A Multi-Agent Architecture for Intelligent Gathering Systems

David Camacho [a,*] Ricardo Aler [a] and Daniel Borrajo [a] and José M. Molina [a]

[a] *Departamento de Informática, Universidad Carlos III de Madrid, Avenida de la Universidad nº 30, CP 28911, Leganés, Madrid, Spain*
E-mail: {dcamacho,dborrajo,molina}@ia.uc3m.es, aler@inf.uc3.es

This paper presents a model to define heterogeneous agents that solve problems by sharing the knowledge retrieved from the WEB and cooperating among them. The control structure of those agents is based on a general purpose Multi-Agent architecture (SKELETONAGENT) based on a deliberative approach. Any agent in the architecture is built by means of several interrelated modules: control module, language and communication module, skills modules, knowledge base, yellow pages, etc.... The control module uses an agenda to activate and coordinate the agent skills. This agenda handles actions from both the internal goals of the agent and from other agents in the environment. In the paper, we show a high level agent model, which is later instantiated to build a set of heterogeneous specialized agents. The paper describes how SKELETONAGENT has been used to implement different kinds of agents and a specialized Multi-Agent System (MAS). The implemented MAS, MAPWEB-ETOURISM, is the specific implementation of a general WEB gathering architecture, named MAPWEB, which extends SKELETONAGENT. MAPWEB has been designed to solve problems in WEB domains through the integration of information gathering and planning techniques. The MAPWEB-ETOURISM system has been applied to a specific WEB domain (e-tourism) which uses information gathered directly from several WEB sources (plane, train, and hotel companies) to solve travel problems. This paper shows how the proposed architecture allows to integrate the different agents tasks with AI techniques like planning to build a MAS which is able to gather and integrate information retrieved from the WEB to solve problems.

*Corresponding author: dcamacho@ia.uc3m.es

Keywords: Multi-Agent Systems, Agent Architectures, Information Gathering, WEB-based Systems

1. Introduction

The Intelligent Agents and Multi-Agent Systems (MAS) research fields have experimented a growing interest from different research communities like Artificial Intelligence (AI), Software Engineering, Psychology, etc... Those research fields try to solve two distinct goals. On the one hand, to define and design software programs (usually called agents) which implement several characteristics like autonomy, proactiveness, coordination, language communication, etc... This goal tries to obtain an adaptive and intelligent program which is able to provide the adequate request to the inputs received from the environment [22,26,42]. On the other hand, it is possible to coordinate several of those agents to build complex societies. When considering societies of agents, new issues arise, like social organization, cooperation, knowledge representation, coordination, or negotiation. In this situation it is possible to speak about Multi-Agent Systems and the previous problems can be studied within different perspectives [7,34,54]. Those research fields allow to test and simulate theoretical models and architectures in complex and real domains [13,51]. There is a wide range of different domains that can be used to test agent organizations like business management [25,41], robotics [8,37], the WEB [2,5,47], simulation of complex societies [16,45], etc...

A common point of interest for previous research fields are both how to define and design the individual agents that make up these systems [6,27,43,44] and how to coordinate and organize groups of agents [19,35]. Different arquitectures and models have been successfully designed, implemented and deployed in several domains and it is possible to learn from those experiences to

build other agent-based models that could be applied in new domains.

In this paper, we develop an agent and Multi-Agent architecture (SkeletonAgent) based on the CooperA and ABC^2 Multi-Agent architectures [37,49,50]. These models have been applied in several domains like robotics [37] or radar [40]. Our approach has extended the later models so that AI techniques (like Planning and Learning) and Web Information Gathering methods can be integrated into a MAS. Agents in SkeletonAgent use an agenda to store their pending goals. The agenda allows to combine actions from different sources, like the internal goals of the agent, requests from other agents, possible changes in the agent environment, etc. Then, a policy will select the most appropriate action in a particular moment. SkeletonAgent allows to decompose high level tasks that can be achieved by each agent in different ways, which is very appropriate for the domains we are interested in (and many others). For instance, a particular task could be solved by either looking in a database, asking another agent, browsing the web, etc. In SkeletonAgent, such high level tasks can be decomposed, and all the alternative subtasks will be stored in the agenda for (possible) parallel processing. Once one of them has achieved the high level task, the alternative medium level tasks, or their related low level tasks, will be removed from the agenda. We have extended the base model to take into account these issues.

With respect to systems that search and use information from the Web, the closest to our goals is the field of Information Gathering (IG), that intends to integrate a set of different information sources with the aim of querying them as if they were a single source [2,17,18,33]. However, these architectures do not intend to use AI techniques in a generic way, as we do, but only to select the appropriate Web sources or control the behaviour of agents.

In summary, our approach consists of a flexible and generic MAS architecture that can use AI and Web gathering techniques, by means of agenda-based agents. Besides describing the architecture, an important part of the paper deals with its instantiation into a Information Gathering system, named MAPWeb, and how it can be applied into a particular Web domain (MAPWebetourism). The paper is structured into the following sections: Section 2 presents some general characteristics about agenda-based agent models; Section 3 describes how we have instantiated the previous agent model into an specific agent and Multi-Agent architecture. Section 4 analyses how the previous architecture can be used to solve problems in the Information Gathering field. Using this architecture, a new system called MAPWeb, is defined, and an application example in a particular Web domain is shown (MAPWebetourism); Section 5 describes several related systems; and finally, Section 6 summarizes the conclusions of the research.

2. Agenda-based Agents

As explained before, our approach extends some ideas used by agenda based systems such as CooperA [49] and ABC^2 [37]. In this section we describe how agents are controlled by agendas in these systems.

2.1. Agent Model

In the CooperA and ABC^2, intelligent agents are defined as a knowledge structure implemented by a set of static and dynamic attributes as in [37]. The static attributes (A_S) are used to represent the agent as the only element with these characteristics in the system, whereas dynamic attributes (A_D) are used to represent those characteristics that change with time. These attributes represent the actual knowledge about the state of the world that the agent knows, so any agent can be represented as: $A = A_S \cup A_D$ where A_S and A_D represent the tuples of the static and dynamic attributes respectively.

The static attributes, A_S, are used to represent the specific knowledge about any agent. These are: name of the agent (N), the list of its skills (S), the knowledge about other agents in the system and their skills (yellow-pages, Y), the language used to communicate with other agents (L), the ontology that represents the knowledge managed by the agent in a particular domain (O), and the set of heuristic rules that govern the behaviour of the agent (H). Therefore, the static knowledge about any agent can be described as:

$$A_S =< N, S, Y, L, O, H >$$

On the other hand, the following dynamic attributes, A_D, are used to represent the current situation of the agent, which is stored in its agenda (A_g). The agenda contains the acts which are under consideration, the queues of messages (Q) received or pending to be sent, and the information (I) about the current state of the world, defined using the language L. Any agent in a given moment is defined by $A_D =< N, A_g, I, Q >$, and the dynamic situation of the whole team of agents as $A_D^+ =< N, A_g, I, Q >^+$, where N is the name of the agent representing each tuple.

For instance, the following example shows the previous (static and dynamic) attributes for an agent example and how they could be instantiated:

```
Example_Agent1 (static)=
        <N = ExAg1,
         S = {suspend,register, unregister,
              update,...skill_1n},
         Y = {(ExAg2: insert,delete,...),
              (ExAg3: planning,...)
              (ExAg_k: wrapper,
                       WebAccess...)},
         O = {problem-domain},
         H = heuristics-rules>

Example_Agent1 (dynamic)=
        <N = ExAg1,
         Ag= {[act1], [act_2]...[act_n]},
         I = {AGENTS.active: ExAg2, ExAg3,
                             ExAg_k
              AGENTS.new: ExAg_k+1
              AGENTS.suspend: none},
         Q = {[message1], ...}>
```

2.2. Agent Control Cycle

This section describes the control cycle of the agents that follow the model described in the previous section. It can also be seen in Figure 1, this figure illustrates the control algorithm, which can be summarized as:

- First, the control module checks the agenda periodically until the first act is inserted. When a particular task needs to be achieved, the initial act is inserted in the agenda. This act could be considered as the main goal or the initial goal of the agent. Other acts will be generated in order to achieve this initial goal. New acts could arrive from other agents at any moment.
- Once several acts have been inserted into the agenda, a *priority-Policy* is applied to sort them out (this evaluation happens at all cycles of the agent, because the priority of the acts change dynamically). The control module will *select* the act with the highest priority ($Act_i = max$; in Figure 1).
- When a particular act is selected, the control module *Evaluates()* this act and selects the skill that will perform the associated task ($Evaluate(Act_i)$ selects $Skill_s$). For instance, when an *act:Planning* is selected, the act and associated parameters are provided to the planning skill of the agent.
- Now, there are two possible situations:
  - The act needs information from other tasks and cannot be executed (the act is not *Ready()* yet for execution), so it will be inserted newly in the agenda (its priority will be increased in the next cycle: $Increment_{priority}(Act_i)$).
  - The act is *Ready()* for execution (the act does not need any other information). However, it is necessary to take into account two new situations to know if the act is directly *Executable()*:
    - If the act can be decomposed into simpler acts, the skill will expand this act into them, assigning new priorities to these acts, and finally adding them into the agenda ($\forall_{j=1}^{n} Act_{ij}.priority()$).
    - If the act is directly executable (all the information is available and it is not possible to expand), it has provided to the correspondent *Execute()* function associated to this skill.

This control cycle continues while the agenda contains acts. If the agents have their agendas empty, they will wait for new tasks to perform.

This algorithm integrates the agenda, the heuristics and the available skills as related modules to implement different behaviours in the agents. This approach is very flexible because it is possible to modify the behaviour of the agent, or the way to achieve a goal by just modifying one or several of the following characteristics of acts:

- The *priority* of a particular act can be modified by the control module or by the skill. The

```
Initialize(Agenda);
while (Agenda ≠ 0)
   (∀Act_i), calculate Priority(Act_i) using a Control-Policy;
   Sort the acts in the Agenda by Priority;
   Remove(Act_i) such as Act_i = max;
   Evaluate(Act_i) selects Skill_s
   if Skill_s(Act_i).Ready() then
      if not Skill_s(Act_i).Executable() then
         Skill_s(Act_i).expand();
         ∀_{j=1}^{n} Act_ij.priority();
         ∀_{j=1}^{n} Insert(Act_ij);
      else
         Execute(Act_i);
   else
      Increment_priority(Act_i);
      Insert(Act_i);
```

Fig. 1. Agent control cycle in SkeletonAgent using the ControlPrio policy.

priorities of the acts can be redefined by using rules or policies, thus changing the global solving algorithm performed by the agents.
- The *evaluation* of an act determines which skill will execute the act. If no skill is available, the control module will generate a fail.
- Acts in the agenda are *selected* by means of policies. By changing the policy it is possible to change the behaviour of the agent.

3. SkeletonAgent

In this section we describe how we have instantiated the general model shown in the previous section and also our extensions to that model. We address both the agent and the Multi-Agent architecture.

3.1. Agent Architecture

This subsection defines the agent architecture of SkeletonAgent, which is based on the agent model described in the previous section. Agents in SkeletonAgent are composed of several modules. Figure 2 shows those modules and their interconnections.

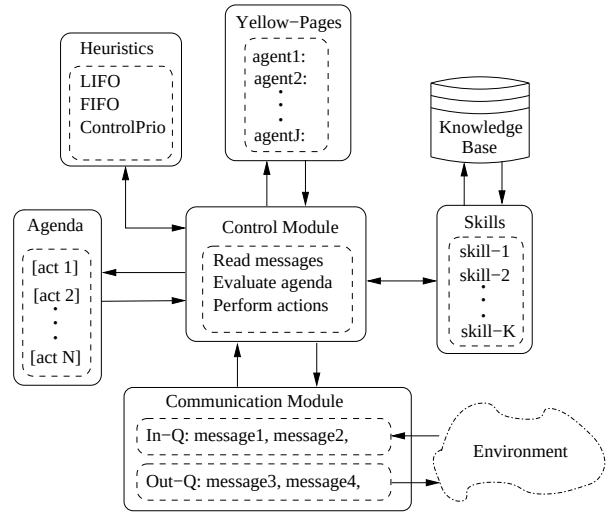

Fig. 2. SkeletonAgent Architecture of an agent.

3.1.1. Agenda.

The Agenda is a dynamic structure (A_g) that stores items named *acts* [14,15]. These acts represent the actions that the agent is considering at a given moment. The agents implemented using SkeletonAgent architecture share a standard communication language (Kqml [20,21]) to perform actions over their environment. A message in Kqml is called *performative* (this term is from the speech act theory [15]) and can be understood like a request for an specific action to be carried out. Any performative can be translated into one

or several acts that could be performed by the receiver agent. When any agent receives a performative, its related act is inserted into the agenda. Once this act is selected, its execution could generate new acts that will be recursively added to the agenda. These performatives are used by different agents to implement several acts like:

- (*achieve, tell*): are used by different agents to require the **execution** of a specific task and to answer with the obtained results. When any agent receives *achieve*, the related act contains the type of skill that will be necessary. For instance, when a planning agent receives an achieve performative to solve a problem, several skills can be used: *planning*, *case-based planning*, or *ask-others*. When this same performative is received by a WEB agent other skills like: *caching*, or *access-Web* could be used by the agent to execute the act.
- (*insert, delete*): are used to insert or delete a specific fact (from the sender agent) in the Knowledge Base of the receiver agent. For instance if an agent is temporarily unavailable in the society, and this fact is known by the manager agent, it is necessary to delete it from the yellow pages of all agents.
- (*register, unregister*): are used by the control-agents in the Multi-Agent systems to manage the insertion and deletion of the agents in the society.
- (*ok, ping*): are used by the control-agents when it is necessary to know the state of a particular agent.
- (*wake-up, sleep*): are used by different agents to activate or suspend temporarily their functions.
- (*request-info, tell-info, finish*): are used among agents to implement a protocol that allows them to send all the desired information in several steps.

The parameters of the different deployed acts for the agents can be summarized as:

- The *type* of the performative (achieve, insert, delete, ok,...). Those performatives are translated into acts that the agent will try to achieve. When these acts are handled, some of them should be expanded to allow their execution (see *Skills*).
- A list of *skills* that could be used by the agent to perform the task.
- The *identification code* (ID-code) of the act, that is used to relate the performative that has been executed with the possible subtasks that could be originated to perform the task. This code is used to build the answer to the performative. For instance, if a performative to solve a general problem is inserted in the agenda, when this act is extracted from the agenda, the skill will generate several new acts to achieve its goal, these new acts are inserted into the agenda and their ID-codes are built using the previous general code. So, if one of the acts fail, all the related acts are deleted from the agenda.
- *TimeStamp, TimeOut*. Any act has two temporal parameters: the *TimeOut* that represents the maximum time to execute the task, and the *TimeStamp* that represents when this act has being inserted in the agenda. When any act is inserted in the agenda, it is possible to use these parameters to:
  - ∗ know how much time it has been stored the act in the agenda, and
  - ∗ modify dynamically the priority of any act using a simple expression.

  If an act cannot be executed within the *TimeOut*, it will be removed from the agenda. The control module will remove any other act that needs the results of, or that could be related to, the rejected act.
- *Priority* of the act. This priority has a default (initial) value which is assigned depending of the type of the act. The priority is modified by the control module using other parameters (*TimeStamp* and *TimeOut*). For instance, the performative *ok* has the maximum priority because it is used for the sender agent to know if the receiver is working, so this performative has a lower TimeOut and a high priority value. This priority will be automatically increased by the control module in any cycle (see Section 2.2) to avoid that an act would never be executed. When a skill expands a complex act into subacts, the new acts will have the same values for the *TimeStamp* and *TimeOut* parameters. The skill will assign different values to the subtasks to select the order to execute them.

The set of possible acts that can be used by the agents and the heuristic used to schedule how these

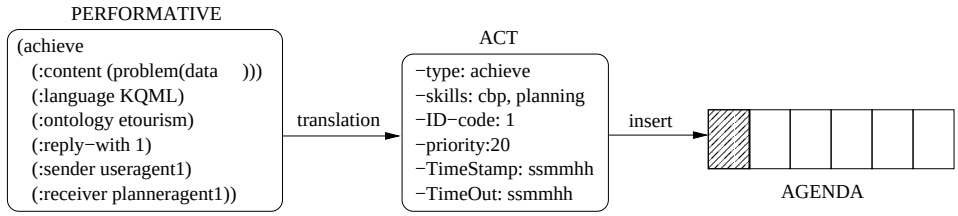

Fig. 3. Relation between a performative and its translated act.

acts are selected and executed in the agenda (see *Heuristics*) defines the behaviour of the agents. Section 2.2 describes in detail how the agenda works. Figure 3 shows the relations between an achieve performative sent by an UserAgent to request for the solution of a problem to the receiver PlannerAgent.

3.1.2. Heuristics.

Agents have a set of heuristics that are used to decide at any time what act to select from the agenda. Actually, any agent can be implemented using in its control module three different types of heuristics (*LIFO, FIFO and ControlPrio*). The first two heuristics correspond to the LIFO (Last In First Out) and FIFO (First In First Out) policies. These heuristics can be used when it is not necessary to select the acts in a particular order and allow to test the agents with simple behaviours. However, these heuristics present several problems when it is necessary to apply a priority in the execution of the acts, because there could be acts with high priority that need to be executed quickly. For this reason a simple priority control heuristic (*ControlPrio*) was implemented. This heuristic is the default behaviour for a given agent in SkeletonAgent. In the future, we plan to use other techniques like fuzzy logic or machine learning to implement more sophisticated heuristics to provide other behaviours for the agents.

3.1.3. Skills.

As explained before, an agenda contains acts. Some of them can be decomposed into lower level acts, which are subsequently introduced into the agenda. When a particular act (atomic, or low level) cannot be decomposed further, it will be executed by the agent. Those executable acts are actually the skills S_i of the agent. Automatic access to the Web, or executing a planner are examples of skills. Figure 4 shows how a high level act (to solve a planning problem) can be decomposed into several subacts, taking into account that the PlannerAgent who receives the problem has several skills to obtain solutions for planning problems. From the point of view of decomposition, it is useful to divide acts into two types: AND acts and OR acts. AND acts require all its subacts to finish successfully whereas OR acts need only one of them to end. For instance, *act:WebAgentsCooperation* of Figure 4 is an AND act because it needs its three subacts to end successfully. On the other hand, *act:Solve-Planning-Problem* will end with success if any of its subacts (*act:SearchPlanBase*, *act:Planning*, ...) obtains a solution to the planning problem.

Figure 5 displays an example of a high level act decomposition. When the OR-type *act:Solve-Planning-Problem* is selected by the control module, it is expanded into its component subacts *act:cbp* (which allows to search for solutions previously stored in a Plan Base) and *act:Planning* (which uses a planner to obtain solutions). These new subacts acquire the characteristics (like the *TimeOut* or *TimeStamp* values) of *act:Solve-Planning-Problem*. Different subacts may have different priorities. Although in this case, *act:cbp* has a higher priority than *act:Planning*, this only means that it will be selected first. But once it is selected, *act:cbp* will be executed in a separated process, and then *act:Solve-Planning-Problem* will be executed in another process (i.e. they will actually work in parallel). If desired, it is also possible that both subacts work in sequence: execute the first subact, and if it fails, execute the second one.

3.1.4. Knowledge Base.

This module stores knowledge that can be used by the agent skills. For instance, in MAPWeb a set of agents specialized in planning (PlannerAgents) are used in the team. These agents need to use several knowledge sources to achieve their main goal: to solve a problem using planning. The knowledge used by these agents is: a description of the problem to be solved, the operations (*oper-*

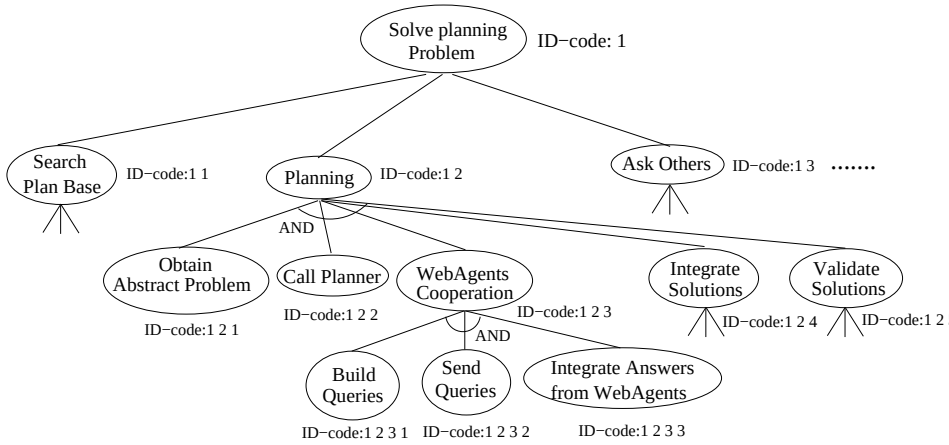

Fig. 4. Expansion of an initial act into several subacts.

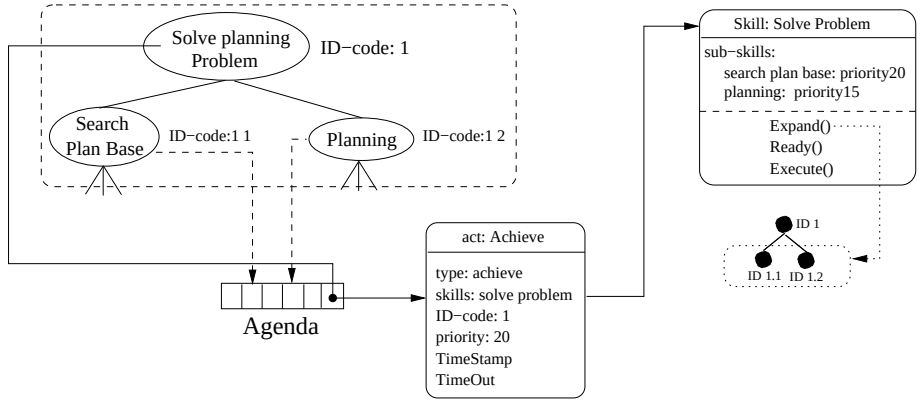

Fig. 5. Relations between the selected act from the agenda (to solve a planning problem) and the skill which is able to perform the task.

ators) that can be used (represented in a *Domain* description), several domain-dependent heuristics, etc. . .

3.1.5. Yellow Pages.

This module stores the knowledge that an agent has about all the agents belonging to its team (T_i). This information consists of a list made by the name of its partners (N), and the name of the skills they can accomplish (S_i). Any skill can be considered as an abstraction of an action that will be accessible to other agents in the team. In fact, it means that the agent has meta-knowledge about itself (through its skills definition) and its partners (using the yellow-pages). Although, SkeletonAgent has standard yellow pages which are initially used by all the agents, there exist other specialized agents that use an extended version of the yellow pages. For instance, the *ManagerAgent* in the MAPWeb-etourism application uses more information about the agents like what is the team that has been used to allocate a particular agent.

3.1.6. Communication Module.

Agents in SkeletonAgent use a communication language to share information. It is the control module which decides to send a message and also the module which receives messages from other agents. Once the control module has received a message, it can be distributed to any other module in the agent. The communication module hides the physical network transport layer from the specific language (Kqml) that is used by the agent. In order to do so, the communication module encapsulates the message, adding information such as *sender*, *receiver*, *performative*, . . . , as shown in Figure 6.

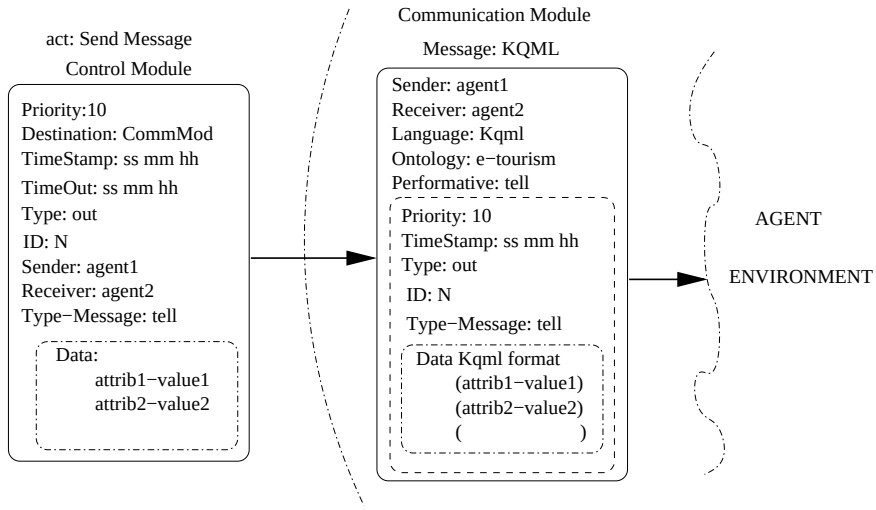

Fig. 6. SkeletonAgent Internal Data Communication.

The communication process is implemented by several interconnected submodules that allow to transform the data correctly. They are:

- The *Manager Communication Layer*. It inserts the messages received in the input queue (*In-Q*) or in the output queue (*Out-Q*) depending on their destination.
- The *Language Layer*. This module translates the internal data into an standard Kqml message or vice versa.
- The *Communication Layer*. It is responsible to serialize and deserialize the information, so that it can be sent (received) through (from) the network.

Figure 7 shows how the serialized messages received by an agent are properly translated into the internal communication data structure before it is sent to the control module of the agent. This design allows to change the technology used to send the messages (i.e. change TCP/IP to RMI) or the language used by the agents (i.e. Kqml or FIPA-ACL) in an independent way.

3.2. Muti-Agent Model

In this section, how to build societies of agents by means of the agent model outlined in the previous section will be described. Any MAS defined using SkeletonAgent can be implemented using one or several teams (T_i). Every team is managed by a specialized agent named CoachAgent (CCH). To manage the different teams it is necessary to use a single ManagerAgent (MNG). This agent (known as the Agent Name Server, ANS, in other architectures) is used to manage the insertion and deletion of other agents in/from the MAS, or which is the selected team that will use the new agent. Finally, it is possible to represent the whole Multi-Agent system (S_{kel}) as:

$$S_{kel} = MNG + \bigcup_{i=1}^{m} T_i$$

Where m represents the available operative teams in the system. Any *team* (T) is made of at least one agent and can be represented as:

$$T = \bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup ... \cup A_n = [A_i \equiv < N_i, S_i, Y_i, L_i, O_i, H_i > \cup < N_i, A_g, I_i, Q_i >]^+$$

Where n represents the number of agents that compose the team. In SkeletonAgent (like other MAS) it is necessary to use a minimun number (and types) of agents to build an operative team. We define an "operative team" as the minimun set of agents which is able to achieve the goal or goals that the system was designed for. These minimun number of agents may change for every domain. For instance, the real implementation of MAPWeb-etourism teams need at least five types of agents (manager agent, coach agents, user agents, planner agents and Web agents) to be an operative group. Other MAS implemented from SkeletonAgent, like SimpleNews, requires only four types of agents (manager, coach, user and Web agents) [10].

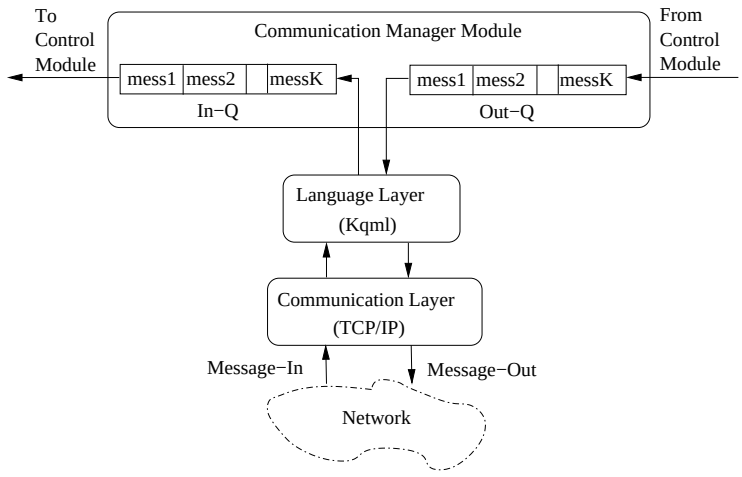

Fig. 7. SkeletonAgent Communication Module.

3.3. Multi-Agent Architecture

Using the previous MAS model, any system implemented from our architecture will need at least the following agents to work properly:

- *Control Agents*. They manage the different agents in the system. There are two types of them:
  - ManagerAgent (MNG). This agent is similar to any ANS with can perform the following roles in the system:
    - It is responsible to add and remove other agents from the system.
    - It controls which agents are active in the agent society.
    - It groups agents in teams. To do this, when any agent requests to be inserted in the society, the MNG determines which teams require this agent.
  - CoachAgent (CCH). They control a team of agents, guaranteeing stability and smooth operation of the active agents. Those agents perform the following roles:
    - They report problems to the MNG. For instance, when a new agent is required for the team.
    - They guarantee that the yellow pages of the team members are coherent.
- *Execution Agents*. These agents are responsible to achieve the differents goals of the system. To coordinate different teams of agents it is possible to include a new skill in the control module of the agents. Currently, three different execution agents have been implemented: planner agents, user agents, and Web agents.

Figure 8 shows the general architecture for any SkeletonAgent MAS. The main characteristics for any MAS implemented in this way can be summarized in:

- Agents in the system use message-passing to communicate with other agents.
- All the agents have the same architecture and they are specialized in different tasks through the implementation of different skills.
- Although the communication language is the same for all the agents in SkeletonAgent, it is possible to distinguish two different types of communication messages. On the one hand, there are control messages whose main goal is to manage the behaviour of the system. On the other hand, execution messages are used to share knowledge and tasks among the agents, to achieve desired goals.

To start correctly the MAS, it is necessary to perform the following steps, as it is shown in Figure 9 in more detail:

1. First, the MNG is executed.
2. Agents in the system need to register themselves to the MNG. Once a CCH has registered, the MNG will select the necessary execution agents from its white pages and will build an operative team. If there are not enough agents, the CCH will wait for them.

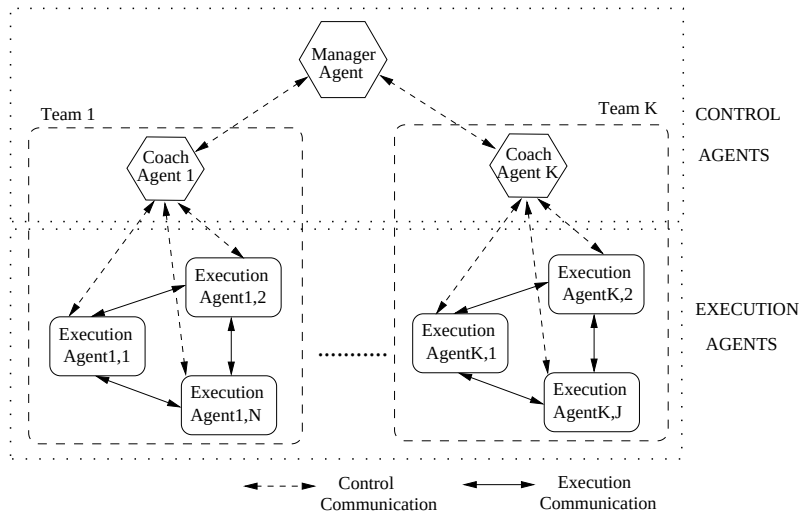

Fig. 8. SkeletonAgent Multi-Agent Architecture.

To build a team the MNG selects the execution agents and provides the necessary information to the CCH. Once the information of the agents has been stored in CCH's yellow pages, it updates the yellow pages of its execution agents. To select the necessary agents to build a group, the MNG uses the Ontology of the CCH agent.

3. Once a team is built, the execution agents can only communicate with the agents belonging to its team or with its CCH.

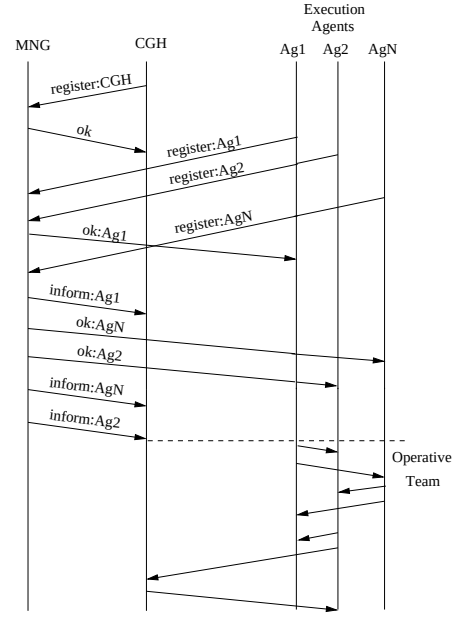

Fig. 9. Operative team building process for SkeletonAgent .

4. MAPWeb: Multi-Agent Planning in the Web

MAPWeb [11,12] (**M**ulti-**A**gent **P**lanning in the **Web**) is a generic MAS information gathering architecture which has been implemented using the SkeletonAgent architecture. MAPWeb is a Multi-Agent framework that integrates planning and Web information gathering (IG) agents. This framework provides a reusable code to help with the development of new Web gathering systems. The main goal of this framework is to deal with problems that require to integrate planning with information gathered from the Web.

4.1. Information Gathering in the Web

Unlike traditional Information Retrieval (IR) techniques, IG systems extract knowledge from documents stored in the Web by taking advantage of their inner structure [3,18,33,36]. IG systems like SIMS [4] use information retrieved from relational databases but other IG systems, like Heracles [2,29], can use other kinds of sources that provide the information into a semi-structured way (the useful information is stored inside the retrieved document and it is necessary to extract or filter the information previously). Several problems arise when Web IG systems are designed and implemented:

1. An IG system needs to select, access and filter the information from the appropriate sources, and finally, reason with the gathered knowledge to build a solution.
2. IG systems have to deal with multiple, distributed, and heterogeneous WEB repositories. WEB sources can be heterogeneous in both content and format. Besides, the number and types of those repositories grow over time.
3. WEB servers can be down at some times.
4. The IG system might find a large number of solutions, so it is necessary to manage this overload to provide them in a comprehensive way to the user.

MAPWEB implements a generic IG WEB architecture which deals with the previous problems using Multi-Agent techniques to allow the cooperation and coordination among heterogeneous agents (specialized in different taks) and planning techniques to integrate information managed by those agents.

4.2. MAPWEB *IG Architecture*

Figure 10 shows one possible MAPWEB topology, or configuration. This configuration is built by two operative teams managed by a MNG, and every team is locally managed by a CCH. $Team_1$ has the minimun set of agents to be operative, whereas $Team_2$ is built by K UserAgents, P PlannerAgents and J WebAgents. In addition, the following execution agents are needed:

- UserAgents (UA) are the bridge between the users and the system. They only implement basic input/output skills to acquire problem descriptions from users and to show the solutions found to them.
- PlannerAgents (PLN) are able to solve planning problems using the information gathered from the WEB.
- WebAgents (WA) are able to provide the requested WEB information like a set of relational records to the PlannerAgents using wrapping techniques. These agents use caching techniques to optimize the number of accesses to the WEB [12].

Following the terminology explained in previous sections, MAPWEB can be described as follows. For the first team (T_1) the static attributes of the control agent ($CoachAgent_1 : CCH_1$), and the specialized planner agent ($PlannerAgent_{11} : PLN_{11}$) are:

```
CoachAgent1 (CCH1)=
        <N = CoachAgent1,
         S = {suspend,update,...skill_1n},
         Y = {(ManagerAgent: insert,
                             delete,...),
              (PlannerAgent11:planning,...)
              (WebAgent11: wrapper,
                           WebAccess...)},
         L = {Kqml},
         O = {e-tourism},
         H = hr-agenda>

PlannerAgent1,1 (PLN11)=
        <N = PlannerAgent11,
         S = {planning,...skill_1j},
         Y = {(CoachAgent1: insert,
                            delete,...),
              (WebAgent11: wrapper,
                           WebAccess...)},
         L = {Kqml},
         O = {e-tourism},
         H = hr-agenda>
```

Where $S = \{suspend, ..., skill_{1n}\}$ and $S = \{$planning, ..., $skill_{1j}\}$ represent the available skills of both agents. For instance, the $skill_{1n}$ implemented by $CoachAgent_1$ could be the delete-item skill. This skill allows to delete a particular item from the yellow pages. This skill is necessary if agents can be temporally unavailable. In that case, they will be removed from the yellow pages of same-team agents to avoid communication problems. On the other hand, the $skill_{1j}$ ($PlannerAgent_{11}$) can implement the search-plans skill that allows the agent to search for old stored plans in its local plan-base. The information that the $CoachAgent_1$ has about the j skills of $PlannerAgent_{11}$ is represented using a list which contains the names of the skills of that agent ($Y = \{(AgentName : skill_1, skill_2, ..., skill_n), (.....)\}$). Then the description of the environment is given. This information is described using the ontology O. For instance, the e-tourism ontology to represent travel problems, the domain-dependent heuristics, etc ... Finally, different sets of heuristic rules can be defined for

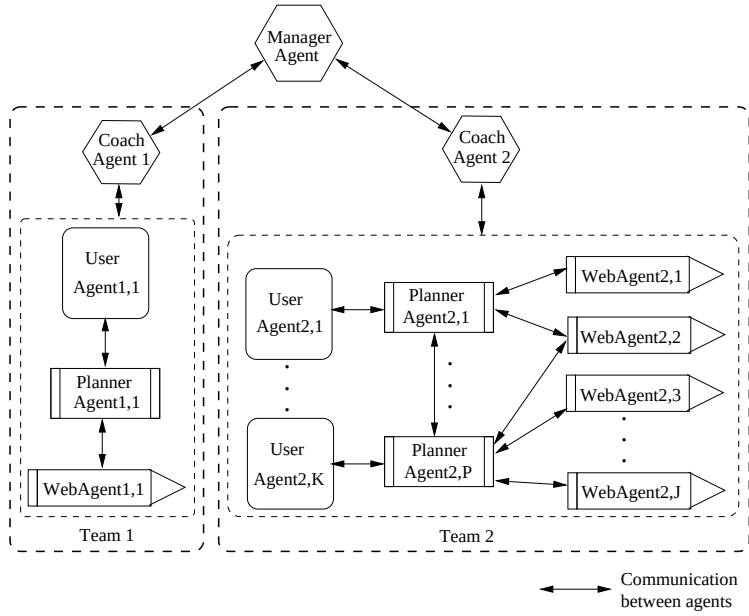

Fig. 10. MAPWeb Architecture.

controlling the agent behaviour. In the previous example the $PlannerAgent_{11}$ shows two different set of heuristics. The $hr - agenda$ is used for controlling the execution of the acts inserted in the agenda. These heuristics are actually implemented like a set of control policies which can be used to modify the control cycle. On the other hand, the dynamic attributes, can be described as:

```
CoachAgent1 (CCH1)=
  <N = CoachAgent1,
   Ag= {[ACHIEVE:insert,WebAgent12,...]
        [act_2]},
   I = {AGENTS.active: ManagerAgent,
                       PlannerAgent11,
                       WebAgent11
        AGENTS.new: WebAgent12
        AGENTS.suspend: none},
   Q = {[REQUESTED: achieve:insert,
         ManagerAgent]}>

PlannerAgent1,1 (PLN11)=
  <N = PlannerAgent11,
   Ag= {[ACHIEVE:solve-problem]...[act_n]},
   I = {obj_1.attribute1,...,
        obj_n.attribute1n},
   Q = {[null]}>
```

This dynamic information, A_D, means that the $CoachAgent_1$ has only one act in the agenda (to insert a new agent in the team: $WebAgent_{12}$). This act is to be performed with the skill *insert*. The information that $CoachAgent_1$ has about its environment is basically which other agents in the team are active or suspended. Finally, the queue of messages shows that the $ManagerAgent$ has requested the $CoachAgent_1$ to perform the skill *insert* (the execution of this skill will allow to include $WebAgent_{12}$ as a new agent member in the team).

4.3. Implemented Skills

Although MAPWeb is a generic architecture that combines Web information retrieval with planning, its skills are better understood in a particular domain. In this section, we will use the e-tourism domain, where the goal is to assist a user in planning his/her trips. This domain will be described in detail in Section 4.4.

MAPWeb's process can be described as follows. First, the user interacts with the UserAgent to input his/her query. The query captures information like the departure and return dates and cities, one way or return trip, maximum number of transfers, and some preference criteria. This information is sent to the PlannerAgent, which transforms it into a planning problem. This planning problem retains only those parts that are essential for the plan-

ning process, which is named the *abstract representation* of the user query. Then, the agent generates several abstract solutions for the user query. The planning steps in the abstract solutions require to be completed and validated with actual information which is retrieved from the WEB. To accomplish this, the PlannerAgent sends information queries to specialized WebAgents, that return several records for every information query. Then, the PlannerAgent integrates and validates the solutions and returns the data to the UserAgent, which in turn displays it to the user.

In order to achieve the previous process, several skills have been implemented for the PlannerAgent and the WebAgents. Figure 11 shows how these skills can be decomposed into subskills.

– *PlannerAgent skills.* This agent has a general skill $Solve - Problem$ which is used to begin the problem solving task. When the act associated to this skill is selected, it is decomposed into three subacts, associated to three skills: $Search - Plan$, $Ask - Others$, and $Planning$. Any of these skills might provide a solution for the planning problem, so $Solve - Problem$ is an OR act. The idea behind these three subacts is that if a relevant plan can be located in the local database ($Search - Plan$ skill), then no actual planning has to be carried out. Otherwise, the agent can ask other PlannerAgents to provide solutions ($Ask - Others$ skill). Finally, the planning skill inside the agent can also provide a solution ($Planning$). These three subacts can be executed in parallel, and removed once one of them returns an answer.
When selected, the $Planning$ act will be decomposed into three subacts, although in this case all of them are required to finish successfully (AND act). These three acts are: $Obtain - Abstract - Representation$ (which obtains a general representation of the problem), $Call - Planner$ (to execute a planner (PRODIGY4.0) which is able to obtain plans for the given problem), and $Cooperate - WebAgents$ (which allows to request information to the WebAgents available in the team, so that the plans can be completed with WEB information).
Other PlannerAgent skills are: $Integrate - Solutions$ (which is used to build a set of specific solutions for the given problem once the WebAgents have answered), and $Validate - Solutions$ (which is used to filter those solutions for which every step in the plan can be actually executed). For instance, flying by plane from city A to city B can only be executed if there is at least one company that flies between those cities). And finally the skill $Send - Solution$ sends to the requesting agent the solution(s) found. Figure 11 shows the representation of some of the skills defined in the PlannerAgents.

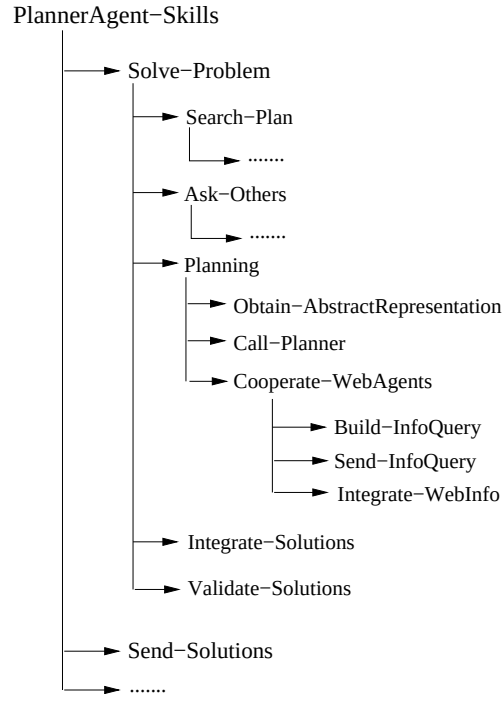

Fig. 11. Skills hierarchy for a PlannerAgent in MAPWEB.

– *WebAgent skills.* The $Search - Record$ skill is used by these agents to look for stored records in their local databases, that were previously gathered by the agents. Otherwise, the skill *Gather-WebInfor* is then used to automatically access, retrieve, and filter the requested information from the WEB source. The skill $Send - Solutions$ is used to communicate to the PlannerAgents the records found.

4.4. MAPWEB-ETOURISM

In this section, we will instantiate MAPWEB in the e-tourism WEB domain. The resulting system, MAPWEB-ETOURISM, is able to assist a

user in planning his/her trips. The e-tourism domain is a modified version of the Logistics domain [52], where the user needs to find a plan to travel between several places. We have selected this domain because it requires different and heterogeneous WEB sources (like WEB sources for plane, train, hotels, taxi, ... companies). Currently, MAPWEB-ETOURISM is able to access, gather, and reuse information from the following WEB sources:

- Flight companies: Iberia Airlines, Avianca Airlines, Amadeus flights, 4Airlines flights.
- Train companies: Renfe, BritRail, RailEurope.
- Rental Car companies: Avis, Hertz, Amadeus car rental, 4Airlines car rental.
- Hotel companies: Amadeus hotels, 4Airlines hotels.

The previous WEB sources can be classified into two main groups: Metasearch systems like Amadeus, 4Airlines, BritRail, and RailEurope which extract information from several companies, and individual sources which belong to a particular company (Iberia, Avianca, Renfe, ...).

To describe the instantiation of MAPWEB in this domain, it will only be necessary to detail how the execution agents have been specialized from the MAPWEB IG architecture:

- UserAgents. Graphical interfaces were implemented to allow the user to interact with the system.
- PlannerAgents. They were supplied a description of the planning domain (e-tourism) and related information. Also, they were given a new skill: a simple Case-Based planning technique was implemented to improve performance (see [11,12] for more details).
- WebAgents. For every WEB source described previously, an specialized agent has been implemented. They use wrappers [32,46] to access and retrieve the WEB information.

A topology has been implemented using an isolated team composed of one CoachAgent, one UserAgent, one PlannerAgent and four WebAgents. Formally, the topology is:

$$MAPWeb - etourism = MNG \cup T = MNG \cup [CCH_1, UA_1, PLN_1, WA_{Amadeus}, WA_{RailEurope}, WA_{Iberia}, WA_{Renfe}]$$

Let us suppose that a user gives the problem described in Table 1 to the UserAgent (UA_1). The problem consists of a sequence of stages. Each stage is a template that represents a leg of the trip, and contains several fields to be filled by the user. The solving process starts when the UserAgent supplies the problem to the PlannerAgent (PLN_1).

From the point of view of acts inserted into the agenda of the PlannerAgent PLN_1, the sequence to solve the user problem is as follows:

- Initially the problem is received by PLN_1 from UA_1 and the act $[ACHIEVE : Solve - Problem]$ is inserted into the agenda. Then, the act is selected.
- The previous act is expanded: $[ACHIEVE : Search - Plan]$, $[ACHIEVE : Planning]$, $[ACHIEVE : Integrate - Solutions]$, and $[ACHIEVE : Validate - Solutions]$ are inserted into the agenda. The first act is selected because it has been given the highest priority.
- If no solution is found (see Figure 5), i.e. the act $[ACHIEVE : Search - Plan]$ failed, then the act $[ACHIEVE : Planning]$ is selected (because it is the next one in priority).
- Then, it will be expanded into $[ACHIEVE : Obtain - Abstract - Representation]$, $[ACHIEVE : Call - Planner]$, and $[ACHIEVE : Cooperate - WebAgents]$.
- Once the act $[ACHIEVE : Obtain - Abstract - Representation]$ has finished, the agent starts looking for new plans using its planning skill (i.e. executes $[ACHIEVE : Call - Planner]$). When the first possible abstract plan is found, the act $[ACHIEVE : Cooperate - WebAgents]$ will be $Ready()$ for execution. Then, a new act $[ACHIEVE : Cooperate - WebAgents]$ will be inserted for every abstract plan found.
- When selected, the act $[ACHIEVE : Cooperate - WebAgents]$ it is expanded into several acts: $[ACHIEVE : Build - InfoQuery]$, $[ACHIEVE : Send - InfoQuery]$, and $[ACHIEVE : Integrate - WebInfo]$.
- When the information queries are built ($[ACHIEVE : Build - InfoQuery]$) the agent uses its yellow pages to select the appropriate WebAgents. Then the act $[ACHIEVE : Send - InfoQuery]$ is executed.

Table 1
Travel problem example to go from Salzburg to Segovia by airplane or train.

Leg	Stage	Date	Transport	Restrictions	Nº Transfers
1	Salzburg → Madrid	May 11th	Plane or train	none	0 or 1
2	2 nights stay	May 11-13th	none	$<$ 100 euros	-
3	Madrid → Segovia	May 13th	Plane or train	none	0 or 1
4	3 nights stay	May 13-16th	none	$<$ 90 euros	-
5	Segovia → Salzburg	May 16th	Plane or train	none	0 or 1

– The answers from the WebAgents are inserted into the PlannerAgent PLN_1 agenda as acts: $[TELL : WA_{Iberia}, information...]$, $[TELL : WA_{Renfe}, null]$, ... Then, an $[ACHIEVE : Integrate - WebInfo]$ will be inserted into the agenda.
– Next, if there is an $[ACHIEVE : Integrate - WebInfo]$ act in the agenda, the appropriate TELL acts will be extracted from the agenda and integrated into the abstract plan.
– Finally the acts $[ACHIEVE : Integrate - Solutions]$, and $[ACHIEVE : Validate - Solutions]$, will be selected, and the final solutions will be constructed.
– A new act $[TELL : Send - Solutions]$ is inserted into the agenda.The execution of this act sends the solutions found to UA_1.

It is important to remark that requests and answers from the agents arrive to the agenda asynchronously. That is, agents do not know when they will be queried or when the expected information will be provided. For instance, while the planner is generating new abstract plans, the PlannerAgent is able to work on other acts in the agenda, like completing previously generated abstracts plans. If one of the acts fails, all the dependent acts will be automatically removed from the agenda. For instance, for leg 1 of the example trip: "travel from Salzburg to Madrid using plane or train", one possible (abstract) solution could be travelling by train directly from the departure city to the arrival city. To achieve this, several acts are inserted so that the information is requested to the WebAgents. At the same time, the planner could find other abstract solutions, like travelling by train from Salzburg to Madrid using one transfer, or travelling directly by plane, etc. Likely, not all the abstract solutions will be possible. For instance, in leg 5: "travel from Segovia to Salzburg", it is not possible to find a solution which uses a plane from Segovia, because this city does not have an airport. On the other hand, the abstract solution that uses a train to travel from Segovia to Madrid, and then take an airplane to Salzburg is feasible.

The previous example used a simple configuration of MAPWeb-etourism. However, it is possible to use many UserAgents, PlannerAgents, and WebAgents in the same team. In that case, a single agent could receive many requests and answers from many different agents at the same time. But the agenda would still work in exactly the same way, which shows the flexibility of this architecture for coordinating agents.

5. Related Work

The aim of this section is to describe the ideas related to the two main issues of this work: agenda architectures and Web information systems. With respect to agenda-based systems, the most closely related to our work are the CooperA [48,50] and ABC^2 [37,38,39].

The CooperA (cooperating agents) platform is a software framework that supports the cooperation of heterogeneous, distributed and semiautonomous Knowledge-Based (KB) systems. In CooperA the KB systems are translated into application agents that are finally be integrated into one system. The users can interact with all the agents using a user interface agent. The CooperA architecture is built by a set of interconnected layers. These layers are: *The CooperA kernel*, *the message-passing mechanism*, *the collection of CooperA system Agents*, and finally *the Community of Application-Specific Agents*. Any agent in the CooperA architecture is a

dynamic structure that communicates with other agents in the system through a message passing skill.

The ABC^2 architecture is based on the CooperA architecture. It uses predefined skills (managed as reactive components) that each agent composes in an opportunistic way to achieve the intelligent behaviour (this architecture has been implemented and tested in the RoboSoccer domain and the Khepera robots). The agenda in ABC^2 has been used to keep a list of pending actions, where each action can require (or not) some of other actions. These actions can be inserted into the agenda by other actions, by events from the environment or by requests received from other agents.

Our work is more closely related to ABC^2, although there are several differences. First, our approach allows to define acts which are composed of subacts that can be executed in parallel and can have AND/OR structures. Second, ABC^2 has only been used to implement a planning oportunistic reactive behaviour for agents, which is appropriate in domains like the RoboSoccer. We have oriented our architecture to facilitate integrating AI solving techniques with IG techniques to work in WEB domains. Some of these features have already been tested in other domains like problem solving through Genetic Programming [1] and WEB News Gathering [9,10].

Several systems have been designed to deal with different information sources. Some of those systems like SIMS [3,4] allow to integrate information from different sources like databases and knowledge bases. The SIMS approach uses a semantic model of a problem domain to integrate the information from several sources, and several algorithms for automatically improving the efficiency of queries using knowledge about both the domain and the information sources. These kinds of systems, usually named mediators, implement several mechanisms that provide access to heterogeneous data and knowledge bases. When these techniques are used to build agents that are able to extract, query, and integrate data from electronic sources it is possible to define an information agent. These information agents have been used to implement different systems that are able to retrieve and integrate information from the WEB [28,31]. The closest systems to our work are:

- WebPlan [23]: it is a WEB assistant for domain-specific search on the Internet based on dynamic planning and plan execution techniques. The existing planning system *CA-Plan* [24,53] has been extended in different ways in order to deal with incomplete information, information seeking operators, user interaction, and interleaving planning and execution. WebPlan is specialized in localizing specific PC software on the Internet. Planning is used in this system to select the most appropriate sources to look for information, whereas MAPWEB uses planning to select the appropriate WEB sources **and** to build the solution to a user problem.
- Ariadne [30]: This system includes a set of tools to construct wrappers that make WEB sources look like relational databases. It also uses mediation techniques based on SIMS [3, 31]. The main focus of these systems is how to access the distributed information, so the integration problem is not a hard problem. However, besides accessing the appropriate information, we are interested in integrating the different sources and solve complex problems with the retrieved information.
- Heracles [2,29]: This framework is used to develop different information assistant systems that employ a set of information agents (Ariadne, Theseus, Electric Elves). A dynamic hierarchical constraint propagation network (CPN) is used to integrate the different information sources. Two assistant systems have been implemented: *The Travel Planning Assistant* (specialized in assisting tourists to plan their trips) and *The WorldInfo Assistant* (for a user-specified location, it integrates information from different information sources like weather, news, holidays, maps, airports, ...). In this framework the integration of the retrieved information is made by a CPN. Therefore, if the problem changes, the CPN needs to be rewritten by hand. MAPWEB is more flexible because it uses a planner to automatically generate the plans, which are the structures analogous to the CPN. For instance, if new transport sources like taxi or buses become available, it is only necessary to add a new planning operator for every new source and the PlannerAgent will use them to access these sources.

6. Conclusions

In this paper we have presented SkeletonAgent, an agenda-based flexible architecture for building agents that can participate in a MAS. The utilization of an agenda-based architecture for agents, allows to coordinate multiple agents with heterogeneous skills in a flexible way. Also, it is very simple to change the behaviour of the agents by modifying the policies used to manage the agenda.

Although our approach is based on work described in [37,49], their ideas have been extended in several ways. First, we allow to define acts which are composed of subacts that can be executed in parallel and can have AND/OR structures. This allows to define alternative ways to achieve a goal and to mix different high level tasks at the minimum level of granularity. For instance, if there are two tasks A and B composed of several subtasks A_1, A_2 ... and B_1, B_2, ..., every subtask will be processed when they are ready, thus interleaving the two tasks A and B in the most appropriate order.

Second, we have oriented our architecture to facilitate integrating AI solving techniques with IG techniques to work in Web domains. To do so, we have instantiated SkeletonAgent to build a MAS, named MAPWeb, that combines AI planning and Web information gathering techniques. MAPWeb is a generic architecture that can be used by developers in any domain requiring planning and Web sources. We have used it to solve travel assistant problems in the e-tourism domain (MAPWeb-etourism). This example has also been used to illustrate the behaviour of the agenda-based architecture. Some of SkeletonAgent features have also been tested in other domains like problem solving through Genetic Programming [1] and Web News Gathering [9,10], which shows that it is a general framework.

In the future we would like to extend MAPWeb-etourism by adding new AI techniques to the architecture, like Machine Learning, so that more complex problems can be solved by using information from the Web.

Acknowledgements

The research reported here was carried out as part of the research project funded by CICYT TIC2002-04146-C05-05.

References

[1] Ricardo Aler, David Camacho, and Alfredo Moscardini. *Cooperation Between Agents to Evolve Complete Programs*, chapter In Intelligent Agent Software Engineering, pages 213–228. Valentina Plekhanova. University of Sunderland, United Kingdom. Ed. by Idea Group Publishing, 2003.

[2] José Luis Ambite, Greg Barish, Craig A. Knoblock, Maria Muslea, Jean Oh, and Steven Minton. Getting from here to there: Interactive planning and agent execution for optimizing travel. In *The Fourteenth Innovative Applications of Artificial Intelligence Conference (IAAI)*, Edmonton, Alberta, Canada, 2002.

[3] Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Cooperative Information Systems*, 2(2):127–158, 1993.

[4] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems, Special Issue on Intelligent Information Integration*, 6(2/3):99–130, 1996.

[5] M. Balabanovic, Y. Shoham, and T. Yun. An adaptive agent for automated web browsing, 1995.

[6] Ronen I. Brafman and Moshe Tennenholtz. Modeling agents as qualitative decision makers. *Artificial Intelligence*, 94(1-2):217–268, 1997.

[7] W. Brenner, R. Zarnekow, and H. Wittig. *Intelligent Software Agents. Foundations and Applications.* Springer-Verlag. ISBN: 3-540-63411-8, New York, 1998.

[8] Rodney A. Brooks. Intelligence without representation. Number 47 in Artificial Intelligence, pages 139–159. 1991.

[9] David Camacho, Ricardo Aler, César Castro, and José M. Molina. Analysis of internet multi-agent based system for zeus and skeletonagent frameworks. In *Second WSEAS International Conference on Multimedia, Internet and Video Technologies (ICOMIV 2002)*, Skiathos, Greece, September 2002. IEEE.

[10] David Camacho, Ricardo Aler, César Castro, and José M. Molina. Performance evaluation of Zeus, Jade and SkeletonAgent frameworks. In *Proceedings of the IEEE Systems, Man, and Cybernectics Conference (SMC-2002)*, Hammamet, Tunisia, October 2002. IEEE.

[11] David Camacho, Daniel Borrajo, José Manuel Molina, and Ricardo Aler. Flexible integration of planning and information gathering. In *Proceedings of the European Conference on Planning (ECP-01)*, Toledo, Spain, September 2001. Springer-Verlag. Series LNAI.

[12] David Camacho, José Manuel Molina, Daniel Borrajo, and Ricardo Aler. Solving travel problems by integrating web information with planning. In *XIII. International Symposium on Methodologies for Intelligent Systems (ISMIS 2002)*, Lyon, France, June 2002. Springer-Verlag. Series LNAI.

[13] Philip R. Cohen, Adam Cheyer, Michelle Wang, and Soon Cheol Baeg. An open agent architecture. In *In Working Notes of the AAAI Spring Symposium: Software Agents*, pages 1–8. AAAI, Menlo Park, CA, 1994.

[14] Philip R. Cohen and Hector J. Levesque. Performatives in a rationally based speech act theory. In *Meeting of the Association for Computational Linguistics*, pages 79–88, 1990.

[15] Philip R. Cohen and C.R. Perrault. Elements of a planbased theory of speech acts. *Cognitive Science*, 3(3):177–212, 1979.

[16] Kerstin Dautenhahn. The art of designing socially intelligent agents: science, fiction and the human in the loop. *Applied Artificial Intelligence Journal*, 1(7), 1998.

[17] Keith Decker and Katya Sycara. Intelligent adaptive information agents. *Journal of Intelligent Information Systems*, 9:230–260, 1997.

[18] Yizhong Fan and Susan Gauch. Adaptive agents for information gathering from multiple, distributed information sources. In *Proceedings of 1999 AAAI Symposium on Intelligent Agents in Cyberspace*. Stanford University, March 1999.

[19] I. A. Ferguson. *Integrated control and coordinated behaviour: A case for agent models*, chapter In Intelligent Agents: Teories, Architectures and Languages (LNAI Volume 890). M. Wooldridge and N.R. Jennings, editors, pages 203–218. Springer-Verlag: Heidelberg, Germany, January 1995.

[20] Tim Finin and Jay Weber et. al. *Draft specification of the KQML agent communication language*. Jun 15 1993.

[21] Tim Finin, R. Fritzson, D. Mackay, and R. McEntire. Kqml as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM94)*, pages 456–463, Gaithersburg, Maryland, 1994. New York: Association of Computing Machinery, ACM Press.

[22] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 1994.

[23] J. Hüllen, Ralph Bergmann, and F. Weberskirch. Webplan: Dynamic planning for domain-specific search in the internet. In *Workshop Planen und Konfigurieren (PuK-99)*, 1999.

[24] J. Hüllen and F. Weberskirch. Extracting goal orderings to improve partial-order planning. In *Workshop Planen und Konfigurieren (PuK-99)*, 1999.

[25] N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien, and B. Odgers. Autonomous agents for business process management. *Int. Journal of Applied Artificial Intelligence*, 14(2):145–189, 2000.

[26] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):275–306, 1998.

[27] Nicholas R. Jennings and Michael J. Wooldridge. *Agent-Oriented Software Engineering*, chapter Handbook of Agent Technology. AAAI/MIT Press, 2000.

[28] Craig A. Knoblock and José Luis Ambite. *Agents for Information Gathering*, chapter In Software Agents. J. Bradshaw editor, AAAI/MIT Press, Menlo Park, CA, 1997.

[29] Craig A. Knoblock, Steve Minton, José Luis Ambite, Maria Muslea, Jean Oh, and Martin Frank. Mixed-initiative, multi-source information assistants. In *The Tenth International World Wide Web Conference (WWW10)*. ACM, May 1-5 2001.

[30] Craig A. Knoblock, Steven Minton, José Luis Ambite, and Naveen Ashish. Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*. Madison, WI, 1998.

[31] Craig A. Knoblock, Steven Minton, José Luis Ambite, Naveen Ashish, Ion Muslea, Andrew G. Philpot, , and Sheila Tejada. The ariadne approach to web-based information integration. *To appear in the International the Journal on Cooperative Information Systems (IJCIS) Special Issue on Intelligent Information Agents: Theory and Applications*.

[32] Nickolas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[33] Eric Lambrecht and Subbarao Kambhampati. Planning for information gathering: A tutorial survey. Technical report, Arizona State University, May 1997. ASU CSE Techincal Report 96-017.

[34] S. Lander and Victor R. Lesser. Understanding the role of negotiation in distributed search among heterogeneous agents. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 438–444, 1993.

[35] Victor R. Lesser. Cooperative multiagent systems: A personal view of the state of the art. *Knowledge and Data Engineering*, 11(1):133–142, 1999.

[36] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Databases*, pages 251–262, Bombay, India, 1996. VLDB Endowment, Saratoga, California.

[37] Vicente Matellán and Daniel Borrajo. ABC^2 an agenda based multi-agent model for robots control and cooperation. *Journal of Intelligent and Robotic Systems*, 32(1):93–114, October 2001.

[38] Vicente Matellán and Daniel Borrajo. Combining classical and reactive planning: The *abc*^{2} model. In *Procceedings of the Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments Workshop. Fourth Artificial Intelligence Planning Systems (AIPS98).*, June Pittsburgh (USA), 1998.

[39] Vicente Matellán, Daniel Borrajo, and Camino Fernández. Using *abc*^{2} in the robocup domain. In

RoboCup-97: Robot Soccer World Cup I. Hiroaki Kitano, editor. Lecture Notes in Artificial Intelligence, pages 475–483, 1998.

[40] José M. Molina. *Técnicas Multiagente de Cooperación para la Gestión Coordinada de Multisensores*. PhD thesis, Departamento de Señales, Sistemas y Radiocomunicaciones. Universidad Politécnica de Madrid, Julio 1997.

[41] Timothy Norman, Nick Jennings, Peyman Faratin, and Abe Mamdani. Designing and implementing a multi-agent architecture for business process management. In Jörg P. Müller, Michael J. Wooldridge, and Nicholas R. Jennings, editors, *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, volume 1193, pages 261–276. Springer-Verlag: Heidelberg, Germany, 12–13 1997.

[42] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–224, October/November 1996.

[43] Hyacinth S. Nwana, L. C. Lee, and Nicholas R. Jennings. Coordination in software agent systems. *The British Telecom Technical Journal*, 14(4):79–88, 1996.

[44] Charles Petrie. Agent-based software engineering. In *Proceedings of PAAM 2000. The Practical Application of Intelligent Agents and Multi-Agents*, 2000.

[45] S. Picault. A multi-agent simulation of primate social concepts. In *European Conference on Artificial Intelligence*, pages 327–328, 1998.

[46] Arnaud Sahuguet and Fabien Azavant. Building intelligent web applications using lightweight wrappers. *Data Knowledge Engineering*, 36(3):283–316, 2001.

[47] Erik Selberg and Orentz Etzioni. The metacrawler architecture for resource aggregation on the web. In *IEEE Expert*, pages pp. 8–14. IEEE, January/February 1997.

[48] Lorenzo Sommaruga, Nikos M. Avouris, and Marc Van Liedekerke. An environment for experimentation with interactive cooperating knowledge based systems. In *Proceedings of Expert Systems*. Cambridge Press, London, 1989.

[49] Lorenzo Sommaruga, Nikos M. Avouris, and Marc Van Liedekerke. *Foundations of Distributed Artificial Intelligence*, chapter The evolution of the CooperA platform, pages 365–400. John Willey. London, 1996.

[50] Lorenzo Sommaruga and N. Shadbolt. The cooperative heuristics approach for autonomous agents. In *Proceedings of the Cooperative Knowledge Based Systems Conference*, pages 49–61. University of Keele, Keele, U.K., 1994.

[51] Milind Tambe. Implementing agent teams in dynamic multi-agent environments. *Applied Artificial Intelligence*, 12, 1998.

[52] Manuela Veloso. *Planning and Learning by Analogical Reasoning*. Springer-Verlag, December 1994.

[53] F. Weberskirch. Combining snlp-like planning and dependency-maintenance. Technical report, Technical Report LSA-95-10E, Centre for Learning Systems and Applications, University of Kaiserslautern, Germany, 1995.

[54] Michael J. Wooldridge. Coherent social action. In *In A. Cohn, editor, Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-94)*, pages 15–26. John Wiley & Sons, August 1994.