COOWS: ADAPTIVE BDI AGENTS MEET SERVICE-ORIENTED COMPUTING (EXTENDED VERSION) [1]

Luigi Bozzo [a] Viviana Mascardi [a] Davide Ancona [a] Paolo Busetta [b]

[a] *DISI - Genova University, Italy,*
luigi.bozzo@gmail.com, mascardi@disi.unige.it, davide@disi.unige.it
[b] *IRST - ITC, Trento, Italy,*
busetta@itc.it

Abstract

Mainstream research in Web Services is currently looking at two main aspects, namely formally describing interactions among services, and finding and combining services. Much work made in the intelligent agents area is being applied to these issues. In this paper, we investigate the application of agent research to Web Services from a different perspective, that is, procedural learning. The final objective is to enable an adaptive system (an agent in our terminology) to discover or being fed with knowledge concerning how to solve a specific set of problems in a specific software or physical environment. Our work is a very preliminary step into the issue, with the main objective of assessing how current Web Services technology can support a component, described in terms of beliefs, desires and intentions, dynamically adapting its behaviour to new environments.

1 Introduction and Motivation

Mainstream research in Web Services (WS) is looking at two main aspects: first, formally describing interactions among services (possibly over long periods of time and having multiple real-world effects, including legally binding actions); second, finding and combining services (e.g., by extending the simple catalogue contained in UDDI repositories with semantically rich descriptions and using the latter for formal verification and for automated composition via planning). As observed by many researchers [4, 7, 13, 3] much work made in the intelligent agents area can be applied to these issues.

In this paper, we investigate the application of agent research to WS from a different perspective, that is, procedural learning. The final objective is to enable an adaptive system (an agent in our terminology) to discover or being fed with knowledge concerning how to solve a specific set of problems in a specific environment (no matter whether the latter is physical, e.g. as in the case of a personal assistant on board of a mobile device, or virtual, e.g. a manufacturing scheduler integrating with its peers in a virtual enterprise). The main difference between procedural learning and approaches based on service discovery and planning is that the goals that have to be achieved are neither satisfied by a third-party, ad-hoc service (as it is envisaged in the world of composite services built by means of orchestration languages), nor via explicit reasoning by the agent on how to combine whatever elementary components are locally available. The first approach would imply the existence of additional computational servers in the target environment: this is often not appropriate especially when sensitive data would need to be exported from the agent trying to achieve its goals to an intermediate service. The second approach is computationally expensive, sometimes unable to reach a final solution without substantial human input (e.g. when multiple paths are equally possible), and practically not applicable when agents are on board of computationally limited devices but have to react in real-time.

[1] The short version of this paper appeared in the Proceedings of the WWW/Internet 2005 Conference, edited by P. Isaìas and M. B. Nunes, Volume II, pages 205–209.

Procedural learning is related to machine learning (in particular, to agents that learn from other agents via instruction or observation), to semantic web and semantic web services (in particular, to express goals and knowledge more flexibly than with current WS interfaces, e.g. WSDL), and to belief-desire-intention (BDI) and teamwork theories (which have been investigating, among other things, how agents and teams of agents can achieve multiple goals in the face of failures and rapidly changing environments). Our work is a very preliminary step into the issue, with the main objective of assessing how current WS technology can support a component, described in terms of beliefs, desires and intentions, that dynamically adapts its behaviour to new environments (namely, a CooBDI agent in our terminology). CooBDI agents [1] are suitable for modeling all kinds of "procedural learning agents" which learn from the interaction with other agents. This is made possible by the main feature of CooBDI consisting of a built-in mechanism for retrieving external plans from collaborative agents, for example when no local plans suitable for achieving a certain desire are available. This feature turns out to be useful in many application fields such as:

- Personal Digital Assistants (PDAs), whose physical resources are limited and dynamic loading and linking of code is required. CooBDI can cope with these issues by allowing an agent to discard the external plans after their usage: this is useful when the space resources of the PDA have strict bounds.

- Self-repairing agents, namely agents situated in a dynamically changing software environment and able to identify the portions of their code that should be updated to ensure their correct functioning in the evolving environment. Since CooBDI allows dynamic replacement of (obsolete) local plans with (updated) external ones, it can be used for modeling such a self-repairing agent.

- Digital butlers, i.e. agents that assist a human user in some task such as managing her/his agenda, filtering incoming mail, retrieving interesting information from the web. A typical feature of a digital butler is its ability to dynamically adapt its behavior to the user needs by cooperating both with more experienced digital butlers and with the assisted user. In our setting, through a user friendly interface, the user might train the CooBDI agent by showing the right sequence of actions (i.e. the right plan) to perform in particular situations. In this case, the external plan should be provided by the human trainer of the digital butler.

One of the main limitations of CooBDI and of its CooAgentSpeak implementation [2], is that, in order to profitably exchange plans, agents must share the set of basic actions that may appear in their plans' bodies. For example, let us suppose that a digital butler `A`, implemented according to the CooBDI architecture, wants to achieve the goal "`reserve_hotel(Hotel)`" for its human user, and that it does not know how to achieve it (namely, it has no plans whose head matches with the "`achieve(reserve_hotel(Hotel))`" triggering event). In the CooBDI setting, `A` would ask for plans whose triggering event is "`achieve(reserve_hotel(Hotel))`" to its digital friends. Let us suppose that `B` answers to this request, by providing the plan (expressed as a first-order logic term) "`plan(trigger(achieve(reserve_hotel(Hotel))), body(retrieve_email_address(Hotel, Address), send_reservation_email_to(Address))).`".

The problem here is that both `retrieve_email_address` and `send_reservation_email_to` are names of actions that are implemented by `B`, but, in the most general case, not by `A`: agent `B` exactly knows how to perform them, but what does it happen, when `A` executes the plan provided by `B`, and must perform them? It would be the same as installing on a PC (the digital butler `A`) a new program (the plan provided by `B`) that makes calls a set of routines (the basic actions `retrieve_email_address` and `send_reservation_email_to`). The program works only if the routines are already present on the PC, otherwise, its execution fails. Thus, without assuming that all the agents in the system share the same set of basic actions, the mechanism of plan exchange is useless. The CooBDI architecture works if this assumption holds.

Since we could not drop this assumption from the CooBDI architecture, we looked for an application domain where this assumption were naturally satisfied. In the WS setting, basic actions can be seen as invocations of WSs. As long as an agent is able to execute an invocation of a WS, it automatically shares the set of basic actions with all the other agents in the system. In the example of the

digital butlers, `B` would send to `A` the plan[2] "`plan(trigger(achieve(reserve_hotel(Hotel))), body(invoke(retrieve_email_address(Hotel, Address), hotel_catalogue_WS), invoke (send_reservation_email_to(Address), email_sender_ws))).`". Provided that `A` is able to execute the `invoke` statement, it can easily execute the body of this plan without further assumptions.

According to this rationale, we have implemented the ideas behind the CooBDI theory by means of WS technologies. In particular, plan bodies are expressed in BPEL4WS (BPEL for short), a high-level scripting language for WSs built on top of WSDL, that includes a mixture of control statements (if/then, while, etc.) and WSs invocation. Provided that an agent is able to execute a BPEL specification, it can execute the body of any plan, making the exchange of plans among agents a fruitful extension of the basic BDI architecture, and without requiring strong assumptions.

In order to exploit the WSs technologies to implement CooBDI agents, we have performed a number of simplifications. We ignore issues with semantics and drastically reduce the representation of desires to what is currently possible by means of WSDL, an XML-based service description language used to describe how to communicate using WSs. Both plans' triggers and desires, in fact, are represented as strings, and thus the matching of a plan trigger with the desire that the plan should achieve is just a string comparison. The representation of the agent's beliefs remains implicit in the plan's description. A prototype, built on open-source software and available to the research community, shows the feasibility of our approach.

The paper is organised as follows: Section 2 describes the background of CooWS, namely CooBDI and the current available WS technologies. Section 3 describes CooWS as an instance of the CooBDI model, and Section 4 summarises the related work and outlines the future directions of research.

2 Background

2.1 Intelligent agents and CooBDI

According to M. Luck,

> agent-based systems are one of the most vibrant and important areas of research and development to have emerged in information technology in the 1990s, and underpin many aspects of modern computing infrastructures and applications. [6]

Agents, following the well known definition of N. Jennings, K. Sycara, M. Wooldridge [5], are usually characterised as computer systems situated in some environment, that are capable of flexible autonomous actions in order to meet their design objectives. Besides the "weak definition" given above, the notion of an intelligent agent as an entity which appears to be the subject of beliefs, desires, intentions (BDI) proposed by A. S. Rao and M. Georgeff [9], is well known and accepted by many researchers.

BDI-style languages allow agents both to be aware of the environment in which they are located thanks to the perception of external events that enter the agents' event queue, and to modify the environment thanks to a set of external actions with which every agent is equipped. The external actions are not specified at the language level since they strictly depend on the application domain. In this sense, BDI-style languages allow the programming of situated agents that are able to perceive the environment where they live and to act on it.

BDI-style languages also support the sociality of agents by letting them exchange data. CooBDI and its CooAgentSpeak implementation, together with JAM (`www.marcush.net/IRS/irs_downloads.html`), represent a step forward toward the implementation of the agents' cooperativity since they allow agents to exchange procedural knowledge (plans), besides mere data (beliefs).

The BDI model is characterised by the following concepts:

- beliefs: the agent's knowledge about the world;
- desires: objectives to be accomplished;
- intentions: stacks of plans currently under execution;

[2]The syntax used here is a simplification of the BPEL syntax that we use to define plan bodies, made just for sake of clarity.

- plans: "recipes" representing the procedural knowledge of the agent, usually characterised by a trigger which fires the adoption of the plan, a precondition that the current state must satisfy for the plan to be applicable, a body of actions to perform, an invariant condition that must hold during the whole plan execution, a set of actions to be executed if the plan execution terminates successfully and a set of actions to be executed in case of plan failure.

Most BDI systems also include an event queue where both events (either perceived from the environment or generated to notify an update of its belief base) and internal subgoals (generated by the agent itself while trying to achieve a main goal) are stored.

The typical BDI execution cycle is characterized by the observation step, where new external events are perceived; the generation of the plan instances that can be adopted to cope with the perceived event (relevant plans) and the insertion of the chosen plan instance on top of the intention stack whose execution generated the perceived event (if any); and finally the execution of the topmost plan of one intention stack.

In most existing BDI-style languages the plan library is static: agents can neither extend their procedural knowledge at run time, nor exchange plans. CooBDI, instead, overcomes these limitations by introducing cooperations among agents to retrieve external plans for achieving desires. It also extends plans with access specifiers and extends intentions to take into account the mechanism for retrieving external plan instances. The CooBDI execution cycle is also modified to cope with all these issues. CooBDI, in its original formulation, distinguishes between external events and desires. Here, for simplicity, we consider ordinary events perceived from the outside as if they were in a one-to-one relation with desires: in our simplified setting, the perception of an ordinary event from the environment, generates the desires that has exactly the same syntax as the perceived event. Basically, we treat ordinary events as if they were desires, and for this reason we will use "event/desire" to refer to them.

The novel features of CooBDI w.r.t. the original BDI architecture are:

- **CooBDI cooperation strategy.** The cooperation strategy of an agent `A` includes the set of agents with which is expected to cooperate (partner agents, or "friends"), the plan retrieval policy that states when an external plan should be looked for, namely only when no suitable plans are locally stored, or in any case, and the plan acquisition policy that states what should be done with a plan retrieved from the extern, namely, if it should be saved in the local plan library, discarded or used to replace those plans with the same trigger.

- **CooBDI plans.** CooBDI plans share with "classical" BDI ones the trigger, the precondition, the body, the invariant and the two sets of success and failure actions. Besides these components, they also have an access specifier which determines the set of agents the plan can be shared with. It may assume three values: `private` (the plan cannot be shared), `public` (the plan can be shared with any agent) and `only(TrustedAgents)` (the plan can be shared only with the agents in the `TrustedAgents` set).

- **CooBDI intentions.** CooBDI intentions are in a one-to-one relation with event/desires: each intention is created when the agent acquires a new event/desire to achieve, and it is deleted when the event/desire fails or is achieved. Intentions are characterized by "standard" BDI components plus components introduced to manage the external plan retrieval mechanism.

- **CooBDI execution cycle.** The execution cycle of CooBDI departs from the classical one to take into account both the generation of event/desires and cooperations. It is characterized by three macro-steps:

  1. process the event queue;
  2. process suspended intentions;
  3. process active intentions.

  Here, we only discuss what happens when relevant plans for an event/desire (i.e. those plans whose triggering event "matches" with the event/desire itself) must be generated: 1) The intention that generated the event/desire is suspended. 2) The local relevant plans for the

event/desire are generated and associated with the intention. 3) According to the cooperation strategy, the set of the agents expected to cooperate to retrieve relevant plans is defined. 4) A plan request for the event/desire is created and it is sent to all these agents.

2.2 Web Services

The term Web service typically refers to a modular application that can be invoked through the Internet. The consumers of Web services are other computer applications that communicate, usually over HTTP, using XML standards including SOAP, WSDL, and UDDI (`philip.greenspun.com/seia/glossary`).

The most relevant tools that make the implementation of a web service feasible, and that we have used in the implementation of CooWS, are SOAP, WSDL, BPEL and UDDI.

- **SOAP.** SOAP (Simple Object Access Protocol), is a light-weight, XML-based protocol originally designed by Microsoft and IBM for exchanging messages between computer software. It is an extensible and decentralised framework that can work over multiple computer network protocol stacks. Remote procedure calls can be modeled as an interaction of several SOAP messages. SOAP can be run on top of all the Internet Protocols, although HTTP is the most common and the only one standardized by the World Wide Web Consortium (W3C).
- **WSDL.** The Web Services Description Language (WSDL) is an XML format published for describing the public interface to the web service. This is an XML-based service description on how to communicate using the web service; namely the protocol bindings and message formats required to interact with the web services listed in its directory. The supported operations and messages are described abstractly, and then bound to a concrete network protocol and message format. WSDL is often used in combination with SOAP and XML Schema to provide web services over the Internet.
- **BPEL4WS.** The Business Process Execution Language for Web Services (BPEL4WS, or BPEL for short) is a programming language, serialized in XML, that is intended to enable programming code that represents the high level state transition logic of the program. This logic encodes information such as when to wait for messages, when to send messages, when to compensate for failed transactions, etc. BPEL adopts Web Services as its external communication mechanism. That is, BPEL's messaging facilities depend on the use of WSDL to describe outgoing and incoming messages.
- **UDDI.** UDDI (Universal Description, Discovery, and Integration) is a platform-independent, XML-based registry for businesses worldwide to list themselves on the Internet. A UDDI business registration consists of three components:
  - White Pages - address, contact, and known identifiers;
  - Yellow Pages - industrial categorizations based on standard taxonomies; and
  - Green Pages - technical information about services exposed by the business

  UDDI is designed to be interrogated by SOAP messages and to provide access to WSDL documents describing the protocol bindings and message formats for interacting with the WSs listed in its directory.

3 CooWS

A CooWS agent adopts the following metaphor inspired by both CooBDI, and the WS technologies.

- **Beliefs.** The variables local to the BPEL processes that constitute the body of the agent's plans can be considered as a metaphor for the agent's beliefs local to that plan, that are not explicitly represented.

- **Desires.** Desires are represented by instances of the `CooDesire` class. In our implementation of CooWS, we adopt two simplifications: the first is that the only means for perceiving the environment is by receiving messages from other agents in the system (namely, the environment is the MAS itself, and perception coincides with communication); the second is that desires are in a one-to-one relation with ordinary events perceived from the outside, thus desires are perceived from the outside as part of a communicative action.

  For this reason, desires are messages. To enhance flexibility, messages may either be structured according to the FIPA ACL standard (`www.fipa.org`), or be unstructured Java strings. The `CooOrdinaryEvent` class that implements ordinary events perceived from the outside, consists of one attribute of type `CooDesire`, besides the sender and the receiver of the message, and other fields described later in this section.

- **Queries.** As happens for beliefs, queries on the beliefs of the agents are not explicitly represented.

- **Actions.** There are two kinds of actions, those that may appear inside the BPEL specification of the agent's plan body (plan's actions), and those that must be executed in case of success or failure of the achievement of an event/desire (success and failure actions). Plan's actions are thus only external ones, and they consist of the invocation of a web service by means of the BPEL `invoke` statement. In the sequel, we use "external web service" for web services offered by providers that live outside of the CooWS platform, such as Google, eBay, Amazon, etc, and we use "coo web service" for the web service offered by the CooWS platform in order to ensure its functioning. There are three types of plan's actions:

  – Delivery of ordinary events. It is implemented by invoking the operation `dispatchEvent (CooEvent anEvent)` offered by the coo web service. Ordinary events are described later in this section.

  – Achievement of internal desires. It is implemented by invoking the `achieve` operation offered by the coo web service. This action allows the implementation of the mechanism of plan nesting, since the invocation of the achieve operation generates a `CooEventAchieve` event that is managed by looking for a plan relevant for the internal desire to achieve[3].

  – Invocation of external web services. They are implemented as "standard" invocations of WSs.

  Cooperative requests for plans are managed transparently to the agent, and do not belong to the set of actions that can be programmed by the user. Success and failure actions, implemented by the `CooAction` class, are associated with each external event that enters the event queue, and are executed in case the management of that event succeeds (resp. fails). They may be calls to any Java method implemented by the developer of the system.

- **Plans.** Plans are instances of the `CooPlan` class, and are defined by a unique plan identifier, a trigger, a body, and an access specifier.

  The trigger is an instance of the `CooDesire` class; the body is a string that uniquely identifies the `serviceName` of a BPEL process (which is represented by a file, maintained in the file system of the server); and the access specifier is a string (`CooAccessSpecOnlyTrusted`, `CooAccessSpecPrivate`, and `CooAccessSpecPublic`) that identifies one of the three access policies described in Section 2.1.

  Our choice of including a reference to a BPEL process file instead of the BPEL file itself as the plan body is only due to efficiency reasons, since specifications of BPEL processes may be very large (just to make an example, the WSDL representation of the web service offered by eBay is 35000 lines of code, for an amount of more than one MB). From a logical viewpoint, however, a plan body **is** a BPEL specification, that contains both invocations of external web services (corresponding to calls to external actions), and invocations of the coo web service (corresponding to calls to internal actions such as `achieve`). These invocations

[3]Internal desires, or sub-goals, are different from the event/desires since the latter are generated by the perception of the environment, and the former are generated by the agent itself.

are composed together by means of the statements that BPEL provides, such as sequence, if-then-else, while, etc.

The self-notification of the plan outcome is automatically added to each BPEL process that represents a plan by the CooWS platform, and consists in invoking the `dispatchPlanOutcome` operation offered by the coo web service. The actions for managing failure or success of the plans are not part of the plan itself, as in the BDI and CooBDI models, but they are associated with each external event that enters the event queue.

- **Plan instances.** The instance of a plan is the BPEL process identified by the `mBodyID` field of the plan, that executes on the BPEL engine. This process is characterised by the identifier of the agent that launched its execution, and by the identifier of the intention that contains the desire that generated the plan. This information is enough to identify the plan instance without ambiguity.
- **Intentions.** An intention is an instance of the `CooIntention` class, and contains a stack of `CooDesire` objects, a boolean attribute defining the intention's state (either active or suspended), and the success and failure actions. With each desire in the stack, the set of relevant plans currently available to the agent is associated. The set of plans is generated as follows:
  - The relevant plans are first looked for in the local plan library.
  - If the set of local relevant plans is empty, or if the retrieval strategy is `RETRIEVAL_STRATEGY_ALWAYS`, the retrieval of external plans is started: an event of type `CooEventRequested` is delivered to the partner agents, the agent goes on with its activities (which may include executing a plan associated with the topmost desire, if there are any), and, when answers to the `CooEventRequested` (namely, `CooEventProvided` events) enter its event queue, it associates the received plans with the desire.
  - Intentions are suspended when there are no active plans associated with their topmost desire.
  - If an intention remains suspended for more than a given amount of time, it fails. This "time-out" mechanism is necessary to avoid that an intention stays in an idle state forever, waiting for answers (namely, relevant plans to be associated to some desire in the intention itself) that might not arrive due to network problems.
- **Events.** There are four kinds of events: cooperation, plan outcome, achieve, and ordinary events.
  - A cooperation event is either an instance of the `CooEventRequested` class, characterised by sender, receiver, unique request identifier, and desire for which the request has been issued, or an instance of the `CooEventProvided` class, characterised by sender, receiver, unique request identifier, and set of plans that are relevant for the desire appearing in the corresponding request.
  - Plan outcome events are instances of the `CooEventPlanOutcome`, and are used to notify the agent either of the failure or of the success of a plan execution. This information is necessary to update the intention stack that generated the plan instance in the right way.
  - Achieve events are instances of the `CooEventAchieve` class, that is characterised by the agent's identifier, the desire to be achieved, and the intention that generated the desire. This kind of events allows the implementation of the mechanism of plan nesting.
  - Ordinary events consist of the reception of messages from other agents (instances if the `CooEventOrdinary` class), and they are characterised by a sender, a receiver, a desire, and the two sets of success and failure actions.

  Events are managed by a first-in-first-out event queue.
- **Retrieval and acquisition strategies.** They are integer fields that can assume values corresponding to the different types of retrieval and acquisition strategies described in Section 2.1.

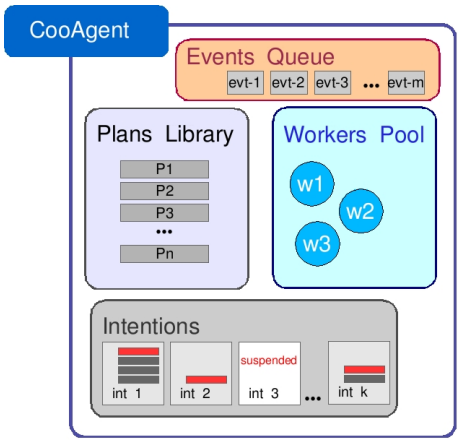

Figure 1: The most important data structures of a CooWS agent

- **Agent.** The structure of the `CooAgent` class consists of seven fields that define the agent's identifier, its partners, the agent's plan library, its retrieval and acquisition strategies, its set of intentions and its event queue, respectively. A graphical representation of the most important fields is given in Figure 1.

  As far as the methods are concerned, besides the standard setter, getter and constructor methods, the `CooAgent` class defines the `onEvent(CooEvent anEvent)` method for handling incoming events; for each event, a worker thread is activated. Concurrence among threads is demanded to the `commons-pool` package offered by Apache Software Foundation. Each worker calls the right method among the following ones, according to the received event.

  - `onEventAchieve(CooEventAchieve anEvent)`: one plan currently under execution issued an achieve event containing a desire. The intention associated with the plan is retrieved, the desire contained in the CooEventAchieve instance is put on top of that intention, and the retrieval of relevant plans is activated in the same way as described in Section 2.1 and recalled in the "Intentions" paragraph.
  - `onEventOrdinary(CooEventOrdinary anEvent)`: an ordinary event is characterised by a desire `aDes`, the sender, the receiver, the success and the failure actions. Its management consists in the creation of a new intention with the `aDes` desire at the top, followed by the retrieval of relevant plans for the management of the desire itself, according to the retrieval strategy.
  - `onEventPlanOutcome(CooEventPlanOutcome outcome)`: this event implements the notification of the success/failure of the plan generated by the intention whose identifier is contained in the `CooEventPlanOutcome` instance. In case of success, the desire at the top of the intention is popped (since it has been successfully achieved). In case of failure, the identifier of the plan is removed from the set of plans associated with the desire at the top of the intention, and the agent tries another plan among the remaining ones (if any, otherwise the desire fails and backtracking to the previous desire in the intention stack is performed; the intention fails if the stack is empty).
  - `onEventProvided(CooEventProvided anEvent)`: the intention that originated the desire for which plans have been provided is retrieved, and the set of provided plans is associated to the right desire in the stack (which is not necessarily the topmost one, since in the meanwhile the nesting of plans might have taken place). If the desire is the topmost one, and the intention was `suspended` (namely, there were no plans associated with its topmost desire), one among the retrieved plans is chosen for execution, and the intention status is switched to `active`.
  - `onEventRequested(CooEventRequested anEvent)`: the set of relevant and sharable plans (namely, those whose trigger is syntactically equal to the desire in the `CooEventRequested` event, and whose access specifier is either `public` or `only trusted`, and the

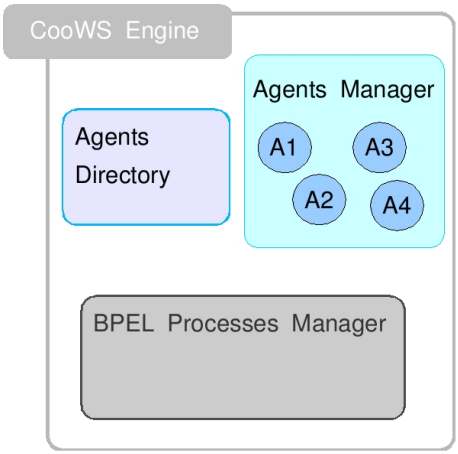

Figure 2: The CooWS engine

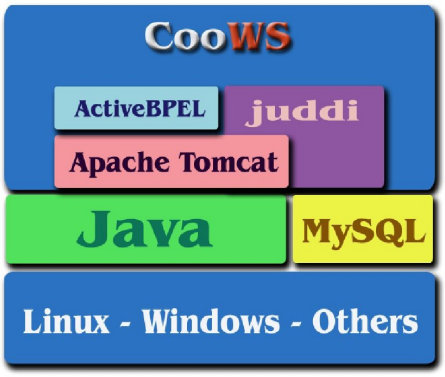

Figure 3: The CooWS platform and its implementation

trusted agents' set includes the sender) are retrieved from the agent's plan library. If the set is not empty, a `CooEventProvided` including the set of relevant plans is built and is dispatched to the requesting agent. The delivery of the message is demanded to the `CooAgentsDirectory` module.

In order to make the CooWS platform work, a Java package named `org.coows.engine`, sketched in Figure 2, has been defined. It provides functionalities for ensuring the persistence of the agents' state, for handling the global event queue, for managing the UDDI catalogue to which all the agents must register in order to enter the CooWS platform, and for accessing the BPEL engine. The implementation of the CooWS platform, downloadable from the web site `http://coows.altervista.org` and graphically described in Figure 3, relies entirely on opensource tools that include ActiveBPEL, Apache Tomcat and Axis, jUDDI, UDDI4J, and MySQL.

4 Related and Future Work

In this paper we have described CooWS, an implemented system, that represents a very preliminary step towards an adaptive BDI framework based on WS. The advantages of integrating CooBDI and WS are twofold. On the one hand, CooWS provided an hint to implement BDI-style agents that are situated in a real software environment, the Web. Very few applications of BDI agents situated in the Web can be found in the literature, and this hint might be a stimulus to the BDI community for strengthening the research in this direction. On the other hand, it made possible to understand some limitations of current WS technology with respect to our long term research, namely the limited support given to adaptivity of procedural knowledge, and gave hints on an architecture both for learning agents and the environment in which they have to act.

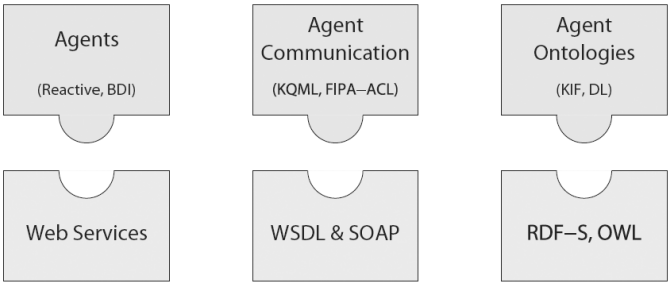

Figure 4: Contrasting WS technologies and their analogies with MAS techniques, from [13]

The most recent literature in the agents' field devotes a large amount of space to describing the existing and potential relationships between agents and WSs. For example, in the last issue of the AgentLink III Agent Technology Roadmap, edited in September 2005 [7], we can read that

> *... Web services thus provide a ready-made infrastructure that is almost ideal for use in supporting agent interactions in a multi-agent system. More importantly, perhaps, this infrastructure is widely accepted, standardised, and likely to be the dominant base technology over the coming years. Conversely, an agent-oriented view of web services is gaining increased traction and exposure, since provider and consumer web services environments are naturally seen as a form of agent-based system.*

A very nice representation of the key agent techniques, and their approximate Web service equivalents, has been given by C. Walton in [13], where agents have their counterpart in WSs (that can be expressed in BPEL), agent communication is put in relation to SOAP and WSDL, and agent ontologies are related to RDF/S and OWL (Figure 4). In this paper, C. Walton proposes to decompose agents into a stub, that executes Agent Interaction Protocols and is responsible for communication between agents, and a body, which encapsulates the reasoning processes, and is encoded as a set of decision procedures. Both the stub and the body are implemented as WSs. A tool for composing services, called MagentA [14] can be used to execute the protocols defined for agents.

In [3] P. A. Buhler and J. M. Vidal, express their point of view on the relationships between agents and web services, that

> *can be summarized by the aphorism "Adaptive Workflow Engines = Web Services + Agents". In this context, the Web services provide the computational resources and the Agents provide the coordination framework. We propose the use of the Business Process Execution Language for Web Services (BPEL4WS) as a specification language for expressing the initial social order of the multiagent system.*

Another proposal that is driven by the will of exploiting the WS technologies for implementing agents is that by K. Sycara, M. Paolucci, J. Soudry, and N. Srinivasan [12], who suggest to extend the OWL-S Model Processing Language by adding a new statement called `exec`. The `exec` statement takes a process model as input and then executes it in order to support a broker agent in both discovery and mediation.

The main limitation of CooWS, is that its development is still in its early stages and we have not produced significant examples yet. We are currently working on the implementation of "digital butlers" that query Google (which can be accessed as a web service) to arrange travels and to organise meetings for their principals. The plans available to the digital butlers do not cover all the requests that may arrive from their principals, and the lack of plans for coping with an incoming request fires the collaborative exchange of plans.

Currently, our main direction of research is to implement this case study in order to obtain experimental results on the CooWS framework. In the future, we are willing to explore:

1. The ability to integrate an ontology into the system, so that matching between events and triggers of plans can become more sophisticated than a simple comparison of strings; in

particular, we plan to follow the proposal of N. Srinivasan, M. Paolucci, K. Sycara [11], of using OWL-S plus UDDI to implement an advanced matchmaker.

2. The ability to dynamically update the set of trusted partners following reputation mechanisms such those suggested by J. Sabater in [10].

3. The theoric issues related to the definition of a BDI logic for collaborative agents. One of the authors of this paper is also involved in the definition of an ATL-based logic for modelling cooperative BDI-agents, referred to as BDI^{ATL} [8], that seems very promising for modelling *cooperation modalities* among rational agents.

References

[1] D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI model with cooperativity. In *Post-proc. of DALT'03*, pages 109–134. 2004.

[2] D. Ancona, V. Mascardi, J. F. Hübner, and R. H. Bordini. Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange. In *Proc. of AAMAS'04*, pages 698–705. 2004.

[3] P. A. Buhler and J. M. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management*, 6:61–87, 2005.

[4] M. N. Huhns. Agents as web services. *Internet Computing*, 6(4):93–95, 2002.

[5] N.R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.

[6] M. Luck. Guest editorial: Challenges for agent-based computing. *Autonomous Agents and Multi-Agent Systems*, 9(3):199–201, 2004.

[7] M. Luck, P. McBurney, O. Shehory, S. Willmott, and the AgentLink Community. *Agent Technology: Computing as Interaction – A Roadmap for Agent-Based Computing*. AgentLink III, 2005.

[8] R. Montagna, G. Delzanno, M. Martelli, and V. Mascardi. BDI^{ATL}: An alternating-time BDI logic for multiagent systems, 2005. In this volume.

[9] A. S. Rao and M. P. Georgeff. BDI agents: from theory to practice. In *Proc. of ICMAS'95*, pages 312–319, 1995.

[10] J. Sabater. Trust and reputation for agent societies. *IIIA Monographs*, 20, 2003.

[11] N. Srinivasan, M. Paolucci, and K. Sycara. Adding OWL-S to UDDI, implementation and throughput. In *Proc. of SWSWPC'04*, 2004.

[12] K. Sycara, M. Paolucci, J. Soudry, and N. Srinivasan. Dynamic discovery and coordination of agent-based semantic web services agents. *IEEE Internet Computing*, 8(3):66–73, 2004.

[13] C. Walton. Uniting agents and web services. *AgentLink News*, 18:26–28, 2005.

[14] C. Walton and A. Barker. An agent-based e-science experiment builder. In *Proc. of the 1st International Workshop on Semantic Intelligent Middleware for the Web and the Grid*, 2004.