Contract no. 511632

CASCOM

Context-Aware Business Application Service Co-ordination in Mobile Computing Environments

Instrument: STREP

Thematic Priority: [FP6-2003-IST-2]

Deliverable D5.2: Service Composition and Execution in IP2P Environments

Due date of deliverable: August 31, 2006
Actual submission date: August 31, 2006

Start date of project: 01.09.2004 Duration: 36 months

Project coordinator name: Dr. Oliver Keller
Project coordinator organisation name: DFKI

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Released

Deliverable D5.2: Service Composition and Execution in IP2P Environments

Document Properties:

Document Number[1]:	IST-2003-511632-CASCOM/WP5/D5.1
Document responsible:	ADETTI
Author(s)/editor(s):	ADETTI {António Lopes, Rui Fonseca, Luís Botelho}, URJC {Alberto Fernandez, Matteo Vasirani, Sascha Ossowski}, UniBasel {Thorsten Möller, Heiko Schuldt}, DFKI {Bastian Blankenburg}
Document Type[2]:	Deliverable
Security Type[3]:	Public
Status of the Document[4]:	Released
Reviewed by[5]:	TMC
Version	1.0

Revision History:

Revision	Date	Issued by	Description
0.1	2006-06-21	António Lopes	Initial Table of Contents
0.2	2006-07-18	António Lopes, Rui Fonseca, Alberto Fernandez, Matteo Vasirani, Thorsten Möller	Added content to sections 2.1.2, 2.4 and 3
0.3	2006-07-20	António Lopes, Luís Botelho, Bastian Blankenburg	Updated content on section 3.2 Added content on sections 2.1.1 and 2.3
0.4	2006-07-25	Alberto Fernandez, Matteo Vasirani, António Lopes	Small changes and comments in the entire document
0.5	2006-07-31	António Lopes, Alberto Fernandez,	Changed the structure of sections 2 and 3, corrected some errors and comments throughout

[1] Format: IST-2003-511632-CASCOM/<Source ID>/(<Partner ID)/<Type> <number>
Example: IST-2003-511632-CASCOM/WP3/D01 (Document comes from the WP1, Deliverable 1)
[2] Document Type: Report,/Deliverable/ PM
[3] Security Type:

Confidential	Internal circulation within project (and Commission project Officer if requested)
Restricted	Restricted circulation list (specify in footnote) and Commission PO only
Internal	With no confidential content but intended for internal use
Public	Public document

[4] Draft <#>, Draft <#> Reviewed, Final Draft, Released, Revised Draft <#>, etc
[5] Reviewed by: <Appointed Reviewer>/TMC / PMC

		Thorsten Möller	the entire document.
0.6	2006-08-04	António Lopes, Alberto Fernandez, Bastian Blankenburg	Updated the content on sections 2.1 and 2.2
0.7	2006-08-11	Thorsten Möller, António Lopes, Alberto Fernandez, Matteo Vasirani, Sascha Ossowski	Updated content in sections 3.2.3, 3.2.4, 3.2.5. Merged the terms “Plan”, “Composite Service” and “Composite Service” into “Composite Service”. Updated section 2.1.3.3 Corrected some text throughout the entire document based on comments from partners.
1.0	2006-08-14	António Lopes, Alberto Fernandez, Sascha Ossowski, Matteo Vasirani, Heiko Schuldt	Added the introduction Corrected an error in section 2.1.3.3.5 Reviewed section 3
1.0	2006-08-31	Heikki Helin	Final editing

Abstract:

This document is the Deliverable 5.2 of the CASCOM project. This deliverable describes the work done in the project regarding service composition and execution in IP2P Environments. The deliverable is divided in two different main sections: the first main section of this deliverable describes methods for service composition planning in IP2P environments. The second main section of the deliverable describes the design of the reliable CASCOM service execution platform suitable for IP2P environments.

Table of Contents

List of Figures

List of Tables

Abbreviations

ACL	Agent Communication Language
AIC	Agent Interaction Component
FIPA	Foundation for Intelligent Physical Agents
IOPEs	Inputs, Outputs, Pre-conditions and Effects
IP2P	Intelligent Peer-to-Peer
JADE	Java Agent DEvelopment framework
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
P2P	Peer to Peer
PA	Personal Agent
PDDL	Planning Domain Definition Language
PcECP	Pre-conditions and Effects Composition Planner
RTPG	Relaxed Temporal Planning Graph
SCPA	Service Composition Planner Agent
SDA	Service Discovery Agent
SEA	Service Execution Agent
SOAP	Simple Object Access Protocol
REST	REpresentational State Transfer
WSDL	Web Services Description Language

1 Introduction

The present document is the deliverable 5.2 of the CASCOM project. CASCOM's main objective is to implement, validate, and trial value-added supportive infrastructure for business application services for mobile workers and users across mobile and fixed networks. The driving vision of CASCOM is that ubiquitous business application services are flexibly coordinated and pervasively provided to the mobile worker/user by intelligent agents in dynamically changing contexts of open, large-scale, and pervasive environments. In compliance with the research objectives of the project, we primarily focus on the W3C standard OWL-S [OWLS03][MBHL04] for describing semantic Web services (based on the W3C standard OWL [OWL03] language) such that standard reasoning services can be used by intelligent agents to automatically find and compose relevant services without too much interaction with its user.

The deliverable is a part of CASCOM's work package 5 whose primary objectives include:

- To develop and deploy generic agents and advanced mechanisms for service co-ordination in IP2P environments.
- To develop agent-based distributed service directories IP2P environments with fixed and dynamic topology.
- To investigate concepts, architectures, and methods to incorporate situation-awareness into service coordinating and providing agents in IP2P environments.
- To develop a reliable service execution platform.

This deliverable describes the methods for service composition planning and execution in IP2P environments. This includes the description of the Service Composition Planner Agent and the approaches for Service Execution in the CASCOM environment.

The first part of the deliverable (Section 2) is dedicated to describing the methods for service composition planning. This main section is divided into three sub-sections.

The first one (Section 2.1) is dedicated to describing the Service Composition Planner Agent, which is an integrated agent for service composition planning activities within the CASCOM environment. This agent has 3 components, described in Sections 2.1.2 and 2.1.3, specifically designed to optimize the composition planning process of Semantic Web services.

The second one (Section 2.2) describes methods for open service composition. In this section a dynamic version of the OWLS-XPlan composition planner, OWLS-XPlan+ is described.

The third sub-section (Section 2.3) describes a specialization of the composition planning process specifically oriented to information services: a broker agent specialized in creating value-added information services based on the preferences provided by a specific user.

The second part of the deliverable (Section 3) is dedicated to describing the methods for execution of Semantic Web services in the CASCOM environment. This main section is divided into three sub-sections as well.

The first sub-section (Section 3.1) makes a general description of the execution process of OWL-S/WSDL services. This sub-section describes this process independently of any chosen approach for executing processes.

The second sub-section (Section 3.2) describes a distributed approach to address the execution of Semantic Web Services. It explains how to deploy a set of several service execution agents to coolaborate in the execution of a composite service.

The third sub-section (Section 3.3) describes an alternative and somehow complementary approach to the one presented in Section 3.2, by presenting a centralized approach for executing Semantic Web services. In this approach a single specialized service execution agent is used to execute composite services as requested by client agents.

1.1 Definition of Terms

A lot of different terms can be used to describe the result of structured composition planning of several atomic services into one single service, such as plan, compound service or composite service. To avoid ambiguity, we have decided to use the official term as used in the W3C's OWL-S standard: composite service.

Please note that, even though sometimes the term "process" can be used to describe a composite service as well, we decided not to exclude this term from the text of the deliverable because it can also be used to denote an atomic service, as opposed to a composite one.

2 Service Composition in IP2P Environments

Service composition is the problem of the dynamic creation of a new service that satisfies given requirements, received in the service composition request. The new service is created by the composition of a specific set of other component services, which must be described to the service composition agent within the received request.

2.1 Service Composition Planner Agent

In the CASCOM architecture the Service Composition Functionality is delivered by the Service Composition Planner Agent, which is described in the following sections.

2.1.1 Typical Usage and Interface

In a typical scenario, the Personal Agent (PA) sends a composition request message to the SCPA. This is done using the standard FIPA request protocol. This message must contain two components, the initial state of the world and the goal to achieve (both in OWL).

After receiving the request, the SCPA cannot immediately perform the composition and before that, it must contact the SDA in order to retrieve OWL-S descriptions of relevant services. When the SDA replies, its message must include a sequence of OWL-S complete service descriptions.

After receiving the set of atomic services that will be used for composition, the SCPA performs the second step, which refers to internal processing and the planning process itself.

Once the planning completed successfully, the SCPA converts the plan to an OWL-S composite service description and informs the PA about the plan. Simultaneously, the SCPA requests the SEA to execute the plan. Any subsequent messages from the SEA about the execution status of the plan are forwarded to the PA.

2.1.1.1 The Agent's Interface

When requesting the composition of a new service, client agents interact with SCPA through the FIPA-request interaction protocol. Such protocol states that when the receiver agent receives an action request, it can either agree or refuse to perform the action. Whichever should be its decision, the agent should then notify the other agent of its decision through the corresponding communicative act (FIPA-agree or FIPA-refuse). By default, if no time constraints apply, the SCPA should always agree to perform the action. If should the answer be a refuse (e.g., the agent cannot perform the request in the given time), the protocol ends and no more messages are exchanged between the agents.

Responding with an agree message establishes a compromise between those two agents, in which, the FIPA-request protocol states that, after a successful action execution the executor agent should return the results through a FIPA-inform message. The SCPA returns a composite service description in OWL-S, which states the result of the composition. Thus, the SCPA will send a FIPA-inform-result message containing the OWL-S description of the new composite service.

However, before composition can be done, the SCPA needs to ask the SDA for relevant services. So, the SCPA itself sends a request to the SDA, in order to obtain the services that may be included in the final composite service. It also uses the FIPA-request protocol and the SDA replies also with a FIPA-agree first, and then with the FIPA-inform-result.

2.1.1.2 Testing the Agent

In this section we present a succinct yet well-aimed example of this agent's behaviour. Let's picture the following scenario to produce this example: the PA requests the SCPA to create a composite service that should contain these three component services: RegisterPatientService, SellMedicineService and DeliverService.

2.1.1.2.1 doComposition

The received FIPA-ACL [FIPA00b] message with FIPA-SL [FIPA00c] content is depicted in Figure 1.

```
(REQUEST
 :sender (agent-identifier  :name pa@cascom)
 :receiver (set (agent-identifier :name scpa@cascom))
 :content  "((action
              (agent-identifier :name scpa@cascom)
              (doComposition
                :categories \"http://.../OnlineMedicineSelling.owl\"
                :init \"http://.../OMS_InitialOntology.owl\"
                :goal \"http://.../OMS_GoalOntology.owl\")))"
:language  fipa-sl
:protocol  fipa-request
:ontology scpa-ontology)
```

Figure 1 – ACL Message received by the SCPA to perform composition

The FIPA-ACL message with FIPA-SL content in Figure 1 is a request message, meaning that the receiver (e.g., SCPA) will, if agreed, perform an action. This action, "doComposition" in the example, has three arguments: the OWL-S request description that should contain IOPEs of the composite service and component services categories; the OWL description of the initial state of the world and the OWL description of the goal to achieve.

OMS_InitialOntology.owl and OMS_GoalOntology.owl files content are presented in annexes A and B, respectively. The OnlineMedicineSelling.owl has the Composite Service IOPEs plus the component services categories (e.g., Register-Patient, Sell-Medicine and Deliver-Service).

2.1.1.2.2 Get Matching Services from the SDA

The next step implies sending a message to the SDA, to obtain the services description to be part of the composite service. This interaction with the SDA takes place exactly as described in Deliverable 5.1.

The returned services by the SDA are the RegisterPatientService.owl, SellMedicineService.owl and DeliverService.owl files, which are presented in annexes C, D and E, respectively.

2.1.1.2.3 Answer to doComposition

At this point the SCPA performs some pre-processing of the input files and then begins the composition and by the time it finishes, the generated plan is then transformed to an OWL-S description of the composite service (see CompositeService.owl file, which is presented as in annex F), and finally the SCPA returns the result to the PA. The reply message is presented in Figure 2.

```
(INFORM
 :sender (agent-identifier  :name scpa@cascom)
 :receiver  (set (agent-identifier  :name pa@cascom))
 :content  "((result
              (action
               (agent-identifier :name scpa@cascom)
               (doComposition
                :categories \"http://.../ OnlineMedicineSelling.owl\"
                :init \"http://.../OMS_InitialOntology.owl\"
                :goal \"http://.../OMS_GoalOntology.owl\"))
          (\"http://.../CompositeService.owl\"))"
 :language  fipa-sl
 :protocol  fipa-request
 :ontology scpa-ontology)
```

Figure 2 – ACL Message return by the SCA with composition's result

Figure 2 shows the message that the SCPA sends to the Personal Agent. It is an Inform message containing the composite service description.

2.1.2 Internal Architecture

Figure 3 shows the internal architecture of the SCPA, which consists of three main components.

The Filter (detailed in section 2.1.3.3) reduces the set of relevant services that will be used for composition. This filter is applied to the services retrieved by the SDA. Its goal is to select a smaller set of services to be used for composition in order to increase the efficiency of the planning process.

The OWLS-XPlan (see section 2.1.3.1.1) is a service composition planner that takes a set of OWL-S services, a description of the initial state and the goal state to be achieved as input, and returns a plan. The set of service descriptions is obtained from the filter.

In addition to the OWLS-XPlan the SCPA includes a pre-conditions and effects composition planner (PcECP) (see section 2.1.3.2). This planner is used to check whether the plan created by the OWLS-XPlan is valid in terms of the pre-conditions and effects specified in the query. To carry out this evaluation, the PcECP will receive as input the set of atomic services that make up the composite plan and will try to generate a plan. If that plan is equal to the one created by the OWLS-XPlan then it is returned as the valid plan[6]. If the plan is not valid, the OWLS-XPlan is invoked again to generate a different plan. The process is repeated until a compatible plan is found or no more different plans can be generated.

[6] Plans that have the exact same actions structured in a Parallel-like construct are considered to be equal, as the order in which the actions are executed in the plan are irrelevant for the effects that the actions produce.

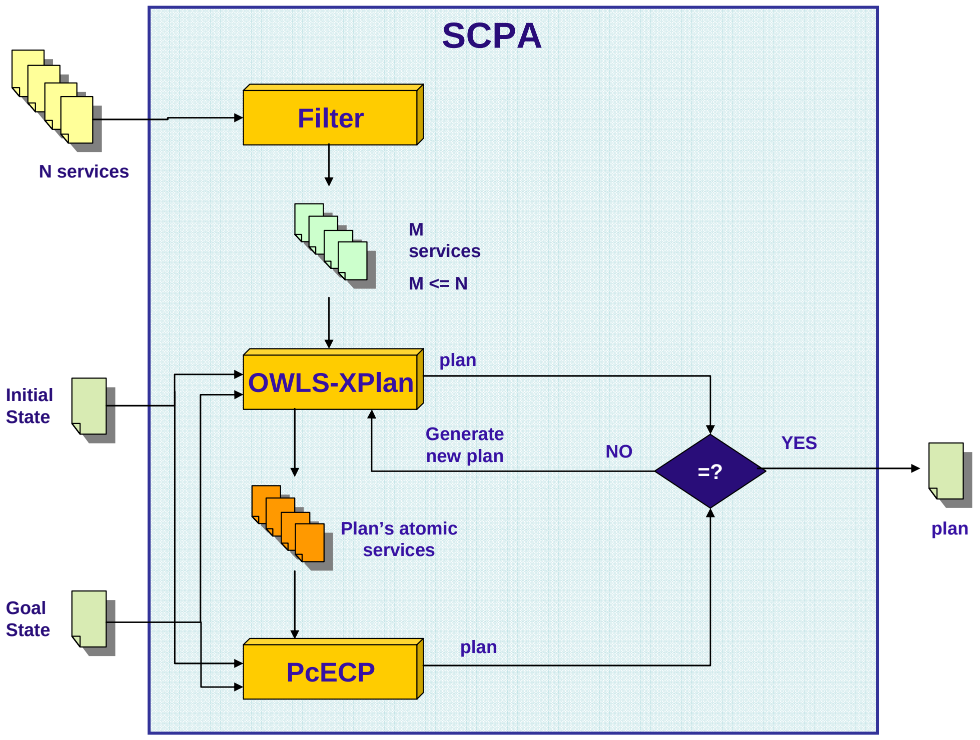


Figure 3 – SCPA internal architecture

2.1.3 Components Detailed Description

In this section, we provide a detailed description of each of components that make up the internal architecture of the Service Composition Planner Agent.

2.1.3.1 Directory-based Service Composition with OWLS-XPlan

Hierarchical task network (HTN) planners such as SHOP2 perform well in domains for which complete and detailed knowledge on at least partially hierarchically structured action execution patterns is available, such as, for example, in scenarios of rescue planning. In domains in which this is not the case, i.e., no concrete set of methods and decomposition rules that lead to an executable plan are provided, an HTN planner would not find the solution due to the fixed structure of hierarchical action decompositions stored in its database. That inherently limits the degree of quality of any HTN planner to that of its used methods that are created by human experts.

In contrast, action based planners are able to find a solution based on atomic actions as they are described in the methods, but without using the structure of the latter. Atomic actions can be combined in multiple ways to solve a given planning problem. The problem then is to cope with planning problems that are in part hierarchically structured according to decomposition rules and methods but not solvable exclusively by means of HTN planning.

For this purpose, we developed a hybrid AI planner Xplan which combines the benefits of both approaches by extending an efficient graph-plan based FastForward-planner with a HTN planning component. To use Xplan for semantic Web-Service composition, XPlan is complemented by a conversion tool that converts OWL-S 1.1 service descriptions to corresponding PDDL 2.1 [M98] descriptions that are used by Xplan as input to plan a service composition that satisfies a given goal. In contrast to HTN planners, Xplan always finds a solution if it exists in the action/state space over the space of possible plans, though the problem is NP-complete. Xplan also includes a re-planning component to flexibly react to changes in the world state during the composition planning process. Together the implementations of Xplan and OWLS2PDDL converter make up the semantic Web service composition planner OWLS-Xplan.

We briefly describe OWLS-XPlan[7] in the following section; for more details we refer to the paper [KGS05].

2.1.3.1.1 OWLS-XPlan

OWLS-XPlan consists of several modules for pre-processing and planning. It takes a set of available OWL-S services, a domain description and a planning query as input. The domain description and the planning query contain OWL individuals (facts) which are true initially or are to be achieved by the plan, respectively. They also contain the necessary OWL ontologies. OWLS-XPlan then returns a plan sequence, i.e. a composite service, which satisfies the planning query. For this purpose, it first converts the domain ontology and service descriptions in OWL and OWL-S, respectively, to an equivalent PDDL 2.1 problem and domain descriptions using the OWLS2PDDL converter.

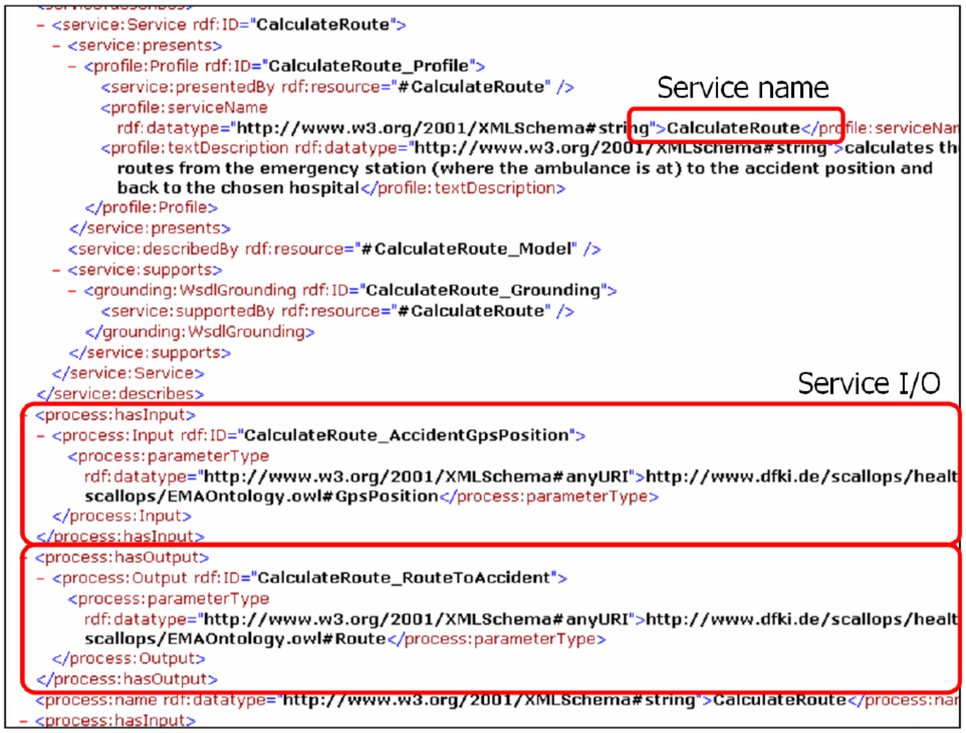
The resulting domain description contains the definition of all types, predicates and actions, whereas the problem description includes all objects, the initial state, and the goal state. Both descriptions are then used by the AI planner XPlan to create a plan (representing a composite Web-Service) in PDDL that solves the given problem in the actual domain. For reasons of convenience, we developed a XML dialect of PDDL, called PDDXML that simplifies parsing, reading, and communicating PDDL descriptions using SOAP. We also developed a module to convert the PDDXML plan description to an OWL-S process model, making it possible to seamlessly integrate OWLS-XPlan into a purely OWL-S based system.

The OWLS2PDDL converter and the XPlan planner are integrated in the CASCOM architecture via the Service Composition Planner Agent SCPA. While the OWLS2PDDL converter and Xplan planner are developed in the liaison project SCALLOPS, the SCPA is developed in CASCOM.

2.1.3.1.1.1 OWLS2PDDXML Converter

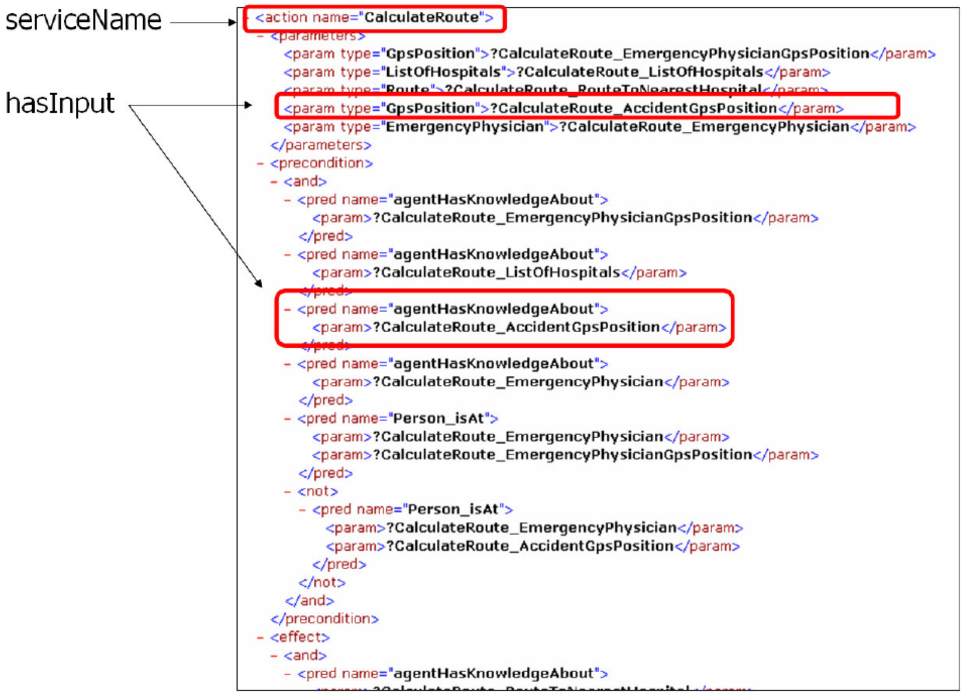
The conversion of OWL-S 1.1 service descriptions to PDDXML requires the transcription of types and properties to PDDL predicates as well as the mapping of services to actions. Any OWL-S service profile input parameter correlates with an equally named one of a PDDL action, and the *hasPrecondition* service parameter can directly be transformed to the precondition of the action by use of predicates. The same holds for the *hasEffect* condition parameter. For the conversion of the output of an individual OWL-S service to PDDL, the service output parameter is mapped to a special type of the service hasEffect parameter. This is because the service *hasEffect* condition explicitly describes how the world state will change while this is not necessarily the case for a *hasOutput* parameter value, though it could implicitly influence the composition planning process. However, PDDL does not allow describing such non-physical knowledge. As an example, Figure 4 and Figure 5 show a part of an OWL-S service description and the corresponding PDDXML action, respectively, which is obtained from the OWLS2PDDXML converter.

[7] OWLS-Xplan is available as open source software under Mozilla Public License 1.1 at http://projects.semwebcentral.org/projects/owls-xplan/



```
    </service:describes>
  - <service:Service rdf:ID="CalculateRoute">
    - <service:presents>
      - <profile:Profile rdf:ID="CalculateRoute_Profile">
          <service:presentedBy rdf:resource="#CalculateRoute" />
          <profile:serviceName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string">CalculateRoute</profile:serviceNar
          <profile:textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">calculates th
            routes from the emergency station (where the ambulance is at) to the accident position and
            back to the chosen hospital</profile:textDescription>
        </profile:Profile>
      </service:presents>
      <service:describedBy rdf:resource="#CalculateRoute_Model" />
    - <service:supports>
      - <grounding:WsdlGrounding rdf:ID="CalculateRoute_Grounding">
          <service:supportedBy rdf:resource="#CalculateRoute" />
        </grounding:WsdlGrounding>
      </service:supports>
    </service:Service>
  </service:describes>
  <process:hasInput>
  - <process:Input rdf:ID="CalculateRoute_AccidentGpsPosition">
      <process:parameterType
        rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://www.dfki.de/scallops/healt
        scallops/EMAOntology.owl#GpsPosition</process:parameterType>
    </process:Input>
  </process:hasInput>
  <process:hasOutput>
  - <process:Output rdf:ID="CalculateRoute_RouteToAccident">
      <process:parameterType
        rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://www.dfki.de/scallops/healt
        scallops/EMAOntology.owl#Route</process:parameterType>
    </process:Output>
  </process:hasOutput>
  <process:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">CalculateRoute</process:nar
- <process:hasInput>
```

Figure 4 – Part of OWL-S 1.1 Service Description



```
  <action name="CalculateRoute">
  - <parameters>
      <param type="GpsPosition">?CalculateRoute_EmergencyPhysicianGpsPosition</param>
      <param type="ListOfHospitals">?CalculateRoute_ListOfHospitals</param>
      <param type="Route">?CalculateRoute_RouteToNearestHospital</param>
      <param type="GpsPosition">?CalculateRoute_AccidentGpsPosition</param>
      <param type="EmergencyPhysician">?CalculateRoute_EmergencyPhysician</param>
    </parameters>
  - <precondition>
    - <and>
      - <pred name="agentHasKnowledgeAbout">
          <param>?CalculateRoute_EmergencyPhysicianGpsPosition</param>
        </pred>
      - <pred name="agentHasKnowledgeAbout">
          <param>?CalculateRoute_ListOfHospitals</param>
        </pred>
      - <pred name="agentHasKnowledgeAbout">
          <param>?CalculateRoute_AccidentGpsPosition</param>
        </pred>
      - <pred name="agentHasKnowledgeAbout">
          <param>?CalculateRoute_EmergencyPhysician</param>
        </pred>
      - <pred name="Person_isAt">
          <param>?CalculateRoute_EmergencyPhysician</param>
          <param>?CalculateRoute_EmergencyPhysicianGpsPosition</param>
        </pred>
      - <not>
        - <pred name="Person_isAt">
            <param>?CalculateRoute_EmergencyPhysician</param>
            <param>?CalculateRoute_AccidentGpsPosition</param>
          </pred>
        </not>
      </and>
    </precondition>
  - <effect>
    - <and>
      - <pred name="agentHasKnowledgeAbout">
          <param>?CalculateRoute_RouteToNearestHospital</param>
```

Figure 5 – Part of an action description in PDDLXML converted by OWLS2PDDLXML

2.1.3.1.1.2 AI Planner XPlan

The AI planner XPlan is a heuristic hybrid search planner based on the FF-planner developed by Hoffmann and Nebel [HN01]. It combines guided local search with graph planning, and a simple form of hierarchical task networks to produce a plan sequence of actions that solves a given problem. This yields a higher degree of flexibility compared to pure HTN planners like SHOP2 [SPW04] whereas the use of predefined workflows or methods improves the efficiency of the FF-planner. In contrast to the general HTN planning approach, a graph-plan based planner is guaranteed to always find a solution independent from whether the given set of decomposition rules for HTN planning would allow building a plan that contains only atomic actions. In fact, any graph-plan based planner would test every combination of actions in the search space to satisfy the goal which, of course, can quickly become prohibitively expensive.

XPlan combines the strengths of both approaches. It is a graph-plan based planner with additional functionality to perform decomposition like a HTN planner. Figure 6 shows an example of how XPlan of OWLS-XPlan uses only those parts of a given method for decomposition that are required to reach the goal state with a sequence of composite services *WS_1* and *WS_3*. In contrast, HTN planning would completely decompose *M* into *WS_1* followed by *WS_2*, hence output also *WS_2* which is of no use for reaching the goal.

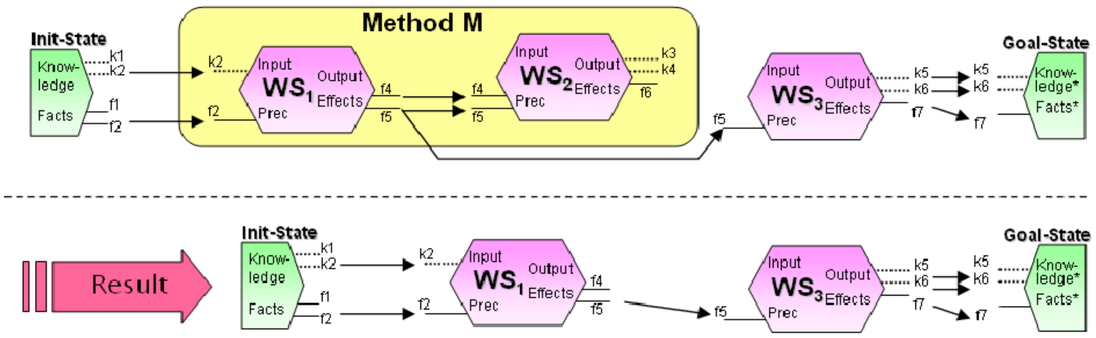


Figure 6 – Using parts of methods to reach a goal state in OWLS-XPlan

2.1.3.1.1.2.1 Architecture

The XPlan system consists of one XML parsing module, and following preprocessing modules. First, required data structures for planning are created and filled, followed by the generation of the initial connectivity graph and goal agenda. The core planning modules concern the heuristically relaxed graph-plan generation and enforced hill-climbing search (cf. Figure 7).

After the domain and problem definitions have been parsed, Xplan compiles the information into memory efficient data structures. A connectivity graph is then generated and efficiently realized by means of a look up table, which contains information about connections between facts and instantiated operators, as well as information about numerical expressions which can be connected to facts. This connectivity graph is maintained during the whole planning process and used for the actual search.

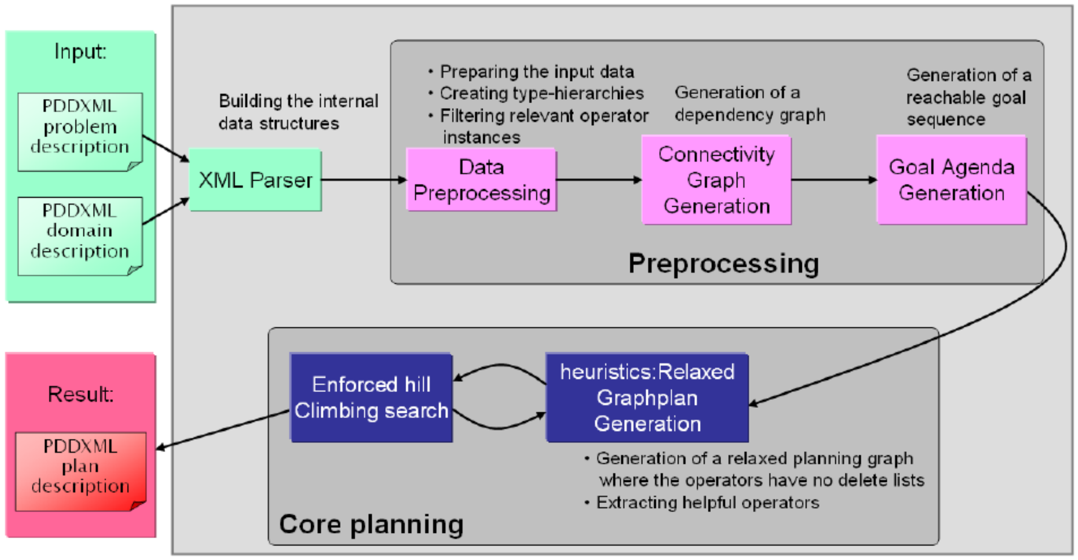
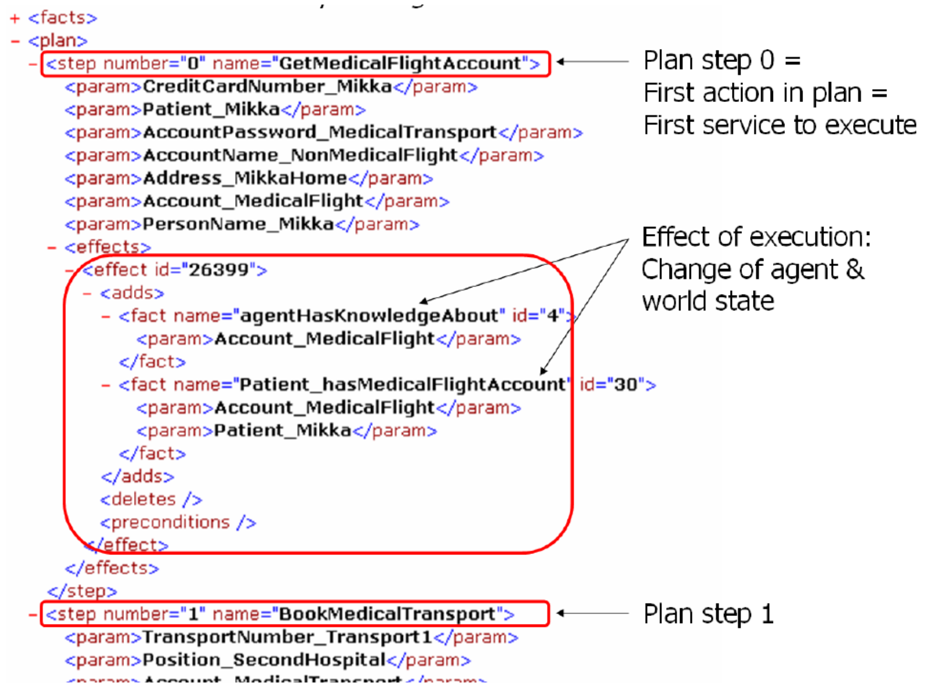

Input:
PDDXML problem description
PDDXML domain description
Building the internal data structures
XML Parser
• Preparing the input data
• Creating type-hierarchies
• Filtering relevant operator instances
Data Preprocessing
Generation of a dependency graph
Connectivity Graph Generation
Generation of a reachable goal sequence
Goal Agenda Generation
Preprocessing
Result:
PDDXML plan description
Enforced hill Climbing search
heuristics:Relaxed Graphplan Generation
• Generation of a relaxed planning graph where the operators have no delete lists
• Extracting helpful operators
Core planning

Figure 7 – Architecture of XPlan

The first of the core planning modules is concerned with the Relaxed Graphplan generation. This is done efficiently using a heuristic [H00] which approximates the distance between the initial state to all reachable states. These distance values are then used to guide the forward directed search. After each successful step the distance values are updated again using the heuristic. Additionally, (decomposition) information from hierarchical task networks is used, if required, to cope with partially hierarchical domains.

The graphplan generation is interleaved with an enforced hill-climbing search method to prune the search space during planning. Information on the quality of an action (execution of a service) is utilized by the local search to decide upon two or more steps that are equally weighted by the heuristic. This is done by computing the set of helpful executable actions for every search state such that the goal eventually can be reached. A helpful action of a search state S is an action that satisfies at least one proposition of the goal set of the first layer in the plan graph. If there are many helpful actions, then actions of an HTN decomposition are preferred. The reason is that such actions are more likely to be succeeded by an useful action in the task network as part of the relaxed plan.

Figure 8 shows a fragment of the plan description produced by Xplan, i.e., a sequence of actions, that is, the composed sequence of corresponding OWL-S services that should be executed by the agent.



```
+ <facts>
- <plan>
  - <step number="0" name="GetMedicalFlightAccount">
      <param>CreditCardNumber_Mikka</param>
      <param>Patient_Mikka</param>
      <param>AccountPassword_MedicalTransport</param>
      <param>AccountName_NonMedicalFlight</param>
      <param>Address_MikkaHome</param>
      <param>Account_MedicalFlight</param>
      <param>PersonName_Mikka</param>
    - <effects>
      - <effect id="26399">
        - <adds>
          - <fact name="agentHasKnowledgeAbout" id="4">
              <param>Account_MedicalFlight</param>
            </fact>
          - <fact name="Patient_hasMedicalFlightAccount" id="30">
              <param>Account_MedicalFlight</param>
              <param>Patient_Mikka</param>
            </fact>
          </adds>
          <deletes />
          <preconditions />
        </effect>
      </effects>
    </step>
  - <step number="1" name="BookMedicalTransport">
      <param>TransportNumber_Transport1</param>
      <param>Position_SecondHospital</param>
```

Figure 8 – Part of plan description in PDDLXML

2.1.3.1.1.2.2 Implementation

We implemented Xplan modularly in C++, using the Microsoft MSXML Parser for reading PDDXML definitions and generating plans in XML format. Alternatively, XPlan also provides an interface for direct interchange of planning data without having to use PDDXML as interchange format.

2.1.3.2 Pre-conditions and Effects Composition Planner

To enhance the planning capabilities of the Service Composition meta-service of the CASCOM Architecture, a pre-conditions and effects composition planner was developed in the scope of the task Service Composition in IP2P Environments. The PcECP provides a way of composing services using the pre-conditions and effects of the atomic services.

This composition planner uses Sapa [DK01], a Multi-objective Metric Temporal Planner, to create a composite service. Sapa is a domain-independent heuristic forward chaining planner that can handle durative actions, metric resource constraints, and deadline goals. It is designed to handle the multi-objective nature of metric temporal planning. This composition planner uses standards such as OWL/OWL-S [OWL03] [OWLS03] service descriptions and PDDL [M98] descriptions. The OWLS2PDDL converter was adapted from the OWLS2PDDXML [DFKI05] (see section 2.1.3.1.1.1) in order to convert OWL-S to PDDL, suitable for the Sapa planner inputs.

The Pre-conditions and Effects Composition Planner (PcECP) was developed as a single component that can easily be integrated into an autonomous agent. In the scope of the CASCOM Architecture, the Service Composition Planning Agent (SCPA) is equipped with this component in order to provide a way of composing services using the pre-conditions and effects expressed in the clients' requests. During the trial of the CASCOM project the added value of this component will be evaluated.

2.1.3.2.1 PcECP's Internal Architecture

The PcECP component has two sub-components: the OWLS2PDDL converter and the Sapa Planner. As described before, after receiving the composition request, the SCPA must first contact the Service Discovery Agent (SDA) in order to obtain a set of candidate services to perform the composition planning. After receiving the set of atomic services that will be used for composition, the SCPA performs the second step, which refers to internal processing of the request.

In the PcECP this internal processing is related to input Sapa parameters. The Sapa has two input parameters, the description of the Domain and the description of the Problem (both in PDDL). Thus, the SCPA uses an internal component named OWLS2PDDL, which will provide those two translated inputs, ready to be used by Sapa. When the two inputs are translated into PDDL, the SCPA begins the composition process using the Multi-Objective Metric Temporal Planner Sapa.

Sapa's output will be a chaining of services mostly based on their own pre-conditions and effects that satisfy the composition's goal. Once the planning is done, all information regarding the chained services is extracted (e.g., pre-conditions and effects) to be then processed by an algorithm that will provide the information about the composite service. Such information is displayed through the composite service inputs, outputs, pre-conditions and effects (IOPEs) considering the generated chain of services. After all those steps, the information (IOPEs) produced by the algorithm and the chain of services obtained by the planning, are then loaded in OWL-S objects in the OWL-S API [S04] in order to produce a complete OWL-S description of the composite service.

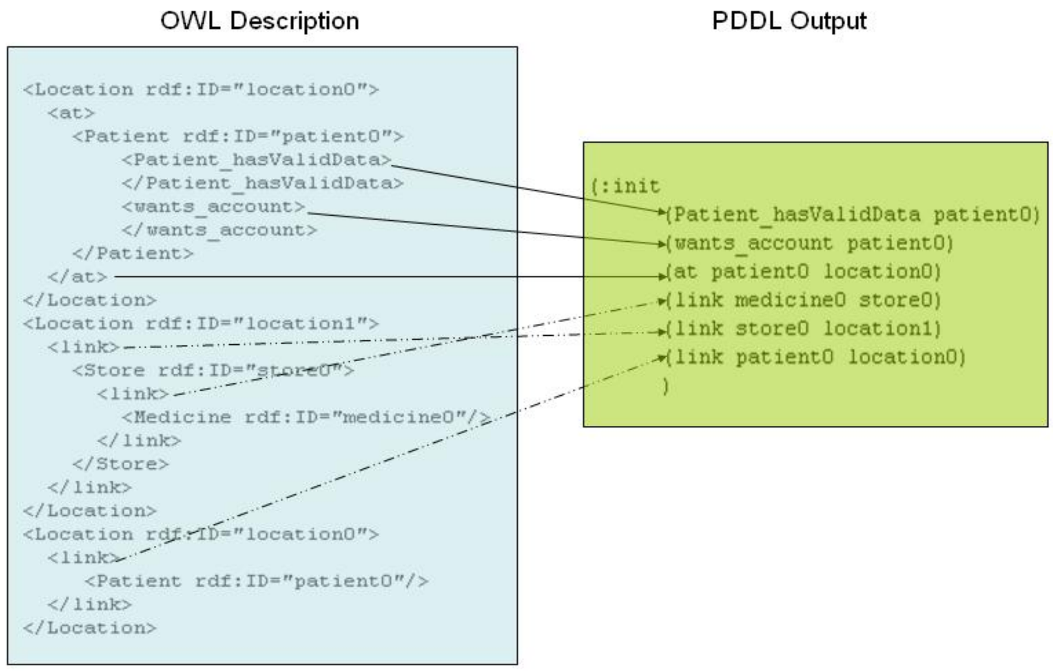
2.1.3.2.1.1 The Converter: OWLS2PDDL

As previously explained, when a service composition is needed, the Service Composition Planner Agent receives the initial state of the world, the goal to achieve both in OWL and the request description in OWL-S with the service categories to be part of the new composite service. Considering the fact that Sapa inputs must be in PDDL, a conversion needs to be done.

A typical problem file in PDDL has three sections: the Objects, the Init and the Goal definitions. The Objects section has all objects mapped to their own types (e.g., location0 location1 – Location). The Init section has all predicates that, given their arguments, produce the initial state of the world (e.g., at patient0 location0, wants_account patient0). At last, the Goal section has all predicates that also, given their arguments, produce the goal to achieve (e.g., Patient_ownsMedicine patient0 medicine0, at medicine0 location0).

To produce the problem file in PDDL, the converter only needs to process the Init and Goal descriptions received by the SCPA. Such files follow a unique pattern. They all have the base ontology followed by the predicates that describe their best interests (Init, Goal).

Given OWL/OWL-S description format, it is possible to map all our needs knowing what each tag represents. So, the converter processes those two files and creates the referred objects attached to their types, the Init section with the predicates and arguments extracted from the initial state of the world ontology file, and the goal section similar to the Init, but using the Goal ontology file. Figure 9 shows an example of a tipical Init description conversion to PDDL.



```
<Location rdf:ID="location0">
  <at>
    <Patient rdf:ID="patient0">
        <Patient_hasValidData>
        </Patient_hasValidData>
        <wants_account>
        </wants_account>
    </Patient>
  </at>
</Location>
<Location rdf:ID="location1">
  <link>
    <Store rdf:ID="store0">
      <link>
        <Medicine rdf:ID="medicine0"/>
      </link>
    </Store>
  </link>
</Location>
<Location rdf:ID="location0">
  <link>
     <Patient rdf:ID="patient0"/>
  </link>
</Location>
```

```
(:init
      (Patient_hasValidData patient0)
      (wants_account patient0)
      (at patient0 location0)
      (link medicine0 store0)
      (link store0 location1)
      (link patient0 location0)
      )
```

Figure 9 – Example of Init conversion to PDDL

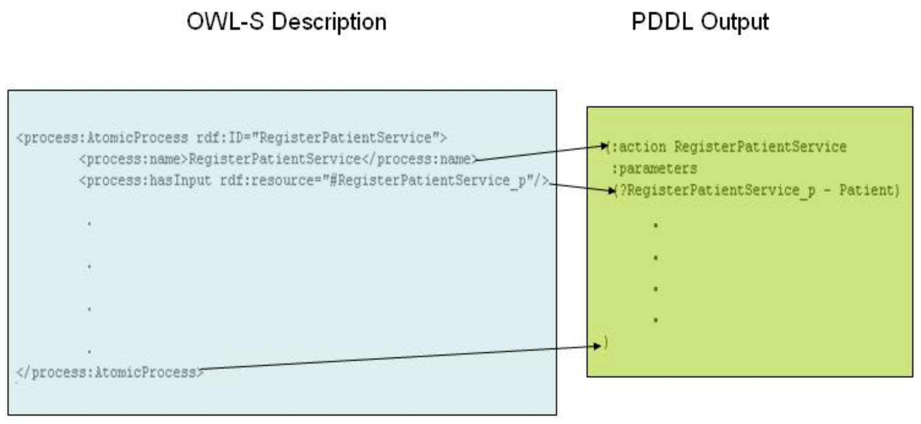
A typical domain file in PDDL has also three sections, yet, obviously different from the problem. Those sections are: the Types, the Predicates and the Actions. Usually the Types section contains object types matched to their super types (e.g., Location – Object, Medicine - Product).

The Predicates section presents the predicates that can and/or will represent the action's pre-conditions and/or effects (e.g., Patient_ownsMedicine ?p – Patient ?m – Medicine). Finally the Actions section contains all actions (services), belonging to the domain, that will be used to perform the plan. Such actions may or may not be chosen by the planner to be part of the plan.

Each Action (Service) has three sub-sections: the Parameters (e.g., ?RegisterPatientService_p – Patient), the Pre-conditions (e.g., Patient_hasValidData ?RegisterPatientService_p) and the Effects (e.g., not(wants_account ?RegisterPatientService_p)).

To create the domain file in PDDL the converter needs to process the ontology files (Init, Goal) as well as the service files. These service files refer to the services returned by the SDA and each service will become an action to be part of the domain file. Hence, to create the Types and the Predicates section, the converter only has to process the ontology files and figure out which types are sub types of others and which predicates exist in the domain.

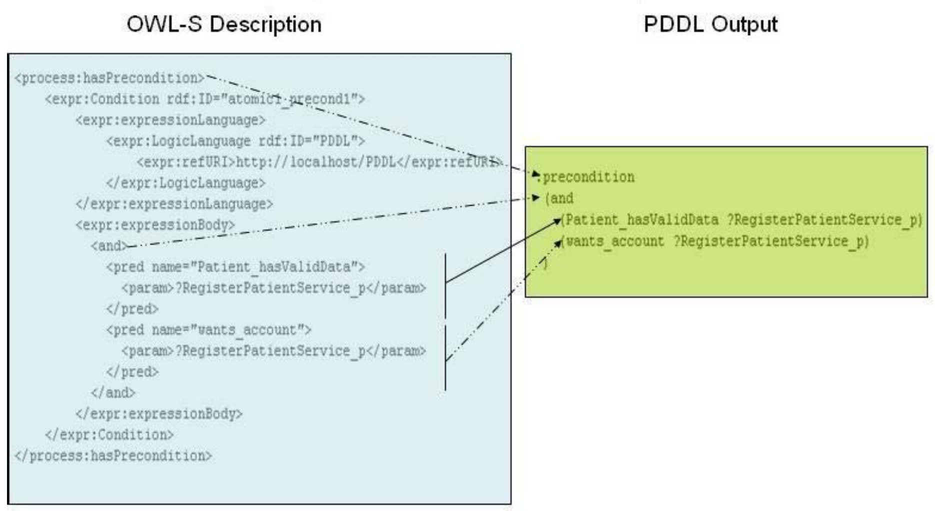
Similar to these two sections (Types and Predicates) but slightly more complex, comes the Actions section. Each Service file, will state an Action in the domain file. In order to ease the comprehension of the conversion process, a few pictures are presented next.



```
<process:AtomicProcess rdf:ID="RegisterPatientService">
        <process:name>RegisterPatientService</process:name>
        <process:hasInput rdf:resource="#RegisterPatientService_p"/>
         .
         .
         .
         .
</process:AtomicProcess>
```

```
(:action RegisterPatientService
 :parameters
 (?RegisterPatientService_p - Patient)
      .
      .
      .
      .
)
```

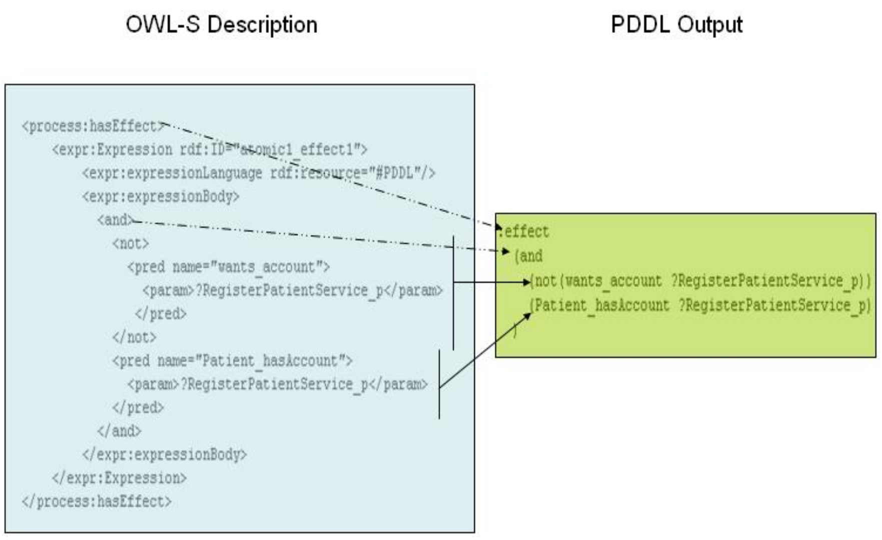
Figure 10 – Beginning of the Action conversion process



```
<process:hasPrecondition>
    <expr:Condition rdf:ID="atomic1-precond1">
        <expr:expressionLanguage>
            <expr:LogicLanguage rdf:ID="PDDL">
                <expr:refURI>http://localhost/PDDL</expr:refURI>
            </expr:LogicLanguage>
        </expr:expressionLanguage>
        <expr:expressionBody>
          <and>
            <pred name="Patient_hasValidData">
              <param>?RegisterPatientService_p</param>
            </pred>
            <pred name="wants_account">
              <param>?RegisterPatientService_p</param>
            </pred>
          </and>
        </expr:expressionBody>
    </expr:Condition>
</process:hasPrecondition>
```

```
:precondition
 (and
  (Patient_hasValidData ?RegisterPatientService_p)
  (wants_account ?RegisterPatientService_p)
 )
```

Figure 11 – Action’s pre-conditions conversion process



```
<process:hasEffect>
    <expr:Expression rdf:ID="atomic1_effect1">
        <expr:expressionLanguage rdf:resource="#PDDL"/>
        <expr:expressionBody>
          <and>
            <not>
              <pred name="wants_account">
                <param>?RegisterPatientService_p</param>
               </pred>
            </not>
            <pred name="Patient_hasAccount">
              <param>?RegisterPatientService_p</param>
            </pred>
          </and>
        </expr:expressionBody>
    </expr:Expression>
</process:hasEffect>
```

```
:effect
 (and
   (not(wants_account ?RegisterPatientService_p))
   (Patient_hasAccount ?RegisterPatientService_p)
 )
```

Figure 12 – Action's Effects conversion process

Figure 10, Figure 11 and Figure 12 show which parts the converter uses to perform the conversion. As stated before, the Action has to be composed of three sub-sections. These pictures show an example of the RegisterPatientService (action) and its parameters, pre-conditions and effects.

2.1.3.2.1.2 The Planner: Sapa

Sapa is a forward chaining planner, which searches in the space of time-stamped states. Sapa handles durative actions as well as actions consuming continuous resources. It was developed regarding heuristics for focusing Sapa's multi-objective search. These heuristics are derived from the optimistic reachability information encoded in the planning graph. Unlike classical planning heuristics, which need only estimate the "length" of the plan needed to achieve a set of goals, Sapa's heuristics need to be sensitive to both the cost and length ("makespan") of the plans for achieving the goals.

Sapa improves the temporal flexibility of the solution plans by post-processing these plans to produce order constrained (or partially-ordered) plans. This way, Sapa is able to exploit both the ease of resource reasoning offered by the position-constrained plans and the execution flexibility offered by the precedence-constrained plans.

All definitions aside, Sapa's purpose in this component is to generate, if possible, a chain of services based on their pre-conditions and effects that satisfy the composition's goal. Such output will then be loaded in the OWL-S API to produce the composite service description as previously explained.

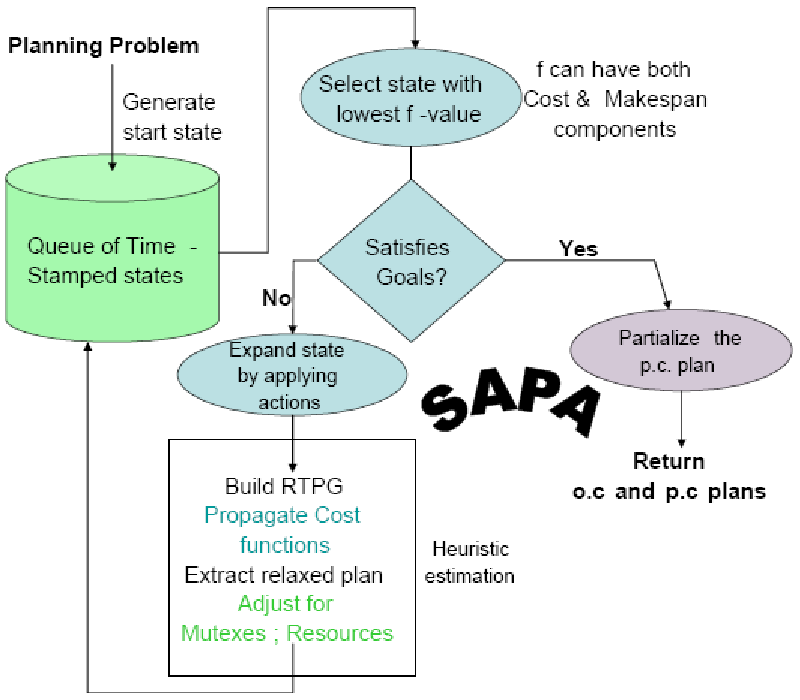


Figure 13 – Sapa's architecture

Figure 13 shows the high-level architecture of Sapa. Sapa uses a forward chaining A* search to navigate in the space of time-stamped states. Its evaluation function is multi-objective and is sensitive to both *makespan* and action cost. When a state is picked from the search queue and expanded, Sapa computes heuristic estimates of each of the resulting children states. The heuristic estimation of a state S is based on (i) computing a relaxed temporal planning graph (RTPG) from S, (ii) propagating cost of achievement of literals in the RTPG with the help of time-sensitive cost functions (iii) extracting a relaxed plan P_r for supporting the goals of the problem and (iv) modifying the structure of P_r to adjust for *mutex* and resource-based interactions. Finally, P_r is used as the basis for deriving the heuristic estimate of S. The search ends when a state S0 selected for expansion satisfies the goals. In this case, Sapa post-processes the position-constrained plan corresponding to the state S to convert it into an order constrained plan. This last step is done to improve the *makespan* as well as the execution flexibility of the solution plan.

2.1.3.3 Filters for Service Composition: A Role and Category based Approach

According to the CASCOM Architecture, when a Personal Agent requests a service, it first contacts the SDA to search for it. If no providers are found, then the PA asks the SCPA (Service Composition Planning Agent) to create a composite service that includes several pre-existing services. In order to be able to generate such a plan that matches the original query, the SCPA needs a set of input services to set out from.

Ideally, the set of services taken into account to create the composite plan should comprise all services registered in the directory. However, this can be impracticable as the number of services increases, as it is expected to occur in the open IP2P environments that CASCOM targets. To overcome that problem, it is necessary to reduce the set of input services that are passed on to the planner. For this purpose, we propose filters that sort out those services registered within the directories that are less relevant to the planning process.

Selecting the set of candidate services to form the plan is not an easy task. Several ad-hoc heuristics can be thought of (e.g. services that share at least one input or output with the query etc…). In this section we propose a more informed method for filtering services that makes use of *service class information*. We first develop a generic framework for service-class based filtering, and then instantiate it to different filters on the basis of (a) organizational information obtained from the role ontology and (b) the service category derived from the directory structure.

The SCPA will be equipped with a component that implements the approach presented in this section. During the trial of the CASCOM project the added value of this component will be evaluated.

2.1.3.3.1 Overview

In this section we give a birds-eye view of our approach to service filtering for composition. Setting out from an ideal situation, we outline the major difficulties that need to be overcome, so as to motivate our service filtering mechanism. The technical details of the process are described in subsequent subsections.

At a high level of abstraction, the service composition planning problem can be conceived as follows: Let $P = \{p_1, p_2, \ldots, p_m\}$ be the set of all possible plans (composite services) for a given service request R, and $D = \{s_1, s_2, \ldots, s_n\}$ the set of input services for the proper service composition planner (i.e. the directory[8] available). The objective of a filter F is to select a given number l of services from D, such that the search space is reduced, but the best plan of P can still be found.

In an *ideal situation* with complete information (i) the set of all plans P is known and (ii) the *quality* of each plan can be evaluated (obviously, the plan that requires the least number of services is not always the one that best matches a query). In this case, the set of services returned by the filter should include all the services of the best plan (as well as some others until the number of l services is reached). However, it is obvious that this ideal case is not realistic since the problem would be already solved (i.e. the plan of maximum quality is supposed to be known beforehand).

In a next step, suppose that it is not possible for the filter to evaluate the quality of the plans in P. In this case, if the number of services necessary for the execution of all plans in P is bigger than the number of services l that are allowed to pass our filter, the latter should make sure that the pruning of the search space for the planner is minimal. Put in another way: the bigger the subset of plans $P' \subset P$ that the planner can choose from, the bigger the probability that the plan of maximum quality is among them. Therefore, the filter should select those services that maximise the cardinality of P', i.e. that maximise the number of plans from P that are available to the planner. A good heuristic to this respect is based on *plan dimension* and on the *number of occurrence*s of services in plans: a service is supposed to be the more important, the bigger the number of plans from P that it is necessary for, and the shorter the plans from P that it is required for.

Again, it is unrealistic to assume complete information respecting the set of plans P for a given query R in general, and respecting their length and number of occurrences of services in them in particular. Nevertheless, we can approximate this information by storing and processing the plans historically created. So, in principle, we can build up matrices as the shown in Table 1 for every possible query. In the example, for some query R, service s_2 was part of 10 plans composed by two services and 55 plans formed by 3 services. In the example, the matrix also stores the number of plans generated of each dimension.

[8] In this context, we use the term "directory" to denote the set of services retrieved by the SDA from the CASCOM Directory Services.

Historical information about plans for service request *R*				
Dimension	1	2	3	…
# of plans	0	50	70	
s_1	0	7	24	
s_2	0	10	55	
s_3	0	33	21	
…				

Table 1 – Example of information about historical plans

However, it soon becomes apparent that the number of services and possible queries is too big to build up all matrices of the above type. The memory requirements would be prohibitively high and the filtering process would become computationally too expensive. Furthermore, the continuous repetition of a very same service request *R* is rather unlikely. And, even more important, this approach would not be appropriate when a new service request (not planned before) is required (which, in fact, is quite usual).

To overcome this drawback, we make use of *service class information* available in the CASCOM framework, so as to cluster services based on certain properties (in particular, we will use the service categories provided by the directory structure, and the role taxonomy reflecting the organisational structure underlying CASCOM interactions – see D5.1). So, there will only be matrices for each *class* of service request (query), and the matrix information stored for *classes* of services instead of services. If the number of classes is not too big, the aforementioned approach can become feasible computationally.

Service classes need not be disjoint. This will allow, for instance, describing that an ambulance service can belong to both transport and medical categories, or search for a service able to play an advisor and explainer role when engaging in interactions to provide the service. In particular, for service advertisements we allow specifying a set of classes, and for service requests we allow a formula in disjunctive normal form (i.e. a disjunction of conjunction of classes). This perfectly fits the requirements of role-based service composition filters (see the role-based service matchmaking approach described in D5.1 for further details on the structure of queries and service advertisements).

Figure 14 depicts the structure of our approach to service composition filtering. With each outcome of a service composition request, a Historical Information Matrix *H* (an abstraction of Table 1) is updated. Setting out from this information, a Relevance Matrix *v* is revised and refined. Based on this matrix, service relevance can be determined in a straightforward manner. For each service composition request, the filtering method is based on this service relevance function.

In the following we go into the details of the different aspects of the outlined approach (depicted in Figure 14). First, in section 2.1.3.3.2, we will see how the relevance matrix is obtained. Afterwards, the calculus of service relevance (making use of the relevance matrix) is described. In section 2.1.3.3.4, different options for the instrumentation of service composition filters are outlined. Subsequently, a filter based on both role-based and service category-based classification is presented. Finally, in section 2.1.3.3.6, our approach to the implementation of service composition filters is sketched.

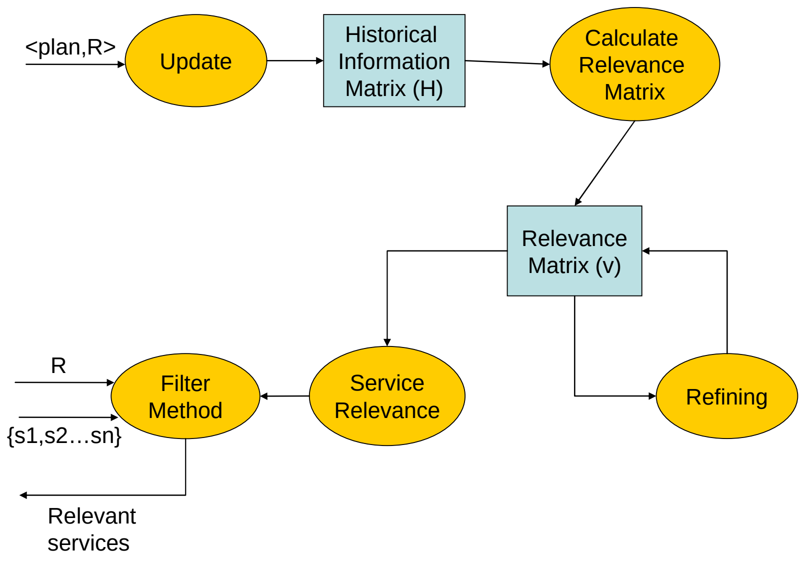


Figure 14 – Architecture of the filter component

2.1.3.3.2 Obtaining the Relevance Matrix

In this section we describe how the relevance matrix *v(s,r)* can be obtained from past plans. We first describe how the information about plans is stored and how the relevance matrix is calculated from this information. Then, we propose a method to refine the relevance matrix. Finally, several options for bootstrapping are described.

Historical information about plans

The *relevance matrix v* represents the estimation of a *service class s* to be included in a (the best) composite plan to provide the requested *service class r*. In order to create the relevance matrix we set out from a set of plans (composite services) that were created in the past. We deal with simplified versions of those plans. In particular we are only interested in the *classes of services* that compose every plan. More precisely, for each plan we use the following information:

a) Request classes expression (disjunctive normal form) included in the request. We denote C_{R1}, C_{R2}, ..., C_{Rn} the classes included in that expression.

b) Plan classes: is the set of classes PC = {C_{P1}, C_{P2}, ..., C_{Pm}} obtained after mapping the services included in the composite plan.

The information about past plans created is stored in a matrix like Table 2 (adapted from Table 1 by considering classes instead of services). A matrix like this will exist for every different class of services. We will name H^R the matrix with information about request class *R*.

H^R: Historical information about plans for service class R (Request)				
Dimension	1	2	3	…
# of plans	0	50	70	
C_1	0	7	24	
C_2	0	10	55	
C_3	0	33	21	
…				

Table 2 – Example of classes information about historical plans

The information in this table is computed as follows. If only one service class is included in the request expression, the dimension (cardinality of *PC*) is calculated and the value of the "# of plans" and every class of *PC* is incremented by one, for that dimension[9].

However, when more than one class are present in the request expression, things are not that simple. The problem is that, when there is more than one class in the request, it is not straightforward to determine the parts of the request that the service (classes) of the composite plan are relevant for.

In the case of a conjunction expression (e.g. $C_{R1} \wedge C_{R2}$), we assume that all the classes in the plan are relevant for both C_{R1} and C_{R2}, since both conditions must be fulfilled. However, in the case of a disjunction this is not the case. For instance, if the request expression is ($C_{R1} \vee C_{R2}$) and the plan includes the classes $\{C_{P3}, C_{P4}, C_{P5}\}$, it is not clear whether these three classes are participating in providing both C_{R1} and C_{R2}, or if, for instance, C_{P3} and C_{P4} are relevant for C_{R1}, and C_{P5} only for C_{R2}. In this last case we decrease the possible negative impact of adding one unit to the wrong class by weighting the contribution of the plan information by the inverse of the number of terms in the disjunction expression. For example, suppose a query (request expression) of the following shape: ($C_{R1} \vee (C_{R2} \wedge C_{R3}) \vee C_{R4}$). Furthermore, assume that the classes of services required by the plan (composite service) that has been determined to best match this query are PC = $\{C_{P1}, C_{P2}\}$. According to our model, the dimension of that plan is 2, and the historical matrixes corresponding to the four classes in the request expression (C_{R1}, C_{R2}, C_{R3}, C_{R4}) are updated in line with this. In particular, the values of H^{CR1}("# of plans",2), $H^{CR1}(C_{P1},2)$, $H^{CR1}(C_{P2},2)$, H^{CR2}("# of plans",2), $H^{CR2}(C_{P1},2)$, $H^{CR2}(C_{P2},2)$, and so on for H^{CR3} and H^{CR4}, are incremented by 1/3 as the number of disjunctive terms in the query is 3. Figure 15 shows the algorithm for updating the historical matrix.

[9] As all services need to be available for the plan to be executable, every service class has the same relevance for the plan, independently of whether one or several services of that class are used by the plan.

```
<PC> = SET OF Class         // plan classes
<RE> = <DisjunctionExpr>    // request expression
<DisjunctionExpr> = SET OF <ConjunctionExpr>
<ConjunctionExpr> = SET OF Class

UpdateHistorical(H: historical matrix;RE: request expression;PC: plan classes)
{
  dim = Card(RE)
  FOR ALL Conj IN RE {
    FOR ALL r IN Conj {
      $H^r$("# of plans",dim) = $H^r$("# of plans",dim) + 1/dim
      FOR ALL p IN PC {
        $H^r$(p,dim) = $H^r$(p,dim) + 1/dim
      }
    }
  }
  return H
}
```

Figure 15 – Historical information matrix update algorithm

Calculus of the relevance matrix

As commented above, by ranking services we try to select a set of services that cover the largest subset of the plan space, as an attempt to maximise the chance of the best plan to be contained in it. Services that formed smaller plans in the past are considered more relevant, since it is easier to cover small plans that large ones, so with less services more plans can be covered.

We use the following function to aggregate the information about plans (remember that all this information is about a single request class *R*):

$$\text{Relevance}(\text{C, R}) = \frac{\sum_{d=1}^{m} \frac{n_d}{d^c}}{\sum_{d=1}^{m} \frac{N_d}{d^c}},$$

Where *d* is the dimension of the plan, *m* is the dimension of the longest plan stored, n_d is the number of times that *C* was part of a composite plan of dimension *d* for the request *R*, and N_d is the total number of plans of dimension *d* (“# of plans”) for that request. *c* is a constant > 0 that allows giving more importance to plans of smaller dimension (a straightforward value is k=1). Only dimensions with more than 0 plans are considered.

In the example of Table 2,

$$\text{Relevance}(\text{C}_1, \text{R}) = \frac{7/2 + 24/3}{50/2 + 70/3} = 0{,}24$$

With this calculus we obtain a *relevance* value between 0 and 1 for every given service class *C* with respect to the composition of a service of class *R*.

Table 3 shows a partial example of the relevance table. In that table *v(i,j)* is the estimated relevance of service class C_i for a request (query) class C_j.

v		Requests							
		C_1	C_2	…	C_i	…	C_j		C_n
Services in directory	C_1		$v(1,2)$		$v(1,i)$		$v(1,j)$		$v(1,n)$
	C_2								
	…								
	C_i				$v(i,i)$		$v(i,j)$		
	…								
	C_j								
	…								
	C_n								

Table 3 – Relevance matrix.

Note that each cell of this table is obtained from one row of Table 2 and that each column of Table 3 is obtained from one of the *n* Historical Information Matrices. Thus the time complexity for the calculation of the relevance matrix is $O(m \cdot n^2)$, being *m* the dimension of the longest plan stored and *n* the number of classes (recall that the number of classes *n* is supposed to be fixed and not overly high).

Refining the relevance matrix

The matrix *v(s,r)* specifies the relevance of a service class *s* to be part of a plan (composite service) that matches the query for a certain service class *r*. However a situation like the following may occur: Suppose that a plan that achieves C_1 is searched for, and that a potential solution is to compose the services C_2 and C_3 ($C_2 \oplus C_3$ for short[10]). However there is no service provider for C_3, but instead C_3 can be composed as $C_4 \oplus C_5 \oplus C_6$, so the final plan is $C_2 \oplus C_4 \oplus C_5 \oplus C_6$. Unfortunately, the value $v(C_4,C_1)$ is low and the service providing C_4 is discarded and not taken into account in the planning process, so the aforementioned plan cannot be found by the planner. Therefore, we will refine the relevance matrix by taking *transitivity* into account, e.g. through the following update: $v(C_4,C_1) = v(C_4,C_3)*v(C_3,C_1)$. The same holds for third-level dependencies (e.g.: $v(C_7,C_1) = v(C_7,C_4)*v(C_4,C_3)*v(C_3,C_1)$). This example motivates the definition of the $v^k(s,r)$ as a *k*-step relevance matrix

$$v^1(s,r) = v(s,r)$$

$$v^k(s,r) = Max\ (v^{k-1}(s,r),\ v^{k-1}(s,s_1) * v(s_1,r),\ v^{k-1}(s,s_2) * v(s_2,r),\ \dots\ v(s,s_n) * v(s_n,r))$$

As shown in the equation, we use the product as combination function and the maximum to aggregate the results (as always, other more complex aggregation functions are possible).

Table 4 shows a relevance matrix for this example. In that case, $v(C_4,C_1) = 0.1$, but $v^2(C_4,C_1) = v(C_4,C_3)*\ v(C_3,C_1) = 0.8*0.7 = 0.56$.

[10] Here $\oplus$ denotes a composition operator.

v		Requests							
		C_1	C_2	C_3	C_4	C_5	C_6	...	C_n
Services in directory	C_1	1							
	C_2	0.8	1						
	C_3	0.7		1					
	C_4	0.1		0.8	1				
	C_5	0.5				1			
	C_6	0.6					1		
	…							1	
	C_n								1

Table 4 – Example of relevance matrix

Note that the higher the value of *k* the better the estimation of the relevance of service classes. The refinement of the relevance matrix is repeated until it converges (i.e. $p^{k+1}(s,r) = p^k(s,r)$) or until a timeout is received. Figure 16 shows the algorithm. The elevated time complexity of $O(n^3)$ for each refinement step is attenuated by the *anytime properties* of the approximation algorithm. Furthermore, recall that the number of classes *n* is supposed to be fixed and not overly high. Finally, note that several updates and refinements can be combined into a “batch” to be executed altogether when the system’s workload is low.

```
RefineMatrix(v: relevance matrix)
{
  REPEAT
    v = v_next
    v_next = RefineStep(v)
  UNTIL v = v_next
}

RefineStep(v: relevance matrix)
  v': relevance matrix
{
  FOR ALL s IN C1..Cn {
    FOR ALL r IN C1..Cn {
      v'(s,r) = v(s,r)
      FOR ALL i IN C1..Cn {
        v'(s,r) = max(v'(s,r),v(s,i)*v(i*r))
      }
    }
  }
  return v'
}
```

Figure 16 – Relevance matrix refinement algorithm

Bootstrapping

There are several ways of obtaining the initial relevance matrix:

a) If there are historical records of plans they can be used to calculate the matrix

b) An a priori distribution can be assigned using expert (heuristic) knowledge

c) The SCPA can work for a while without filtering services until the number of plans generated is considered representative enough. Then the relevance matrix is calculated and refined.

These options can also be combined. For instance, the heuristic a priori distribution can be combined with the historical database as the starting matrix. In addition, if that matrix is supposed to be insufficiently informed, then it may be completed with option c). An a priori distribution based on expert (heuristic) knowledge can be incorporated into our model directly as the *relevance matrix* but preferably as a (fictitious) *historical matrix*. The latter has the advantage that it can be combined with historical records and/or other expert knowledge (there might be several expert sources) by adding all the historical matrixes, followed by the calculus of the *relevance matrix* as explained above. If the a priori distribution were included directly as a *relevance matrix* it could not be combined with historical information since the *relevance matrix* is obtained from the *historical matrix* and, in consequence, the a priori information would be lost.

2.1.3.3.3 Service Relevance Calculus

In this section we describe how the relevance of a service *S* for a request *R* is calculated using the relevance matrix *v*.

The first step to calculate the *relevance* of a service *s* for a request *r* is the mapping of both to *classes of services*. Then, the relevance between the classes is calculated. We will use the following notation:

$v(s,r)$: relevance of *class s* for the *class r* in the request, and

$V(S,R)$: relevance of service *S* for the service request *R*

Depending on whether the services are mapped to one or several classes we apply the following.

1) The simplest case is a request *R* that only includes a class (*r*) in its description. Two cases are possible:

a) The service *S* only belongs to one class (*s*): in this case $V(S,R) = v(s,r)$

b) The service *S* belongs to *several* classes (s_1, s_2 … s_n). In this case we take the highest relevance of the different classes, i.e. $V(S,R) = \max(v(s_1,r), v(s_2,r), \ldots, v(s_n,r))$. Again, although more sophisticated functions are conceivable, we use the maximum for aggregation as it is easy to compute and intuitive for humans.

2) The request specifies a logical expression containing several classes of services (r_1, r_2 … r_m). Now, again we have two cases:

a) The service *S* only belongs to one class (*s*). We evaluate logical formulas using the *maximum* for disjunctions and the *minimum* for conjunctions. For example, if the request *R* includes the formula $r_1 \vee (r_2 \wedge r_3)$, then $V(S,R) = \max(v(s,r_1), \min(v(s,r_2), v(s,r_3)))$.

b) The service *S* belongs to several classes (s_1, s_2 … s_n). In this case we combine the two previous options: the request formula is evaluated by decomposing it as 2a) and using inside the *maximum* to aggregate the service classes specified by the provider. For instance, if in the last example the service *S* belonged to the classes s_1 *and* s_2, the calculus would be:

$V(S,R) = \max[\max(v(s_1,r_1), v(s_2,r_1)), \min(\max(v(s_1,r_2),v(s_2,r_2)),\max(v(s_1,r_3),v(s_2,r_3)))]$.

Figure 17 shows the algorithm to calculate the relevance of a service *S* for a request *R*. This is done by the *ServiceRelevance* function. This function uses the *SingleRelevance* function, which returns the relevance of the service advertisement *S* for one single request's class *r*. As described before, the request may not only include a *class of service* but also an expression (a disjunction of conjunction of classes). The two loops decompose that expression, using the minimum as combination function for the values in a conjunction and the maximum for disjunctions. Assuming that the maximum number of classes that a service can belong to is negligible, the time complexity of the algorithm is linear in the number literals in the query.

```
ServiceRelevance(S: service advertisement; R:service request; v: relevance matrix)
{
  rel = 0
  FOR ALL ConjunctionExpr IN R {
    rel' = inf
    FOR ALL r IN ConjunctionExpr {
      rel = min(rel',SingleRelevance(r,S,v))
    }
    rel = max(rel, rel')
  }
  return rel
}
SingleRelevance(r: request class; S: service advertisement; v: relevance matrix)
{
  rel = 0
  FOR ALL s IN S {
      rel = max(rel,v(s,r))
  }
  return rel
}
```

Figure 17 – Service relevance algorithm

2.1.3.3.4 Types of Generic Service Composition Filters

When a service request is analysed by our filter, the set of services are first ranked by an estimation of the relevance of the service class for that request. Then, only the services belonging to the best ranked classes are passed on to the planner.

In order to determine the concrete services that pass the filter we consider three major options:

a) To establish a *threshold* and filter out those services whose classes have a degree of relevance lower than that threshold.

b) To return the estimated *k best* services based on the relevance of their corresponding classes. In this case the number of services that pass the filter is pre-determined.

c) To return a *percentage* of the original set of services (based on the relevance of their corresponding classes). In this case the number of services considered in the planning process depends on the directory size.[11]

When designing the algorithms corresponding to these filters configurations, an additional problem needs to be taken into account. Services with low (or even zero) relevance values would never be considered for planning, so they could never be part of a plan (composite service), remaining with low relevance forever. This is obviously too restrictive, as our relevance values are only estimations based on the information available at some point in time. To overcome this we allow some services to be fed into the planner even though they are not supposed to be relevant enough according to the filter policy. Those additional services are chosen randomly. This random option is combined with the three aforementioned filter types to allow for an exploration of the service (class) space.

Figure 18 shows the three algorithms for *class*-based filtering of services, depending on the desired filter type. The three functions receive a set of service descriptions, a service request, and the relevance matrix.

[11] Recall that, in this context, the directory size is given by the number services that the SDA provides to the SCPA.

The *ThresholdFilter* function returns all the services whose relevance is above a given threshold (received as parameter). Its result also includes additional services which are chosen randomly with a probability given in a parameter.

The *K-Filter* function returns the *k* best estimated services. In addition, a *K-RANDOM* number of services is randomly selected among the rest of the directory.

The *%-Filter* function returns a pre-determined number of services. In this case the number is not specified directly but as a percentage of the directory size. Also a percentage is specified to be included randomly. Note that the algorithm for this function is the *K-Filter* where the *k* values are calculated from the cardinality of the set of input services and the desired percentage.

Note that, if necessary, the random option can be inhibited by passing the value 0 in the corresponding parameter.

```
ThresholdFilter(S: set of services; R: service request; v: relevance matrix,
                THRESHOLD: [0..1]; RANDOM_PROBABILITY: [0..1])
{
  result = ∅
  FOR ALL s IN S {
    IF (ServiceRelevance(R,s,v) > THRESHOLD) OR
       (random(0..1) < RANDOM_PROBABILITY) THEN
      result = result ∪ s
  }
  return result
}

K-Filter(S: set of services; R: service request; v: relevance matrix,
         K: Integer; K-RANDOM: Integer)
{
  relevances = ∅
  FOR ALL s IN S {
    SortInsert(<s, ServiceRelevance(R,s,p)>,relevances)
  }
  result = relevances[1..K]
  i = 0
  WHILE i < K-RANDOM {
    x = random(K+1..Card(relevances))
    IF relevances[x] ∉ result THEN {
      result = result ∪ relevances[x]
      i := i+1
    }
  }
  return result[1..k]
}

%-Filter(S: set of services; R: service request; v: relevance matrix,
         PERCENTAGE: [1..100]; RANDOM_PERCENT: [0..100])
{
  n = Cardinal(S)
  return K_Filter(S,R,v,n*PERCENTAGE/100,n*RANDOM_PERCENT/100)
}
```

Figure 18 – Filter of services. The three possible configurations are described.

The mode of operation and its parameters allow adapting the filter depending on the context (domain, available resources, response time, etc). For instance, if the resources of the SCPA are very limited a *K-Filter* with small value of *k* or a high value *Threshold* filter might be used. If more resources are available, the threshold can be lowered and/or a higher number of randomly selected services may be added.

2.1.3.3.5 Service Composition Filter Instantiation

In the following we present two different approaches to apply the filtering framework proposed in this section. For each approach the mapping of services to classes is defined. Both methods are based on information available in the OWL-S service descriptions used by CASCOM.

a) Role-based filtering

This method is based on organizational concepts such as roles and type of social interactions. The idea is to relate roles searched in the query to roles played by agents in the composite service, that is, what are the roles typically involved in a plan when a role *r* is included in the query. For example, it is common that a *medical assistance* service includes *travel arrangement, arrival notification, hospital log-in, medical information exchange* and *second opinion* interactions.

In the CASCOM deliverable D5.1 (section 4) the role-based information included in service advertisements and service requests is explained (Figure 19 shows part of the types of interaction taxonomy). In CASCOM, service descriptions include information related to the interactions in which the service provider agent can engage. Each service provider can advertise several interactions. In the case of a service requests, it is allowed to specify the roles searched as an expression in disjunctive normal form. This information fits in the generic framework presented above by simply considering roles as classes of services, both for services advertisements and requests.

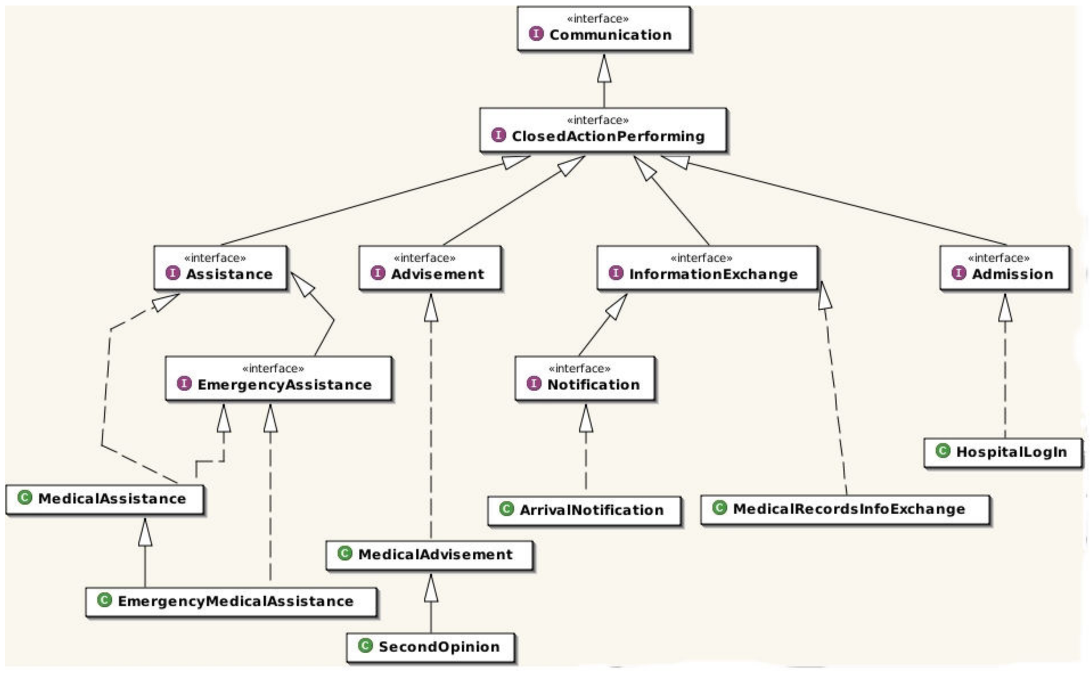

«interface» Communication
«interface» ClosedActionPerforming
«interface» Assistance
«interface» Advisement
«interface» InformationExchange
«interface» Admission
«interface» EmergencyAssistance
«interface» Notification
HospitalLogIn
MedicalAssistance
ArrivalNotification
MedicalRecordsInfoExchange
MedicalAdvisement
EmergencyMedicalAssistance
SecondOpinion

Figure 19 – Partial CASCOM interaction ontology

In our role based modelling approach (deliverable D5.1), we use a role taxonomy that is supposed to be static over significant amounts of time. Still, the ontology *can* be extended to include new roles and types of interaction not considered before. In that case, the relevance matrix is updated with new rows and columns for those new roles. The relevance values for those new roles are unknown initially, but this can be overcome by randomly including some services with low relevance (as outlined in section 2.1.3.3.4) and, in general, by applying the bootstrapping techniques described in section 2.1.3.3.2.

b) Category-based filtering

Another pertinent strategy for service classification is based on the *categories* (travel, medical…) they belong to. Such categories are considered important information in service descriptions (in fact, the OWL-S language includes a specific field for this characteristic). There are several well known category taxonomies (NAICS, UNSPSC,…). However, CASCOM does not choose one in particular, keeping it open to the service describer.

In our filter framework, each category is considered a *class of service*. Service descriptions include a set of categories. In the case of a service advertisement, this fits exactly our *classes* approach (set of classes). In the case of service requests, the set of categories specified are interpreted as a logical formula by connecting them with the operator *or* ($\vee$).

If the number of different classes (categories) is too big, the computational complexity (regarding both space and time) can become rather high. In that case, the granularity of the classes can be decreased by clustering several categories into the same class based on inheritance relations in the taxonomy tree.

The two types of classification of services presented can be combined as follows:

$$\text{Relevance}(S,C) = \alpha * \text{Rel}^{RB}(S,C) + (1-\alpha) * \text{Rel}^{CB}(S,C), \text{ with } \alpha \in [0..1].$$

2.1.3.3.6 Implementation

In this section we sketch the implementation of the filter component that will be part of the SCPA, together with the service composition planners. The general architecture of the component is depicted in Figure 20. The *RelevanceFilter* is the generic interface used to filter the available services, given a specific request. It has a unique method to be implemented, called *calculateServiceRelevance()*, which returns how relevant a service is for fulfilling a user request. The *RelevanceFilter* interface filters out the available services using one of the implemented filters (*PercentageFilter*, *KFilter* or *ThresholdFilter*).

Two specific relevance filters use different kinds of classifications of services: roles and service categories. The role-based relevance filter (*RoleRelevanceFilter*) calculates how a specified service advertisement is relevant respecting the requested service, using the role structures contained in both service advertisement and service request. The corresponding relational values between roles are stored in a *RoleRelevanceMatrix*. In the case of category-based relevance filters an analogous structure is used where the involved classes are *CategoryRelevanceFilter* and *CategoryRelevanceMatrix*. Our framework is quite general and extensible, so that others relevance filters can be implemented.

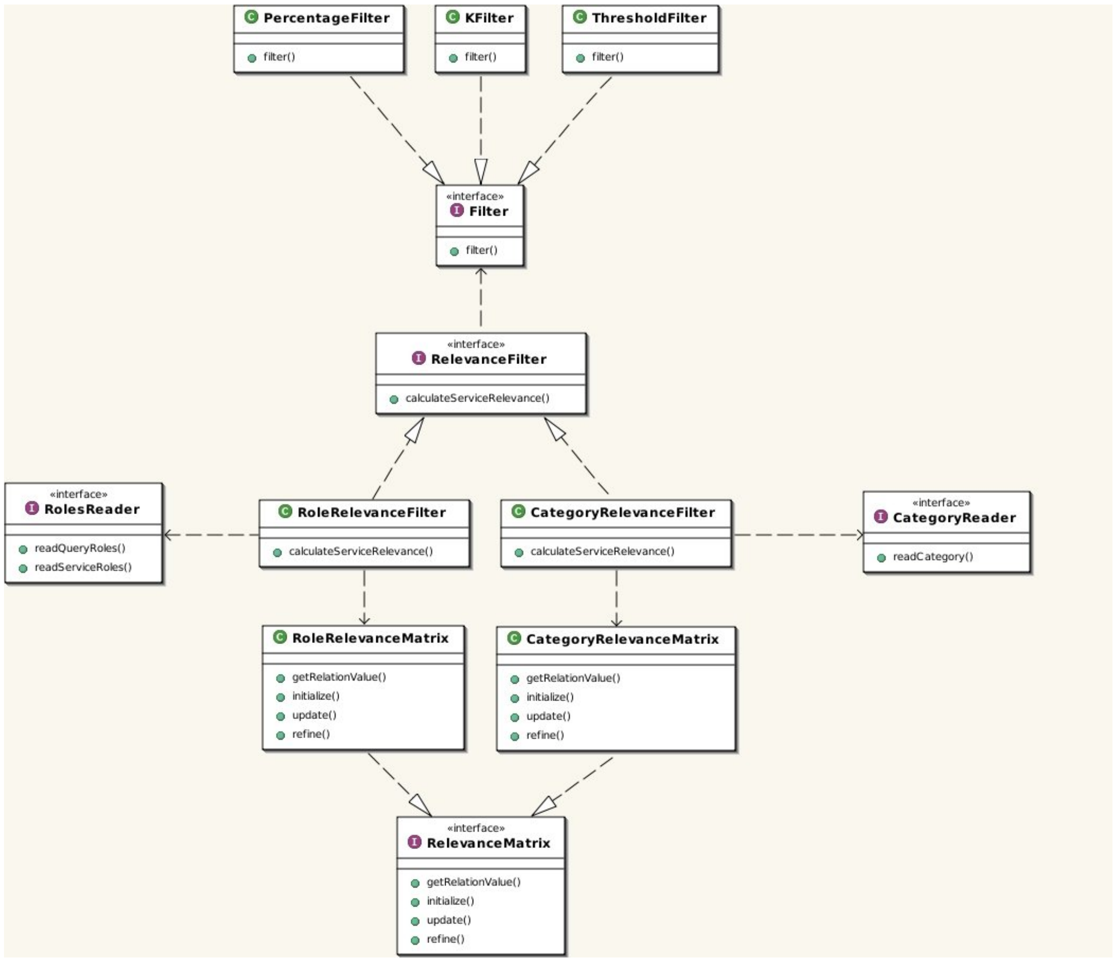


Figure 20 – Class diagram of the filter component

As an example, a general view of the control flow and the relations between the different classes of the role-based filter component is depicted in Figure 21 (the category-based component is similar). The SCPA uses the *RoleRelevanceFilter* to get, from the original set of services, a reduced set of relevant services. For every service, the *RoleRelevanceFilter* calculates its value of relevance, using the relation values stored in the *RoleRelevanceMatrix,* and then applies one of the implemented filters, returning a reduced set of services to the SCPA. The SCPA performs its planning action and, if a plan is found, it gives a feedback to the *RoleRelevanceFilter.* The feedback is simply the original service request and the composite service created. This information is used by the *RoleRelevanceFilter* to update the *RoleRelevanceMatrix.*

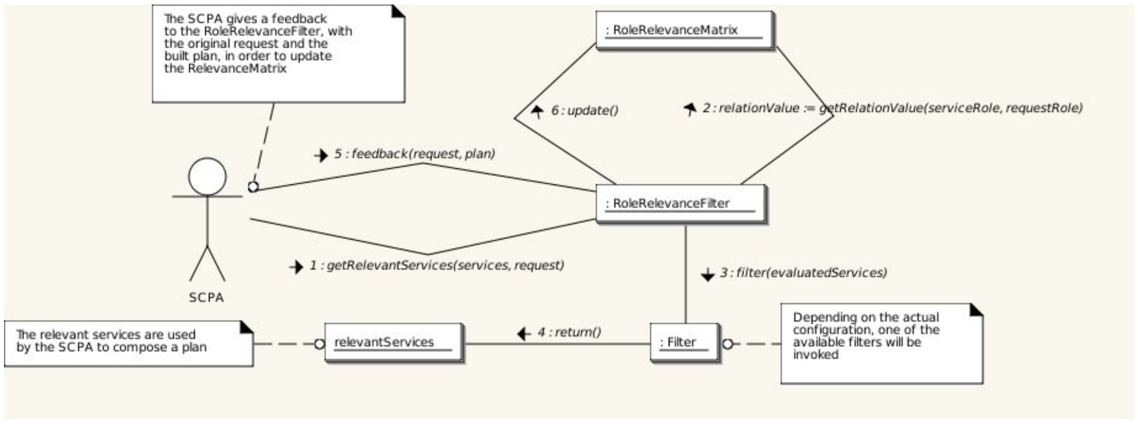

The SCPA gives a feedback to the RoleRelevanceFilter, with the original request and the built plan, in order to update the RelevanceMatrix
: RoleRelevanceMatrix
6 : update()
2 : relationValue := getRelationValue(serviceRole, requestRole)
5 : feedback(request, plan)
: RoleRelevanceFilter
1 : getRelevantServices(services, request)
3 : filter(evaluatedServices)
SCPA
The relevant services are used by the SCPA to compose a plan
relevantServices
4 : return()
: Filter
Depending on the actual configuration, one of the available filters will be invoked

Figure 21 – General view of class relations

2.2 Open Service Composition

In open environments such as the CASCOM system, non-deterministically occurring events during service composition planning or composition plan execution are possible. On the one hand, such events might make existing plans or plan parts invalid. On the other hand, better plans might become available. Examples for the first case include planned services becoming unavailable or facts that belong to a precondition for a planned service. An example for the second case is that a new service becomes available which could replace a number of services in an existing plan, leading more directly to the plan goal.

In order to extend OWLS-XPlan such that it can dynamically react to such events, we considered two possible techniques: contingency planning and heuristic re-planning. In a contingency planning approach, a number of plans for recognized classes of events are pre-computed. If such an event is then observed, the appropriate pre-computed plan is instantiated. Thus, contingency planning anticipates possible events and allows for a quick reaction in the case that some of these events do occur. The downside is the additional computational effort for all those possible plans, many of which will never be instantiated.

In contrast, heuristic re-planning is a pure reactive approach, where a re-planning is carried out only if it becomes necessary. To minimize the re-planning effort, an appropriate re-entry point in the existing (part of) the plan is computed using the heuristic. The idea is to re-use as much as possible of the existing plan, introducing only small changes to make it valid for the changed world state

In the following section, we introduce OWLS-XPlan+, a dynamic extension of OWLS-XPlan with heuristic quasi-online re-planning.

2.2.1 Dynamic Version of OWLS-XPlan

As it is described in section 2.1.3.1.1, the input for OWLS-XPlan consists of OWL ontologies and individuals for the domain description and planning query, as well as available OWL-S services. These inputs are then converted to PDDXML with OWLS2PDDXML converter. Thus, changes in the OWL ontologies, individuals and the set of available services each might affect operators, actions, predicates, facts and objects in the PDDXML problem and domain descriptions as well as existing plan parts. The OWLS-XPlan+ API thus extends the OWLS-XPlan API such that it might be informed about such events. In CASCOM, the SCPA will handle OWLS-XPlan+, and thus it will also be responsible for forwarding events to the planner. The SCPA itself will learn about events via the CASCOM context awareness subsystem.

Because a plan is a sequence of actions, i.e. operator instances, whose executions achieve the goal state, the following three main cases are relevant when considering a re-planning:

1. **An action becomes available.** This might happen if
    a. A new operator (service) is introduced.
    b. The world state (set of facts) is changed such that an operator whose instantiation was impossible before can be instantiated now.
    c. New predicates which are part of the pre-conditions or effects of an operator are introduced, making it possible to instantiate this operator.
2. **An action in the plan is not possible anymore.** This is possible if any of the opposites of points 1.a – 1.c happens.
3. **The goal state is changed.** This is the case if the planning request is changed.

Therefore, on an incoming event, OWLS-XPlan+ first decides which of these cases are given in the current situation. Each is handled separately by a respective algorithm. The three algorithms are shortly outlined in the following sections.

It has, however, to be noted that some events, such as the removal of an object, might lead to an inconsistent state if the set of facts is not also updated appropriately. Since OWLS-XPlan+ does not include a consistency checker for its inputs or incoming events, consistency has to be ensured by the caller (i.e. SCPA in CASCOM) to guarantee correct planning results.

2.2.1.1.1.1 New Action Becoming Available

In this case, OWLS-XPlan+ first checks whether the new operator might lead to a better plan, i.e., a plan containing less services. If this is the case, the point in the plan where the new operator might first be helpful is identified to start the re-planning from there. In more detail, the individual steps are as follows

1. **Re-planning decision**
    a. Using the same initial state as for the original plan plus the new operator *o*, build the relaxed plan graph and extract a new relaxed plan *P*.
    b. Estimate the length *L* of the new plan *P* via the heuristic
    c. If the length *L* is smaller than the heuristically estimated length of the relaxed plan in the original planning process, continue with step 2. Otherwise *o* provides no advantage and no re-planning is done.
2. **Re-planning**
    a. Find the position *e* of the first occurrence of the new operator *o* in the new relaxed Plan *P*. This is to find out how many operators in the old plan have to be applied before the new operator might become helpful.
    b. Apply all operators from the old plan occurring before position *e* in the new plan.
    c. Identify instances of o which are applicable in the current state.
    d. If no instance of *o* is applicable, apply more operators from the old plan until an instance of *o* is applicable.
    e. Apply the new operator *o*.
    f. Identify a re-entry point in the old plan by searching for planned actions in the old plan which corresponds to helpful actions in the current state. Continue with step 2.a from this position. If no such position can be found, the remainder of the plan has to be re-planned completely.

2.2.1.1.1.2 Planned Action Becoming Unavailable

In this situation a re-planning has to be done because the plan is not valid anymore. OWLS-XPlan+ tries to replace the affected action(s) by replacing it with alternative actions which achieve the same effect. If this is possible, the remainder of the plan can be re-used, thus reducing re-planning time significantly.

1. **Re-planning decision**
    a. Mark all actions in the plan, which are affected (because the operator does no exist anymore, or because a precondition does not longer hold).
    b. If no actions are marked, stop.
2. **Re-planning**; for each affected action,
    a. Create a relaxed plan.
    b. Use enforced hill climbing search to circumvent the affected operator by applying alternative operators.
    c. Identify a re-entry point in the old plan by searching for planned actions in the old plan which corresponds to helpful actions in the current state. Continue with step 2.a from this position. If no such position can be found, the remainder of the plan has to be re-planned completely.

2.2.1.1.1.3 Goal State Change

In this situation a re-planning is necessary in the case where the new goal is not achieved by the existing plan. Otherwise, a re-planning might still be beneficial if the new goal can be achieved by a shorter plan than the existing one.

1. **Initialization**
    a. Create a relaxed plan for the new goal.
    b. Mark all actions in the existing plan, which are also contained in the new relaxed plan.
2. **Re-planning**; for each non-marked action,
    a. Use enforced hill climbing search to circumvent the action by applying alternative operators.
    b. Identify a re-entry point in the old plan by searching for planned actions in the old plan which corresponds to helpful actions in the current state. Continue with step 2.a from this position. If no such position can be found, the remainder of the plan has to be re-planned completely.

2.3 User-centered Composition of Value Added Information

Important features of the agent-based paradigm are the specialization of agents in more narrow sub-problems than the global problem and the creation of value-added services. Information Web Services, ontology frameworks, user preferences and intelligent agents offer the baseline for providing a value-added service of composition of information services. In this section we describe the design and development of an information broker agent, Fredo, reflecting the goal of providing user-centered composition of value added information.

This technology is presented as a specialization of the composition process for information services only and is not intended to substitute the described composition planning infrastructure of the CASCOM Architecture. Due to the complexity that it adds to the service providers and the communication with them, it was decided not to include this composition approach in the CASCOM demonstration scenario.

2.3.1 Fredo, the Agent

Fredo is an information broker agent specialized in finding information about specified domains, partially satisfying a set of given preferences, and in providing a value-added information service in the sense that it dynamically integrates information about several domains. Although specialized in finding and integrating information of specified domains, Fredo is a generic agent in the sense that it is not tailored for a specific set of domains. Its functioning is absolutely domain-independent. Fredo is a generic middle agent that can be sent off by other agents to do this and that, in any information domain.

Fredo may be used in several scenarios, such as the one represented in Figure 22. There, a Personal Assistant Agent (PA) sends Fredo its user preferences about restaurants. Fredo interacts with the Ontology Agent (OA) and the Service Discovery Agent (SDA) to discover information providers that can supply information regarding the preferences received from the PA. Then, Fredo queries the discovered information providers in order to gather information relevant to the specified preferences, evaluates it, sorts it and sends part of it to the PA.

In the described scenario, Fredo receives a query containing a set of preferences about possibly different, even though related objects. Since Fredo is not supposed to have information of its own, it must discover other agents that can provide the desired information. Therefore, it interacts with ontology agents[12] until it identifies those ontologies involved in the received query. It then asks the SDA for agents providing information services using the identified ontologies.

[12] In this context, ontology agents are perceived as being specialized agents that hold information about the ontologies used in an environment, which are represented in a hierarchical fashion. These agents are able to provide information on a specific ontology or elements of an ontology, such as classes, predicates and actions.

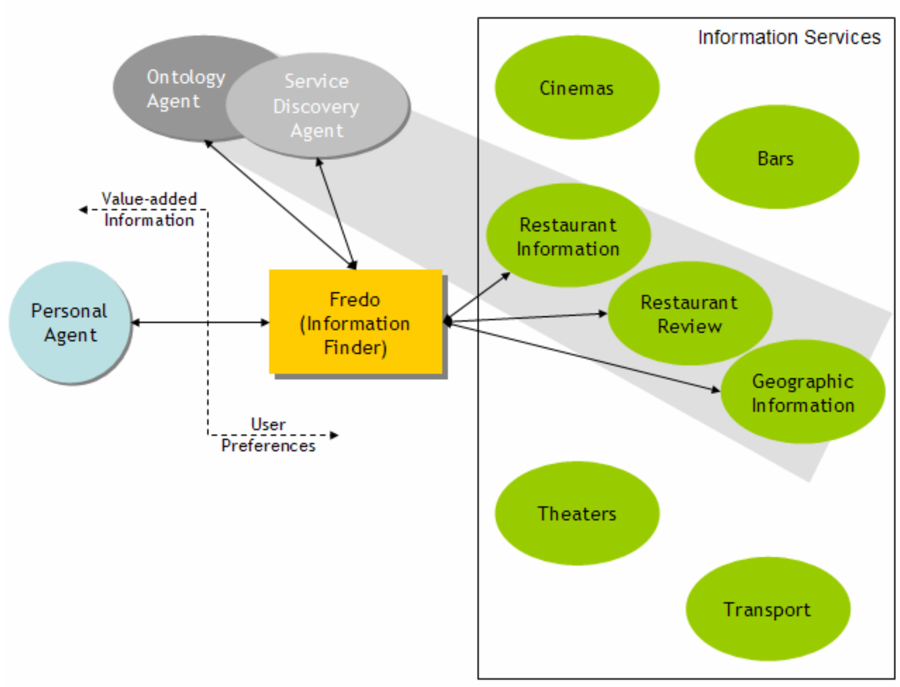


Figure 22 – Fredo Scenario Example

Fredo requests the necessary information from the selected information providers. In order to avoid information overload, Fredo uses a heuristic strategy that allows it to constrain the queries sent to the information providers. This heuristic relies on the fuzzy evaluation of the specified preferences. Furthermore, the interaction with each information provider is governed by a special purpose interaction protocol in which the reply from the information provider is sent one page at a time. If the information received until a certain point in time is good enough, Fredo can halt the interaction with the information provider.

Since, in most cases, it is impossible to satisfy all of the specified preferences, Fredo uses a fuzzy inference engine to evaluate the gathered information, with respect to the specified preferences. This fuzzy evaluation mechanism is the same used in the heuristic planning of the queries sent to the information providers. The best results are then sent to Fredo's client.

2.3.2 Information Broker Agent

Fredo's client (e.g, a personal agent) sends messages requesting objects that satisfy a given set of preferences. Figure 23 depicts the format of a typical message received by Fredo: "Give me any restaurant and its evaluation that satisfy these preferences to a degree larger than 0.8".

```
(query-ref
 :sender (agent-identifier :name pa@cascom)
 :receiver (set (agent-identifier :name fredo@cascom))
 :content "((any (sequence ?rest ?eval)
             (and
              (instance ?rest Restaurant)
              (= ?eval (value Finder Evaluation
                (sequence (set <Pref1> <Pref2> …))))
              (>?eval0.8))))"
 :language fipa-sl)
```
Figure 23 – Request received by Fredo from its client

`query-ref` is the FIPA ACL performative used for open questions. The content of the `query-ref` message must be a referential expression. `(any <Term> <Proposition>)` is a FIPA SL referential expression that refers a Term that satisfies the specified proposition. `(sequence ?rest ?eval)` is an ordered pair of a restaurant and its evaluation with respect to the specified preferences. `(instance ?rest Restaurant)` means the variable `?rest` is an instance of the class Restaurant. `(value Finder Evaluation (sequence (set <Pref1> <Pref2>…)))` represents the value returned by the invocation of the static method `Evaluation` of the class `Finder` with a single parameter sequence: the set of specified preferences. See [BAER02] for a more detailed explanation of the `instance/2` and `value/4` operators.

The method Evaluation receives a set of preferences applied to the corresponding target objects and returns the evaluation of the target objects with respect to the specified preferences.

The evaluation of the target information with respect to the specified preferences is based on a fuzzy reasoning process. The same fuzzy evaluation process is used by Fredo in the heuristic planning of the queries it sends to information providers in order to obtain the required information. The membership functions of the used fuzzy sets were empirically determined.

2.3.2.1 Representing Preferences

We decided to represent preferences as objects so that it is easy to talk about them, using a first order logic based content language. An instance of class Preference represents the preference itself and the target object, which is the object to which the preference is applied. A single preference may relate several attributes of the target object. For instance, “the value of attribute A1 of the target object should be twice the value of attribute A2 of the same object”. A preference has the following attributes: target, which is a term representing the object of the preference; targetClass, which is a word representing the class name of the target; attributes, which is a sequence of strings representing names of attributes involved in the preference; values, which is an expression that evaluates to a sequence of the constrained values of the sequence of attributes; cut-off, which is a cut-off expression (while values expresses a preference, cut-off expresses a hard constrains that must be fulfilled); and weight, which is a float in the interval [-1, 1] representing the importance of the preference, where -1 means the property is highly undesired, 0 means the client is indifferent with respect to the property, and 1 means the property is highly desired.

Attributes:		Ai		
(Constrained) Values:		Vi		
Cut-off Values:		Ci		

Vi is the constrained value of attribute Ai, and Ci is its cut-off values

Table 5 – Preference structure

The main idea, clarified in Table 5, is the following: each specified attribute must be associated with a constrained value and possibly with a cut-off value.

```
(Preference
 :target <Restaurant>
 :targetClass Restaurant
 :attributes (sequence maxPrice foodType)
 :values (any (sequence ?price ?type)
      (and (= ?type Italian)
        (<= ?price 10)))
 :cut-off (sequence
     (AttributeCutOff :similarity 0.8)
     (AttributeCutOff))
 :weight0.7)
```

Figure 24 – Preference for cheap Italian Restaurants

As shown in Figure 24, when expressing the preference for cheap Italian restaurants, the restaurant is the target object of the preference and the target class is "`Restaurant`"; attributes is `(sequence maxPrice foodType)`; values is `(any (sequence ?price ?type) (and (= ?type Italian) (<= ?price 10)))`; the cut-off value could be a sequence of cut-off specifications, each one referring to the value of the corresponding attribute `(sequence (AttributeCutOff :similarity 0.8) (AttributeCutOff))`, in which `(AttributeCutOff)` is the null cut-off specification; and weight could be the importance of the whole preference, say `0.7`.

The cut-off specification can be a sequence of instances of the class `AttributeCutOff`, a single instance of the class `GlobalCutOff`, or an instance of the class `CutOff`, which includes both the sequence of `AttributeCutOff` and the `GlobalCutOff`.

Class AttributeCutOff

similarity: Float in the interval [0, 1]

Class GlobalCutOff

evaluation: Float in the interval [0, 1]

Class CutOff

global: GlobalCutOff

attributes: AttributeCutOff

If cut-off is a sequence of class `AttributeCutOff` instances, each instance corresponds to the attribute in the same position in attributes. If cut-off is a single cut-off specification then it refers to the degree to which the complete constraint expression is true. If cut-off is an instance of the class `CutOff`, it is a global cut-off plus a sequence of cut-off specifications for all constrained attributes. In any of the above cases, each specified cut-off value is the degree to which the target object fulfills the preference expression, which is evaluated by a fuzzy reasoning mechanism.

2.3.2.2 Identifying Information Providers

Since Fredo is a broker of information services, it must consult agents/web services that provide information services in the domains specified by the received set of preferences. In the first place, Fredo must determine the domain of the received query. To accomplish this, it analyses the received preferences in order to identify the referred classes and attributes. Since each Preference is always about an object of a specified class and involves a set of attributes of that class, the agent loops over all preferences and creates a data structure called domain, which identifies all the attributes involved in the preferences for each target class:

```
domain = {(c1, {a1.1, …, a1.n}), (c2, {a2.1, …, a2.m}), (c3, {a31, …, a2.k}), …}
```

In the case of the cheap Italian restaurant, domain is represented in Prolog by the list [class('Restaurant', [maxPrice, foodType])].

Afterwards, Fredo asks ontology agents for ontologies that include each class c_i containing all attributes in $A_i=\{a_{i.1}, …, a_{i.n}\}$. The result of this question is a data structure called ontologies containing a set of ontologies associated to the set of pairs (<class>/<attribute set>):

ontologies = {(o1, {(c1.1, A1.1), … (c1.n, A1.n)}), (o2, {(c2.1, A2.1), ...(c2.m, A2.m)}), ...} , in which $A_{i.j}$ is the set of referred attributes of class $c_{i.j}$.

Often, each ontology includes more than one referred class. It is even frequent that the same ontology includes all classes referred in the received preferences. Additionally, Fredo asks the Ontology Agent what are the key attributes of each class in the set domain.

In the next step, Fredo asks the SDA for agents that provide information services using each of the ontologies in the set ontologies. In the case of the cheap Italian restaurant, the set ontologies is represented in Prolog by a list like [ontology('http://.../ontologies/restaurant.owl', class('Restaurant', [maxPrice, foodType]))]. The SDA uses this ontology information to search services through their categories. This search can also be carried out through the Directory Facilitator (DF) agent, using the parameter ontologies in the DF service description.

The answer provided by the SDA plus the information previously acquired about the key attributes of each class is organized in the set called providers. providers contains the service providers that provide the desired information associated to the relevant classes and attributes known to the agent, plus the key attributes of each class:

providers = {(a1, {(c1.1, A1.1, K1.1), … (c1.n, A1.n, K1.n)}), (a2, {(c2.1, A2.1, K2.1), ...(c2.m, A2.m, K2.m)}), ...}, in which a_i is the identification of agent i, and $K_{i.j}$ is the set of key attributes of class c_i containing attributes $A_{i.j}$. The set providers is passed on to the next stage of Fredo's processing.

The above process, by which Fredo identifies the necessary information providers, is absolutely generic. There is nothing that depends on a specific domain. Furthermore, the received set of preferences can refer to several ontologies possibly provided by several agents. Therefore, Fredo is a generic agent (i.e., domain independent) that provides a value-added information service since it may integrate information services using several ontologies provided by different information providers.

2.3.2.3 Querying the Information Providers

There are three important issues that deserve mentioning about the way Fredo interacts with the information providers to acquire the desired information: (i) what information is asked from each provider; (ii) how to create questions that constrain the amount of information sent in the reply; and (iii) the interaction protocol.

Using the set providers, Fredo knows what information it can request from each provider. It will ask only for the set of attributes referred in the received preferences plus the set of attributes that form the key for that class. We have assumed that preferences are not rigid constraints. A given target object might be poor with respect to one of the specified preferences but it may be good with respect to others. Therefore, Fredo cannot just request the instances of the desired classes that satisfy all specified preferences that apply to them. That would be too rigid.

Fredo cannot also ask for the values of the selected attributes for all possible instances of each target class. That would result in large amounts of information being exchanged requiring considerable processing. This trade-off between the desire to evaluate all instances with respect to all applicable preferences, and the desire to avoid information overload lead to the definition of a heuristic strategy that is used to constrain the amount of information requested from the information providers without ignoring instances that would likely be well evaluated.

The first step towards this goal is to use the cut-off values that are specified within the preferences. Since Fredo's clients are not interested in receiving information that does not satisfy the specified cut-off criteria, there is no point in asking information providers to supply information that does not meet those same cut-off criteria. This means the query to be sent to the information providers must contain constraining clauses that exclude information that does not meet the cut-off criteria.

However, the cut-off criteria are not mandatory and therefore they may be omitted in the specification received by Fredo. For this reason, we devised a heuristic decision rule to create a cut-off criterion for each attribute involved in the preferences that has not been attached to one.

The general heuristic is the following: if some desired property is very important when compared to the other desired properties, then we are interested only in instances of the specified class that satisfy well the specified preference. If the preference is not important, we are less restrictive – we may even be interested in instances that are not very good with respect to that preference. For negative preferences, similar criteria apply mutatis mutandis.

The above principle is used by Fredo to create constraining clauses that will be used in the queries it sends to the providers. The queries sent to the information providers request information about the instances of a given class that satisfy the cut-off criteria, either explicitly specified or heuristically created by Fredo.

Fredo uses a fuzzy approach to determine the heuristic cut-off value of an attribute. The desired value of an attribute of the target object as specified in a given preference is construed as a fuzzy set. Within this framework, the degree to which the actual value of the target object attribute matches the preference is determined by the membership function of the fuzzy set representing that preference. For instance, the preference of a restaurant with `maxPrice` less than 10 € may be represented by the fuzzy set shown in Figure 25. With a `maxPrice` of 12 €, the degree to which it matches this particular preference is 0.5.

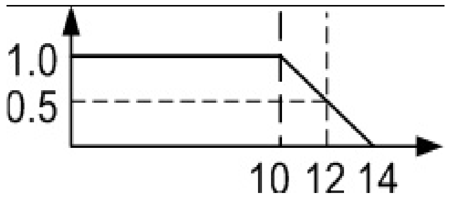


Figure 25 – Fuzzy lessThan(10)

Suppose that, given the specified preference weight, Fredo wants to exclude all restaurants whose price matches the preference to a degree of 0.75 or less. Fredo uses the fuzzy set membership function to determine the prices that belong to the desired set with a degree of 0.75 or less. Then, it creates a logical expression that excludes (restaurants having) those prices. This logical expression is included in the query sent by Fredo to the restaurant information service in order to constrain the amount of returned information. In this case, that logical constraint would exclude all restaurants with price greater then 11 € (i.e., prices that belong to the desired fuzzy set with a degree equal to or less than 0.75):

```
(and (instance ?rest Restaurant) (≤ (value ?rest maxPrice) 11))
```

According to the heuristic already explained, the cut-off value for an attribute depends on the weight of the preference defined for that attribute. The higher the weight, the higher the heuristic cut-off value should be. That is, the target object must meet a given preference to the degree to which the preference is important. (`E1`) is the empirically determined expression used by Fredo to compute the membership degree of the cut-off value (`d`) of an attribute for a preference with weight `w`.

$$(E1)\ d = 3/4 \times |w|$$

As can be seen, this empiric expression is general, in the sense that it does not depend on the particular fuzzy set being used to represent the preference.

Notice that Fredo does not assume that information providers have fuzzy inference engines. Fredo uses its fuzzy evaluation engine to compute the applicable cut-off criteria. Then, it uses the computed criteria in the queries it sends to information providers.

The fuzzy sets used in the heuristic planning of the queries to be sent to information providers are the same used in the evaluation of the received information with respect to the specified preferences.

Even though Fredo uses explicitly stated or heuristically determined cut-off values to limit the amount of information asked to the providers, there is still the possibility of information overload. Therefore the interaction between Fredo and providers must be governed by a special purpose interaction protocol aimed at dealing with large amounts of information being returned by information providers. In the remaining of this section, we describe the paged information-request protocol, according to which, information providers send the requested information, one page at a time. Using the paged information-request protocol, Fredo can interrupt the process at any moment in time if it is satisfied with the information already received.

In the paged information-request protocol (Figure 26), the initiator (Fredo) sends a query to the participant (the provider) containing also the specification of the page size, that is, the number of items that may be received in each page. The page size is specified by the user parameter `Xreply-page-length` of the ACL message (ACL user parameters start with an X). This query is called the information request. After receiving the query, the participant has three alternatives: it does not understand the query, it refuses to provide the requested information one page at a time, or it agrees to reply one page at a time. If the participant does not understand or if it refuses, the protocol halts. If the participant agrees, the protocol internal state changes to the agreed state.

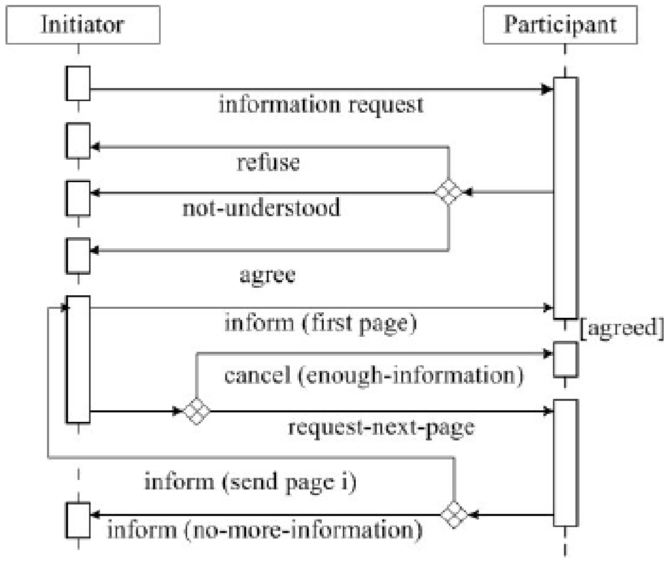


Figure 26 – Paged Information-Request Protocol

In the agreed state, the participant must send the first page of the reply to the initiator (Fredo). When the initiator receives one page of the requested information, it may request the participant to stop sending more pages of the requested information, or request the next page. If the initiator requests the participant to stop sending information, the protocol halts. If the participant exhausts the information to be sent it sends a message informing the initiator that no more information is available.

Any instance of the paged information-request protocol forms a context in which, each of the several pages of the information sent by the participant (provider) to the initiator (requester) may be associated to a sequence number. In this context, the page number has a precise meaning: it is the order by which the page is sent to the initiator. The sender alone exclusively determines the sequencing number of each page. There is no way the requester can reasonably assume any thing about the specific content associated to a given page number. However, the requester may assume that the sender preserves the coherence among page numbers. That is, the same page is not sent twice with different page numbers. And no two different pages are associated with the same page number. As soon as the protocol ends, the page numbers associated to the several sent / received pages loose their meaning. That is, if the protocol was executed again, no one would be entitled to believe that the same information would be paged in the same sequence as in another instance of the protocol.

Given the context mechanism provided by the paged information-request protocol, it is possible to clearly define the operator `page/2`, such that `(page N InformationDescription)` means the Nth page of the information referred to by `InformationDescription` within a given instance of the protocol.

The mechanism described in this section allows Fredo to create the queries it must send to the identified information providers, trying to constrain the amount of returned information to a minimum without discarding possibly relevant information. This mechanism is a generic mechanism in the sense that none of its stages depends on any specific domain.

2.3.2.4 Evaluating Gathered Information

A single preference specifies a preferred condition and its importance. Other components of preferences such as the cut-off values were omitted because they are irrelevant to the present explanation. If the agent can determine the degree to which the preferred condition is satisfied by the target object, than it is easy to evaluate a given object with respect to a set of preferences $P=\{(\varphi_1, w_1), \ldots, (\varphi_n, w_n)\}$, in which φ_i is the preferred condition specified in preference `i`, and w_i is the importance of preference `i`.

Let the closed proposition φx denote the application of the preferred condition φ to the object `x`, and let also $\mu(\Psi)$ denote the degree to which the closed proposition Ψ is satisfied. The evaluation of object `x` with respect to the set of preferences $P=\{(\varphi_1, w_1), \ldots, (\varphi_n, w_n)\}$ is given by $\varepsilon(x) = (\Sigma i\ w_i \times \mu(\varphi_i x))/(\Sigma i\ |w_i|)$ for all preferences in P.

If the closed sentence Ψ is a proposition of the first order predicate calculus, $\mu(\Psi)=1$ iff Ψ is true, and $\mu(\Psi)=0$ iff Ψ is false. If Ψ is a fuzzy proposition, then $\mu(\Psi)$ is the truth-value of Ψ. $\mu(\Psi)$ takes values in the interval [0, 1].

Let us now describe the way the preferred condition is applied to the target object of the preference exemplified in Figure 24. The application of the preferred condition to the specified target restaurant results in the closed proposition `Prop ≡ (and (= (value <Restaurant> foodType) Italian) (<= (value <Restaurant> maxPrice) 10))` in which `<Restaurant>` is a specific restaurant. This application is a simple process if we remember that the variable `?price` (i.e., the first variable) in values corresponds to the attribute `maxPrice` (i.e., the first attribute) in attributes, and the variable `?type` (i.e., the second variable) in values corresponds to the attribute `foodType` (i.e., the second attribute) in attributes.

The degree to which the preferred condition is satisfied by the target object is the truth-value of Prop (considering that `<Restaurant>` is a concrete restaurant), for which we propose to use a fuzzy evaluation mechanism.

For using the fuzzy evaluation mechanism we have to convert propositions such as `(and (= (value <Restaurant> foodType) Italian) (≤ (value <Restaurant> maxPrice) 10))` into fuzzy propositions. In this case we end up with [`value(<Restaurant>, foodType`) is `ItalianFood`] $\wedge$ [`value(<Restaurant>, maxPrice`) is `lessThan(10)`], in which `value(<Restaurant>, foodType)` and `value(<Restaurant>, maxPrice)` play the role of fuzzy variables, and `ItalianFood` and `lessThan(10)` are fuzzy values.

It is necessary to define the universes and membership functions of the fuzzy values `ItalianFood` and `lessThan(10)`. `lessThan(N)` represents a family of fuzzy values whose membership functions are shifted versions of one another.

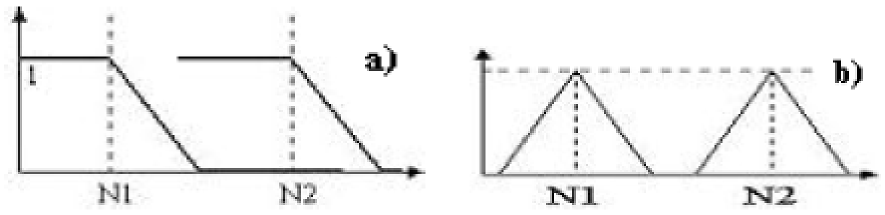


Figure 27 – Membership functions: a) lessThan(N) b) equalTo(N)

For each value of `N`, `lessThan(N)` is a fuzzy value. An alternative solution could be used, where the slope of the `lessThan(N)` membership function would be less pronounced for larger values of `N` in order to reflect the fact that 1010 is as less than 1000, as 101 is less than 100. Each of these alternatives is more adequate for specific domains. Other fuzzy values could be defined, such as `greaterThan(N)`, which would be defined in the same way as `lessThan(N)`, and `equalTo(N)`, which would have a representation as shown in Figure 27 b). A possible alternative would be to use Gaussian instead of triangular membership functions.

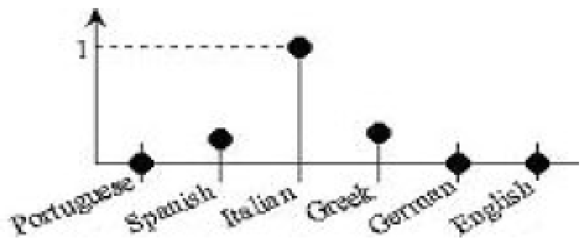


Figure 28 – Fuzzy value Italian Food

For the fuzzy value `ItalianFood` we assumed that the universe of discourse is formed by the set of all food types in the domain (e.g., Italian, Portuguese, Greek, Spanish, French, German). Maybe Greek and Spanish food can be considered more similar to Italian than other food types (Figure 28). In order to use this approach, for each food type, we must define the degree to which the other food types are similar to it.

Certainly, a finer grained fuzzy value could be defined, for instance one in which the universe of discourse would contain several food types for the same country, such as food from Toscana, Neapolitan food, Milanese food and so on.

One problem of using a fuzzy evaluation is to preserve the generality of the approach. In general, the definition of fuzzy concepts such as “Italian food” is dependent of the domain. Most often these concepts even depend on the point of view of the evaluator. One way to go about this would be to include the definition of fuzzy concepts in the domain ontology. However, no work has yet been done about the representation of fuzzy values in commonly used formalisms for ontology representation such as OWL.

Another alternative is to put the responsibility of maintaining these domain specific fuzzy sets on the side of the Personal Agent (PA). This alternative is more realistic and even more appropriate since such subjective concepts as being similar to “Italian food” should be tailored to each user. PAs can easily build such fuzzy sets through simple statistical analysis of their user preferences and feed back. Currently the required membership functions are kept in Fredo, but this is only a temporary solution.

3 Service Execution in IP2P Environments

This task develops a reliable service execution platform which takes into account both single and composite services. This includes augmenting the planned service composition with monitoring assertions that can be checked to verify successful execution of services, and in the case of failure, will allow necessary re-planning.

As input, service execution takes service descriptions represented by the Web Ontology Language for Services (OWL-S) and a set of actual input values which are associated to the services input parameters. On successful execution a set of output values (results) will be returned to the invoker whereby the values are associated to the output parameters in the service description. Both input and output sets might be empty—in this case invoked service(s) produce just side effects. Furthermore, execution requires grounding for each service, that is, a reference (address) to a concrete service endpoint (instance) and information about the protocol to be used to interact with the service instance. We use OWL-S in combination with WSDL to specify the grounding for services (see section 3.1).

Two approaches were developed for delivering the service execution functionality in the CASCOM environment: a distributed approach (see Section 3.2), where different agents in the environment, well coordinated and co-operating, can contribute to the execution of parts of a composite service; and a centralized approach (see Section 3.3), where a single specialized agent can execute an entire composite service.

Even though the term “Service Execution Agent” is used to represent the agents capable of executing services in both approaches, it does not denote the same agent in different approaches.

In the distributed approach, a Service Execution Agent is an agent that, after being added a specific behaviour, is capable of contributing to parts of the execution of a composite service. In spite of being able to execute a complete composite service, the distributed approach’s Service Execution Agent cooperates with other Service Execution Agents to avoid scalability and robustness problems.

In the centralized approach, a Service Execution Agent is not a normal agent that was extended to be able to contribute to the execution of parts of a composite service. It is a specialized agent capable of executing any composite service described in OWL-S. This specialized agent should be used by other agents that do not have any execution capabilities (such as the Personal Agent) and yet need to execute some specific composite service.

3.1 Executing OWL-S/WSDL Services

OWL-S is an OWL-based (Web Ontology Language) ontology used to describe semantic web services. OWL-S Services are described in three parts: a *Profile* (which tells "what the service does"); a *Process Model* (which tells "how the service works"); and a *Grounding* (which tells "how to access the service"). The Profile and Process Model are considered to be *abstract* specifications, in the sense that they do not specify the details of particular message formats, protocols, and network addresses by which a Web service is instantiated. This role of providing more concrete details belongs to the grounding part. WSDL (Web Service Description Language) provides a well-developed means of specifying these kinds of details. For the execution process, the most relevant parts of an OWL-S service description are the *Process Model* and the *Grounding*. The *Profile* part is more relevant for discovery, matchmaking and composition processes, hence no further details will be provided in this section.

Process Model. The *Process Model* (or *Service Model*) describes the steps that should be done for a successful execution of the service. These steps represent two different views of the process: first, a process produces a data transformation of the set of given inputs into the set of produced outputs; second, a process produces a transition in the world from one state to another. This transition is described by the pre-conditions and effects of the process [OWLS03]. The *Process Model* identifies three types of processes: *atomic*, *simple*, and *composite*. *Atomic* processes are directly invokable (by passing them the appropriate messages). Atomic processes have no sub-processes, and can be executed in a single step, from the perspective of the service requester. *Simple* processes are not invokable and are not associated with a grounding description, but, like atomic processes, they *are* conceived of as having single-step executions. *Composite* processes are decomposable into other (non-composite or composite) processes. These represent several steps of executions, which can be described using different control constructs, such as sequence (representing a sequence of steps) or If-Then-Else (representing conditioned steps).

Grounding. The *Grounding* specifies the details of how to access the service. These details mainly include protocol and message formats, serialization, transport, and addresses of the service provider. The central function of an OWL-S Grounding is to show how the abstract inputs and outputs of an atomic process are to be concretely realized as messages, which carry those inputs and outputs in some specific format. The *Grounding* can be extended to represent specific communication capabilities, protocols or messages. WSDL and *AgentGrounding* (described in the next sub-section) are two possible extensions. WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate [CCMW01].

Execution process. The execution of an OWL-S service is made in the following steps:

1. Validation of the service's pre-conditions. The execution process continues only if all preconditions are true;
2. If the service to be executed is a composite service, it must be recursively decomposed into individual atomic services, which are executed by evoking their service providers. Invocation is done through the description of the service providers (and their interfaces) contained in the grounding section of the service description (WSDL or *AgentGrounding – see section 3.3.1.4*);
3. After execution, validation of the service's effects is made by comparing them with the actual service execution results (if the service was executed as expected in the effects, then proceed);
4. Collect the results (if any – the service may be only a "*change-the-world*" kind of service).

3.2 Distributed Approach

The distributed approach for service execution differs from the centralised approach (see Section 3.1) in the sense that at runtime execution is not limited to take place at just one software entity but might involve several distinct entities. Since functionality of the execution system is encapsulated by agents the term distinct software entity referrers to distinct agent instances. Whether they are actually physically separated, i.e., run on different hosts[13], is a matter of the agent platform used and the concrete deployment of agent instances. In other words, the distributed approach imposes no constraints on the organisation of agents beyond what is implicated by the agent platform used.

Distributed execution requires a certain strategy to organise and co-ordinate distribution. Several approaches exist and we will describe our approach used in Section 3.2.1. We begin by describing the general software structure of the execution agent implementation in Section 3.2.1 and the agent interface for ACL messages exchange in Section 3.2.3.

[13] on the other hand, not physically separated would mean that they run on one host within different threads or processes

3.2.1 Execution Strategy

Carring out execution of composite services in a distributed environment consisting of more than one execution agent requires the definition of a certain strategy. The strategy basically defines tree properties. First, it defines how to divide the composite service into sections which can be executed in a distributed way afterwards. Second, it defines where and when to distribute those sections to different execution agents. Finally, it defines a mechanism for control between the participating agents to guarantee consistent execution. Altogether they imply the way execution will be actually done at runtime.

There are various possibilities to define such strategies all of them contain advantages as well as disadvantages when being applied to different kinds of service environments, i.e., their usefulness strongly depends on the setting and characteristics of the environment. Consequently, selection or design of a certain execution strategy should always be accompanied by an analysis of the service environment, in particular the technologies used (like SOAP based Web service interactions) and the deployment structure of service providers and their (physical) relation to service clients (agents in the CASCOM architecture). In the following, we will shortly describe three different dimensions for categorisation of execution strategies. In Section 3.2.1.1 and 3.2.1.2 we describe two general-purpose approaches which can be further optimised for specific environments. The latter one was choosen for implementation of the distributed service execution agent and we will outline some advantages and disadvantages.

In general, a strategy for distributed execution of composite services must initially define whether execution control takes place in a centralised or decentralised way. The centralised approach requires a dedicated agent who manages and co-ordinates execution after receipt of the composite service and co-ordinates execution. In doing so, the manager agent invokes according to the structure of the composite service the actual execution agents which in turn invoke the actual services and awaits the results of the service invocations from the agents. In contrast to the centralised approach the decentralised approach does not require a dedicated agent. Whereas the manager agent in the centralised approach turns out to become a bottleneck and a single point of failure both problems are eliminated in the decentralised approach. On the other hand, a decentralised approach comes with the tradeoff that more advanced control mechanisms are required, accompanied by higher (communication) efforts in case of failure handling and initial preparation before execution (see Section 3.2.1.1). In addition, the decentralized approach allows for balancing the load among different agents and for taking into account the dynamics of a system where agents might leave or others join.

Starting from the process model/structure of OWL-S services, execution can be further classified depending on whether the services are atomic or composite (see Section 3.1). Whereas execution of an atomic OWL-S service implicates just one service invocation, the number of service invocations for composite OWL-S services intuitively relates to the number of atomic services out of which they are *composed*. In general, we always refer to composite services since execution of atomic services in fact does not require an execution strategy. It can not be considered useful even possible to distribute a service consisting of exactly one service invocation between different agents. The reason is that invocation of a (Web) service is defined as a request-reply pattern between a service client and the provider and cannot be further split up. As a conclusion, it is now obvious what the smallest granularity for sections of a composite service is: the atomic services. At the same time it is also the preferred granularity as it directly maps to the OWL-S process model[14]. Hence, it is not remarkable that most execution strategies would size the sections for distribution equal to the atomic services. Finally, the general assumption is that agents execute those sections of the composite service. For instance, the most straightforward approach would associate each atomic service within a composite service to one execution agent which invokes it, i.e., a composite service consisting of *n* atomic services would require *n* execution agents.

[14] The process model basically relates to a directed graph, whereby the nodes are (atomic) services and vertices represent the control flow.

Yet another dimension for categorisation of execution strategies relates to whether execution agents and (Web) service providers are tightly integrated/coupled or not. By tightly integrated we mean that an execution agent basically wraps one or more services, that is, both run on the same machine probably in the same process – or virtual machine in case of Java. In addition, the implementation of the invocation of the Web service by the wrapper agent itself might bypass the whole Web services communication stack and might be done without this overhead by direct programmatic calls. Furthermore, this type mostly comes along with the intention that an execution agent wraps a fixed set of services, i.e., the set of services is expected to never change. As opposed to tight integration, remote coupling means that the execution agent and the (Web) service provider are distant from each other, that is, the execution agent invokes the service in the standard way using its communication layer and protocol – for instance, the common Web service stack, that is HTTP based SOAP or REST. Finally, this type mostly comes along with a generic design of the execution agent which is able to invoke any remote service.

3.2.1.1 Fully-Decentralised Execution Strategy with Tight Integration of Execution Agent and Web Services

The determinative characteristic of this strategy is the assumption that execution agents and (Web) service providers are tightly or locally integrated, as described above. Consequently, execution agents need to be deployed and made available for every pre-existing service provider in the infrastructure.

Figure 29 illustrates the course of a very simple scenario where a composite service containing a sequence of three Web service invocations WS1 to WS3 should be executed. The client agent – according to the setting in the CASCOM architecture this would actually be the SCPA – submits the valid OWL-S composite service description together with the input data to any available execution agent, for instance SEA_0. This execution agent parses the given service description for the first atomic service provider, which is WS1. With this information the agent resolves the execution agent SEA_1 as it is the wrapper of WS1 – note that we assume the availability of some kind of directory containing a mapping from service providers to execution agents. In the next step SEA_0 forwards the complete service description and input data to SEA_1 co-requesting execution start. By forwarding the complete service description each agent is able to resolve the next agent to forward control and results to because this information can be resolved from the process model of the service description. In the scenario, after completion of execution of WS1, SEA_1 would resolve the agent for WS2 which is SEA_2 and again forward the complete service description together with the input data and result 1 produced by WS1. This procedure continues until SEA_3 has finished invocation of WS3 with result 3. In step 5 SEA_3 sends results 1 to 3 back to SEA_0 which just maps them to the composite service’s output (result). Finally, in step 6 the result is sent back to the client agent.

In case of failures on one execution agent or in case of violated pre- or post-conditions execution either has to be rolled back or a re-planning could be initiated trying to find alternative services and continue execution if an alternative was found. Assuming transactional properties of the services a roll back can be done based on the fact that each execution agent knows its adjacent predecessor(s) from the process model of the composite service. A rollback then requires moving back along the control path step by step and rolling back each service invocation done.

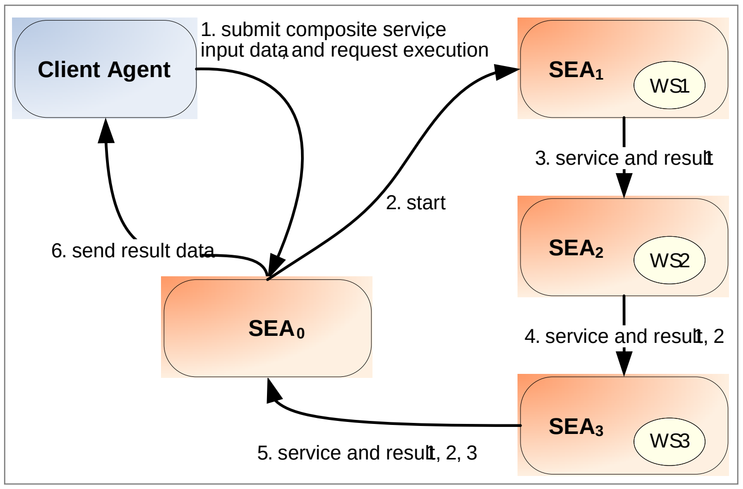

Client Agent
1. submit composite service input data and request execution
SEA$_1$
WS1
3. service and result 1
2. start
SEA$_2$
WS2
6. send result data
SEA$_0$
4. service and result 1, 2
SEA$_3$
WS3
5. service and result 1, 2, 3

Figure 29 – Course of fully-decentralised execution strategy with tight integration of SEA and service provider

The downside of the simple variant of this approach is that intermediate results produced by service invocations and input data will be forwarded in any case no matter if they are actually required on every execution agent, i.e., the data flow is not optimal. For instance, in the scenario illustrated in Figure 29, if WS1 produces large volumes of data as result 1 they will be forwarded to SEA$_2$ and SEA$_3$ in any case. In a first optimised version data (input and output data, intermediate results) would only be forwarded between the agents where it is required, thus reducing the overall data communication amount. Furthermore, it is not necessary to forward the composite OWL-S service description from agent to agent. In a second optimisation step this can be optimised to split the composite service into its atomic services sections and extend the execution strategy with an initial distribution of the sections to each agent, i.e., each execution agent receives just its own task within the composite service. In the scenario above this initial step could be done by SEA$_0$. To complete this optimisation each agent also needs knowledge about its adjacent predecessor(s) and successor(s) for control flow navigation. The successor(s) is/are required for normal forward navigation whereas the predecessor(s) is/are required for backwards navigation in case of roll back.

3.2.1.2 Dynamically-Decentralised Execution Strategy with Remote Coupling of Execution Agent and Web Services

This approach was designed to address the development in current service oriented architectures in which the spreading and application of Web service standards like WSDL and SOAP have reached a level where Web service providers almost always apply those technologies to publish and provide access to their Web services. As a matter of fact, the question had to be raised whether a tight coupling of execution agents and Web services is appropriate, assumable, even feasible in practice or whether a strategy can be found which fits well to the the current exploitation of Web services which considers remote invocation of services using the common Web services communication stack and which is still robust against failures. All in all, this approach supports the requirement that service providers want to remain Web services conform with respect to the service interfaces.

Figure 30 illustrates the course of the quasi-decentralised and distributed execution strategy. Likewise Figure 29 it also assumes a very simple scenario where a composite service containing a sequence of three Web service invocations WS1 to WS3 should be executed. It is also assumed that a composite service is split up into sections equal to the atomic services inside. The client agent submits a valid OWL-S composite service description together with the input data to one available execution agent, SEA$_1$ in this case. After that SEA$_1$ parses the service description and immediately starts invocation of the first Web service WS1. After return of the result from WS1 the the first section is finished and the strategy now continues with a decision step to determine whether execution of the next section should be made by the same agent or whether the execution state should be transferred to another execution agent and continue there. In this decision step arbitrary heuristics can be used to figure out how to continue, i.e., on which agent to continue. For instance, it would be possible to design context based heuristics making it possible to adapt dynamically to overloaded, slow, expensive, or failure-prone context situations and transfer execution state to other execution agents with superior context situations. Additionally, it is also possible to face the effort for transferring execution state to another execution agent with the benefit available there. However, the design of this heuristics based continuation decision is generic and allows configuring a custom implementation for the execution agent. For the prototype we have implemented a very basic CPU load based heuristics, that is, transfer of the execution state takes place if the average CPU load on the current execution agent exceeded a certain threshold in a preceding timeframe. To come back to the sample scenario in Figure 30, after the decision step execution of the next section might continue on SEA_1 until WS3 returned its result. Finally, in step 5 the composite service output (result) is returned to the client agent.

The heuristics based approach for distribution is able to cope with external failure situations but not with crash situations of the current execution agent itself. To overcome this problem, two possibilities exist. It is either possible to persist the current execution state at the current agent, or to replicate the execution state online to another execution agent. Whereas the first solution is only appropriate for short interruptions whereby the agent gets rebooted immediately afterwards and restarts the interrupted execution, the second solution is more robust since execution can be immediately overtaken by the agent which has the replicated execution state. Therefore we intend to extend the implementation of the execution agent with the latter solution.

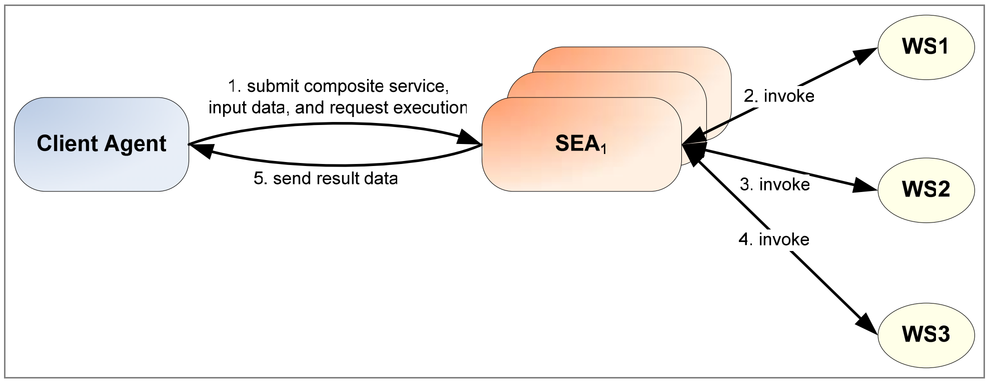


Figure 30 – Dynamically-decentralised execution strategy with remote coupling of execution agent and Web services

Provided that the heuristics based decision computation always returns with the result to continue at the same execution agent, execution is in fact not distributed and the communication effort is minimal. According to the actual heuristic used, this situation represents the optimal execution environment then. On the opposite side, transfer to another agent on each decision computation represents the most suboptimal execution environment according to the heuristic and also involves the highest communication overhead. Therefore it turns out that the more precarious the context environment is the more efforts must be undertaken.

Another characteristic of this strategy is that its properties do not deteriorate assuming that (Web) service providers would be tightly integrated with agents. Assuming this change the execution agent which currently does invocations should then communicate to other agents using ACL messages instead of using the Web services stack. Consequently, only the message layer for service invocations needs to be replaced/extended by a communication layer for ACL messages.

3.2.2 General Structure

The distributed execution system has an agent based structure. All its functionality is represented to the outside by agents, whereby for a minimum deployment just one agent is sufficient (of course, such a deployment is in fact not distributed). Furthermore, all execution agents in a distributed deployment are equal in their functionality accessible by client agents, which means that all of them are available to be used in the same way, thus presenting a true P2P structure.

As illustrated in Figure 31, the service execution agent (SEA) is represented first by the class `OnAgent` and secondly by the behaviour `OnBehaviour`[15]. The `OnAgent` has no functionality implemented except to register itself to the JADE directory facilitator (DF) and to instantiate and add the `OnBehaviour` to itself. This means that the `OnBehaviour` comprises the execution functionality completely. This way, a great flexibility is achieved with respect to who can implement OWL-S service execution functionality: The class `OnBehaviour` can be added to any agent, thus extending the agent to which it is added with OWL-S service execution functionality.

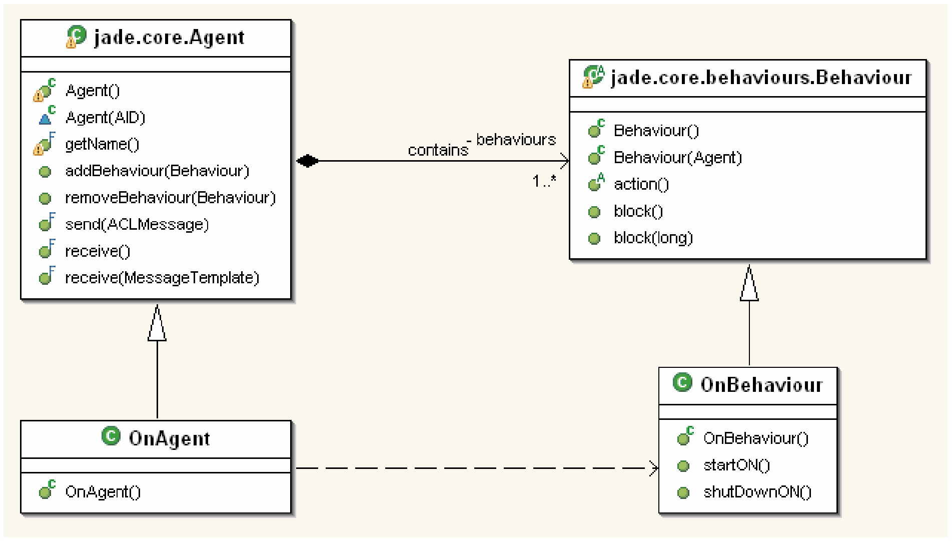


Figure 31 – Execution Agent – General Class structure

Figure 32 shows the dependencies and how the On system is integrated into the `OnBehaviour`, i.e., the execution agent. Basically, the lifecycle of the On system is managed by the class `BasicController` which contains methods for startup, reboot and shutdown of the system. After the `OnAgent` has instantiated a new `OnBehaviour` in its `setup()` method it calls `startOn()`, which simply delegates the call to `BasicController.init()` followed by `BasicController.start()`. On successful start up of the On System, the agent adds the behaviour instance to itself and is then able to accept execution requests. In case start up fails the agent won't add the behaviour to itself and deactivates itself. In case the agent was requested to shut down externally or by the agent platform its `takeDown()` method is called which delegates the call to `OnBehaviour.shutDownON()` which in turn delegates to `BasicController.shutdown()`.

Receiving and sending of ACL messages is not done in the usual way by using the `receive()` and `send()` methods of the agent since this is handled by the `OnBehaviour` implementation, which extends an interaction protocol, see section 3.2.3.1.

[15] The prefix „On" stands for *OSIRIS next*, for further information see Section 3.2.1.

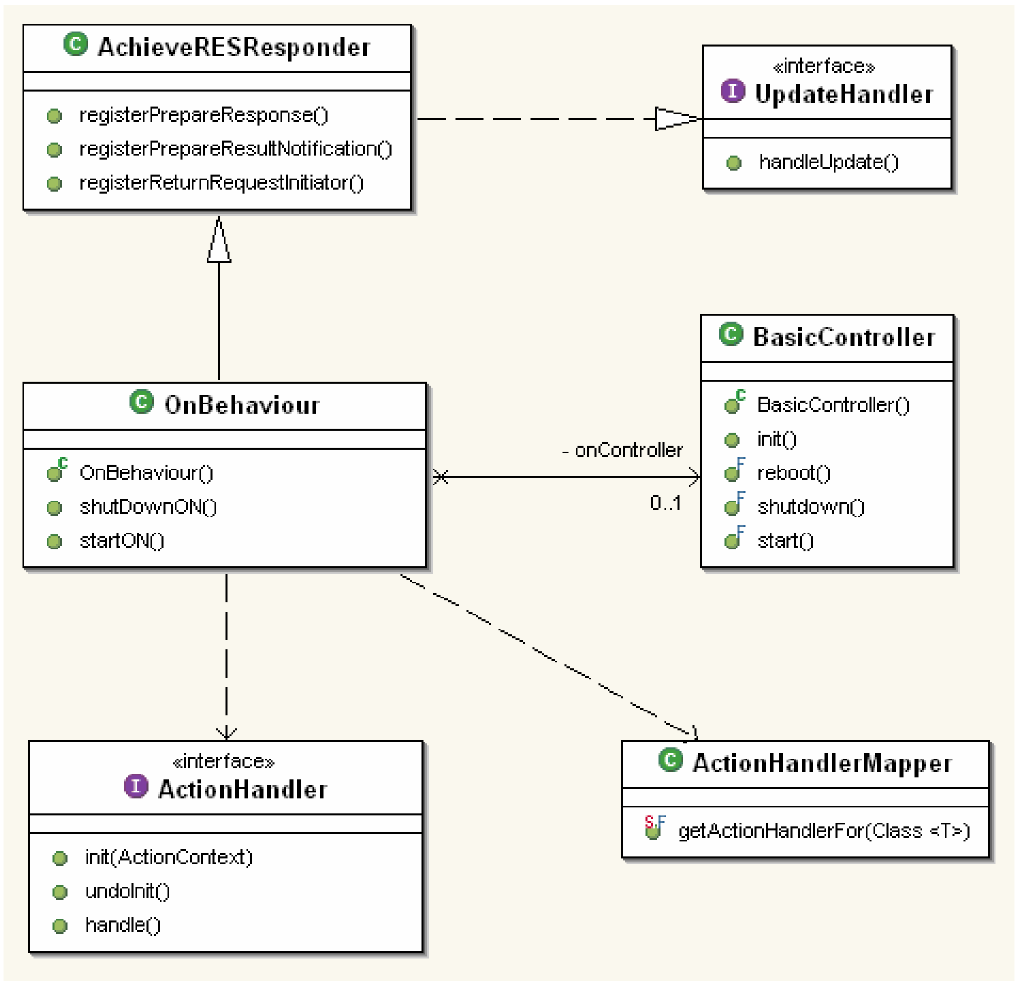


Figure 32 – On Execution System to Agent integration – Class structure

Figure 32 also displays the interface `ActionHandler` and the class `ActionHandlerMapper`. These two represent the link from a received ACL message which contains a `AgentAction` (see Section 3.2.3.2) to associated action commands which encapsulate the logic associated to a certain action. `ActionHandlerMapper` contains the mapping for each `AgentAction` to exactly one `ActionHandler`. For each arriving request message the mapper resolves an implementation of `ActionHandler` and instantiates it. Consequently, for each `AgentAction` which the execution agent is supposed to understand and handle one action handler has to be implemented. After instantiation each `ActionHandler` might need to do several initialisation steps in order to execute the actual action logic afterwards. This has to be done inside the `init()` method. In order to signal initialisation failures, the implementor might throw an exception in which case the agent refuses the request message. To undo side effects which might have been created during a failed initialisation the implementor can implement `undoInit()`, i.e., this method is called whenever `init()` returned prematurely with an exception. Finally, the implementor has to implement `handle()` which contains the actual action logic. The method neither has a return value nor does it throw an exception. This means that it forces an asynchronous semantic for action handling and thus decouples the actual execution logic from the agent thread. This was necessary since the JADE framework carries out a round-robin non-preemptive scheduling policy[16] among all behaviours attached to an agent and thus ensures that `OnBehaviour` does not interfere with other behaviours since it releases control at the time handling of an action starts. So from the point on of invoking `handle()` a new thread has to be spawn and any further internal progress is done based on the notify/wait pattern realised by the specific protocol implementation, see Section 3.2.3.1.

3.2.3 Interaction Interface

SEA publishes one interaction interface by which all ACL message interactions are to be done. As the SEA currently does not show any proactive behavior, i.e., it has to be accessed actively by other agents, the type of supported interactions are similar to asynchronous remote procedure calls (RPC). Not just because of the asynchronous invocation model but especially because of the required usage patterns which are beyond the simple request/reply pattern, all interactions between calling agent and execution agent are stateful, thus forming an interaction protocol. Each new invocation of one of the execution agents methods implicitly creates a new session which lasts until the final result – no matter if the result is positive or negative – was sent back to the invoking agent. The state on both sides is encapsulated by finite state automata as provided by the JADE framework. As a recommended starting point for simple request/reply agent conversations FIPA has specified the standard ***Achieve Rational Effect*** protocol [FIPA00a], implemented in JADE by the Java classes pair `jade.proto.AchieveREInitiator` and `jade.proto.AchieveREResponder`. However, since the protocol is not sufficient with respect to the requirements of service execution interaction, see Deliverable 3.2, an extended version of the protocol, named ***Achieve Rational Effect **** protocol, was implemented. For instance, when some agent requests execution of some composite OWL-S service to the execution agent, it was specified that the requestor might want to get notified about execution progress, that is, the position of control flow within the composite service, respectively the effects achieved so far.

In the following section we will describe the protocol and the implementation in detail.

3.2.3.1 Achieve Rational Effect * Protocol

In short, the Achieve Rational Effect * protocol extends the standard FIPA Achieve Rational Effect protocol [FIPA00a] with two optional features:

1. The possibility to send any kind of intermediate or feedback ACL messages (information) to the initiator before the final result (inform or failure) is sent to the initiator. Consecutive messages of this kind can be sent, but may alternate with 2.)
2. The possibility to send return requests (ACL messages) back to the initiator to “ask” for additional information which might be required to achieve the original rational effect. A return request must be answered by the initiator until a new return request can be done or a new feedback message can be sent.

Both parts are designed and implemented generically to be (re)used in any situation which would fit to the functionality provided. Consequently, the protocol and the implementation are not limited to be used within CASCOM and can be delivered back to the FIPA, JADE community. However, for the interfacing with the execution agent the first part is used to provide the requesting agent with up to date information about the current state of the execution. The second part is used to trigger re-planning of composite OWL-S services in case of problems during execution, for instance, if a certain atomic service part of the composite service became suddenly unavailable.

Whereas the standard FIPA Achieve Rational Effect protocol supports 1:N conversations, i.e., can handle several responders at the same time, the extended version has been tested only for 1:1

[16] In a non-preemptive scheduling policy multiple tasks (behaviours) execute by voluntarily ceding control to other tasks at programmer-defined points within each task, i.e., programmers control when to return from their task.

conversations, although its implementation was not restricted with respect to the number of responders.

Figure 33 shows the defined order and cardinalities of the ACL message flow for the standard Achieve Rational Effect protocol and the newly integrated ACL messages, altogether comprising the Achieve Rational Effect * protocol. As it can be seen, the protocol starts with the mandatory ACL message from the initiator to the responder, whereby the performative is set to request, followed by the optional ACL message with performative set to either refuse, not understood, or agree. After that zero to n propagate or return request messages can be sent to the initiator. In case a return request message is sent it must be answered with inform or failure message by the initiator. Each return request is encapsulated by a sub Achieve Rational Effect protocol on both initiator and responder side. Finally the protocol ends with the final result message, whereby the performative is set to either inform or failure depending on whether the responder succeeded in achieving the effect or not.

The protocol identifies the proper association of ACL messages in an ongoing conversation with more than two partners based on the FIPA ACL slot *conversion-id*. Furthermore, the proper order of messages is ensured by the slots *reply-with* and *in-reply-to*. This way, and by the internal state model of initiator and responder (see Figure 34 and Figure 35), it can be ensured that all messages arriving out of sequence will be discarded. Nevertheless, the protocol defines methods which make it possible to process them, but they do not show any effect to the flow of the protocol.

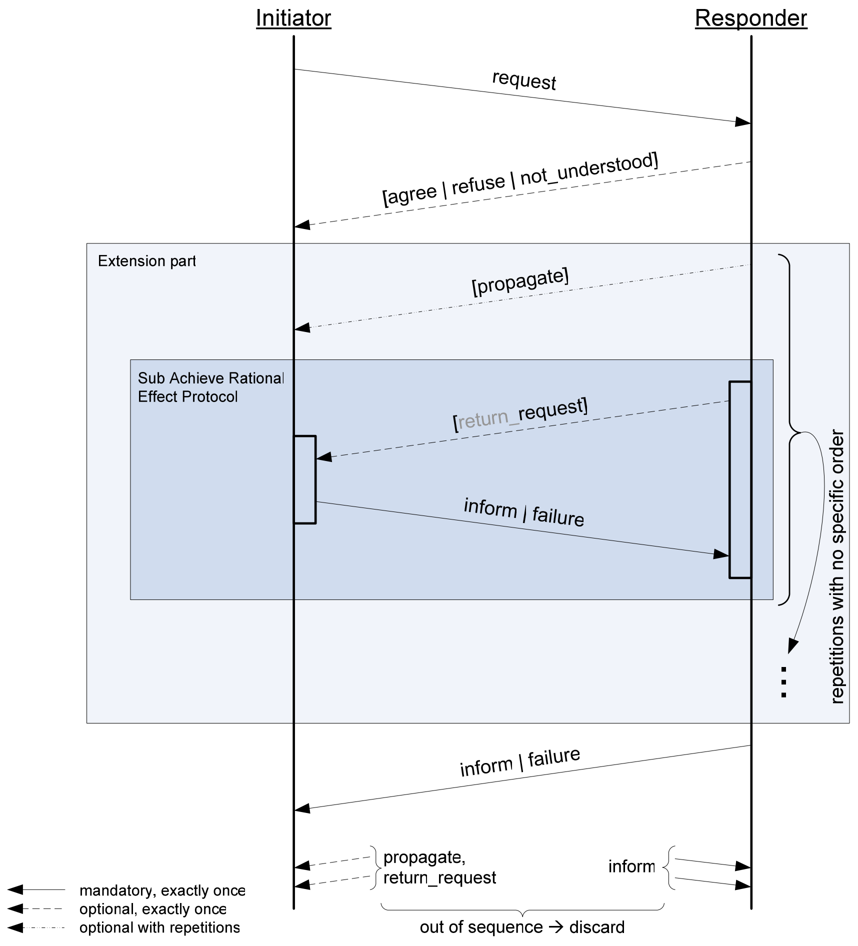

Initiator
Responder
request
[agree | refuse | not_understood]
Extension part
[propagate]
Sub Achieve Rational Effect Protocol
[return_request]
inform | failure
repetitions with no specific order
inform | failure
propagate, return_request
inform
out of sequence → discard
mandatory, exactly once
optional, exactly once
optional with repetitions

Figure 33 – Message Flow

For the intermediate/feedback messages the ACL performative propagate had to be chosen since all other ACL performatives either do not fit according to their associated semantics or would interfere with the protocols semantics. However, in case of the general CASCOM architecture the performative propagate truly fits to the setting since the client for the SEA is the service composition planner agent (SCPA) which propagates the current execution state to the personal agent anyway.

The Java implementation of the protocol is provided by the two correlating classes (they always have to be used in combination) `AchieveRESInitiator` and `AchieveRESResponder`, see Figure 36. Each of them implements finite state automata for the initiator and responder part, altogether realising the whole protocol.

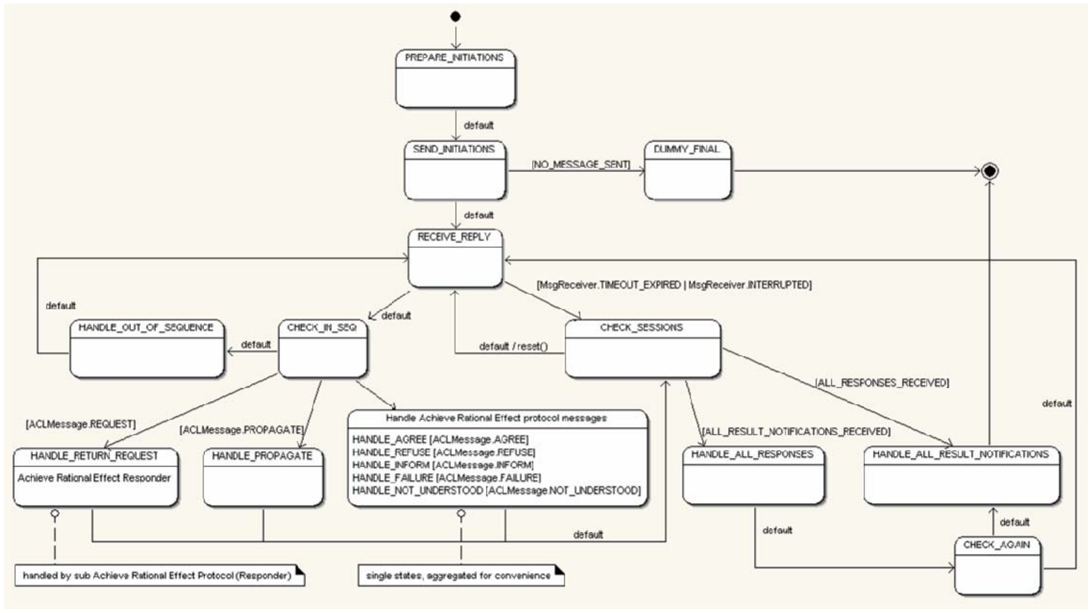

PREPARE_INITIATIONS
default
SEND_INITIATIONS
[NO_MESSAGE_SENT]
DUMMY_FINAL
default
RECEIVE_REPLY
[MsgReceiver.TIMEOUT_EXPIRED | MsgReceiver.INTERRUPTED]
default
default
HANDLE_OUT_OF_SEQUENCE
default
CHECK_IN_SEQ
default / reset()
CHECK_SESSIONS
[ALL_RESPONSES_RECEIVED]
default
[ACLMessage.REQUEST]
[ACLMessage.PROPAGATE]
Handle Achieve Rational Effect protocol messages
HANDLE_AGREE [ACLMessage.AGREE]
HANDLE_REFUSE [ACLMessage.REFUSE]
HANDLE_INFORM [ACLMessage.INFORM]
HANDLE_FAILURE [ACLMessage.FAILURE]
HANDLE_NOT_UNDERSTOOD [ACLMessage.NOT_UNDERSTOOD]
[ALL_RESULT_NOTIFICATIONS_RECEIVED]
HANDLE_RETURN_REQUEST
Achieve Rational Effect Responder
HANDLE_PROPAGATE
HANDLE_ALL_RESPONSES
HANDLE_ALL_RESULT_NOTIFICATIONS
default
default
default
CHECK_AGAIN
handled by sub Achieve Rational Effect Protocol (Responder)
single states, aggregated for convenience

Figure 34 – Achieve Rational Effect * Protocol – Initiator state model

Figure 34 and Figure 35 show the state models of the initiator and responder. In case of the initiator the relevant states are all the *handler states* containing the prefix `HANDLE_`. Each of them gets activated after an ACL message was received and it was checked according to its performative and slot *in-reply-to* to be in defined sequence order. Furthermore, each handler state corresponds to a `handle`… method and `registerHandle`… respectively. For the extension to the Achieve Ration Effect * protocol basically two changes had to be made, i.) alignment of the `CHECK_IN_SEQ` state, and ii.) addition of the handler states `HANDLE_RETURN_REQUEST` and `HANDLE_PROPAGATE`. For this reason it was possible to directly derive the implementation from `AchieveREInitiator`, see Figure 36.

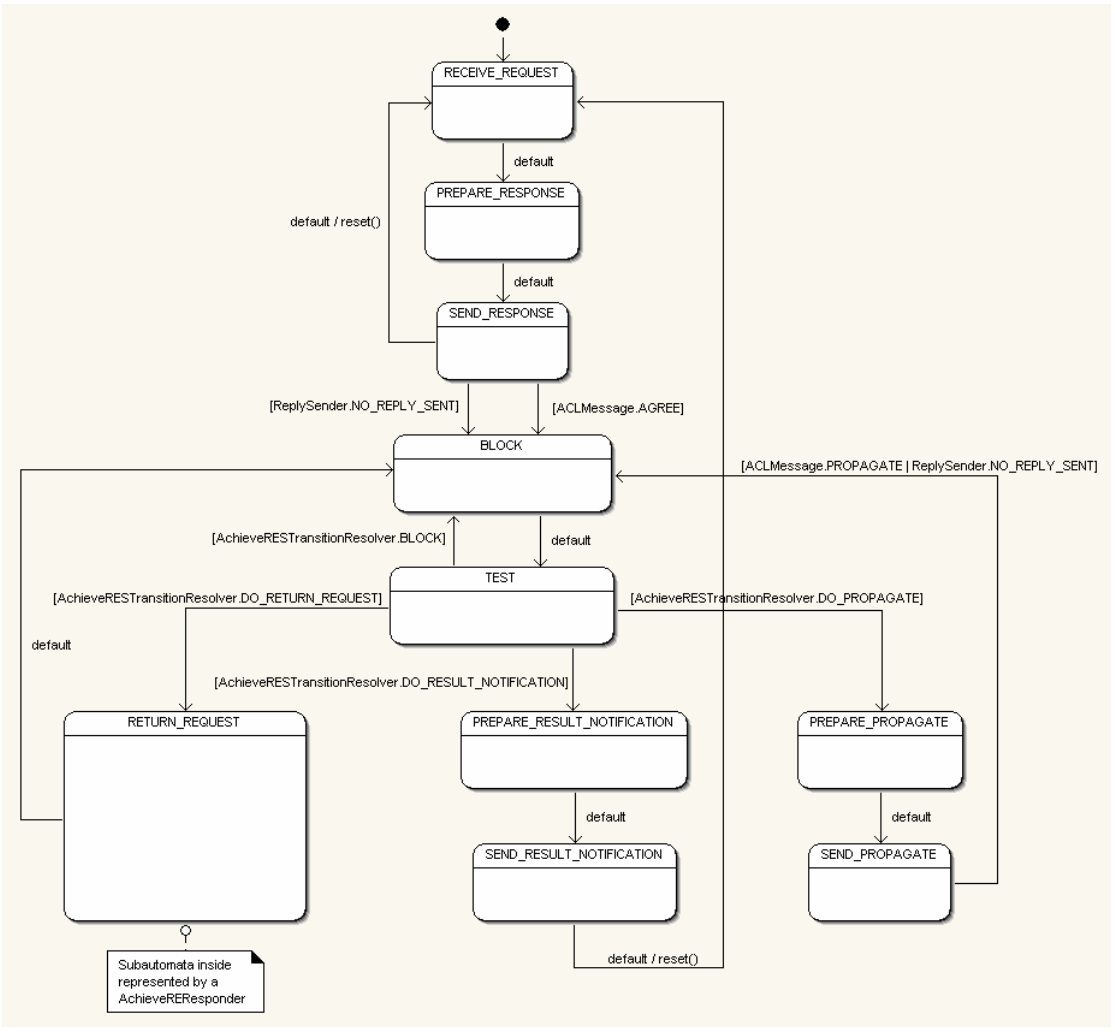

RECEIVE_REQUEST
default
PREPARE_RESPONSE
default / reset()
default
SEND_RESPONSE
[ReplySender.NO_REPLY_SENT]
[ACLMessage.AGREE]
BLOCK
[ACLMessage.PROPAGATE | ReplySender.NO_REPLY_SENT]
[AchieveRESTransitionResolver.BLOCK]
default
TEST
[AchieveRESTransitionResolver.DO_RETURN_REQUEST]
[AchieveRESTransitionResolver.DO_PROPAGATE]
default
[AchieveRESTransitionResolver.DO_RESULT_NOTIFICATION]
RETURN_REQUEST
PREPARE_RESULT_NOTIFICATION
PREPARE_PROPAGATE
default
default
SEND_RESULT_NOTIFICATION
SEND_PROPAGATE
Subautomata inside represented by a AchieveREResponder
default / reset()

Figure 35 – Achieve Rational Effect * Protocol – Responder state model

Parts of the state model for the responder part, see Figure 35, were reused from the existing class AchieveREResponder class but had to extend essentially. The relevant handler states are the PREPARE_... states and RETURN_REQUEST. Except for PREPARE_RESPONSE they get activated depending on the outcome of the TEST state. After the responder part of the protocol finishes by sending the final result notification the state machine does not terminate but transits to its initial state RECEIVE_REQUEST. This is done on purpose as an object resources optimisation during runtime and allows instances of the class to get reused after each cycle.

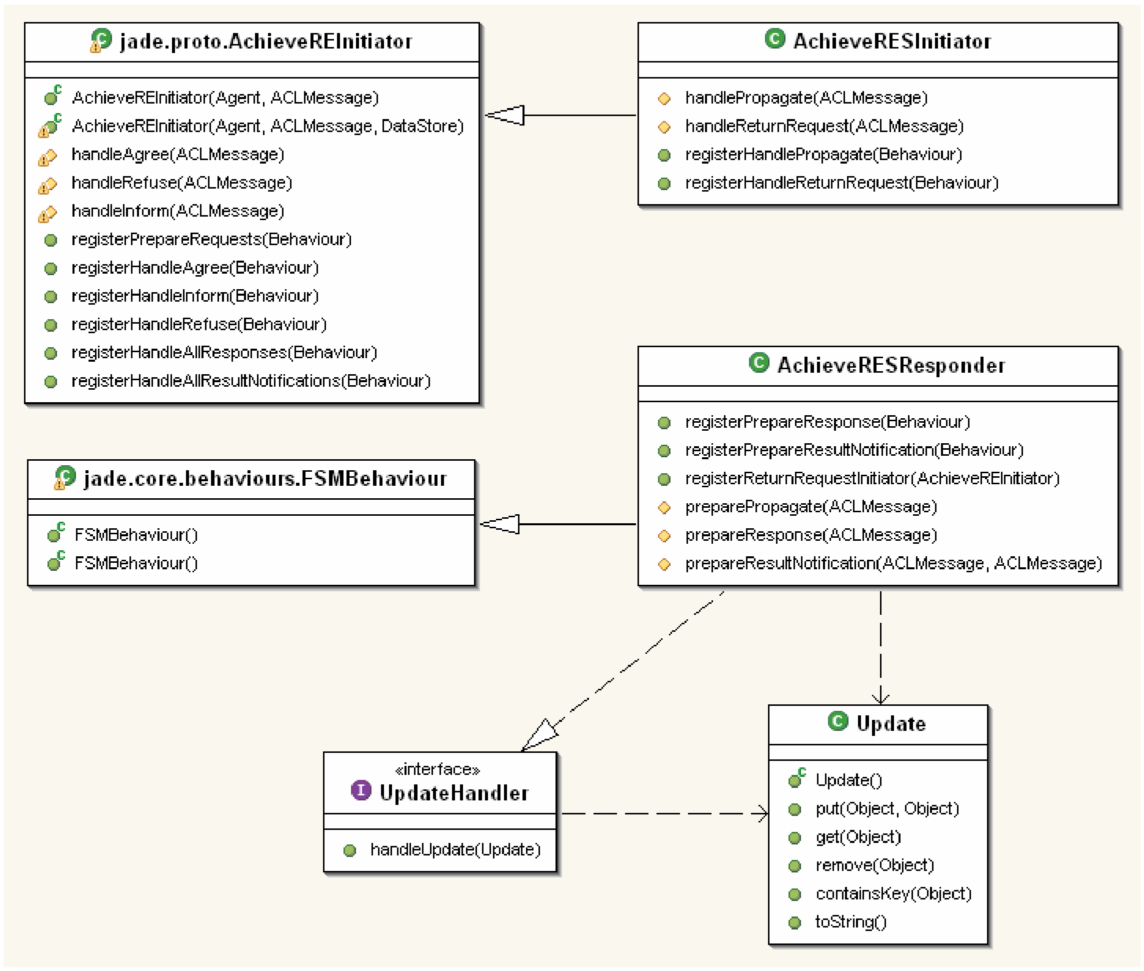

jade.proto.AchieveREInitiator
AchieveREInitiator(Agent, ACLMessage)
AchieveREInitiator(Agent, ACLMessage, DataStore)
handleAgree(ACLMessage)
handleRefuse(ACLMessage)
handleInform(ACLMessage)
registerPrepareRequests(Behaviour)
registerHandleAgree(Behaviour)
registerHandleInform(Behaviour)
registerHandleRefuse(Behaviour)
registerHandleAllResponses(Behaviour)
registerHandleAllResultNotifications(Behaviour)
AchieveRESInitiator
handlePropagate(ACLMessage)
handleReturnRequest(ACLMessage)
registerHandlePropagate(Behaviour)
registerHandleReturnRequest(Behaviour)
AchieveRESResponder
registerPrepareResponse(Behaviour)
registerPrepareResultNotification(Behaviour)
registerReturnRequestInitiator(AchieveREInitiator)
preparePropagate(ACLMessage)
prepareResponse(ACLMessage)
prepareResultNotification(ACLMessage, ACLMessage)
jade.core.behaviours.FSMBehaviour
FSMBehaviour()
FSMBehaviour()
«interface»
UpdateHandler
handleUpdate(Update)
Update
Update()
put(Object, Object)
get(Object)
remove(Object)
containsKey(Object)
toString()

Figure 36 – Achieve Rational Effect * Protocol – Class structure

Figure 36 displays the general class structure of both implementations `AchieveRESInitiator` and `AchieveRESResponder`. In general, they follow the JADE framework conventions for naming of state handler methods. `AchieveRESInitiator` contains four new methods required to represent the processing of intermediate/feedback messages and return requests. The class can be easily extended by overriding one of the `handle`… call back methods, which provide hooks to handle the corresponding states of the protocol. For instance, `handlePropagate()` is called when a propagate message is received. The more advanced possibility would be instead of extending the class and overriding some of its methods to register application-specific behaviours by using one of the `register`… methods. With respect to each state both possibilities are mutuallly exclusive but for different states both can be mixed. The same principles for handling the protocol states apply for the `AchieveRESResponder` class except that for handling the return request state, registration of a subsequent `AchieveREInitiator` is mandatory, i.e., no `handle`… method exists.

As noted in Section 3.2.1, progress of the protocol has to be achieved by an asynchronous notify/wait interaction pattern in order not to conflict with the round-robin non-preemptive scheduling policy for behaviours within agents. Support for this pattern is provided by the interface `UpdateHandler` and the class `Update`. This means that transitions between the states of the protocol are achieved by sending updates to the update handler – note that

`AchieveRESResponder` implements `UpdateHandler`. Each update will be processed and according to pre-defined markers part of each update it can be decided which handler state to choose next. Each update also stores the data to be used for sending ACL messages within each handler state. As soon as an update was processed the behaviour gives control back to the agent's behaviour scheduler and waits until the next update arrives.

3.2.3.2 Agent Ontology

This section describes the set of concepts, predicates, and actions and the semantics attached to them, i.e., the ontology used to create meaningful ACL message content for every interaction with the SEA, thus comprising the content definition of the speech act of the SEA. Generally, content expressions inside ACL messages are represented using the content language SL0 as it is common to the CASCOM agent architecture. The ontology is formally described based on the concepts realised inside the JADE platform, which means that it consists first of the vocabulary used within the content expressions and second a set of Java classes (see Figure 37) which directly reflect the concepts, predicates, and actions. The vocabulary is omitted here for convenience since its literals directly map to the attribute names of the Java classes.

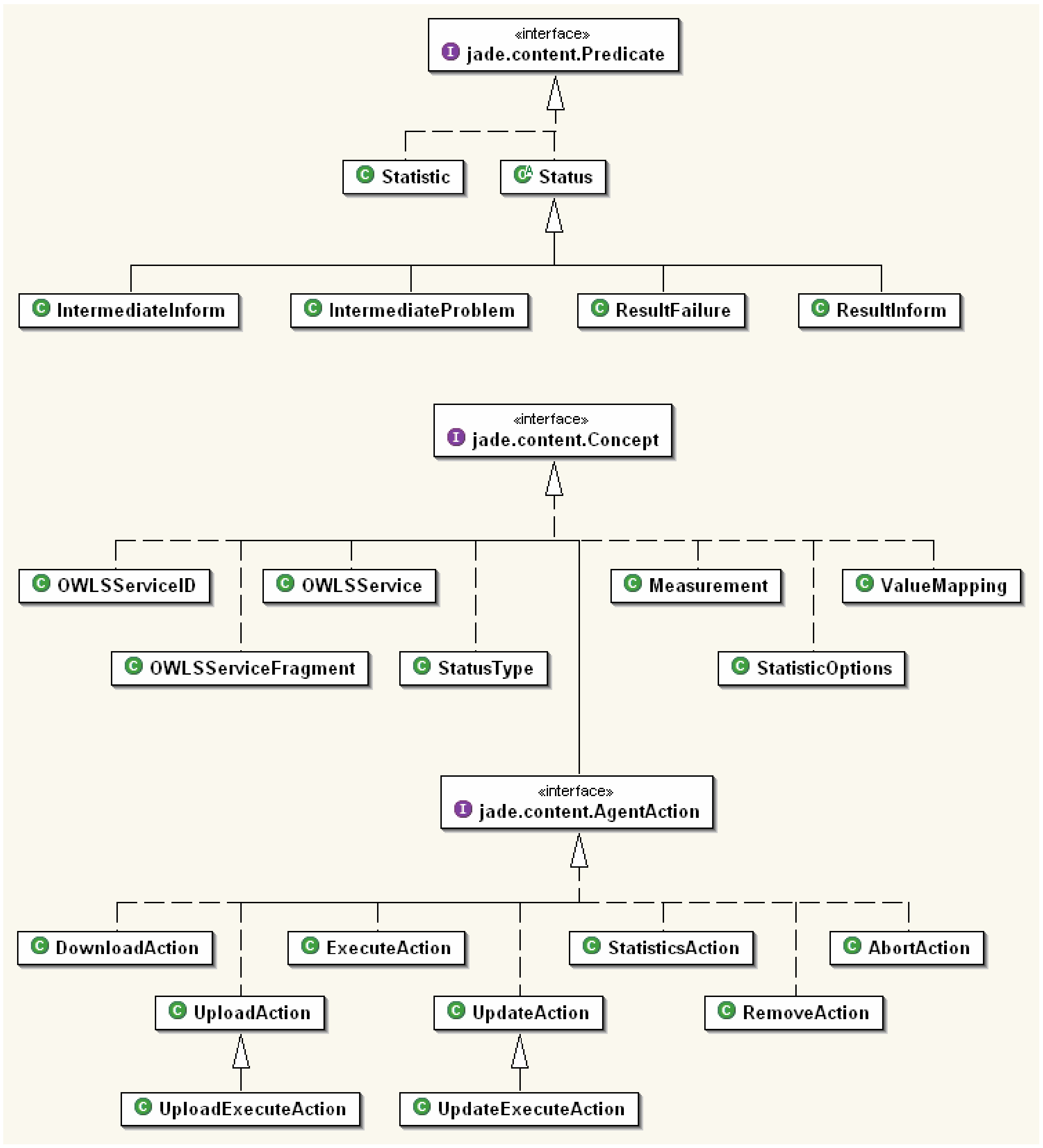


Figure 37 – Overview of ontology of execution agent

3.2.3.2.1 Description of Predicates

The following tables describe the semantics of all predicates and their attributes.

Class	**`ch.unibas.on.jade.onto.schema.Status`**		
Description	Abstract class which contains common attributes used as content for replies of the SEA.		
Attribute	**Description**	**Cardinality**	**Type**
message	A human readable string containing informal information about the status.	0…1	String
statusType	The type of the status.	1	StatusType
owlsServiceID	The identifier to which OWL-S service the status relates.	0…1	OWLSServiceID
payload	A list of Concept objects representing the result content.	0…*	Concept

Table 6 – Description of the predicate object Status

The classes (predicates) `IntermediateInform`, `IntermediateProblem`, `ResultInform`, and `ResultFailure` are subclasses of `Status` and do not add new attributes nor overwrite functionality of the super class except that the attribute `statusType` is fixed set to one of the constant `StatusType` instances.

Class	**`ch.unibas.on.jade.onto.schema.Statistic`**		
Description	Class which contains a list of measurement values sent as reply to agent action StatisticsAction.		
Attribute	**Description**	**Cardinality**	**Type**
measurements	The list of measurement objects.	0…*	Measurement

Table 7 – Description of the predicate object Statistic

3.2.3.2.2 Description of Concepts

The following tables describe the semantics of all concepts and their attributes.

Class	**`ch.unibas.on.jade.onto.schema.OWLSServiceID`**		
Description	Class which contains the unique identifier associated to each OWL-S service uploaded to the SEA, i.e., the identifier is assigned globally unique within the distributed execution system. Optionally, it stores a unique service instance identifier associated as soon as the service gets executed.		
Attribute	**Description**	**Cardinality**	**Type**
host	The agent where the identifier was created.	1	String
serviceId	The service identifier (alphanumerical).	1	String
serviceInstanceId	The service instance identifier (alphanumerical)	0…1	String

Table 8 – Description of the concept object OWLSServiceID

Class	`ch.unibas.on.jade.onto.schema.OWLSService` and `ch.unibas.on.jade.onto.schema.OWLSServiceFragment`		
Description	Both classes hold an OWL-S service description XML string. They are used either for initial execution of services or for execution after re-planning.		
Attribute	**Description**	**Cardinality**	**Type**
serialisation	The XML document of the OWL-S service.	1	String

Table 9 – Description of the concepts OWLSService and OWLSServiceFragment

Class	`ch.unibas.on.jade.onto.schema.StatusType`		
Description	Class used to identify the available status types within agent replies. Furthermore, it is used to specify which kind of status messages a initiator agent is interested in. The class contains four constants of this type which represent the pre-defined status types: INTERMEDIAT_INFORM, INTERMEDIATE_PROBLEM, RESULT_INFORM, RESULT_FAILURE.		
Attribute	**Description**	**Cardinality**	**Type**
value	A unique identifier to distinguish each status type from each other.	1	String

Table 10 – Description of the concept StatusType

Class	`ch.unibas.on.jade.onto.schema.Measurement`		
Description	Class used to store single, generic runtime measurements of the execution system which can be queried actively by other agents.		
Attribute	**Description**	**Cardinality**	**Type**
name	A formal or informal description of what the measurement represents.	1	String
value	The numerical or alphanumerical measurement value.	1	String
timestamp	Optional date when the measurement was taken.	0…1	Date

Table 11 – Description of the concept Measurement

Class	`ch.unibas.on.jade.onto.schema.StatisticOptions`		
Description	Class used to store the keys of measurements/statistic values to query for. Note that the syntax and semantics of key names is application specific.		
Attribute	**Description**	**Cardinality**	**Type**
measurementKeys	A list of keys identifying each measurement/statistic value.	1…*	String

Table 12 – Description of the concept StatisticOptions

Class	**`ch.unibas.on.jade.onto.schema.ValueMapping`**		
Description	Class used to store input and output data associated to a (composite) service for execution. The key references a certain input or output parameter of the OWL-S service profile and the value stores the concrete value used as input or output		
Attribute	**Description**	**Cardinality**	**Type**
complexValue	A flag which indicates whether the value is simple or complex.	1	Boolean
key	The key which maps to an existing input or output parameter within the OWL-S service profile.	1	String
value	Stores the concrete input or output value. In case of a simple type (number, boolean, string) the value is represented as string. In case of a complex type it stores an XML document.	1	String

Table 13 – Description of the concept ValueMapping

3.2.3.2.3 Description of Agent Actions

The following tables describe the semantics of all agent actions and their attributes. The tables also contain descriptions of the final reply messages sent back in either case failure or success.

Class	**`ch.unibas.on.jade.onto.schema.DownloadAction`**		
Description	Agent action used to download an OWL-S service description which was uploaded before.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary and the concept OWLSService containing the XML document of the service description.		
Attribute	**Description**	**Cardinality**	**Type**
serviceID	The identifier to indicate which OWL-S service to download.	1	OWLSServiceID

Table 14 – Description of the agent action DownloadAction

Class	**`ch.unibas.on.jade.onto.schema.UploadAction`**		
Description	Agent action used to upload an OWL-S service description. Start of execution has to be requested afterwards.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary and the concept OWLSServiceID containing the unique service identifier.		
Attribute	**Description**	**Cardinality**	**Type**
service	The service to upload.	1	OWLSService

Table 15 – Description of the agent action UploadAction

Class	**`ch.unibas.on.jade.onto.schema.UploadExecuteAction`**		
Description	Agent action used to upload an OWL-S service description and immediately trigger execution of the given service.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary and zero to more concepts ValueMapping containing the output values.		
Attribute	**Description**	**Cardinality**	**Type**
service	The service to upload and execute.	1	OWLSService
statusType	Indicates whether only final result message should be sent back or intermediate execution progress messages should be sent too.	1	StatusType
valueMappings	The set of input value associated to the input parameters of the service profile.	0…*	ValueMapping

Table 16 – Description of the agent action UploadExecuteAction

Class	ch.unibas.on.jade.onto.schema.ExecuteAction		
Description	Agent action used to trigger execution of an OWL-S service description which was uploaded already before.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary and zero to more concepts ValueMapping containing the output values.		
Attribute	**Description**	**Cardinality**	**Type**
serviceID	The identifier of the OWL-S service to execute.	1	OWLSServiceID
statusType	Indicates whether only final result message should be sent back or intermediate execution progress messages should be sent too.	1	StatusType
valueMappings	The set of input value associated to the input parameters of the service profile.	0…*	ValueMapping

Table 17 – Description of the agent action ExecuteAction

Class	ch.unibas.on.jade.onto.schema.UpdateAction		
Description	Agent action used to update an OWL-S service description which was uploaded before either completely or.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary.		
Attribute	**Description**	**Cardinality**	**Type**
serviceID	The identifier indicating which OWL-S service should be updated.	1	OWLSServiceID
serviceFragment	The new service (fragment) replacing the old one.	1	OWLSServiceFragment

Table 18 – Description of the agent action UpdateAction

Class	**`ch.unibas.on.jade.onto.schema.UpdateExecuteAction`**		
Description	Agent action used to update an OWL-S service description which was uploaded before either completely or partially and immediately trigger execution of the given service.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary and zero to more concepts ValueMapping containing the output values.		
Attribute	**Description**	**Cardinality**	**Type**
serviceID	The identifier indicating which OWL-S service should be updated and executed.	1	OWLSServiceID
serviceFragment	The new service (fragment) replacing the old one.	1	OWLSServiceFragment
statusType	Indicates whether only final result message should be sent back or intermediate execution progress messages should be sent too.	1	StatusType

Table 19 – Description of the agent action UpdateExecuteAction

Class	**`ch.unibas.on.jade.onto.schema.StatisticsAction`**		
Description	Agent action used to query for certain statistics/measurements of the execution system.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary and zero to more concepts Measurement according to the query parameters.		
Attribute	**Description**	**Cardinality**	**Type**
statisticOptions	The identifier to which OWL-S service the status relates.	1	StatisticOptions

Table 20 – Description of the agent action StatisticsAction

Class	`ch.unibas.on.jade.onto.schema.RemoveAction`		
Description	Agent action used to remove an OWL-S service description which was uploaded before.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary.		
Attribute	**Description**	**Cardinality**	**Type**
serviceID	The identifier indicating which OWL-S service should be removed.	1	OWLSServiceID

Table 21 – Description of the agent action RemoveAction

Class	`ch.unibas.on.jade.onto.schema.AbortAction`		
Description	Agent action used to abort ongoing execution of an OWL-S service instance.		
Failure message content	Agent answers with ResultFailure status predicate containing informal text message pre-defined in the ontology vocabulary.		
Success message content	Agent answers with ResultInform status predicate containing informal text message pre-defined in the ontology vocabulary.		
Attribute	**Description**	**Cardinality**	**Type**
serviceID	The identifier indicating which OWL-S service should be removed. Note that both the service and instance identifier must be given within the object.	1	OWLSServiceID

Table 22 – Description of the agent action AbortAction

3.2.4 Execution Monitoring

Service execution monitoring addresses activities that seek to acquire and distribute the extent to which progress is being made according to some reference. This includes time tracking information as well action tracking information and enables that timely actions can be taken in view of detected changes or deficiencies. However, the term execution monitoring here is only to be seen as an functionality for remote event monitoring and does not relate to local logging functionalities in the sense that execution monitoring realises local (and persistant) logging of execution checkpoints used for failure recovery.

For the service execution agent, monitoring is realised in a domain-independent way trying to complement integrated exception and failure handling within the agent infrastructure. Additionally, configuration of which data to collect can be done for each (composite) service execution independently as integral part of the message content, that is, monitoring properties are not globally configured.

In particular, execution monitoring of the implementation comprises the following functionality:

- Progress during execution used as the basis to create monitoring information is defined according to the sections in which a composite OWL-S service is split up, i.e., its atomic services. Consequently, progress can be monitored at the beginning and finish of each atomic service invocation. In particular, if an atomic service contains definition of post-conditions (effects) in its profile those post conditions are directly used together with the service name. If the service profile does not define post-conditions just the service name of the profile is used.
- Each client agent who submits and initiates execution of an OWL-S composite service can monitor execution progress of the submitted service. Client agents can only monitor their own submitted services.
- Client agents can set as part of each action message the monitoring properties. For instance, the agent actions `ExecuteAction` and `UploadExecuteAction` of the agent ontology – see Section 3.2.3.2 – contain the slot `statusType` to configure which monitoring information they are interested in. The properties can be set for each service execution independently.
- Client agents who have submitted execution of one OWL-S composite service will be actively notified after a certain progress was made according to some reference. In other words, the push based monitoring information delivery does not require client agents to actively poll for monitoring information.

3.2.5 Context-awareness

In the distributed approach the service execution agents interact with the generic CASCOM context system to both obtain and provide relevant contextual information. Provision and acquisition of context information is done throughout execution process by using the specific interface of the context framework. For example, each service execution agent constantly provides average CPU load information within a preceding time interval and the average time for service execution. This would allow other agents to select non-busy execution agents. Furthermore, each execution agent uses its own context information and probably context information from other sources as input to decide about where to continue with an ongoing execution – see description of the execution strategy in Section 3.2.1.2. Nevertheless, the design of the distributed execution agent aimed at preventing the situation where the usage of the context system becomes indispensable to proper functioning, i.e., integration of context-awareness is meant to be a benefitial but not crucial component.

3.3 Centralised Approach

Both approaches for Service Execution in IP2P environments that were implemented in the scope of the service coordination layer of the CASCOM Architecture present some (dis)advantages.

In the distributed approach (see Section 3.1) the execution process is carried out by a group of different entities (agents running in different places). Even though this prevents the single point of failure problem, it requires coordination mechanisms, which may be lengthy, to avoid conflicts in the execution of a composite service carried out by several different agents. In the centralized approach, the execution process is carried out by a single agent that has the necessary skills to execute complex composite services. Although it avoids having coordination mechanisms, it doesn’t prevent the single point failure problem.

Even though the distributed approach has managed to avoid heavy (time and resource consuming) coordination mechanisms (see section 3.2.1), it still requires agent developers to add a specific behavior in their agents in order to be able to be a part of the cooperative execution environment. Also, this extra development is only available for a single programming language (Java) and a single FIPA-complaint platform (JADE).

To complement the distributed approach, the centralized approach was designed to offer an alternative in a form of a single specialized execution agent, which can easily be contacted by a client agent (an agent requesting the execution of a certain service), using FIPA-compliant communication mechanisms and languages (quite in the same way as a client agent can contact a service execution agent in the distributed approach), thus avoiding any work performed by the agent developer.

3.3.1 Service Execution Agent

The Service Execution Agent (SEA) was implemented using Java and component-based software as well as other tools that were extended to incorporate new functionalities into the service execution environment. These tools are the JADE agent platform [BPR99] and the OWL-S API [S04].

The agent was developed with an internal architecture that was clearly designed to enable the agent to engage in these required interactions: receive and reply to requests from client agents; acquire\provide relevant context information; request for re-discovering and re-planning of services; and execute remote semantic web services.

3.3.1.1 Agent’s Interface

When requesting the execution of a specified service, client agents interact with SEA through the FIPA-request interaction protocol [FIPA00a]. This protocol states that when the receiver agent receives an action request, it can either agree or refuse to perform the action. It should then notify the other agent of its decision through the corresponding communicative act (FIPA-agree or FIPA-refuse).

SEA performs this decision process through a service execution’s request evaluation algorithm that involves acquiring adequate context information. SEA will only agree to perform a specific execution if it is possible to execute it, according to currently available context information. For example, if necessary service providers (for the execution of the service) are not available and the time that takes to find alternatives is longer than the timeframe the client agent expects to obtain a reply to the execution request, then SEA refuses to perform it.

On the other hand, if SEA is able to perform the execution request (because service providers are immediately available), but not in the time frame requested by the client agent (again, according to available context information) it also refuses to do so. SEA can also refuse to perform execution requests if its work-load is already too high (if its requests queue is bigger than a certain defined constant).

The FIPA-request also states that after successful execution of the requested action, the executer agent should return the corresponding results through a FIPA-inform message. After executing a service, SEA can send one of two different FIPA-inform messages: one sending the results obtained from the execution of the service; other sending just a notification that the service was successfully executed (when no results are produced by the execution).

3.3.1.2 Context-awareness

Context-aware computing is a computing paradigm in which applications can discover and take advantage of contextual information. A general mechanism for context-aware computing is summarized in the following steps [ADOB98]:

1. Collect information on the user's physical, informational or emotional state;
2. Analyze the information, either by treating it as an independent variable or by combining it with other information collected in the past or present;
3. Perform some action based on the analysis;
4. And repeat from step 1, with some adaptation based on previous iterations.

SEA uses a similar approach to [ADOB98] to enhance its activity, by adapting it to the specific situation that the agent and its client are involved in, at the time of the execution process. SEA interacts with the CASCOM's context system in order to obtain context information, subscribe desired context events and to provide relevant context information. Other agents, web services and sensors (both software and hardware) in the environment will interact with the context system as well, by providing relevant context information related to their own activities, which may be useful to other entities in the environment.

Acquisition of context information is made through a specific interface of the context framework by querying it in a pre-determined query language. Context events are event listeners that monitor certain changes in context. Whenever context information changes, the system notifies the entities that subscribed the corresponding class of events. This is useful for SEA to be aware of availability of certain entities in the environment on which it operates.

Throughout the execution process, SEA provides and acquires context information from and to this context system. For example, SEA provides relevant information such as the queue of its service execution requests and the average time of service execution. This will allow other entities in the environment to determine the service execution agent with the smallest work-load, and hence that can provide a faster execution service. During the execution of a composite service, SEA invokes atomic services from specific service providers (both web services, and service provider agents). SEA also provides valuable information regarding these service providers' availability and average execution time. Other entities can use this information to rate service providers or to simply determine the best service provider to use in a specific situation. Furthermore, SEA uses its own context information (as well as information from other sources and entities in the environment) to adapt the execution process to a specific situation. For instance, when selecting among several providers of some desirable atomic service, SEA will choose the one with better availability (lesser history of down time) and lower average execution time.

In situations such as the one where service providers are unavailable, it is faster to obtain the context information from the context system (as long as service providers can also provide their own availability context information) than by simply trying to use the services and discover that they are unavailable (because of the time lost waiting for connection time-outs to occur). After obtaining this relevant information, SEA can then contact other service-oriented agents (such as service discovery and composition agents) for requesting the re-discovering of service providers and/or re-planning of composite services. This situation-aware approach using context information on-the-fly helps SEA to provide a value-added execution service.

3.3.1.3 Internal Architecture

The developed agent is composed of three components: the Agent Interaction Component (AIC), the Engine Component (EC) and the Service Execution Component (SEC). Figure 38 illustrates the internal architecture of the agent and the interactions that occur between the components.

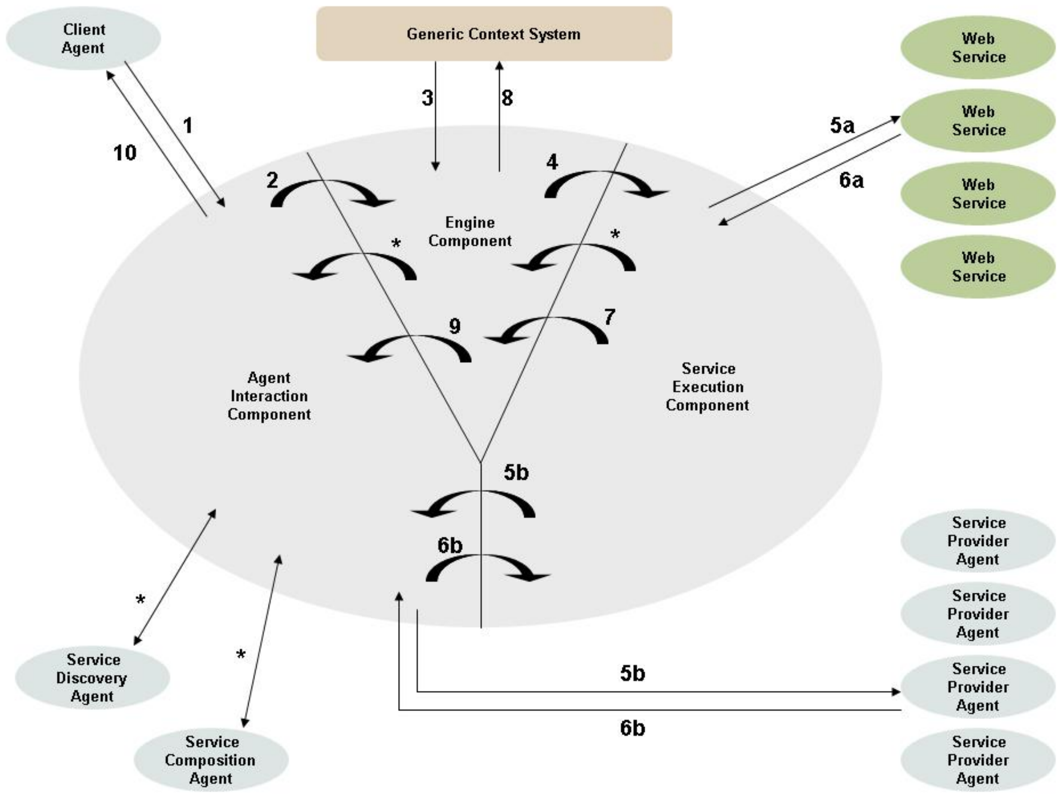


Figure 38 – Internal Architecture of the Service Execution Agent

The AIC was developed as an extension of the JADE platform and its goal is to provide an interaction framework to FIPA-compliant agents, such as SEA's clients (requesting the execution of specified services – Figure 38, step 1) and service discovery and composition agents (when SEA is requesting the rediscovering and re-planning of specific services – Figure 38, steps *). This component extends the JADE platform to provide extra features regarding language processing, behavior execution, database information retrieval and components' communication.

Among other things, the AIC is responsible for receiving messages, parsing them and processing them into a suitable format for the EC to use it (Figure 38, step 2). The reverse process is also the responsibility of AIC – receiving data from the EC and processing it into the agents' suitable format to be sent as messages (Figure 38, step 9). The EC is the main component of SEA as it controls the agent's overall activity. It is responsible for pre-processing service execution requests, interacting with the context system and deciding when to interact with other agents (such as service discovery and composition agents). When the EC receives an OWL-S service execution request (Figure 38, step 2), it acquires suitable context information (regarding potential service providers and other relevant information, such as client location – Figure 38, step 3) and plans the execution process.

If the service providers of a certain atomic service (invoked in the received composite service) are not available, SEA interacts with a service discovery agent (through the AIC – Figure 38, steps *) to discover available providers for the atomic services that are part of the OWL-S composite service. If the service discovery agent cannot find adequate service providers, the EC can interact with a service composition agent (again through the AIC – Figure 38, steps *) asking it to create an OWL-S composite service that produces the same effects as the original service.

After having a service ready for execution, with suitable context information, the EC sends it to the SEC (Figure 38, step 4), for execution. Throughout the execution process, the EC is also responsible for providing context information to the context system, whether it is its own information (such as service execution requests' queue, average time of execution), or other entities' relevant context information (such as availability of providers and average execution time of services).

The SEC was developed as an extension of the OWL-S API [S04] and its goal is to execute semantic web services (Figure 38, steps 5a and 6a) described using OWL-S service descriptions and WSDL grounding information. The extension of the OWL-S API allows for the evaluation of logical expressions in conditioned constructs, such as the *If-then-Else* and *While* constructs, and in the service's pre-conditions and effects. OWL-S API was also extended[17] in order to support the execution of services that are grounded on service provider agents (Figure 38, steps 5b, 6b). This extension is called *AgentGrounding* and it is explained in detail in section 3.3.1.4.

When the SEC receives a service execution request from the EC, it executes it according to the description of the service's process model. This generic execution process is described in section 3.1.

During the execution process, SEC collects relevant context information (such as providers' availability, quality of service and execution times). After execution of the specified service and generation of its results, the SEC sends them to the EC (Figure 38, step 7) for further analysis and post-processing, which includes sending gathered context information to the context system (Figure 38, step 8) and sending the results to the client agent (through the AIC – Figure 38, steps 9, 10).

3.3.1.4 OWL-S Grounding Extension: AgentGrounding

WSDL describes the access to a network service, more specifically, web services provided by network service providers. However, WSDL currently lacks a way of representing agent bindings, i.e., a representation for complex interactions such as the ones that take place with service provider agents. To overcome this limitation, we decided to create an extension of the OWL-S *Grounding* specification, named *AgentGrounding*. This extension is the result of an analysis of the necessary requirements for interacting with agents when evoking the execution of atomic services.

The *AgentGrounding* definition includes the following elements:

- *agentName* – the name of the service provider agent
- *agentAddress* – the address of the service provider
- *serviceName* – name of the service to be evoked
- *serviceType* – type of the service to be evoked (action, referential expression, proposition)
- *hasArgumentParameter* – used to represent arguments of the service
    - *argumentType* – type of the argument (string, integer)
    - *owlsParameter* – reference to OWL-S service defined parameters (inputs)
    - *paramIndex* – order on which the argument appears in the service invocation
- *serviceOutput* – used to represent the outputs of the service
    - argumentType
    - owlsParameter
- *protocol* – communication protocol to be used when invoking the service

[17] This extension constitutes an add-on to the original Service Execution Agent and it was done only to increase the range of classes of service providers that can be used in the CASCOM environment.

- *serviceOntology* – ontology which describes the terms to be used in the invocation of the service
- *agentCommunicationLanguage* – agent communication language to be used when communicating with the agent (FIPA-ACL, KQML)
- *contentLanguage* – content language to be used when communicating with the agent (FIPA-SL, KIF)

Figure 39 is an example of an OWL-S service *Grounding* description using the proposed *AgentGrounding* extension. This description illustrates a service, provided by a FIPA compliant agent, of finding books (within several different sources) with a given input title.

```
<!-- Grounding description -->
<agentGrounding:AgentGrounding rdf:ID="BookFinderGrounding">
  <service:supportedBy rdf:resource="#BookFinderService"/>
  <grounding:hasAtomicProcessGrounding
rdf:resource="#BookFinderProcessGrounding"/>
</agentGrounding:AgentGrounding>
<agentGrounding:AgentAtomicProcessGrounding
rdf:ID="BookFinderProcessGrounding">
  <grounding:owlsProcess rdf:resource="#BookFinderProcess"/>
    <agentGrounding:agentName>bookeeper@cascom
    </agentGrounding:agentName>
    <agentGrounding:agentAddress>http://...
    </agentGrounding:agentAddress>
    <!-- Service Identification -->
    <agentGrounding:serviceName>find-book
    </agentGrounding:serviceName>
    <!-- Service Arguments -->
    <agentGrounding:hasArgumentParameter>
      <agentGrounding:ArgumentParameter rdf:ID="input-string">
       <agentGrounding:argumentType>java.lang.String
       </agentGrounding:argumentType>
       <agentGrounding:owlsParameter rdf:resource="#InputString"/>
       <agentGrounding:paramIndex>1</agentGrounding:paramIndex>
      </agentGrounding:ArgumentParameter>
    </agentGrounding:hasArgumentParameter>
    <agentGrounding:serviceOutput>
     <agentGrounding:ArgumentVariable rdf:ID="book-info">
      <agentGrounding:argumentType>java.lang.String
      </agentGrounding:argumentType>
      <agentGrounding:owlsParameter rdf:resource="#BookInfo"/>
     </agentGrounding:ArgumentVariable>
    </agentGrounding:serviceOutput>
   <!-- Other information -->
   <agentGrounding:serviceType>action</agentGrounding:serviceType>
   <agentGrounding:protocol>fipa-request</agentGrounding:protocol>
   <agentGrounding:agentCommunicationLanguage>fipa-acl
   </agentGrounding:agentCommunicationLanguage>
   <agentGrounding:contentLanguage>
   fipa-sl</agentGrounding:contentLanguage>
   <agentGrounding:serviceOntology>book-finder-ontology
   </agentGrounding:serviceOntology>
</agentGrounding:AgentAtomicProcessGrounding>
```

Figure 39 – Example of AgentGrounding description

The extension made to the OWL-S API, allows the execution of groundings such as the one described in Figure 39 by invoking the specified service. This invocation is made by sending a message directly to the agent providing the service. All the information that is needed for sending the message is included in the *AgentGrounding* description.

The example depicted in Figure 39 describes a service named *BookFinderService*, which is grounded to an action *find-book* that accepts as input a single string named *inputstring*. This *input-string* argument is linked to the OWL-S service input parameter *InputString*. The action returns as output, also a string, named *book-info*, which is linked to the OWL-S service output parameter *BookInfo*. Other information that can be extracted from this grounding is the protocol (*fipa-request*), the agent communication language (*fipa-acl*), the ontology (*book-finder-ontology*) and the content language (*fipa-sl*) to be used in the invocation message. SEA can use this information to send the FIPA message that is described in Figure 40.

```
(REQUEST
 :sender (agent-identifier :name sea@cascom)
 :receiver (set (agent-identifier :name bookeeper@cascom
   :addresses (sequence http://...)))
 :content "((action
     (agent-identifier :name bookeeper@cascom)
     (find-book
      :input-string \"Da Vinci Code\")))"
 :language fipa-sl
 :ontology book-finder-ontology
 :protocol fipa-request
 :conversation-id SEA-AGR-CID-1117800292257)
```

Figure 40 – Message generated from the example in Figure 39

The information extracted from the *AgentGrounding* example in Figure 39 is enough for the agent to be able to create the message. However, the agent must also add the OWL-S Service Input information to the generated grounding message. In this case, the client agent requested the execution of the service with the input *"Da Vinci Code"*.

The *AgentGrounding* specification allows the representation of several instances of messages that can be sent to FIPA compliant agents, including the use of different message performatives, agent communication languages and content languages. However, this is a work in progress and some possibilities are not yet covered.

References

[ADOB98] Abowd, G. D., Dey, A., Orr, R. and Brotherton, J., 1998, “Context-awareness in wearable and ubiquitous computing”, Virtual Reality, 3:200–211.

[BPR99] Bellifemine, F., Poggi, A., Rimassa, G., 1999, “JADE – a FIPA-compliant agent framework”, CSELT internal technical report. Part of this report has been also published in Proceedings of PAAM'99, London, pp.97-108.

[BAER02] Botelho, L.M.; Antunes, N.; Ebrahim, M.; and Ramos, P., 2002, “Greeks and Trojans Together”, In Proc. of the Workshop “Ontologies in Agent Systems” of the AAMAS2002 Conference.

[CCMW01] Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S., 2001, “Web Services Description Language (WSDL) 1.1”. Available on-line at http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[DFKI05] DFKI - Die Deutsche Forschungszentrum für Künstliche Intelligenz, 2005, “OWLS2PDDL converter“, at http://projects.semwebcentral.org/projects/owls2pddl/

[DK01] Do, M.B., Kambhampati, S., 2001, “Sapa: A Domain-Independent Heuristic Metric Temporal Planner”, in Proceedings of the 6th European Conference on Planning

[FIPA00a] Foundation for Intelligent Physical Agents, 2000, “FIPA Communicative Act Library Specification”, Specification number SC00037, Geneva, Switzerland.

[FIPA00b] Foundation for Intelligent Physical Agents, 2000, “FIPA ACL Message Structure Specification”, Specification number SC00061, Geneva, Switzerland

[FIPA00c] Foundation for Intelligent Physical Agents, 2000, “FIPA SL Content Language Specification”, Specification Number SC00008, Geneva, Switzerland

[FIPA02] FIPA Members, 2002, “Foundation for Intelligent Physical Agents website”, http://www.fipa.org/

[HN01] Hoffmann, J., and Nebel, B., 2001, “The FF Planning System: Fast Plan Generation Through Heuristic Search”, Journal of Artificial Intelligence Research (JAIR) (14):253–302.

[KGS05] Klusch, M.; Gerber, A.; Schmidt, M., 2005, “Semantic Web Service Composition Planning with OWLS-Xplan”, Proceedings 1st Intl. AAAI Fall Symposium on Agents and the Semantic Web, Arlington VA, USA

[M98] McDermott, D., 1998, “PDDL - The Planning Domain Definition Language”, AIPS-98 Competition Committee, draft 1.6 edition

[MBHL04] Martin, D.; Burstein, M.; Hobbs, J.; Lassila, O.; McDermott, D.; McIlraith, S.; Narayanan, S.; Paolucci, M.; Parsia, B; Payne, T.; Sirin, E.; Srinivasan, N. and Sycara, K., 2004, "OWL-S 1.1 Release", http://www.daml.org/services/owl-s/1.1/overview/

[OWL03] OWL Coalition, 2003, “OWL – Web Ontology Language Reference W3C Working Draft”, at http://www.w3.org/TR/owl-ref/

[OWLS03] OWL Services Coalition, 2003, “OWL-S: Semantic Markup for Web Services”, DARPA Markup Language Program

[S04] Sirin, E., 2004, “OWL-S API project website”, http://www.mindswap.org/2004/owl-s/api/

[SPW04] Sirin, E.; Parsia, B.; Wu, D.; Hendler, J.; and Nau, D. 2004. “HTN planning for web service composition using SHOP2”. Journal of Web Semantics, 1(4) 377–396.

Annexes

This section presents the annexes that are referred throughout the entire deliverable.

A. OMS_InitialOntology.owl file content

In this section we can see the OWL description of the ontology representing the initial state of the world before the service is executed.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://localhost/OMSOntology.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Account">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Location">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Medicine">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Product"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Object"/>
  <owl:Class rdf:ID="Patient">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Person"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Product">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Person">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Store">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="wants_account">
    <rdfs:range rdf:resource="#Patient"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="Patient_hasAccount">
    <rdfs:domain rdf:resource="#Patient"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="Patient_ownsMedicine">
```

```
        <rdfs:domain rdf:resource="#Patient"/>
      <rdfs:range rdf:resource="#Medicine"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="Patient_hasValidData">
      <rdfs:domain rdf:resource="#Patient"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="bill">
      <rdfs:range rdf:resource="#Medicine"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="link">
      <rdfs:domain rdf:resource="#Object"/>
      <rdfs:range rdf:resource="#Object"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="at">
      <rdfs:domain rdf:resource="#Object"/>
      <rdfs:range rdf:resource="#Object"/>
    </owl:ObjectProperty>
    <Location rdf:ID="location0">
      <at>
        <Patient rdf:ID="patient0">
            <Patient_hasValidData>
            </Patient_hasValidData>
            <wants_account>
            </wants_account>
        </Patient>
      </at>
    </Location>
    <Location rdf:ID="location1">
      <link>
        <Store rdf:ID="store0">
          <link>
         <Medicine rdf:ID="medicine0"/>
          </link>
        </Store>
      </link>
    </Location>
    <Location rdf:ID="location0">
      <link>
         <Patient rdf:ID="patient0"/>
      </link>
    </Location>
  </rdf:RDF>
```

B. OMS_GoalOntology.owl file content

In this section we can see the OWL description of the goal that the composite service is supposed to achieve.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://localhost/OMSOntology.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Account">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Location">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Medicine">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Product"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Object"/>
  <owl:Class rdf:ID="Patient">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Person"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Product">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Person">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Store">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Object"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="wants_account">
    <rdfs:range rdf:resource="#Patient"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="Patient_ownsMedicine">
    <rdfs:domain rdf:resource="#Patient"/>
    <rdfs:range rdf:resource="#Medicine"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="Patient_hasValidData">
    <rdfs:domain rdf:resource="#Patient"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="Patient_hasAccount">
    <rdfs:domain rdf:resource="#Patient"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="bill">
    <rdfs:range rdf:resource="#Medicine"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="link">
    <rdfs:domain rdf:resource="#Object"/>
    <rdfs:range rdf:resource="#Object"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="at">
    <rdfs:domain rdf:resource="#Object"/>
    <rdfs:range rdf:resource="#Object"/>
  </owl:ObjectProperty>
  <Location rdf:ID="location0">
    <at>
      <Medicine rdf:ID="medicine0">
        <Patient_ownsMedicine>
          <Patient rdf:ID="patient0"/>
        </Patient_ownsMedicine>
      </Medicine>
     </at>
   </Location>
 </rdf:RDF>
```

C. RegisterPatientService.owl file content

This section describes the OWL-S description of the first retrieved service by the SDA. This is a typical OWL-S format description, which means that we can find certain things that we need. Things like IOPEs, that later will be used by Sapa to produce the specified chaining of services, based on their pre-conditions and effects. This exact service will be the first of that chain.

```
<?xml version="1.0"?>
<!DOCTYPE uridef [
<!ENTITY ontology "http://localhost/ontologies/Ontology.owl">
]>
<rdf:RDF
    xmlns:expr="http://www.daml.org/services/owl-
s/1.1/generic/Expression.owl#"
    xmlns:process="http://www.daml.org/services/owl-
s/1.1/Process.owl#"
    xmlns:shadow-rdf="http://www.daml.org/services/owl-
s/1.1/generic/ObjectList.owl#"
    xmlns:service="http://www.daml.org/services/owl-
s/1.1/Service.owl#"
    xmlns="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:ontology= "&ontology;#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:swrl="http://www.daml.org/services/owl-
s/1.1/generic/swrlx.owl#"
    xmlns:j.0="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
    xml:base="http://localhost/oms/RegisterPatientService.owl">
    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Service.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Profile.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Grounding.owl"/>
    </owl:Ontology>
    <process:Input rdf:ID="RegisterPatientService_p">
        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
st/OMS_InitialOntology.owl#Patient</process:parameterType>
    </process:Input>

    <j.0:Profile rdf:ID="RegisterPatientService_profile">
        <j.0:hasInput rdf:resource="#RegisterPatientService_p"/>
        <j.0:textDescription>This is the description of thehealthcare
profile</j.0:textDescription>
        <j.0:has_process>

    <j.0:serviceCategory>
    <ontology:NAICS rdf:ID="NAICS-category">
    <j.0:value>Register-Patient</j.0:value>
    <j.0:code>561599</j.0:code>
    </ontology:NAICS>
    </j.0:serviceCategory>

<process:AtomicProcess rdf:ID="RegisterPatientService">
        <process:name>RegisterPatientService</process:name>
        <process:hasInput rdf:resource="#RegisterPatientService_p"/>
        <process:hasPrecondition>
            <expr:Condition rdf:ID="atomic1_precond1">
                <expr:expressionLanguage>
                    <expr:LogicLanguage rdf:ID="PDDL">

<expr:refURI>http://localhost/PDDL</expr:refURI>
                    </expr:LogicLanguage>
                </expr:expressionLanguage>
                <expr:expressionBody>
                  <and>
                 <pred name="Patient_hasValidData">
                   <param>?RegisterPatientService_p</param>
                 </pred>
                 <pred name="wants_account">
                   <param>?RegisterPatientService_p</param>
                 </pred>
               </and>
             </expr:expressionBody>
            </expr:Condition>
        </process:hasPrecondition>
        <process:hasResult>
            <process:Result>
                <process:hasEffect>
                    <expr:Expression rdf:ID="atomic1_effect1">
                        <expr:expressionLanguage
rdf:resource="#PDDL"/>
                        <expr:expressionBody>
                       <and>
                            <not>
                           <pred name="wants_account">
                             <param>?RegisterPatientService_p</param>
                            </pred>
                            </not>
                         <pred name="Patient_hasAccount">
                           <param>?RegisterPatientService_p</param>
                         </pred>
                       </and>
                        </expr:expressionBody>
                    </expr:Expression>
                </process:hasEffect>
            </process:Result>
        </process:hasResult>
    </process:AtomicProcess>
</j.0:has_process>
        <j.0:hasResult>
            <process:Result>
                <process:hasEffect rdf:resource="#atomic1_effect1"/>
            </process:Result>
        </j.0:hasResult>
        <j.0:hasPrecondition rdf:resource="#atomic1_precond1"/>
        <j.0:serviceName>healthcare</j.0:serviceName>
    </j.0:Profile>
    <service:Service rdf:ID="RegisterPatientService_service">
        <service:presents
rdf:resource="#RegisterPatientService_profile"/>
        <service:describedBy rdf:resource="#RegisterPatientService"/>
    </service:Service>
</rdf:RDF>
```

D. SellMedicineService.owl file content

This annex describes the OWL-S description of the second retrieved service by the SDA.

```
<?xml version="1.0"?>
<!DOCTYPE uridef [
<!ENTITY ontology "http://localhost/ontologies/Ontology.owl">
]>
<rdf:RDF
    xmlns:expr="http://www.daml.org/services/owl-
s/1.1/generic/Expression.owl#"
    xmlns:process="http://www.daml.org/services/owl-
s/1.1/Process.owl#"
    xmlns:shadow-rdf="http://www.daml.org/services/owl-
s/1.1/generic/ObjectList.owl#"
    xmlns:service="http://www.daml.org/services/owl-
s/1.1/Service.owl#"
    xmlns="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:swrl="http://www.daml.org/services/owl-
s/1.1/generic/swrlx.owl#"
    xmlns:ontology= "&ontology;#"
    xmlns:j.0="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
    xml:base="http://localhost/oms/SellMedicineService.owl">
    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Service.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Profile.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Grounding.owl"/>
    </owl:Ontology>
    <process:Input rdf:ID="SellMedicineService_p">
        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
st/OMS_InitialOntology.owl#Patient</process:parameterType>
    </process:Input>
    <process:Input rdf:ID="SellMedicineService_m">
        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
st/OMS_InitialOntology.owl#Medicine</process:parameterType>
    </process:Input>
    <process:Input rdf:ID="SellMedicineService_s">
        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
st/OMS_InitialOntology.owl#Store</process:parameterType>
    </process:Input>
    <process:Input rdf:ID="SellMedicineService_from">
```

```
        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
st/OMS_InitialOntology.owl#Location</process:parameterType>
    </process:Input>

    <j.0:Profile rdf:ID="SellMedicineService_profile">
        <j.0:hasInput rdf:resource="#SellMedicineService_p"/>
        <j.0:textDescription>This is the description of thehealthcare
profile</j.0:textDescription>
        <j.0:has_process>

    <j.0:serviceCategory>
    <ontology:NAICS rdf:ID="NAICS-category">
    <j.0:value>Sell-Medicine</j.0:value>
    <j.0:code>561599</j.0:code>
    </ontology:NAICS>
    </j.0:serviceCategory>

      <process:AtomicProcess rdf:ID="SellMedicineService">
        <process:name>SellMedicineService</process:name>
        <process:hasInput rdf:resource="#SellMedicineService_p"/>
        <process:hasInput rdf:resource="#SellMedicineService_m"/>
        <process:hasInput rdf:resource="#SellMedicineService_s"/>
        <process:hasInput rdf:resource="#SellMedicineService_from"/>
        <process:hasPrecondition>
            <expr:Condition rdf:ID="atomic2_precond1">
                <expr:expressionLanguage rdf:resource="#PDDL"/>
                <expr:expressionBody>
               <and>
                 <pred name="Patient_hasAccount">
                   <param>?SellMedicineService_p</param>
                 </pred>
                 <pred name="link">
                   <param>?SellMedicineService_s</param>
                   <param>?SellMedicineService_from</param>
                 </pred>
                 <pred name="link">
                   <param>?SellMedicineService_m</param>
                   <param>?SellMedicineService_s</param>
                 </pred>
               </and>
                </expr:expressionBody>
            </expr:Condition>
        </process:hasPrecondition>
        <process:hasResult>
            <process:Result>
                <process:hasEffect>
                    <expr:Expression rdf:ID="atomic2_effect1">
                        <expr:expressionLanguage
rdf:resource="#PDDL"/>
                        <expr:expressionBody>
                       <and>
                         <pred name="Patient_ownsMedicine">
                           <param>?SellMedicineService_p</param>
                           <param>?SellMedicineService_m</param>
                         </pred>
                         <pred name="bill">
```

```
                          <param>?SellMedicineService_m</param>
                        </pred>
                        <pred name="at">
                          <param>?SellMedicineService_m</param>
                          <param>?SellMedicineService_from</param>
                        </pred>
                      </and>
                    </expr:expressionBody>
                   </expr:Expression>
               </process:hasEffect>
           </process:Result>
       </process:hasResult>
   </process:AtomicProcess>
  </j.0:has_process>
       <j.0:hasResult>
           <process:Result>
               <process:hasEffect rdf:resource="#atomic2_effect1"/>
           </process:Result>
       </j.0:hasResult>
       <j.0:hasPrecondition rdf:resource="#atomic2_precond1"/>
       <j.0:serviceName>healthcare</j.0:serviceName>
   </j.0:Profile>
   <service:Service rdf:ID="SellMedicineService_service">
       <service:presents
rdf:resource="#SellMedicineService_profile"/>
       <service:describedBy rdf:resource="#SellMedicineService"/>
   </service:Service>
</rdf:RDF>
```

E. DeliverService.owl file content

This annex presents the OWL-S description of the third and final retrieved service by the SDA.

```
<?xml version="1.0"?>
<!DOCTYPE uridef [
  <!ENTITY ontology "http://localhost/ontologies/Ontology.owl">
]>
<rdf:RDF
    xmlns:expr="http://www.daml.org/services/owl-
s/1.1/generic/Expression.owl#"
    xmlns:process="http://www.daml.org/services/owl-
s/1.1/Process.owl#"
    xmlns:shadow-rdf="http://www.daml.org/services/owl-
s/1.1/generic/ObjectList.owl#"
    xmlns:service="http://www.daml.org/services/owl-
s/1.1/Service.owl#"
    xmlns="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:swrl="http://www.daml.org/services/owl-
s/1.1/generic/swrlx.owl#"
    xmlns:ontology= "&ontology;#"
    xmlns:j.0="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
    xml:base="http://localhost/oms/DeliverService.owl">
    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Service.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Profile.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Grounding.owl"/>
    </owl:Ontology>
    <process:Input rdf:ID="DeliverService_m">
      <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
st/OMS_InitialOntology.owl#Medicine</process:parameterType>
    </process:Input>
    <process:Input rdf:ID="DeliverService_p">
      <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
st/OMS_InitialOntology.owl#Patient</process:parameterType>
    </process:Input>
    <process:Input rdf:ID="DeliverService_from">
      <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
st/OMS_InitialOntology.owl#Location</process:parameterType>
    </process:Input>
    <process:Input rdf:ID="DeliverService_to">
```

```
        <process:parameterType
    rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://localho
    st/OMS_InitialOntology.owl#Location</process:parameterType>
        </process:Input>

        <j.0:Profile rdf:ID="DeliverService_profile">
            <j.0:hasInput rdf:resource="#DeliverService_p"/>
            <j.0:textDescription>This is the description of thehealthcare
    profile</j.0:textDescription>
            <j.0:has_process>

        <j.0:serviceCategory>
        <ontology:NAICS rdf:ID="NAICS-category">
        <j.0:value>Deliver-Service</j.0:value>
        <j.0:code>561599</j.0:code>
        </ontology:NAICS>
        </j.0:serviceCategory>

    <process:AtomicProcess rdf:ID="DeliverService">
        <process:name>DeliverService</process:name>
        <process:hasInput rdf:resource="#DeliverService_m"/>
        <process:hasInput rdf:resource="#DeliverService_p"/>
        <process:hasInput rdf:resource="#DeliverService_from"/>
        <process:hasInput rdf:resource="#DeliverService_to"/>
        <process:hasPrecondition>
            <expr:Condition rdf:ID="atomic3_precond1">
                <expr:expressionLanguage rdf:resource="#PDDL"/>
                <expr:expressionBody>
                   <and>
                     <pred name="at">
                       <param>?DeliverService_m</param>
                       <param>?DeliverService_from</param>
                     </pred>
                     <pred name="bill">
                       <param>?DeliverService_m</param>
                     </pred>
                     <pred name="link">
                       <param>?DeliverService_p</param>
                       <param>?DeliverService_to</param>
                     </pred>
                   </and>
                </expr:expressionBody>
            </expr:Condition>
        </process:hasPrecondition>
        <process:hasResult>
            <process:Result>
                <process:hasEffect>
                    <expr:Expression rdf:ID="atomic3_effect1">
                        <expr:expressionLanguage rdf:resource="#PDDL"/>
                        <expr:expressionBody>
                          <and>
                            <not>
                           <pred name="at">
                             <param>?DeliverService_m</param>
                             <param>?DeliverService_from</param>
                           </pred>
                         </not>
                         <pred name="at">
                           <param>?DeliverService_m</param>
                           <param>?DeliverService_to</param>
                         </pred>
                       </and>
                     </expr:expressionBody>
                    </expr:Expression>
                </process:hasEffect>
            </process:Result>
        </process:hasResult>
      </process:AtomicProcess>
    </j.0:has_process>

            <j.0:hasResult>
                <process:Result>
                    <process:hasEffect rdf:resource="#atomic3_effect1"/>
                </process:Result>
            </j.0:hasResult>
            <j.0:hasPrecondition rdf:resource="#atomic3_precond1"/>
            <j.0:serviceName>healthcare</j.0:serviceName>
        </j.0:Profile>
        <service:Service rdf:ID="DeliverService_service">
            <service:presents rdf:resource="#DeliverService_profile"/>
            <service:describedBy rdf:resource="#DeliverService"/>
        </service:Service>
    </rdf:RDF>
```

F. CompositeService.owl file content

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:expr="http://www.daml.org/services/owl-
s/1.1/generic/Expression.owl#"
    xmlns:process="http://www.daml.org/services/owl-
s/1.1/Process.owl#"
    xmlns:shadow-rdf="http://www.daml.org/services/owl-
s/1.1/generic/ObjectList.owl#"
    xmlns:service="http://www.daml.org/services/owl-
s/1.1/Service.owl#"
    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
    xmlns="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:j.0="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
    xml:base="http://localhost/compositeService.owl">
    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Service.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Profile.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl"/>
        <owl:imports rdf:resource="http://www.daml.org/services/owl-
s/1.1/Grounding.owl"/>
    </owl:Ontology>
    <expr:Expression rdf:ID="Composite1_effect1">
        <expr:expressionLanguage>
            <expr:LogicLanguage rdf:ID="PDDL">
                <expr:refURI>http://localhost/PDDL</expr:refURI>
            </expr:LogicLanguage>
        </expr:expressionLanguage>
        <expr:expressionBody>
         <and>
              <pred name="patient_hasaccount">
                   <param>patient0</param>
              </pred>
              <pred name="patient_ownsmedicine">
                   <param>patient0</param>
                   <param>medicine0</param>
              </pred>
              <pred name="bill">
                   <param>medicine0</param>
              </pred>
              <pred name="at">
                   <param>medicine0</param>
                   <param>location0</param>
              </pred>
              <not>
                   <pred name="wants_account">
```

```
                        <param>patient0</param>
                   </pred>
      </not>
          </and></expr:expressionBody>
     </expr:Expression>
     <process:Input rdf:ID="DeliverService_deliverservice_m">
         <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Medicine</process:parameterType>
     </process:Input>
     <process:Perform rdf:ID="RegisterPatientService">
         <process:process>
             <process:AtomicProcess
rdf:ID="RegisterPatientServiceProcess">

<process:name>RegisterPatientServiceProcess</process:name>
                <process:hasInput>
                    <process:Input
rdf:ID="RegisterPatientService_registerpatientservice_p">
                        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Patient</process:parameterType>
                    </process:Input>
                </process:hasInput>
                <process:hasPrecondition>
                    <expr:Condition rdf:ID="atomic1_precond1">
                        <expr:expressionLanguage
rdf:resource="#PDDL"/>
                        <expr:expressionBody>
         <and>
              <pred name="patient_hasvaliddata">
                   <param>patient</param>
              </pred>
              <pred name="wants_account">
                   <param>patient</param>
              </pred>
         </and></expr:expressionBody>
                    </expr:Condition>
                </process:hasPrecondition>
                <process:hasResult>
                    <process:Result>
                        <process:hasEffect>
                            <expr:Expression
rdf:ID="atomic1_effect1">
                                <expr:expressionLanguage
rdf:resource="#PDDL"/>
                                <expr:expressionBody>
         <and>
              <pred name="patient_hasaccount">
                   <param>patient</param>
              </pred>
              <not>
                   <pred name="wants_account">
                        <param>patient</param>
                   </pred>
                </not>       </and></expr:expressionBody>
                            </expr:Expression>
                        </process:hasEffect>
                    </process:Result>
                </process:hasResult>
            </process:AtomicProcess>
        </process:process>
        <process:hasDataFrom>
            <process:Binding>
                <process:toParam
rdf:resource="#RegisterPatientService_registerpatientservice_p"/>
                <process:valueSource>
                    <process:ValueOf>
                        <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                        <process:theVar>
                            <process:Input rdf:ID="patient0">
                                <process:parameterType rdf:datatype=

"http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Patient</process:parameterType>
                            </process:Input>
                        </process:theVar>
                    </process:ValueOf>
                </process:valueSource>
            </process:Binding>
        </process:hasDataFrom>
    </process:Perform>
    <process:Input
rdf:ID="SellMedicineService_sellmedicineservice_m">
        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Medicine</process:parameterType>
    </process:Input>
    <expr:Expression rdf:ID="atomic2_effect1">
        <expr:expressionLanguage rdf:resource="#PDDL"/>
        <expr:expressionBody>
         <and>
              <pred name="patient_ownsmedicine">
                   <param>patient</param>
                   <param>medicine</param>
              </pred>
              <pred name="bill">
                   <param>medicine</param>
              </pred>
              <pred name="at">
                   <param>medicine</param>
                   <param>location</param>
              </pred>
         </and></expr:expressionBody>
    </expr:Expression>
    <process:Input rdf:ID="location1">
        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
```

```
>http://localhost/OMSOntology.owl#Location</process:parameterType>
    </process:Input>
    <process:Perform rdf:ID="SellMedicineService">
        <process:process>
            <process:AtomicProcess
rdf:ID="SellMedicineServiceProcess">

<process:name>SellMedicineServiceProcess</process:name>
                <process:hasInput>
                    <process:Input
rdf:ID="SellMedicineService_sellmedicineservice_p">
                        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Patient</process:parameterType>
                    </process:Input>
                </process:hasInput>
                <process:hasInput>
                    <process:Input
rdf:ID="SellMedicineService_sellmedicineservice_s">
                        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Store</process:parameterType>
                    </process:Input>
                </process:hasInput>
                <process:hasInput>
                    <process:Input
rdf:ID="SellMedicineService_sellmedicineservice_from">
                        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Location</process:parameterType>
                    </process:Input>
                </process:hasInput>
                <process:hasInput
rdf:resource="#SellMedicineService_sellmedicineservice_m"/>
                <process:hasPrecondition>
                    <expr:Condition rdf:ID="atomic2_precond1">
                        <expr:expressionLanguage
rdf:resource="#PDDL"/>
                        <expr:expressionBody>
         <and>
              <pred name="patient_hasaccount">
                   <param>patient</param>
              </pred>
              <pred name="link">
                   <param>store</param>
                   <param>location</param>
              </pred>
              <pred name="link">
                   <param>medicine</param>
                   <param>store</param>
              </pred>
         </and></expr:expressionBody>
                    </expr:Condition>
                </process:hasPrecondition>
                <process:hasResult>
                    <process:Result>
                        <process:hasEffect
rdf:resource="#atomic2_effect1"/>
                    </process:Result>
                </process:hasResult>
            </process:AtomicProcess>
        </process:process>
        <process:hasDataFrom>
            <process:Binding>
                <process:toParam
rdf:resource="#SellMedicineService_sellmedicineservice_p"/>
                <process:valueSource>
                    <process:ValueOf>
                        <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                        <process:theVar rdf:resource="#patient0"/>
                    </process:ValueOf>
                </process:valueSource>
            </process:Binding>
        </process:hasDataFrom>
        <process:hasDataFrom>
            <process:Binding>
                <process:toParam
rdf:resource="#SellMedicineService_sellmedicineservice_m"/>
                <process:valueSource>
                    <process:ValueOf>
                        <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                        <process:theVar>
                            <process:Input rdf:ID="medicine0">
                                <process:parameterType rdf:datatype=

"http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Medicine</process:parameterType>
                            </process:Input>
                        </process:theVar>
                    </process:ValueOf>
                </process:valueSource>
            </process:Binding>
        </process:hasDataFrom>
        <process:hasDataFrom>
            <process:Binding>
                <process:toParam
rdf:resource="#SellMedicineService_sellmedicineservice_s"/>
                <process:valueSource>
                    <process:ValueOf>
                        <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                        <process:theVar>
                            <process:Input rdf:ID="store0">
                                <process:parameterType rdf:datatype=
"http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Store</process:parameterType>
                            </process:Input>
                        </process:theVar>
                    </process:ValueOf>
                </process:valueSource>
            </process:Binding>
        </process:hasDataFrom>
        <process:hasDataFrom>
            <process:Binding>
                <process:toParam
rdf:resource="#SellMedicineService_sellmedicineservice_from"/>
                <process:valueSource>
                    <process:ValueOf>
                        <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                        <process:theVar rdf:resource="#location1"/>
                    </process:ValueOf>
                </process:valueSource>
            </process:Binding>
        </process:hasDataFrom>
    </process:Perform>
    <expr:Condition rdf:ID="Composite1_precond1">
        <expr:expressionLanguage rdf:resource="#PDDL"/>
        <expr:expressionBody>
         <and>
              <pred name="patient_hasvaliddata">
                   <param>patient0</param>
              </pred>
              <pred name="wants_account">
                   <param>patient0</param>
              </pred>
              <pred name="link">
                   <param>store0</param>
                   <param>location1</param>
              </pred>
              <pred name="link">
                   <param>medicine0</param>
                   <param>store0</param>
              </pred>
              <pred name="link">
                   <param>patient0</param>
                   <param>location0</param>
              </pred>
         </and></expr:expressionBody>
    </expr:Condition>
    <process:Perform rdf:ID="DeliverService">
        <process:process>
            <process:AtomicProcess rdf:ID="DeliverServiceProcess">
                <process:name>DeliverServiceProcess</process:name>
                <process:hasInput
rdf:resource="#DeliverService_deliverservice_m"/>
                <process:hasInput>
```

```
                    <process:Input
rdf:ID="DeliverService_deliverservice_from">
                        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Location</process:parameterType>
                    </process:Input>
                </process:hasInput>
                <process:hasInput>
                    <process:Input
rdf:ID="DeliverService_deliverservice_p">
                        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Patient</process:parameterType>
                    </process:Input>
                </process:hasInput>
                <process:hasInput>
                    <process:Input
rdf:ID="DeliverService_deliverservice_to">
                        <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Location</process:parameterType>
                    </process:Input>
                </process:hasInput>
                <process:hasPrecondition>
                    <expr:Condition rdf:ID="atomic3_precond1">
                        <expr:expressionLanguage
rdf:resource="#PDDL"/>
                        <expr:expressionBody>
         <and>
              <pred name="at">
                   <param>medicine</param>
                   <param>location</param>
              </pred>
              <pred name="bill">
                   <param>medicine</param>
              </pred>
              <pred name="link">
                   <param>patient</param>
                   <param>location</param>
              </pred>
         </and></expr:expressionBody>
                    </expr:Condition>
                </process:hasPrecondition>
                <process:hasResult>
                    <process:Result>
                        <process:hasEffect>
                            <expr:Expression
rdf:ID="atomic3_effect1">
                                <expr:expressionLanguage
rdf:resource="#PDDL"/>
                                <expr:expressionBody>
         <and>
              <pred name="at">
                   <param>medicine</param>
```

```
                    <param>location</param>
               </pred>
               <not>
                    <pred name="at">
                         <param>medicine</param>
                         <param>location</param>
                    </pred>
               </not>       </and></expr:expressionBody>
                              </expr:Expression>
                         </process:hasEffect>
                     </process:Result>
                 </process:hasResult>
             </process:AtomicProcess>
         </process:process>
         <process:hasDataFrom>
             <process:Binding>
                 <process:toParam
rdf:resource="#DeliverService_deliverservice_m"/>
                 <process:valueSource>
                     <process:ValueOf>
                         <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                         <process:theVar rdf:resource="#medicine0"/>
                     </process:ValueOf>
                 </process:valueSource>
             </process:Binding>
         </process:hasDataFrom>
         <process:hasDataFrom>
             <process:Binding>
                 <process:toParam
rdf:resource="#DeliverService_deliverservice_p"/>
                 <process:valueSource>
                     <process:ValueOf>
                         <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                         <process:theVar rdf:resource="#patient0"/>
                     </process:ValueOf>
                 </process:valueSource>
             </process:Binding>
         </process:hasDataFrom>
         <process:hasDataFrom>
             <process:Binding>
                 <process:toParam
rdf:resource="#DeliverService_deliverservice_from"/>
                 <process:valueSource>
                     <process:ValueOf>
                         <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                         <process:theVar rdf:resource="#location1"/>
                     </process:ValueOf>
                 </process:valueSource>
             </process:Binding>
         </process:hasDataFrom>
         <process:hasDataFrom>
```

```
            <process:Binding>
                <process:toParam
rdf:resource="#DeliverService_deliverservice_to"/>
                <process:valueSource>
                    <process:ValueOf>
                        <process:fromProcess
rdf:resource="http://www.daml.org/services/owl-
s/1.1/Process.owl#TheParentPerform"/>
                        <process:theVar>
                            <process:Input rdf:ID="location0">
                                <process:parameterType rdf:datatype=

"http://www.w3.org/2001/XMLSchema#anyURI"

>http://localhost/OMSOntology.owl#Location</process:parameterType>
                            </process:Input>
                        </process:theVar>
                    </process:ValueOf>
                </process:valueSource>
            </process:Binding>
        </process:hasDataFrom>
    </process:Perform>
    <j.0:Profile rdf:ID="oms_initialontology_profile">
        <j.0:has_process>
            <process:CompositeProcess rdf:ID="Composite1">
                <process:composedOf>
                    <process:Sequence>
                        <process:components>
                            <process:ControlConstructList>
                                <shadow-rdf:first
rdf:resource="#RegisterPatientService"/>
                                <shadow-rdf:rest>
                                    <process:ControlConstructList>
                                        <shadow-rdf:first
rdf:resource="#SellMedicineService"/>
                                        <shadow-rdf:rest>

<process:ControlConstructList>
                                                <shadow-rdf:first
rdf:resource="#DeliverService"/>
                                                <shadow-rdf:rest
rdf:resource="http://www.daml.org/services/owl-
s/1.1/generic/ObjectList.owl#nil"/>

</process:ControlConstructList>
                                        </shadow-rdf:rest>
                                    </process:ControlConstructList>
                                </shadow-rdf:rest>
                            </process:ControlConstructList>
                        </process:components>
                    </process:Sequence>
                </process:composedOf>
                <process:hasInput rdf:resource="#patient0"/>
                <process:hasInput rdf:resource="#store0"/>
                <process:hasInput rdf:resource="#location1"/>
                <process:hasInput rdf:resource="#medicine0"/>
                <process:hasInput rdf:resource="#location0"/>
                <process:hasPrecondition
rdf:resource="#Composite1_precond1"/>
                <process:hasResult>
                    <process:Result>
                        <process:hasEffect
rdf:resource="#Composite1_effect1"/>
                    </process:Result>
                </process:hasResult>
            </process:CompositeProcess>
        </j.0:has_process>
        <j.0:hasResult>
            <process:Result>
                <process:hasEffect
rdf:resource="#Composite1_effect1"/>
            </process:Result>
        </j.0:hasResult>
        <j.0:hasPrecondition rdf:resource="#Composite1_precond1"/>
        <j.0:hasInput rdf:resource="#location1"/>
        <j.0:textDescription>This is the description of
theoms_initialontology profile</j.0:textDescription>
        <j.0:serviceName>oms_initialontology</j.0:serviceName>
        <j.0:hasInput rdf:resource="#location0"/>
        <j.0:hasInput rdf:resource="#medicine0"/>
        <j.0:hasInput rdf:resource="#patient0"/>
        <j.0:hasInput rdf:resource="#store0"/>
    </j.0:Profile>
    <service:Service rdf:ID="oms_initialontology_service">
        <service:presents
rdf:resource="#oms_initialontology_profile"/>
        <service:describedBy rdf:resource="#Composite1"/>
    </service:Service>
</rdf:RDF>
```