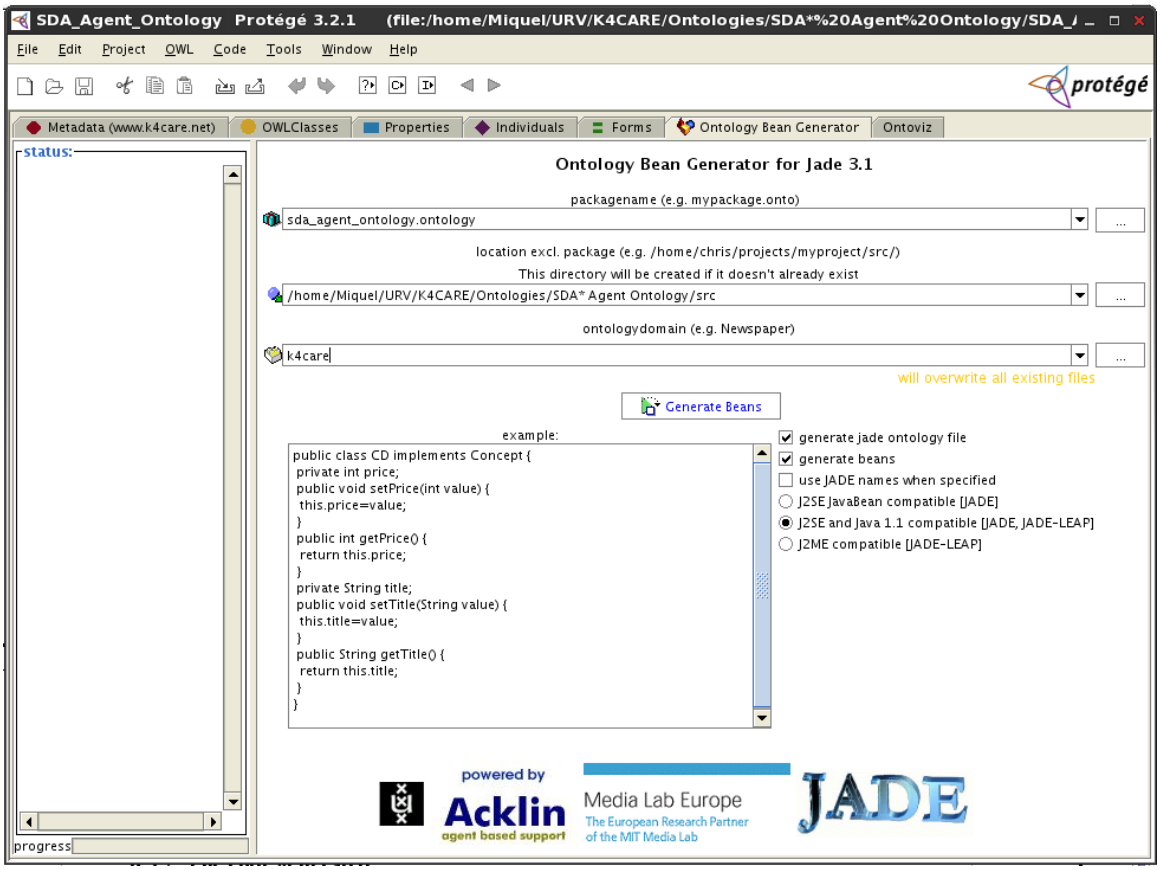


Figure 27. The Bean Generator Tab

After the parameters to create the ontology are introduced, we are able to generate the ontology beans with the *"Generate Beans"* button.

5.5.2. The generated code

After this process, Protégé has generated a set of classes that can be used to communicate concepts, actions and predicates in a JADE conversation. Figure 28 shows the "pool" of generated classes, eachone representing one concept oraction, and its attributes:

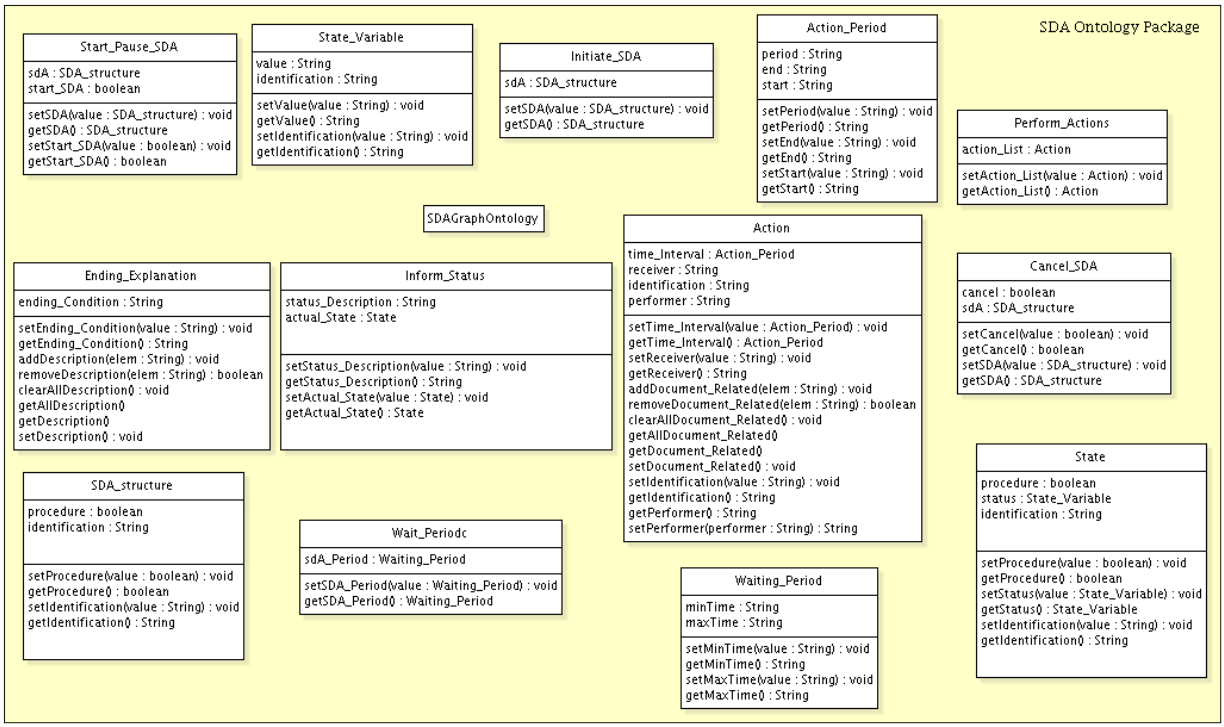


Figure 28. The SDA Agent communication ontology*

5.6. Controlling problematic situations

Finally, to finish the design description there are some problematic situations that have been solved following the next indications:

- As the K4CARE is a project still in development, the checking procedure of the patient variables has been done using the standard input, as there is another subproject in K4CARE whose goal is to obtain this information from the databases.
- The time constraints which are present in the SDA* specification are still not clear nowadays, like the parallelism of action execution. This work only treats the time restrictions as they are specified in the SDA* specification, so evaluates the time restrictions jumping between two nodes and informs about the repetition of one action to the agents requested to. It is planned that in future versions these issues will be treated.
- The error conditions haven't been clearly defined nowadays, so, the SDA* Agent only informs to its parent about any appearing error.

5.7. Developing in a distributed team

As K4CARE is an European project with some teams distributed among the continent the need of usage of collaborative tools is prior. This section comments the tools used to work in the project:

- It is necessary to have a tool to manage all the generated code, as the amount of code generated by all the team's members is sufficiently large to be complex to manage. The used tool has been subversion[16].
- We need to have a tool to compile the code written by each team member, as it's important to share a uniform way of compiling and generating the results of the written code. To achieve this aim, two tools have been used, Maven[15] (the one used for this part of the project) and Ant. This kind of tools have been used as nobody in each working unit of the whole project has to know how it exactly is distributed, so, only installing and using the provided configuration files for each project unit by each responsible developer every one will be capable to generate each part.
- As the code can have errors, it's important to have a testing tool in order to verify the code, and at the same time, use one with standarized characteristics, as Junit[14].
- As the generation of the ontology code is a boring and repetitive process it is interesting to have a tool to generate this code automatically, and also to manage the OWL files generated, as the Protégé + Ontology bean generator[11].

6. Testing

To test the development that has been done, three level of tests have been performed:

- ✔ test of the I2P package as a data structure: To test this part of the project, a testing package has been designed using JUnit. In this package the creation, modification and the reading of this data structure has been tested, and it has worked correctly.

  As the IDE (Eclipse [10]) used has JUnit integration it has been quite easy to integrate the designed tests with the generated code. To do so, a new package has been added into the package structure of the project:

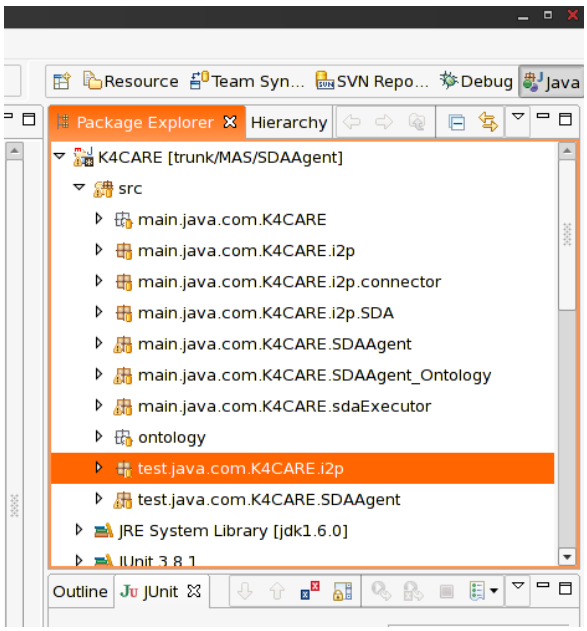


Figure 29. The package structure

This package contains basically one test case, divided in 3 parts:

- ✔ Elements construction test, which tests the correct construction of the different elements in a SDA*, the States, the Decisions and the Actions.
- ✔ Elements referencing test, tests the creation and functionality of the branches.
- ✔ Structure test, tests the SDA* graph structure by itself

In the next piece of code it is possible to see the internal structure of this test case:

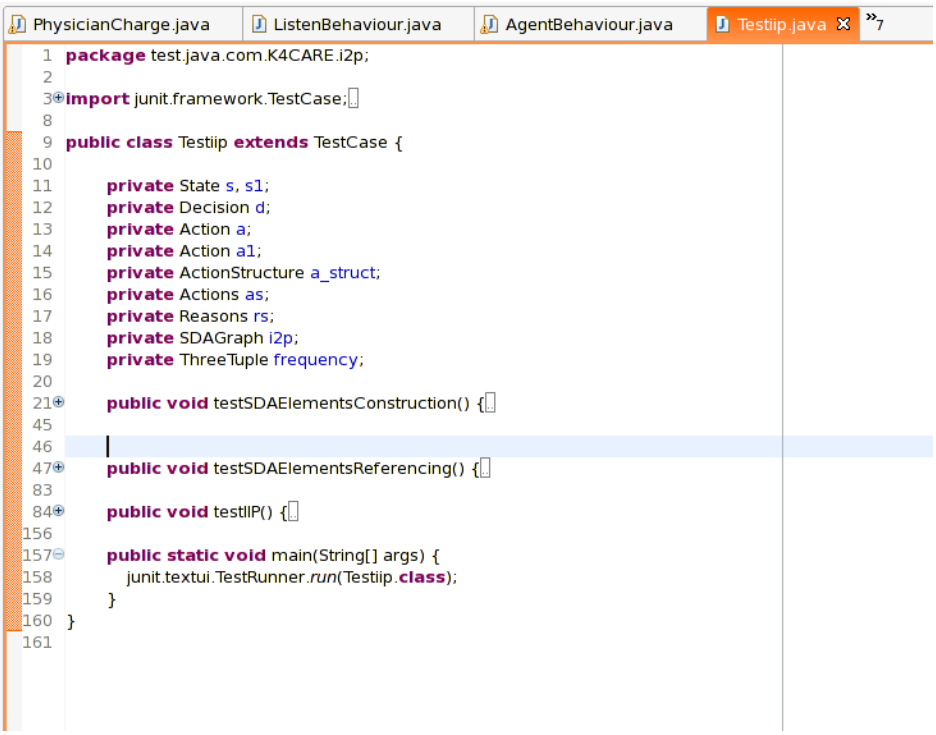


Figure 30. SDA Structure testing*

After developing this code, the test finishes correctly, so it's possible to conclude that the SDA Graph structure is sufficiently robust and bug-free to be used:

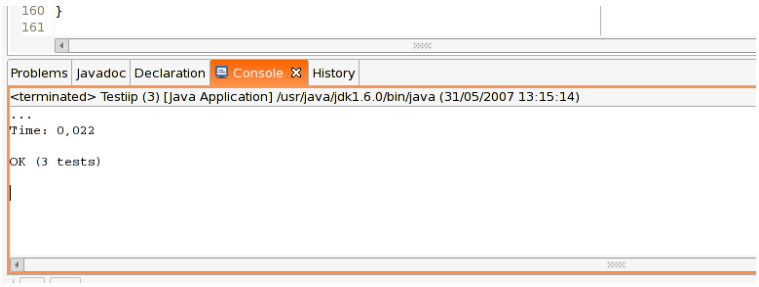


Figure 31. The JUnit testing results

✔ test of the executor engine:

The second level of test is centered in the execution engine. In this test the SDA Agent execution capabilities have been tested. To do so, some simple SDA* structures have been developed using the SDA:

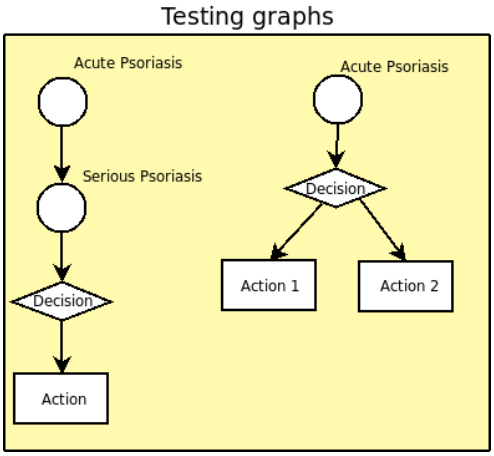


Figure 32. SDA tested graphs structure*

Some changes have been performed over these structures, like the introduction of some temporal restrictions, or cycles between the nodes. The main goals of these tests were accomplished:

- ✔ Test the interpretation of the SDA* graph structure
- ✔ Test the Agent event-based proactive behaviour
- ✔ Continue with the evaluation of the SDA* structure

Due to the amount of tests done and the output generated it is impossible to write all them here, however, an example of the output generated from the execution of the first diagram presented above is shown here:

```
Agent SDAExecutioner: started
Agent Behaviour:I've this possible Entry Points:
0) main.java.com.K4CARE.i2p.SDA.State: Acute Psoriasis
1) main.java.com.K4CARE.i2p.SDA.State: Serious Psoriasis
Agent Behaviour:Choose one Entry Point by its id:
 Serious Psoriasis
Agent Behaviour:Choose one Entry Point by its id:
Serious Psoriasis
Agent Behaviour:I've detected a new state reached event
Agent Behaviour:The state is: Serious Psoriasis
SDAExecutor: I've detected an state element. I need the next info Psoriasis
SDAExecutor: I'm in a main.java.com.K4CARE.i2p.SDA.State node. With 0
cycles.
SDAExecutor: I'm going to jump to another node without any label or time
SDAExecutor: I'm in a main.java.com.K4CARE.i2p.SDA.Decision node. With 1
cycles.
SDAExecutor: I'm going to jump to another node without any label or time
SDAExecutor: An action has been reached
Agent Behaviour:I've detected a new action reached event, now I read the
action: What to do
Agent Behaviour:I've detected that the graph has finished. I finish my
execution
```

In this code extract it is possible to see the interaction between the agentBehaviour and the SDAExecutor and the different messages which these two elements exchange. It's also possible for the reader to notice the interaction with the Data Abstraction Layer via text input, as it is still under development by other member of the project.

✔ test of the SDA* Agent in an Agent environment:

This final test is centered in the SDA Agent interaction with other agents. The main goals in this test are:

- To evaluate the correct functionality of the ontology
- Test the interaction of the SDA Agent in a Multi-Agent System
- Verify tests one and two

To do so, in this test a new SDA Structure has been created, in this case the invented SDA* graph structure from section 3.3.3 has been used (see figure 12), but some modifications have been introduced, concretely no Nurse agent has been created, and the tasks related to the Nurse agent have been assigned to the Head Nurse. This has been done in order to simplify a few the exchanged messages and to remove an actor that does not give any new functionality to the test. So in this test we have 4 static agents:

- A Physician in Charge
- A Family Doctor
- The Head Nurse
- The Patient

And one dynamic agent created by the Head Nurse, the SDA Agent. The first task of these agents is to register to the DF, and after this is done, the Head Nurse creates a new SDA Agent who also registers to the DF. When all the agents have registered, the SDA Agent starts the execution of the SDA structure mentioned before, and we have obtained the following results.

After defining the actors in our test system, the execution of the SDA* structure has been started. Like in the second test, the output generated is quite long. So, in order to reduce it, only the execution of one branch of the SDA* structure is shown. In this case the chosen one is the “No Functionality” branch, related with “Kidney Problems”. Here we have the text output:

```
Agent FD_Julius.Hibert started.
FD_Julius.Hibert is going to register to the DF.
Agent Patient_Homer.Simpson started.
Patient_Homer.Simpson is going to register to the DF.
Agent PC_Nick.Riviera started.
PC_Nick.Riviera is going to register to the DF.
Agent FD_Julius.Hibert: I'm registered.
Julius Hibert: I'm doing something
Agent HN_Edna.Krabappel started.
HN_Edna.Krabappel is going to register to the DF.
Agent Patient_Homer.Simpson: I'm registered.
Homer Simpson: I'm doing something
Agent PC_Nick.Riviera: I'm registered.
Nick Riviera: I'm doing something
Agent HN_Edna.Krabappel: I'm registered.
Edna Krabappel: I'm doing something
Agent HN_sdaAgent: started
Agent HN_sdaAgent: My father is => HN_Edna.Krabappel
Agent Behaviour:I've this possible Entry Points:
     0) main.java.com.K4CARE.i2p.SDA.State: Kidney Problems
     1) main.java.com.K4CARE.i2p.SDA.State: Low Kidney Functionality
     Agent Behaviour: Choose one Entry Point by its id:
     Kidney Problems
Agent Behaviour:I've detected a new state reached event
Agent Behaviour:The state is: Kidney Problems
SDAExecutor: I've detected an state element. I need the next info Kidney
failure
SDAExecutor: I'm in a main.java.com.K4CARE.i2p.SDA.State node. With 1
cycles.
SDAExecutor: I'm going to jump to another node without any label or time
SDAExecutor: I'm in a main.java.com.K4CARE.i2p.SDA.Decision node. With 1
cycles.
SDAExecutor: I'm going to jump to another node without any label or time
SDAExecutor: A decision has been reached
Agent Behaviour:I've detected a new decision reached event
Agent Behaviour:The decision is: Perform adequate actions and has 3
connections
Agent Behaviour:I've this possible connections:
     0) Otherwise
     1) No functionality
     2) Low functionality
     Agent Behaviour: Choose one:
     No functionality
SDAExecutor: I'm in a main.java.com.K4CARE.i2p.SDA.Decision node
SDAExecutor: I'm going to jump to another node with a reasoning to
accomplish
SDAExecutor: An action has been reached
Agent Behaviour:I've detected a new action reached event, now I read the
action: Assistive devices
HN_sdaAgent I've sent: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:39 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:39 CEST 2007")
:Performer PC :Document_Related (sequence "D. Assistive Devices") :Receiver
Patient :Identification "Assistive devices"))))
HN_sdaAgent: I'm going to wait the necessary time
Edna Krabappel I've received: ((action (agent-identifier :name
```

```
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:39 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:39 CEST 2007")
:Performer PC :Document_Related (sequence "D. Assistive Devices") :Receiver
Patient :Identification "Assistive devices"))))
Edna Krabappel: I've received the next order => Assistive devices
Edna Krabappel: I agree with the asked action
HN_sdaAgent: I confirm the response
HN_sdaAgent: I've received an agree response
Edna Krabappel: confirmed
Edna Krabappel: I'm going to wait the necessary time
Nick Riviera: I've received: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:39 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:39 CEST 2007")
:Performer PC :Document_Related (sequence "D. Assistive Devices") :Receiver
Patient :Identification "Assistive devices"))))
Nick Riviera: I've received the next order ==>Assistive devices
Nick Riviera: I agree with the asked action
HN_Edna.Krabappel: I confirm the response
HN_Edna.Krabappel: I've received an agree response
Nick Riviera: confirmed
HN_Edna.Krabappel: Action asked to PC_Nick.Riviera done.
Nick Riviera: I'm doing something
Edna Krabappel: I'm doing something
HN_sdaAgent: Action asked to HN_Edna.Krabappel done.
SDAExecutor: I'm in a main.java.com.K4CARE.i2p.SDA.Action node. With 1
cycles.
SDAExecutor: I'm going to jump to another node without any label or time
SDAExecutor: An action has been reached
Agent Behaviour:I've detected a new action reached event, now I read the
action: Cure the Patient
HN_sdaAgent I've sent: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:40 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:40 CEST 2007")
:Performer PC :Document_Related (sequence "D. Nursing Care") :Receiver
Patient :Identification "Cure the Patient"))))
HN_sdaAgent: I'm going to wait the necessary time
Edna Krabappel I've received: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:40 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:40 CEST 2007")
:Performer PC :Document_Related (sequence "D. Nursing Care") :Receiver
Patient :Identification "Cure the Patient"))))
Edna Krabappel: I've received the next order => Cure the Patient
HN_sdaAgent: I confirm the response
HN_sdaAgent: I've received an agree response
Edna Krabappel: I agree with the asked action
Edna Krabappel: confirmed
Edna Krabappel: I'm going to wait the necessary time
Nick Riviera: I've received: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
```

```
10:50:40 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:40 CEST 2007")
:Performer PC :Document_Related (sequence "D. Nursing Care") :Receiver
Patient :Identification "Cure the Patient"))))
Nick Riviera: I've received the next order ==>Cure the Patient
Nick Riviera: I agree with the asked action
HN_Edna.Krabappel: I confirm the response
HN_Edna.Krabappel: I've received an agree response
Nick Riviera: confirmed
HN_Edna.Krabappel: Action asked to PC_Nick.Riviera done.
HN_sdaAgent: Action asked to HN_Edna.Krabappel done.
HN_sdaAgent I've sent: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:41 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:41 CEST 2007")
:Performer FD :Document_Related (sequence "D. authorize nursing care")
:Receiver Patient :Identification "Cure the Patient"))))
HN_sdaAgent: I'm going to wait the necessary time
Nick Riviera: I'm doing something
Edna Krabappel: I'm doing something
Edna Krabappel I've received: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:41 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:41 CEST 2007")
:Performer FD :Document_Related (sequence "D. authorize nursing care")
:Receiver Patient :Identification "Cure the Patient"))))
Edna Krabappel: I've received the next order => Cure the Patient
HN_sdaAgent: I confirm the response
HN_sdaAgent: I've received an agree response
Edna Krabappel: I agree with the asked action
Edna Krabappel: confirmed
Edna Krabappel: I'm going to wait the necessary time
Julius Hibert: I've received: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:41 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:41 CEST 2007")
:Performer FD :Document_Related (sequence "D. authorize nursing care")
:Receiver Patient :Identification "Cure the Patient"))))
Julius Hibert: I've received the next order ==>Cure the Patient
Julius Hibert: I agree with the asked action
HN_Edna.Krabappel: I confirm the response
HN_Edna.Krabappel: I've received an agree response
Julius Hibert: confirmed
HN_Edna.Krabappel: Action asked to FD_Julius.Hibert done.
HN_sdaAgent: Action asked to HN_Edna.Krabappel done.
HN_sdaAgent I've sent: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:42 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:42 CEST 2007")
:Performer HN :Document_Related (sequence "D. intravenous therapy")
:Receiver Patient :Identification "Cure the Patient"))))
HN_sdaAgent: I'm going to wait the necessary time
Julius Hibert: I'm doing something
Edna Krabappel: I'm doing something
Edna Krabappel I've received: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
```

```
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:42 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:42 CEST 2007")
:Performer HN :Document_Related (sequence "D. intravenous therapy")
:Receiver Patient :Identification "Cure the Patient"))))
Edna Krabappel: I've received the next order => Cure the Patient
Edna Krabappel: I agree with the asked action
HN_sdaAgent: I confirm the response
HN_sdaAgent: I've received an agree response
Edna Krabappel: confirmed
HN_sdaAgent: Action asked to HN_Edna.Krabappel done.
HN_sdaAgent I've sent: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:43 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:43 CEST 2007")
:Performer HN :Document_Related (sequence "D. follow-up report") :Receiver
Patient :Identification "Cure the Patient"))))
HN_sdaAgent: I'm going to wait the necessary time
Edna Krabappel: I'm doing something
Edna Krabappel I've received: ((action (agent-identifier :name
HN_Edna.Krabappel@morpheus:1099/JADE :addresses (sequence
http://morpheus:7778/acc http://morpheus:42311/acc)) (Perform_Action
:Action_List (SDA_Action :Time_Interval (Action_Period :End "Thu Jun 07
10:50:43 CEST 2007" :Period "0" :Start "Thu Jun 07 10:50:43 CEST 2007")
:Performer HN :Document_Related (sequence "D. follow-up report") :Receiver
Patient :Identification "Cure the Patient"))))
Edna Krabappel: I've received the next order => Cure the Patient
Edna Krabappel: I agree with the asked action
HN_sdaAgent: I confirm the response
HN_sdaAgent: I've received an agree response
Edna Krabappel: confirmed
HN_sdaAgent: Action asked to HN_Edna.Krabappel done.
Edna Krabappel: I'm doing something
SDAExecutor: I'm in a main.java.com.K4CARE.i2p.SDA.Action node
SDAExecutor: I'm going to jump to another node with a time reasoning
Agent Behaviour: I've detected a new time event
Agent Behaviour: The execution has been late, so it will finish
Agent Behaviour: The execution has been aborted, so bye bye dear
```

The agents after registering in the DF remain waiting for action requests. Once the SDA Agent arrives to an Action Block from the SDA* structure, he sends to the related agent the tasks that are there written, and sends an agree message, which is replied with a confirm message, and agreed by the SDA Agent. If we analyze the outputs generated by the agents it's possible to see the contents of the request messages, and there we can see an ontology message codified in the SL codec. It's interesting to notice that, in this test case, the patient isn't requested to do anything because the actions in the SDA* structure are requested to the other agents, and the responsible of these agents are who must perform these actions over the patient.

Finally when the execution is finished, the SDA Agent finalizes its execution and the other agents remain waiting new messages.

In the next figure is shown a global view of this message exchange:

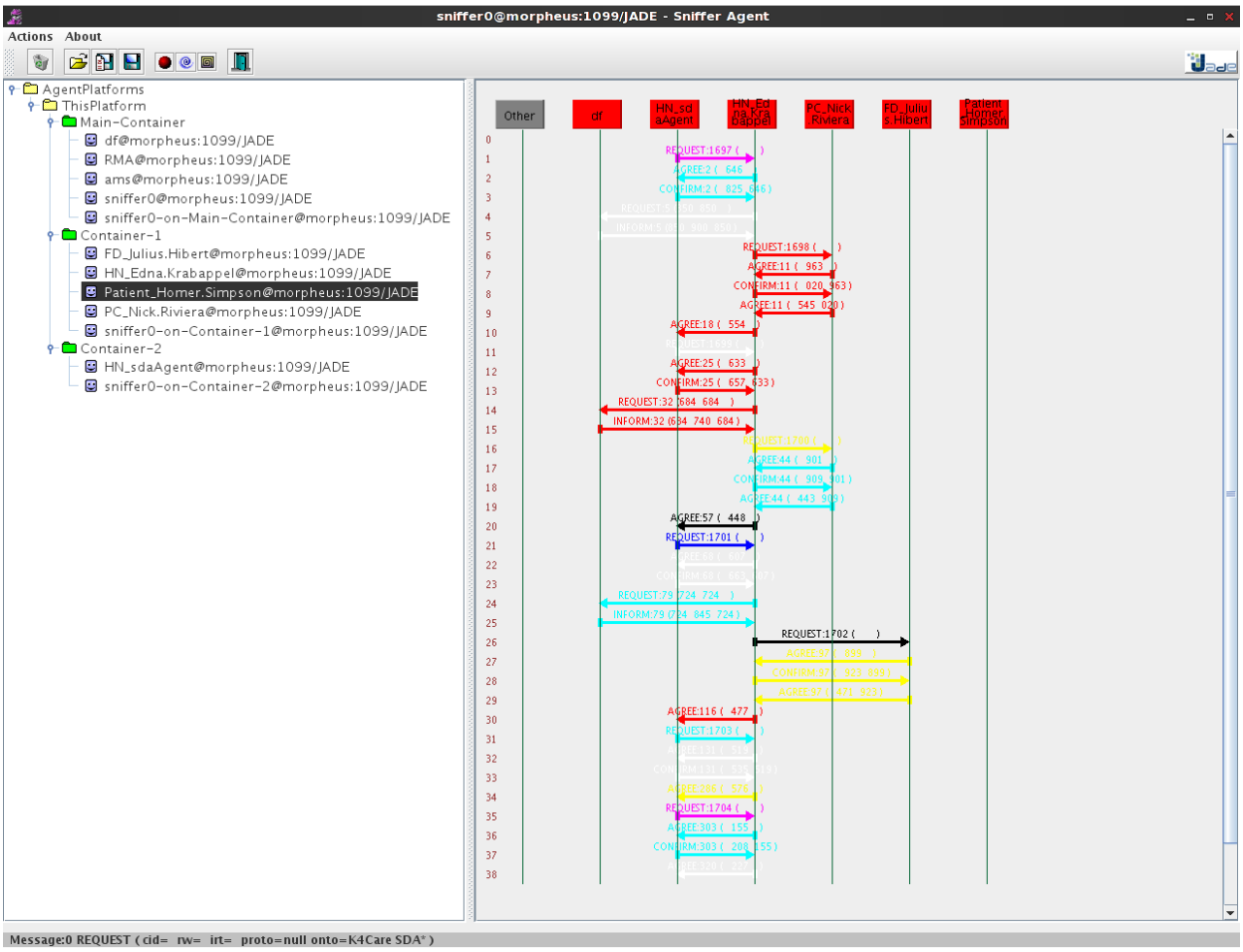


Figure 33. Message exchange between agents in an SDA execution*

So, after these three testing packages it's possible to conclude that the SDA Agent and structure are working fine as the SDA* structure works correctly, the engine interprets correctly the structures, and finally, the communication between agents also work accurately.

7. Conclusions and future work

The most important conclusion to which we have arrived in this project is that Multi-Agent Systems technologies are a great approach in the field of Artificial Intelligence, as they combine some elements from the real world as capabilities to perform intelligent processes, multitask functionalities, and the most important, collaboration possibilities. This last feature is by far the most interesting of all the features given by Multi-Agent Systems, as it has been possible to see during the development of this project where an agent has been designed to coordinate with other agents. It's important to remark the importance of the fact that all the issues concerning Multi-Agent Systems have an standarisation entity like the FIPA (which is an IEEE Computer Society standards organization) who regulates them. This avoids the possible incompatibilities between different implementations and gives detailed documentation about how and why a protocol or the messages between the agents are defined.

In second term, some conclusions about the SDA* formalism. This is a great approach to represent a medical guideline, because it has a graph structure (easily understandable by computer engineers) and thanks to its "tree" visual representation it's also useful for the medical staff (who are the most interested on them). However, there are some aspects that we think that this formalism has to improve. The first one is the possibility of using first order logic (feature that is in progress nowadays), because is more closer to the real world than propositional logic, giving to the doctors (an also to the computer science people) richer semantical content in the representation. In second place, and from the point of view of the agents, it's important to strictly define which parallel behaviours are being represented in a SDA* graph, as nowadays it hasn't a clear representation and it's important to model a more real world.
A global conclusion that can be extracted from the SDA* is the fact that it is a good approximation, but it needs some refinements in order to be more precise.

In third place, it's important to remark the different utilities that Software Engineering gives to the developer, as the design patterns. Without this kind of tools or "recipes" (because the design patterns are more like recipes than tools) it would be very difficult to solve the final design of the system in a clear and understandable way.

In fourth place and to finish the conclusions, we would like to stress the important paper of the tools used to work in the K4CARE team. Without these tools it would be very complicated to maintain, share and use the generated code and the generated designs. So thanks to tools like subversion it has been easy to use the code generated by other team members, or with tools like JUnit it has been possible to test the correct functionalities of the generated code, in order to find the possible bugs present on it.

Nowadays, the K4CARE Project is in its second year. During the first year the design of the ontologies, the documents and the architecture of the system have been done. In this second year the development of the Multi-Agent System will be done, so the work presented in this document will be integrated with the work done by all the other project members, and at the same time it will be revised and extended with the new needed features. As future features it would be interesting to implement the SDA* formalism completely including the first order logic and the parallelism (and all the issues related to these implementations) and also integrating all this code with the rest of the final Multi-Agent System.

To finalise this document, I would like to thank all the comments and revisions given by Dr. Antonio Moreno in his task of directing this work, to David Isern for his comments and ideas for the design and development of the Agent, to David Sánchez for the help given to design and develop the ontology and to Montserrat Batet, Albert Solé and Joan Casals for the help given during the development of some parts of the code. I'd like also to thank the K4CARE Hungarian and Czech people the comments given during the development of the work.

8. Annex

8.1. Annex 1. What is a Multi-Agent System?

Before starting to explain what are the Multi-Agent Systems, it's important to understand what are their basic elements, the *Agents*.

It's easy to find some definitions about what is an agent as this is a large field of research, but one of the most accepted ones is the proposed by Michael Wooldridge collected in [03]:

> - An intelligent agent is a computational process which is capable to perform tasks in an autonomous way, and which could communicate with other agents in order to resolve problems cooperating, coordinating and negotiating with them. The agents live in a complex and dynamic environment with whom they interact in order to achieve a set of objectives.

8.1.1. Agent Properties

An intelligent agent is a hardware or (more usually) software-based computer system that enjoys the following properties:

- *autonomy*: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
- *social ability*: agents interact with other agents (and possibly humans) via some kind of agent-communication language;
- *reactivity*: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;
- *pro-activeness*: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavour by taking the initiative.
- *Reasoning/learning*: agents must capable to increase the performance of their acts using some kind of learning or reasoning techniques.
- *Mobility*: agents must be able to physically move between different computers. This is not an essential property, but it can be desired.
- *Temporal continuity*: agents will be executing processes continuously.
- *Truthfulness*: agent will always communicate true information.
- *Benevolence*: agents will not have conflictive objectives and it will perform what it is asked for.
- *Intelligence*: From the other properties, it's possible to conclude that an agent would use some artificial intelligence techniques in order to solve the problems which it would have.

8.1.2. Agent Types

Agents also can be of some types, in this section the most important of these types will be presented but, it's possible that there will be more agent types that won't be listed here:

- *Collaborative*: This kind of agents collaborate with other agents in order to achieve one concrete objective.
- *Interface*: This type of agents are the ones which join forces with the user in the achievement of one concrete objective.
- *Mobile*: This sort of agents are characterized by having mobility capabilities.
- *Internet*: Typology of agents related to the search and manipulation of information through Internet.
- *Reactive*: Agents which react to external stimulation, without having any explicit world model nor reasoning or planification.
- *Hybrid*: This last sort of agents are combinations of two or more types of the ones listed above.

8.1.3. Multi-Agent Systems

After presenting the concepts related to the term agent we are ready to understand the Multi-Agent theory, so we define a Multi-Agent System as that in one set of agents cooperate, coordinate and communicate in order to achieve a common objective.
In this subsection, the typical properties and advantages of the Multi-Agent Sustems are presented.

8.1.3.1. Multi-Agent Systems advantages

The principal advantages of using Multi-Agent systems are the next:

- *Modularity*: The programming complexity is reduced because the working units are smaller, this advantage also relies in a more structure programming.
- *Eficiency*: The distributed programming permits to distribute the tasks among the agents achieving parallelism.
- *Reliability*: The fact that a system element breaks its work hasn't mean that the rest of the elements stop their work also; although it's possible to achieve more security due it's possible to replicate critical services, andso, obtaining redundancy.
- *Flexibility*: It's possible to add and delete agents dinamically.

8.1.3.2. Multi-Agent Systems management

The agent administration stablishes a logical model for the creation, register, communication, mobility and destruction of agents. As the FIPA is the standarisation organization of all the related protocols, architectures, etc. related to the agents, in this project their management structure will be used. This structure is presented in the figure 34, and has the next components

- An **agent** is a computational process that implements the autonomous, communicating functionality of an application. Agents communicate using an Agent Communication Language. An Agent is the fundamental actor on an AP which combines one or more service capabilities, as published in a service description, into a unified and integrated execution model. An agent must have at least one owner, for example, based on organisational affiliation or human user ownership, and an agent must support at least one notion of identity. This notion of identity is the Agent Identifier (AID) that labels an agent so that it may be distinguished unambiguously within the Agent Universe. An agent may be registered at a number of transport addresses at which it can be contacted.
- A **Directory Facilitator (DF)** is an optional component of the AP, but if it is present, it must be implemented as a DF service. The DF provides yellow pages services to other agents. Agents may register their services with the DF or query the DF to find out what services are offered by other agents, including the discovery of agents and their offered services in ad hoc networks. Multiple DFs may exist within an AP and may be federated.
- An **Agent Management System (AMS)** is a mandatory component of the AP. The AMS exerts supervisory control over access to and use of the AP. Only one AMS will exist in a single AP. The AMS maintains a directory of AIDs which contain transport addresses (amongst other things) for agents registered with the AP. The AMS offers white pages services to other agents. Each agent must register with an AMS in order to get a valid AID.
- An **Message Transport Service (MTS)** is the default communication method between agents on different APs.
- An **Agent Platform (AP)** provides the physical infrastructure in which agents can be deployed. The AP consists of the machine(s), operating system, agent support software, FIPA agent management components (DF, AMS and MTS) and agents.

  The internal design of an AP is an issue for agent system developers and is not a subject of standardisation within FIPA. AP's and the agents which are native to those APs, either by creation directly within or migration to the AP, may use any proprietary method of inter-communication.

  It should be noted that the concept of an AP does not mean that all agents resident on an AP have to be co-located on the same host computer. FIPA envisages a variety of

different APs from single processes containing lightweight agent threads, to fully distributed APs built around proprietary or open middleware standards.

FIPA is concerned only with how communication is carried out between agents who are native to the AP and agents outside the AP. Agents are free to exchange messages directly by any means that they can support.

- **Software** describes all non-agent, executable collections of instructions accessible through an agent. Agents may access software, for example, to add new services, acquire new communications protocols, acquire new security protocols/algorithms, acquire new negotiation protocols, access tools which support migration, etc.

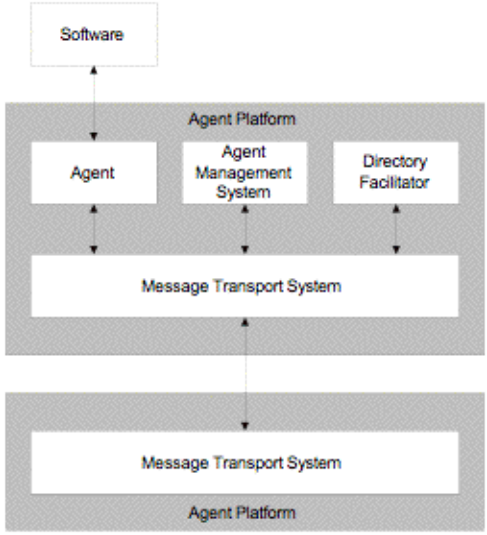


Figure 34. Multi-Agent system structure

8.1.3.3. Multi-Agent Systems Messages Structure

As said in section 4.1.1 agents must be capable to communicate between them. To achieve this, it's prior to have a clear communication protocol. The FIPA, as the organisation in charge of this subjects has published some standards [04] about the communication between agents, the contents of the messages send and the codification of them. In this section the most important of this messages are presented in order to have a clear idea about this communication issues.

A FIPA ACL message contains a set of one or more message parameters. Precisely which parameters are needed for effective agent communication will vary according to the situation; the only parameter that is mandatory in all ACL messages is the `performative`, although it is expected that most ACL messages will also contain `sender`, `receiver` and `content` parameters.

If an agent does not recognize or is unable to process one or more of the parameters or parameter values, it can reply with the appropriate `not-understood` message.

Specific implementations are free to include user-defined message parameters other than the FIPA ACL message parameters specified in Table 15. The semantics of these user-defined parameters is not defined by FIPA, and FIPA compliance does not require any particular interpretation of these parameters. The prefatory string "X-" must be used for the names of these non-FIPA standard additional parameters.

Some parameters of the message might be omitted when their value can be deduced by the context of the conversation. However, FIPA does not specify any mechanism to handle such conditions, therefore those implementations that omit some message parameters are not guaranteed to interoperate with each other.

The full set of FIPA ACL message parameters is shown in Table 15 without regard to their specific encodings in an implementation. FIPA-approved encodings and parameter orderings for ACL messages are given in other specifications. Each ACL message representation specification contains precise syntax descriptions for ACL message encodings based on XML, text strings and several other schemes.

Parameter	**Category of Parameters**
performative	Type of communicative acts
sender	Participant in communication
receiver	Participant in communication
reply-to	Participant in communication
content	Content of message
language	Description of Content
encoding	Description of Content
ontology	Description of Content
protocol	Control of conversation
conversation-id	Control of conversation
reply-with	Control of conversation
in-reply-to	Control of conversation
reply-by	Control of conversation

Table 15. FIPA ACL Message Parameters

The following terms are used to define the ontology and the abstract syntax of the FIPA ACL message structure:

- **Frame**. This is the mandatory name of this entity that must be used to represent each instance of this class.
- **Ontology**. This is the name of the ontology, whose domain of discourse includes their parameters described in the table.

- **Parameter**. This identifies each component within the frame. The type of the parameter is defined relative to a particular encoding. Encoding specifications for ACL messages are given in their respective specifications.
- **Description**. This is a natural language description of the semantics of each parameter. Notes are included to clarify typical usage.
- **Reserved Values**. This is a list of FIPA-defined constants associated with each parameter. This list is typically defined in the specification referenced.

All of the FIPA message parameters share the frame and ontology shown in Table 16.

Frame	**fipa-acl-message**
Ontology	fipa-acl

Table 16. FIPA ACL Message Frame and Ontology

- Performative

Parameter	**Description**
performative	Denotes the type of the communicative act of the ACL message

Notes: The `performative` parameter is a required parameter of all ACL messages.

- Sender

Parameter	**Description**
sender	Denotes the identity of the sender of the message, that is, the name of the agent of the communicative act.

Notes: The `sender` parameter will be a parameter of most ACL messages. It is possible to omit the `sender` parameter if, for example, the agent sending the ACL message wishes to remain anonymous. The `sender` parameter refers to the agent which performs the communicative act giving rise to this ACL message.

- Receiver

Parameter	**Description**
receiver	Denotes the identity of the intended recipients of the message.

Notes: Ordinarily, the `receiver` parameter will be a part of every ACL message. It is

only permissible to omit the `receiver` parameter if the message recipient can be reliably inferred from context, or in special cases such as the embedded ACL message in `proxy` and `propagate`.

The `receiver` parameter may be a single agent name or a non-empty set of agent names. The latter corresponds to the situation where the message is multicast. Pragmatically, the semantics of this multicast is that the sender intends the message for each recipient of the CA encoded in the message. For example, if an agent performs an `inform` act with a set of three agents as receiver, it denotes that the sender intends each of these agents to come to believe the content of themessage.

- *Reply To*

Parameter	**Description**
reply-to	This parameter indicates that subsequent messages in this conversation thread are to be directed to the agent named in the `reply-to` parameter, instead of to the agent named in the `sender` parameter.

- *Content*

Parameter	**Description**
content	Denotes the content of the message; equivalently denotes the object of the action. The meaning of the content of any ACL message is intended to be interpreted by the receiver of the message. This is particularly relevant for instance when referring to referential expressions, whose interpretation might be different for the sender and the receiver.

Notes: Most ACL messages require a content expression. Certain ACL message types, such as `cancel`, have an implicit content, especially in cases of using the `conversation-id` or `in-reply-to` parameters.

- *Language*

Parameter	**Description**
language	Denotes the language in which the content parameter is expressed.

Notes: The ACL `content` parameter is expressed in a formal language. This field may be omitted if the agent receiving the message can be assumed to know the language of the content expression.

- *Encoding*

Parameter	Description
encoding	Denotes the specific encoding of the content language expression.

Notes: The content expression might be encoded in several ways. The `encoding` parameter is optionally used to specify this encoding to the recipient agent. If the `encoding` parameter is not present, the encoding will be specified in the message envelope that encloses the ACL message.

- *Ontology*

Parameter	Description
ontology	Denotes the ontology(s) used to give a meaning to the symbols in the content expression.

Notes: The `ontology` parameter is used in conjunction with the `language` parameter to support the interpretation of the content expression by the receiving agent. In many situations, the `ontology` parameter will be commonly understood by the agent community and so this message parameter may be omitted.

- *Protocol*

Parameter	Description
protocol	Denotes the interaction protocol that the sending agent is employing with this ACL message.

Notes: The `protocol` parameter defines the interaction protocol in which the ACL message is generated. This parameter is optional; however, developers are advised that employing ACL without the framework of an interaction protocol (and thus directly using the ACL semantics to control the agent's generation and interpretation of ACL messages) is an extremely ambitious undertaking.

Any ACL message that contains a non-null value for the `protocol` parameter is considered to belong to a conversation and it is required to respect the following rules:

- the initiator of the protocol must assign a non-null value to the `conversation-id` parameter,
- all responses to the message, within the scope of the same interaction protocol, should contain the same value for the `conversation-id` parameter, and,
- the timeout value in the `reply-by` parameter must denote the latest time by

which the sending agent would like to have received the next message in the protocol flow (not be confused with the latest time by which the interaction protocol should terminate).

- *Conversation Identifier*

Parameter	**Description**
conversation-id	Introduces an expression (a conversation identifier) which is used to identify the ongoing sequence of communicative acts that together form a conversation.

Notes: An agent may tag ACL messages with a conversation identifier to manage its communication strategies and activities. Typically this will allow an agent to identify individual conversations with multiple agents. It will also allow agents to reason across historical records of conversations.
It is required the usage of globally unique values for the `conversation-id` parameter in order to allow the participants to distinguish between several concurrent conversations. A simple mechanism to ensure uniqueness is the concatenation of the globally unique identifier of the sender agent to an identifier (for example, a progressive number) that is unique within the scope of the sender agent itself

- *Reply With*

Parameter	**Description**
reply-with	Introduces an expression that will be used by the responding agent to identify this message.

Notes: The `reply-with` parameter is designed to be used to follow a conversation thread in a situation where multiple dialogues occur simultaneously. For example, if agent *i* sends to agent *j* a message which contains:

`reply-with <expr>`

Agent *j* will respond with a message containing:

`in-reply-to <expr>`

- *In Reply To*

Parameter	**Description**
in-reply-to	Denotes an expression that references an earlier action to which this message is a reply.

- *Reply By*

Parameter	Description
reply-by	Denotes a time and/or date expression which indicates the latest time by which the sending agent would like to receive a reply.

Notes: The time will be expressed according to the sender's view of the time on the sender's platform. The reply message can be identified in several ways: as the next sequential message in an interaction protocol, through the use of the `reply-with` parameter, through the use of a `conversation-id` and so forth. The way that the reply message is identified is determined by the agent implementer.

8.1.3.4. Multi-Agent Systems Communication Protocols

Another important feature of the Multi-Agent Systems are the communication protocols between agents. With these protocols the agents are capable to perform complex activities. As seen in the last section there is a parameter in the category control of conversation which is related to the protocol. This protocol is who defines a set of rules or steps to follow in order to perform a conversation.

As happens in the other Multi-Agent Systems related themes the FIPA defines a set of communicative protocols[05], that are described here:

- *FIPA Request*

The FIPA Request Interaction Protocol (IP) allows one agent to request another to perform some action. The Participant processes the request and makes a decision whether to accept or refuse the request. If a refuse decision is made, then "refused" becomes true and the Participant communicates a refuse. Otherwise, "agreed" becomes true.
If conditions indicate that an explicit agreement is required (that is, "notification necessary" is true), then the Participant communicates an agree. The agree may be optional depending on circumstances, for example, if the requested action is very quick and can happen before a time specified in the reply-by parameter. Once the request has been agreed upon, then the Participant must communicate either:

- A `failure` if it fails in its attempt to fill the request,
- An `inform-done` if it successfully completes the request and only wishes to indicate that it is done, or,
- An `inform-result` if it wishes to indicate both that it is done and notify the initiator of the results.

Any interaction using this interaction protocol is identified by a globally unique, non-null conversation-id parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

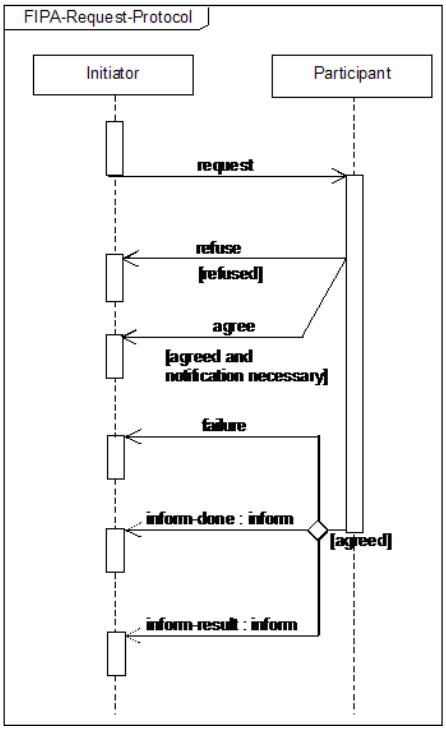


Figure 35. FIPA Request Interaction Protocol

FIPA Query

The Initiator requests the Participant to perform some kind of inform action using one of two query communicative acts, `query-if` or `query-ref`. The `query-if` communication is used when the Initiator wants to query whether a particular proposition is true or false and the `query-ref` communication is used when the Initiator wants to query for some identified objects. The Participant processes the `query-if` or `query-ref` and makes a decision whether to accept or refuse the query request. If the Participant makes a refuse decision, then "refused" becomes true and the Participant communicates a refuse. Otherwise, "agreed" becomes true.

If conditions indicate that an explicit agreement is required (that is, "notification necessary" is true), then the Participant communicates an agree. The agree may be optional depending on circumstances, for example, if the requested action is very quick and can happen before a time specified in the `reply-by` parameter. If the Participant fails, then it communicates a failure.

In a successful response, the Participant replies with one of two versions of inform:

- The Participant uses an `inform-t/f` communication in response to a `query-if` where the content of the `inform-t/f` asserts the truth or falsehood of the proposition, or,
- The Participant returns an `inform-result` communication in response to a `query-ref` and the content of the `inform-result` contains a referring expression to the objects for which the query was specified.

Any interaction using this interaction protocol is identified by a globally unique, non-null conversation-id parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

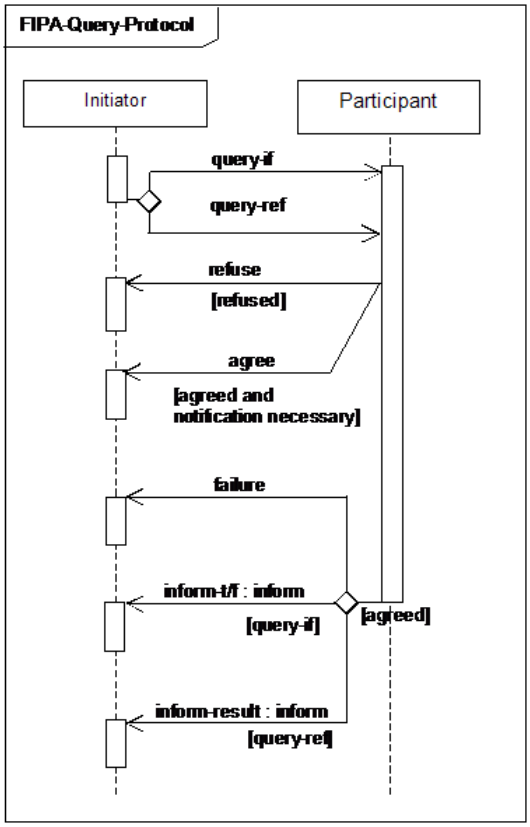


Figure 36. FIPA Query Interaction Protocol

FIPA Contract Net

The Initiator solicits *m* proposals from other agents by issuing a call for proposals (`cfp`) act , which specifies the task, as well any conditions the Initiator is placing upon the execution of the

task. Participants receiving the call for proposals are viewed as potential contractors and are able to generate *n* responses. Of these, *j* are proposals to perform the task, specified as `propose` acts.

The Participant's proposal includes the preconditions that the Participant is setting out for the task, which may be the price, time when the task will be done, etc. Alternatively, the *i=n-j* Participants may `refuse` to propose. Once the deadline passes, the Initiator evaluates the received *j* proposals and selects agents to perform the task; one, several or no agents may be chosen. The *l* agents of the selected proposal(s) will be sent an `accept-proposal` act and the remaining *k* agents will receive a `reject-proposal` act. The proposals are binding on the Participant, so that once the Initiator accepts the proposal, the Participant acquires a commitment to perform the task. Once the Participant has completed the task, it sends a completion message to the Initiator in the form of an `inform-done` or a more explanatory version in the form of an `inform-result.` However, if the Participant fails to complete thetask, a `failure` message is sent.

Note that this IP requires the Initiator to know when it has received all replies. In the case that a Participant fails to reply with either a `propose` or a `refuse` act, the Initiator may potentially be left waiting indefinitely. To guard against this, the `cfp` act includes a deadline by which replies should be received by the Initiator. Proposals received after the deadline are automatically rejected with the given reason that the proposal was late. The deadline is specified by the `reply-by` parameter in the ACL message.

Any interaction using this interaction protocol is identified by a globally unique, non-null `conversation-id` parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations. In the case of 1:N interaction protocols or sub-protocols the Initiator is free to decide if the same `conversation-id` parameter should be used or a new one should be issued. Additionally, the messages may specify other interaction-related information such as a timeout in the `reply-by` parameter that denotes the latest time by which the sending agent would like to have received the next message in the protocol flow.

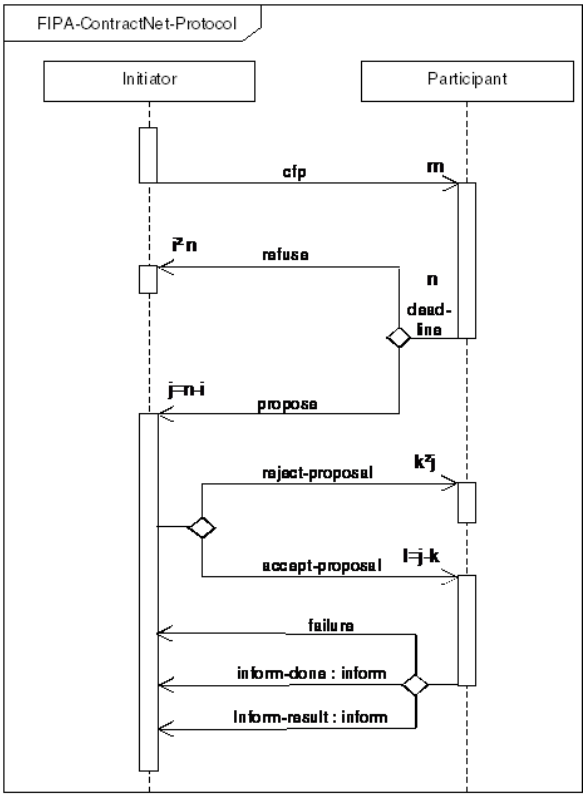


Figure 37. FIPA Contract Net Interaction Protocol

FIPA Iterated Contract Net

As with the FIPA Contract Net IP, the Initiator issues *m* initial call for proposals with the `cfp` act. Of the *n* Participants that respond, *k* are `propose` messages from Participants that are willing and able to do the task under the proposed conditions and the remaining *j* are from Participants that `refuse`.

Of the *k* proposals, the Initiator may decide this is the final iteration and accept *p* of the bids *(0 ≤ p ≤ k)*, and reject the others. Alternatively the Initiator may decide to iterate the process by issuing *a* revised `cfp` to *l* of the Participants and rejecting the remaining *k-l* Participants. The intent is that the Initiator seeks to get better bids from the Participants by modifying the call and requesting new (equivalently, revised) bids. The process terminates when the Initiator refuses all proposals and does not issue a new `cfp`, the Initiator accepts one or more of the bids or the Participants all refuse to bid.

Any interaction using this interaction protocol is identified by a globally unique, non-null `conversation-id` parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each

agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

In the case of 1:N interaction protocols or sub-protocols the Initiator is free to decide if the same `conversation-id` parameter should be used or a new one should be issued. Additionally, the messages may specify other interaction-related information such as a timeout in the `reply-by` parameter that denotes the latest time by which the sending agent would like to have received the next message in the protocol flow.

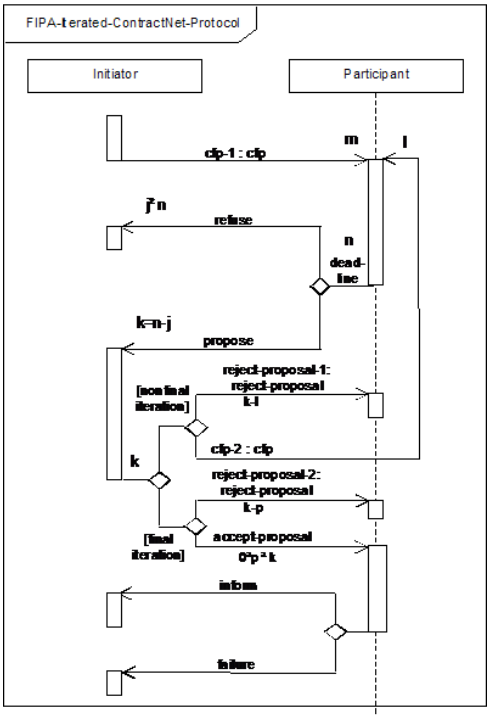


Figure 38. FIPA Iterated Contract Net Interaction Protocol

FIPA Brokering

The FIPA Brokering Interaction Protocol (IP) is a macro IP since the `proxy` communicative act for brokerage embeds a communicative act as its argument and so the IP for the embedded communicative act is also embedded in this IP. This embedded IP guides some parts of the remainder of the interaction, thus parts of this protocol are written very generically.

The Initiator of the brokering interaction begins the interaction with a `proxy` message which contains the following: a referential expression denoting the target agents to which the broker should forward the communicative act, the communicative act to forward and a set of proxy conditions such as the maximum number of agents to which the message should be forwarded. The Broker processes the request and makes a decision whether to agree to or refuse the request and communicates either an `agree` or a `refuse` communicative act accordingly. Communication of a `refuse` terminates the interaction.

Once the Broker has agreed to be a proxy, it then locates agents per the description from the

`proxy` message. If no such agents can be found, the Broker returns a `failure-no-match` and the interaction terminates. Otherwise, the Broker may modify the list of matching agents based on the `proxy-condition` parameter. It then begins *m* interactions with the resulting list of *n* agents with each interaction in its own separate sub-protocol. At this point, the Broker should record some of the ACL parameters, for example, `conversation-id, reply-with` and `sender`, of the received `proxy` message to return in the *r* replies to the Initiator.

Note that the nature of the sub-protocol and the nature of the replies are driven by the interaction protocols specified in the communicative act from the `proxy` message. As the sub-protocol progresses, the Broker forwards the responses that it receives from the sub-protocol to the Initiator. These messages are defined as the `reply-message-sub-protocol` communications, and may be either successful replies as defined by the sub-protocol or `failure`. If the initial proxy was an `inform`, there may in fact be no replies from the sub-protocol (and in fact means that the interaction is identical to a recruited inform). When the sub-protocol completes, the Broker forwards the final `reply-message` from the sub-protocol and the brokering IP terminates. However, there can be other failures that are not explicitly returned from the sub-protocol, for example, the agent that is executing the sub-protocol has failed. If the Broker detects such problems, it returns a `failure-brokering`, which terminates the IP.

A second issue to address occurs because multiple agents may match and therefore multiple sub-protocols (*m* of them) may be initiated by the Broker within the brokering IP. In this case, the Broker may collect the *n* received responses and combine them into a single `reply-message-sub-protocol`, or may forward the `reply-message-sub-protocol` messages from the separate sub-protocols individually ($1 \leq p \leq n$.). This is complicated by situations such as one agent responding with a `failure` while a second agent returns a `reply-message`, or the situation where results are inconsistent. The Broker must determine whether to resolve such situations internally or forward the responses to the Initiator. In doing this, the Broker must also be careful to avoid disruptive acts such as directly forwarding a `failure` from a sub-protocol, which would have the inadvertent effect of ending the brokering IP.

Any interaction using this interaction protocol is identified by a globally unique, non-null `conversation-id` parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

In the case of 1:N interaction protocols or sub-protocols the Initiator is free to decide if the same `conversation-id` parameter should be used or a new one should be issued. Additionally, the messages may specify other interaction-related information such as a timeout in the `reply-by` parameter that denotes the latest time by which the sending agent would like to have received

the next message in the protocol flow.

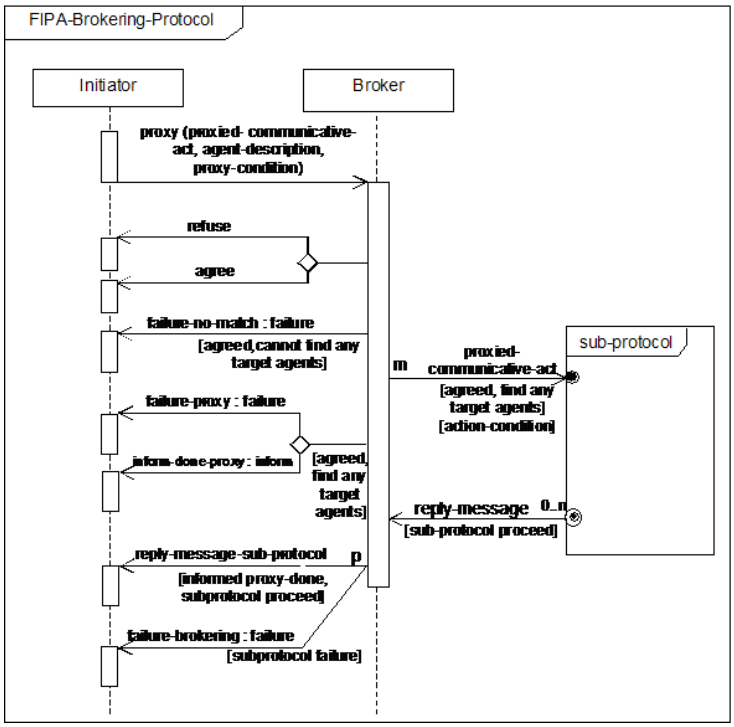


Figure 39. FIPA Brokering Interaction Protocol

FIPA Recruiting

The FIPA Recruiting Interaction Protocol (IP) is a macro IP since the `proxy` communicative actfor recruiting embeds a communicative act as its argument and so the IP for the embedded communicative act is also embedded in this IP. This embedded IP guides some parts of the remainder of the interaction, thus parts of this protocol are written very generically.

The Initiator of the recruiting interaction begins the interaction with a `proxy` message which contains the following: a referential expression denoting the target agents to which the recruiter should forward the communicative act, the communicative act to forward and a set of proxy conditions such as the maximum number of agents to be forwarded. The Recruiter processes the request and makes a decision whether to agree to or refuse the request, and communicates either an `agree` or a `refuse` communicative act accordingly. Communication of a `refuse` terminates the interaction.

Once the Recruiter has agreed to be a proxy, it then locates agents per the description from the `proxy` message. If no such agents can be found, the Recruiter returns a `failure-no-match` and the interaction terminates. Otherwise, the Recruiter may modify the list of matching agents based on the `proxy-condition` parameter. It then begins *m* interactions with the resulting list of *n* agents with each interaction in its own separate sub-protocol. The initiation of the sub-protocol should be done with care, using the ACL parameters to correlate the responses to the

request. If the Recruiter has been given a message containing a separate `designated-receiver` parameter from the interaction Initiator, it needs to start each sub-protocol with a `reply-to` parameter containing the Designated Receiver and the `conversation-id` of the original conversation. If the Recruiter instead is to indicate that the Initiator should receive the replies, then the `reply-to` parameter should designate the Initiator and the `conversation-id` of the recruiting conversation. Other ACL parameters may also need to be propagated.

Note that the nature of the sub-protocol and the nature of the replies are driven by the interaction protocols specified in the communicative act from the proxy message. As the sub-protocol progresses, it forwards its responses back either to the Designated Receiver or to the Initiator, depending on the value of the `reply-to` parameter in the `proxy` message. These messages are defined as `reply-message-sub-protocol` communications and may be either successful replies as defined by the sub-protocol or `failure`. If the initial proxy was an `inform`, there may in fact be no replies from the sub-protocol (and in fact means that the interaction is identical to a brokered `inform`). When the sub-protocol completes, the Recruiter forwards the final `reply-message-sub-protocol` from the sub-protocol and the recruiting IP terminates.

A second issue to address occurs because multiple agents may match and therefore multiple sub-protocols may be initiated by the Recruiter within the recruiting IP. In this case, the sub-protocols may be communicating multiple `reply-message-sub-protocol` communications from the different agents involved in the IP (for a total of *m* responses). This is complicated by such situations as one sub-protocol responding with a `failure` while a second sub-protocol returns a `reply-message-sub-protocol`, or the situation where results are inconsistent. The agent that receives the messages must determine how to detect and resolve such situations internally.

Any interaction using this interaction protocol is identified by a globally unique, non-null `conversation-id` parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

In the case of 1:N interaction protocols or sub-protocols the Initiator is free to decide if the same `conversation-id` parameter should be used or a new one should be issued. Additionally, the messages may specify other interaction-related information such as a timeout in the `reply-by` parameter that denotes the latest time by which the sending agent would like to have received the next message in the protocol flow.

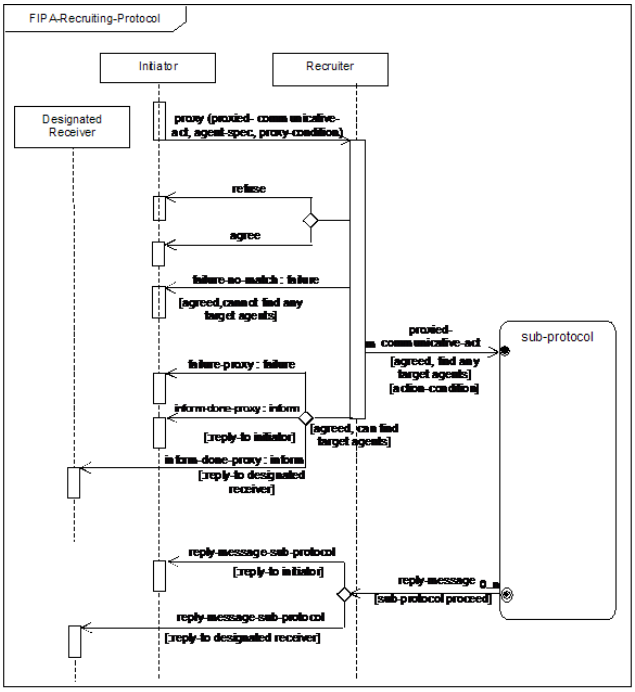


Figure 40. FIPA Recruiting Interaction Protocol

FIPA Subscribe

The Initiator begins the interaction with a subscribe message containing the reference of the objects in which they are interested. The Participant processes the `subscribe` message and makes a decision whether to accept or refuse the query request. If the Participant makes a refuse decision, then "refused" becomes true and the Participant communicates a `refuse`. Otherwise, "agreed" becomes true.

If conditions indicate that an explicit agreement is required (that is, "notification necessary" is true), then the Participant communicates an `agree`. The `agree` may be optional depending on circumstances, for example, if the requested action is very quick and can happen before a time specified in the `reply-by` parameter.

In a successful response, the Participant replies with an `inform-result` communication with the content being a referring expression to the subscribed objects. The Participant continues to send `inform-result` messages as the objects denoted by the referring expression change. If at some point after the Participant agrees, it experiences a failure, then it communicates this with a `failure` message, which also terminates the interaction. Otherwise, the interaction may be terminated by the Initiator using the cancel meta-protocol.

Any interaction using this interaction protocol is identified by a globally unique, non-null `conversation-id` parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

Additionally, because it may be important to preserve the sequence of the `inform-result` messages, it is important that the message transport used for this IP preserve the ordering of messages.

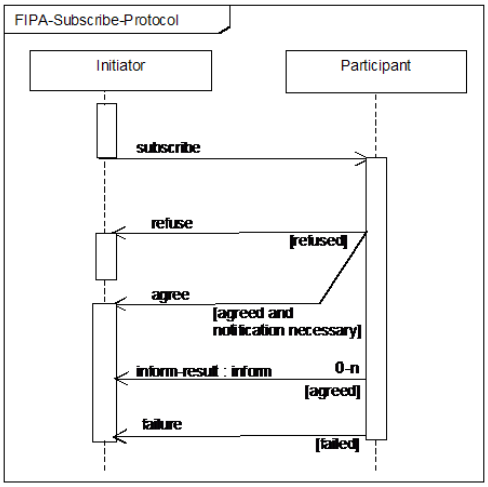


Figure 41. FIPA Subscribe Interaction Protocol

FIPA Propose

The Initiator sends a `propose` message to the Participant indicating that it will perform some action if the Participant agrees. The Participant responds by either accepting or rejecting the proposal, communicating this with the `accept-proposal` or `reject-proposal` communicative act, accordingly. Completion of this IP with an `accept-proposal` act would typically be followed by the performance by the Initiator of the proposed action and then the return of a status response.

Any interaction using this interaction protocol is identified by a globally unique, non-null `conversation-id` parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

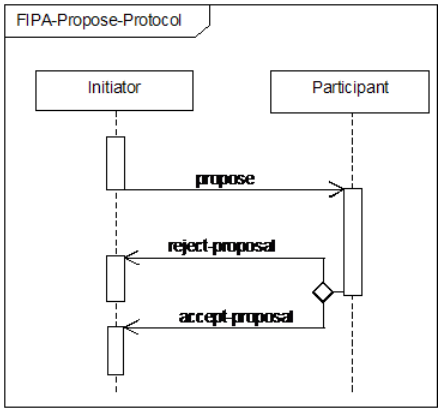


Figure 42. FIPA Propose Interaction Protocol

FIPA Request When

The initiator uses the `request-when` action to request that the participant do some action once a given precondition becomes true. If the requested agent understands the request and does not initially refuse, it will `agree` and wait until the precondition occurs. Then, it will attempt to perform the action and notify the requester accordingly.
If after the initial agreement the participant is no longer able to perform the action, then it will send a `failure` action to the initiator. Once the action has completed and the `failure`, `inform-done`, or `inform-result` has been sent, the conversation ends.

Any interaction using this interaction protocol is identified by a globally unique, non-null `conversation-id` parameter, assigned by the Initiator. The agents involved in the interaction must tag all of its ACL messages with this conversation identifier. This enables each agent to manage its communication strategies and activities, for example, it allows an agent to identify individual conversations and to reason across historical records of conversations.

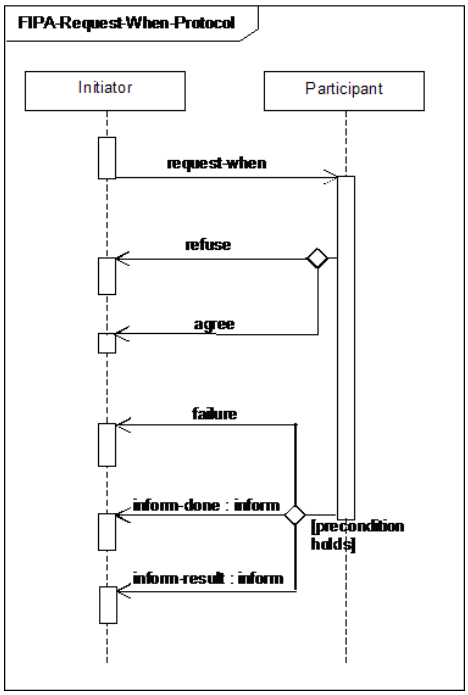


Figure 43. FIPA Request When Interaction Protocol

8.2. Annex 2. Developing Multi-Agent Systems: JADE

JADE [06] is a software development framework aimed at developing multi-agent systems and applications conforming to FIPA standards for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. JADE has been fully coded in Java and an agent programmer, in order to exploit the framework, should code his/her agents in Java.

As a middleware that facilitates the development of multi-agent systems, it includes:

- A *runtime environment* where JADE agents can "live" and that must be active on a given host before one or more agents can be executed on that host.
- A *library of classes* that programmers have to/can use (directly or by specializing them) to develop their agents.
- A suite of *graphical tools* that allows administrating and monitoring the activity of running agents.

8.2.1. JADE Packages

JADE is composed of the following main packages:

jade.core implements the kernel of the system. It includes the Agent class that must be extended by application programmers; besides, a Behaviour class hierarchy is contained in jade.core.behaviours sub-package. Behaviours implement the tasks, or intentions, of an agent. They are logical activity units that can be composed in various ways to achieve complex execution patterns and that can be concurrently executed. Application programmers define agent operations writing behaviours and agent execution paths interconnecting them.

The *jade.lang.acl* sub-package is provided to process Agent Communication Language according to FIPA standard specifications. The jade.content package contains a set of classes to support user-defined ontologies and content-languages. A separate tutorial describes how to use the JADE support to message content. In particular jade.content.lang.sl contains the SL codec, both the parser and the encoder.

The *jade.domain* package contains all those Java classes that represent the Agent Management entities defined by the FIPA standard, in particular the AMS and DF agents, that provide life-cycle, white and yellow page services. The subpackage jade.domain.FIPAAgentManagement contains the FIPA-Agent-Management Ontology and all the classes representing its concepts. The subpackage jade.domain.JADE.AgentManagement

contains, instead, the JADE extensions for Agent-Management (e.g. for sniffing messages, controlling the life-cycle of agents, …), including the Ontology and all the classes representing its concepts. The subpackage jade.domain.introspection contains the concepts used for the domain of discourse between the JADE tools (e.g. the Sniffer and the Introspector) and the JADE kernel. The subpackage jade.domain.mobility contains all concepts used to communicate about mobility.

The *jade.gui* package contains a set of generic classes useful to create GUIs to display and edit Agent-Identifiers, Agent Descriptions, ACLMessages, …

The *jade.mtp* package contains a Java interface that every Message Transport Protocol should implement in order to be readily integrated with the JADE framework, and the implementation of a set of these protocols.

jade.proto is the package that contains classes to model standard interaction protocols (i.e. fipa-request, fipa-query, fipa-contract-net, fipa-subscribe and soon others defined by FIPA), as well as classes to help application programmers to create protocols of their own.

The FIPA package contains the IDL module defined by FIPA for IIOP-based message transport.

Finally, the *jade.wrapper* package provides wrappers of the JADE higher-level functionalities that allows the usage of JADE as a library, where external Java applications launch JADE agents and agent containers.

JADE comes bundled with some tools that simplify platform administration and application development. Each tool is contained in a separate sub-package of *jade.tools*. Currently, the following tools are available:

- *Remote Management Agent*, RMA for short, acting as a graphical console for platform management and control. A first instance of an RMA can be started with a command line option ("-gui") , but then more than one GUI can be activated. JADE maintains coherence among multiple RMAs by simply multicasting events to all of them. Moreover, the RMA console is able to start other JADE tools.
- *The Dummy Agent* is a monitoring and debugging tool, made of a graphical user interface and an underlying JADE agent. Using the GUI it is possible to compose ACL messages and send them to other agents; it is also possible to display the list of all the ACL messages sent or received, completed with timestamp information in order to allow agent conversation recording and rehearsal.
- *The Sniffer* is an agent that can intercept ACL messages while they are in flight, and displays them graphically using a notation similar to UML sequence diagrams. It is useful for debugging your agent societies by observing how they exchange ACL messages.
- *The Introspector* is an agent that allows to monitor the life cycle of an agent, its

exchanged ACL messages and the behaviours in execution.

- *The DF GUI* is a complete graphical user interface that is used by the default Directory Facilitator (DF) of JADE and that can also be used by every other DF that the user might need. In such a way, the user might create a complex network of domains and sub-domains of yellow pages. This GUI allows in a simple and intuitive way to control the knowledge base of a DF, to federate a DF with other DF's, and to remotely control (register/deregister/modify/search) the knowledge base of the parent DF's and also the children DF's (implementing the network of domains and sub-domains).
- The *LogManagerAgent* is an agent that allows setting at runtime logging information, such as the log level, for both JADE and application specific classes that use Java Logging.
- The *SocketProxyAgent* is a simple agent, acting as a bidirectional gateway between a JADE platform and an ordinary TCP/IP connection. ACL messages, travelling over JADE proprietary transport service, are converted to simple ASCII strings and sent over a socket connection. Viceversa, ACL messages can be tunnelled via this TCP/IP connection into the JADE platform. This agent is useful, e.g. to handle network firewalls or to provide platform interactions with Java applets within a web browser.

8.2.2. The Agent Platform

The standard model of an agent platform, as defined by FIPA, is represented in the following figure.

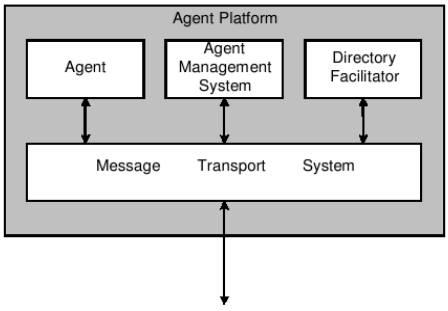


Figure 44. Reference architecture of a FIPA Agent Platform

The Agent Management System (AMS) is the agent who exerts supervisory control over access to and use of the Agent Platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.

The Directory Facilitator (DF) is the agent who provides the default yellow page service in the platform.

The Message Transport System, also called Agent Communication Channel (ACC), is the software component controlling all the exchange of messages within the platform, including messages to/from remote platforms.

JADE fully complies with this reference architecture and when a JADE platform is launched, the AMS and DF are immediately created and the ACC module is set to allow message communication. The agent platform can be split on several hosts. Only one Java application, and therefore only one Java Virtual Machine (JVM), is executed on each host. Each JVM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host. The main-container, or front-end, is the agent container where the AMS and DF lives and where the RMI registry, that is used internally by JADE, is created. The other agent containers, instead, connect to the main container and provide a complete run-time environment for the execution of any set of JADE agents.

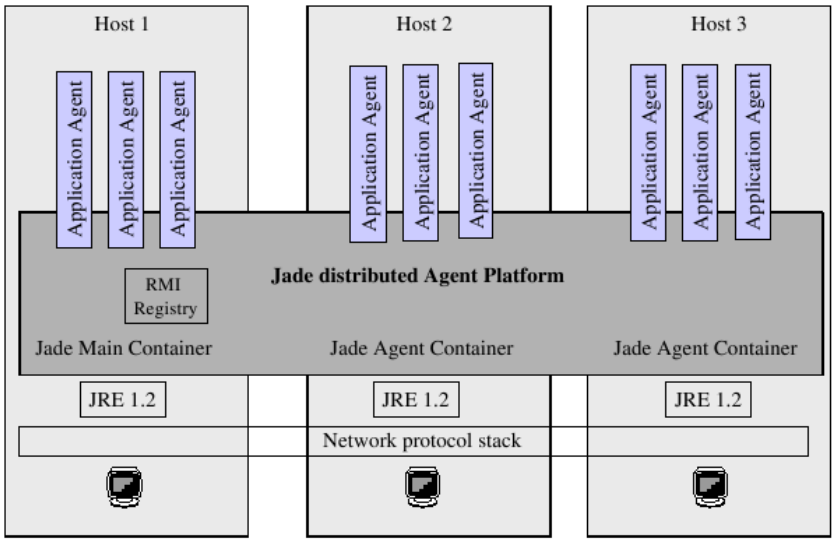


Figure 45. JADE Agent Platform distributed over several containers

According to the FIPA specifications, DF and AMS agents communicate by using the *FIPA-SL0* content language, the fipa-agent-management ontology, and the fipa-request interaction protocol. JADE provides compliant implementations for all thesecomponents:

- the *SL-0* content language is implemented by the class *jade.content.lang.sl.SLCodec*. Automatic capability of using this language can be added to any agent by using the method `getContentManager().registerLanguage(new SLCodec(0));`
- concepts of the ontology (apart from Agent Identifier, implemented by jade.core.AID) are implemented by classes in the *jade.domain.FIPAAgentManagement* package.The FIPAManagementOntology class defines the vocabulary with all the constant symbols of

the ontology. Automatic capability of using this ontology can be added to any agent by using the following code:

```
getContentManager().registerOntology(FIPAManagementOntology.getInstance());
```

➔ finally, the fipa-request interaction protocol is implemented as ready-to-use behaviours in the package jade.proto.

Every class implementing a concept of the fipa-agent-management ontology is a simple collection of attributes, with public methods to read and write them, according to the frame based model that represents FIPA fipa-agent-management ontology concepts. The following convention has been used. For each attribute of the class, named attrName and of type attrType, two cases are possible:

1. The attribute type is a single value; then it can be read with attrType getAttrName() and written with void setAttrName(attrType a), where every call to setAttrName() overwrites any previous value of the attribute.
2. The attribute type is a set or a sequence of values; then there is an void addAttrName(attrType a) method to insert a new value and a void clearAllAttrName() method to remove all the values (the list becomes empty). Reading is performed by a Iterator getAllAttrName() method that returns an Iterator object that allows the programmer to walk through the List and cast its elements to the appropriate type.

8.2.3. Basic concepts of the ontology

The package *jade.content.onto.basic* includes a set of classes that are commonly part of every ontology, such as Action, TrueProposition, Result, , …

Notice that the Action class should be used to represent actions. It has a couple of methods to set/get the AID of the actor (i.e. the agent who should perform the action) and the action itself (e.g. Register/Deregister/Modify).

8.2.4. Simplified API to access DF and AMS services

JADE features described so far allow complete interactions between FIPA system agents and user defined agents, simply by sendng and receiving messages as defined by the standard.

However, because those interactions have been fully standardized and because they are very common, the following classes allow to successfully accomplish this task with a simplified interface.

Two methods are implemented by the class Agent to get the AID of the default DF and AMS of the platform: `getDefaultDF()` and `getAMS()`.

8.2.5. DFService

jade.domain.DFService implements a set of static methods to communicate with a standard FIPA DF service (i.e. a yellow pages agent).

It includes methods to request register, deregister, modify and search actions from a DF. Each of this method has a version with all the needed parameters, and one with a subset of them where the omitted parameters are given default values.

Notice that these methods block every agent activity until the action is successfully executed or a *jade.domain.FIPAException* exception is thrown (e.g. because a failure message has been received by the DF), that is, until the end of the conversation.

In some cases, instead, it is more convenient to execute these tasks in a non-blocking way. In these cases a *jade.proto.AchieveREInitiator* or *jade.proto.SubscriptionInitiator* should be used in conjunction with the `createRequestMessage()`, `createSubscriptionMessage()`, `decodeDone()`, `decodeResult()` and `decodeNotification()` methods that facilitate the preparation and decoding of messages to be sent/received to/from the DF. The following piece of code exemplifies that in the case of an agent subscribing to the default DF.

```
DFAgentDescription template = // fill the template
AID df = getDefaultDF();
ACLMessage subs = DFService.createSubscriptionMessage(this, df, template, null))
Behaviour b = new SubscriptionInitiator(this, subs) {
   protected void handleInform(ACLMessage inform) {
     try {
        DFAgentDescription[] dfds =
          DFService.decodeNotification(inform.getContent());
        // do something
     }
     catch (FIPAException fe) {
        fe.printStackTrace();
     }
   }
};
addBehaviour(b);
```

8.2.6. AMSService

This class is dual of DFService class, accessing services provided by a standard FIPA AMS agent and its interface completely corresponds the the DFService one.

Notice that JADE calls automatically the register and deregister methods with the default AMS respectively before calling `setup()` method and just after `takeDown()` method returns; so

there is no need for a normal programmer to call them.

However, under certain circumstances, a programmer might need to call its methods. To give some examples: when an agent wishes to register with the AMS of a remote agent platform, or when an agent wishes to modify its description by adding a private address to the set of its addresses, …

9. References

[01] Knowledge-Based HomeCare eServices for an Ageing Europe - Annex I - "Description of Work". 76 pag. 04/11/2005

[02] Fabio Campana, Roberta Annicchiarico, David Riaño et al. Knowledge-Based HomeCare eServices for an Ageing Europe – D01 – The K4CARE Model. 182 pag. 2006, http://www.k4care.net.

[03] David Isern, Antonio Moreno, Gianfranco Pedone, Lazslo Varga - Agent-based provision of Home Care Services. Artificial Intelligence in Medicine 07-11 July of 2007 in Amsterdam, http://www.aimedicine.eu/AIME07/.

[04] David Riaño – The SDA Model v1.0: a Set Theory approach, 50 pag. DEIM Report, 2007

[05] David Isern – Avaluació d'entorns de desenvolupament de SMAs, 1998-99. Projecte de Final de Carrera, ET. Inf. Sistemes. ETSE, Universitat Rovira i Virgili.

[06] FIPA ACL Message Structure - http://www.fipa.org/specs/fipa00061/SC00061G.html

[07] FIPA Protocols Specification - http://www.fipa.org/repository/standardspecs.html

[08] JADE Programmer's Guide - http://jade.tilab.com/doc/programmersguide.pdf

[09] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides - Design Patterns: Elements of Reusable Object-Oriented Software, 349 pag. Ed. Addison Wesley, 2003

[10] Eclipse IDE - http://www.eclipse.org

[11] Protege - http://protege.stanford.edu

[12] Google - http://www.google.com

[13] Wikipedia - http://en.wikipedia.org

[14] JUnit - http://www.junit.org

[15] Maven - http://maven.apache.org

[16] Subversion - http://subversion.tigris.org/

[17] Tony Sintes - Learn custom events with a concrete example (http://www.javaworld.com/javaworld/javaqa/2002-03/01-qa-0315-happyevent.html)

[18] Tony Sintes - How do you create a custom event? (http://www.javaworld.com/javaqa/2000-08/01-qa-0804-events.html)

[19] Asbru Description - http://www.openclinical.org/gmm_asbru.html

[20] PROforma Description - http://www.acl.icnet.uk/lab/proforma.html

[21] D. Isern and A. Moreno. *Computer-based management of clinical guidelines: A Survey*. In Proc. of Fourth Workshop on Agents applied in Healthcare on ECAI'06. Riva del Garda (Italy), August 28 - September 1, 2006