Semantic Agent for Distributed Knowledge Management

Julien Subercaze, Pierre Maret

Semantic Agent for Distributed Knowledge Management

Julien Subercaze and Pierre Maret[1]

Introduction and motivation

At the beginning of the decade, the Agent Mediated Knowledge Management workshops series [23, 1, 13] as well as Bonifacio's theoretical approach [3] layed the foundations of a new field of distributed knowledge management based upon the agent paradigm. The agent based approach enables key features for knowledge management. The local management of knowledge by agents allows going beyond the limitations of centralized knowledge management. Thus, knowledge can be maintained in each agent at a coarse-grained level, with different representations. Interactions between agents permit us to take into account the social aspect of knowledge and are well suited to represented organizational memories [9]. Van elst also outlined the importance of pro- activeness for knowledge management [22].

In the mean time, the publishing of the agent roadmap in 2003 [17] pointed out the lack of connection between multi-agent systems and semantic web technologies. Since then, many applications and frameworks have been developed to bridge this gap. Semantic web languages and tools are now widely used to represent agents' knowledge. TAGA [25] uses OWL and RDF as knowledge representation in the field of a trading agent competition, using a FIPA compliant framework. AgentOWL [15] extends JADE agents with OWL support for their knowledge Base (KB). It also introduces an OWL based semantic agent model. Knowledge agents, introduced by [2], are used for domain specific web search. In this application, agents' KBs are represented in RDF. RDF is also used in CORESE [8] which is a semantic web search engine for corporate knowledge developed within the COMMA (Corporate Memory Management through Agents)

[1] Julien Subercaze

Laboratoire Hubert Curien, Saint-Etienne F-42000, France e-mail: julien.subercaze@univ-st-etienne.fr

Pierre Maret

Laboratoire Hubert Curien, Saint-Etienne F-42000, France e-mail: pierre.maret@univ-st-etienne.fr

European IST project. The JADE framework, which is currently the most used in research and industry supports natively RDF representing agents' knowledge. These examples show us that semantic web technologies are widely used for representing agent knowledge, and that we can clearly state that the connection between agent-based knowledge management and semantic web has been made.

However, this example presents the use of semantic for the representation of agent's knowledge and not for the dynamic part of the agent: its behavior. Katasonov proposed a Semantic Agent Programming Language (S-APL) [14], based on BDI reasoning, in which agent behavior are semantically described. Behaviors remain programmed in JAVA but are described in RDF syntax. A closed-world reasoner (CWM[2]) is used for BDI support. S-APL has three main drawbacks. First the reasoning is restricted to the closed world assumption whereas semantic web languages such as OWL make the open world assumption. Secondly each new function in S-APL has to be programmed in JAVA, which reduces the interoperability between agents. An agent having some new functions will not be able to transfer its behavior to another agent since the latter does not own the JAVA code implementing the functions. Using this approach, agent behavior programming is not taking advantage of the semantic web technologies and limits interoperability.

Our motivation is to program agent behavior using semantic web standards using a finite number of actions that will be used to build complex behaviors. This language should not refer explicitly to a lower level language and should support open world reasoning. The latest advances in the field of semantic web have enabled rule languages supporting open world reasoning. We base our approach on the use of semantic rule language to program semantic agents. We aim at designing agents having a knowledge base and behavior base represented using the same syntax. The use of OWL for knowledge base and semantic rules for behavior allows this feature. In the next section we first discuss the choice of the semantic rule language that will be used to program agents and then present the design of an agent programming language and the resulting agent architecture. In section 1.3 we describe the resulting ontological agent model. Section 1.4 shows a practical example of a SAM behavior and details the different steps of its execution. Section 1.5 concerns the implementation of the SAM prototype. We discuss in section 1.6 the perspectives of application of semantic agents in distributed knowledge management. Our conclusions are presented in section 1.7.

[2] http://www.w3.org/2000/10/swap/doc/cwm.html

Building agents with semantic rules

Semantic rules are an important part of the Semantic Web project. Figure 1.1 shows the current status of specification in the Semantic Web layer cake. The logic part, which is of primary interest for us, is still work in progress. This part describes the semantic rule languages. Currently, for this layer, two proposals are pending. The most well known one is the Semantic Web Rule Language (SWRL)[3] [12]; it is based on a combination of the OWL-DL with the RuleML language. The second proposal is the Web Rule Language (WRL[4]) initiative that was in influenced by the Web Service Modeling Language WSML. Whereas WRL is at a draft stage, the semantic web community is focusing its research towards SWRL. Indeed, software such as Protégé[5], Pellet[6] and Jess[7] already provide support for SWRL. Due to these advances in implementation, it is now possible to develop agents based on semantic rules. Thus our choice naturally went to SWRL for the design of the Semantic Agent Model (SAM). SWRL presents two main advantages compared to other rule languages. First because it is an OWL based language, it allows writing the rules in terms of OWL concepts (i.e. classes, individuals, properties and data values). To these OWL concepts, the SWRL specification adds several built-ins functions for comparisons, math, strings and time [12]. From a more agent programming point of view, concepts of the agent knowledge base can be directly treated in the rule language without loss of expressivity.

[3] http://www.w3.org/Submission/SWRL/

[4] http://www.w3.org/Submission/WRL/

[5] http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab

[6] http://clarkparsia.com/pellet/

[7] http://herzberg.ca.sandia.gov/jess/

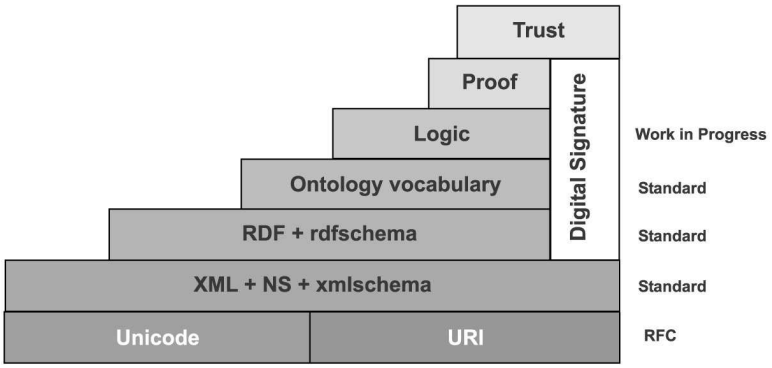

Fig 1.1 TheSemantic Web Layer Cake

The second benefit of SWRL resides in its logical foundations. SWRL combines OWL-DL (decidable version of OWL) with Rule Markup Language (RuleML). Thus, it can be roughly considered as the union of Horn-Logic and OWL based on the description logic SHOIN. Consequently the expressivity of SWRL comes at the price of decidability [19]. SWRL is not decidable (SWRL full). However, a subset called DL Safe SWRL rules is decidable. For the agent development and for reasoning, DL Safe SWRL rules are more expressive compared to other rule languages. Indeed, most of the rule-based agents are based on Prolog supporting Horn Clauses. Researches are currently going on for implementing DL reasoning in Prolog but none is currently available for agents. Practical advantages for using Description Logic in the field of Multi-Agent Systems (MAS) has been shown in [18], especially in the field of information retrieval.

Architecture design

Programming agent behavior using a rule language can be carried out in two ways. The first way consists in extending a logic programming language in order to support traditional agent features (i.e. message passing, threading, etc.). The second way consists in building a layered architecture using the rule language at an upper layer. Some agent features are delegated to a lower layer. In this type of architecture, the lower level language (i.e. Java, C++,etc.) is commonly used to handle communication, file access, thread management, etc. The main idea behind this approach is to reuse the required features for MAS that are already implemented in another language and to define an agent interpreter to support a particular architecture, such as BDI for instance. The literature shows examples of both approaches. Clark et al. [7] follows the first approach by extending Qu-Prolog with multi-threading support and inter-thread message communication. However,

this approach is not scalable and does not comply with the Agen Communication Language (ACL) as specified by the FIPA[8]. FIPA-ACL is currently recognized as the standard for agent communication and ensures interoperability between MAS frameworks. S-APL that we discussed in the previous section follows the same approach but some direct calls to JAVA functions are directly inserted into the rules.

SAM Architecture

Standard MAS languages rely on the second approach. Agent0, the first agent dedicated language, which is an implementation of Shoham's Agent Oriented Programming was developed on top of LISP. Similarly, 3APL, 3APL-m, JA- SON and the BDI agent system Jadex are based on JAVA. Our architecture follows the same approach. The specificity of our approach is to rely on a finite number of actions, in order to ensure interoperability of agents' behaviors.

In short, our approach results in the layered architecture (Fig.1.2) :

1. Knowledge Base
2. Engine
3. MAS Framework and low level actions

[8] http://www.fipa.org/repository/aclspecs.html

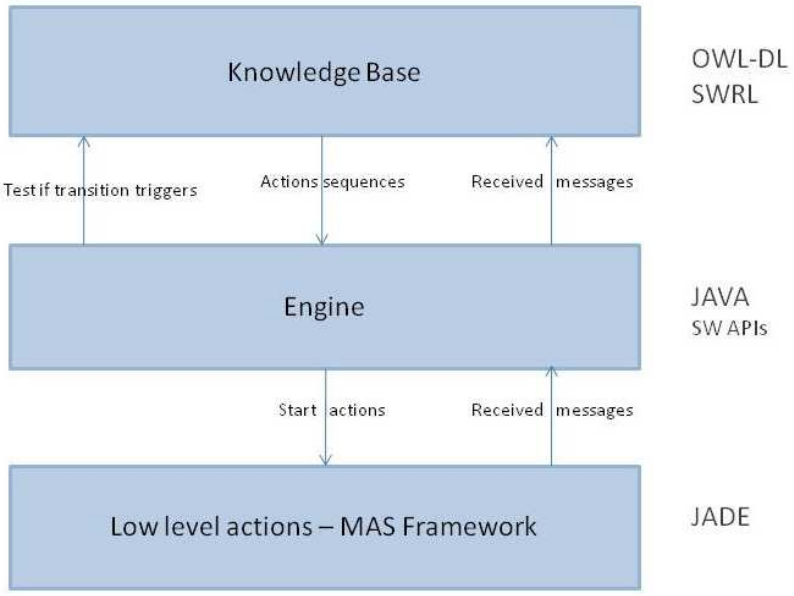

Fig. 1.1 : SAM Agent architecture

Knowledge Base

The knowledge base is the upper layer of the SAM architecture. The knowledge base contains the knowledge of the agent which, in our approach, is composed of static knowledge and behavior. Agent behaviors are expressed using SWRL rules. As SWRL is based upon OWL, Terms (i.e. OWL concepts) of the knowledge base are directly manipulated in the rules. Terms of the knowledge base can appear in both antecedent and consequent of the rules. A formal specification of the rule syntax is given in section 1.2.5.

Engine

As SWRL built-ins do not cover all the requirements for agent programming, we have introduced additional low level actions (3rd layer) and a link between the rules and these actions. This link is given by a middle layer, which is the control structure that interfaces the rules contained in the knowledge base and the low level actions. Rules from the knowledge base are red by the engine, one at a time. If a rule requires a call to low level actions, the engine layer carries out this call.

Low level actions and MAS Framework

This layer contains the implementation of the low level actions that are complementary to SWRL built-ins. An extensive list of these actions is given in section 1.3. Notice that these actions are introduced as instances of OWL class Actions in the syntax of the rules (1st layer). Communication between agents relies on an existing MAS framework. Messages are structured following the FIPA-ACL standard, and consequently the MAS framework has to be FIPA compliant (our implementation is based upon JADE which is FIPA compliant). Messages from other agents are received through the MAS framework, then converted into an OWL representation and finally added to the knowledge base.

Control Structure

Rule-based agents constitute an important part of the research on MAS. In [11], Hindriks et al. define the requirements for a minimal agent programming language that includes rules and goals. They also defined formalization tools that were applied to three standard agent programming languages AGENT-0[21], AgentSpeak(L)[20] (that was later implemented and extended in JASON[4]) and 3APL[10]. Their definition of an agent program for goal directed agents includes a set of rules called the rule base of the agent. They identify rule ordering as a crucial issue in rule-based agents. However, this presents us with the following problem: when several rules from the rule set can be fired, there must be an order to determine the sequence of execution of those rules. So the order in which the rules will be sorted must be defined. Hindriks et al. [11] proposed that all rules fall into one of the following categories: reactive(R), means-end(M), failure(F) and optimization(O) with an order based on intuition :

$$R > F > M > O$$

As SWRL doesn't support rule ordering, we are also confronted with the same issue. However, instead of deciding an arbitrary order, we have decided to use another model of behavior, a slightly modified version of the Extended Finite State Machine (EFSM) model [6]. EFSM guarantees the execution of only one rule at a time. In EFSM, transitions between states are expressed using IF statements. A transition is fired when trigger conditions are valid. Once the transition has been fired, the machine is brought from the current state to the next state and the set of specified operations are performed. Our choice is to use a finite number of actions (called atomic actions) to fulfill basic MAS requirements. We differentiate two kinds of atomic actions, external and internal. Internal actions have an effect on the agent internal knowledge Base.

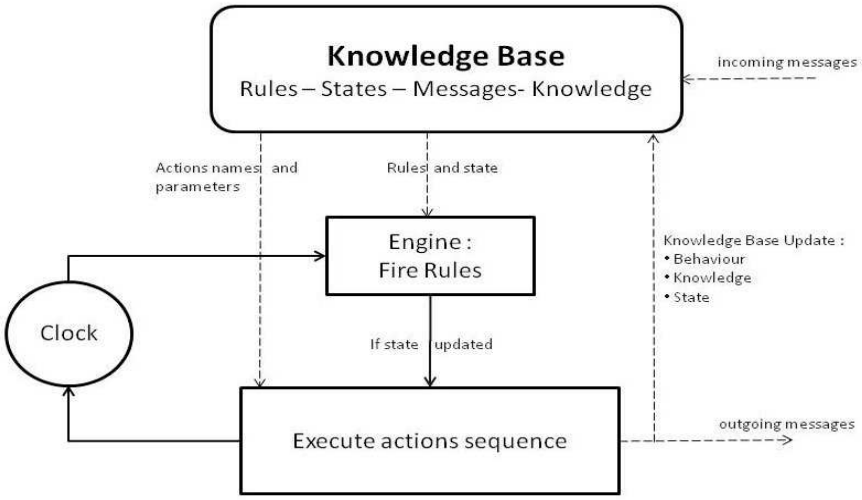

Fig. 1.2 SAM Agent Interpreter

External actions are the interactions of the agent within its environment. These actions include environment perception, action on the environment, message reception and emission. External actions are not included in SWRL built-ins whereas a subset of internal actions is. In section 1.3 we detail the list of atomic actions that are not SWRL built- ins. A deterministic EFSM is a restriction of EFSM in which there is at most one possible transition for each state and set of triggering conditions. We used this restriction to ensure that on one rule can be triggered at a time. A pseudo code algorithm for the interpreter is defined in algorithm 1.

Algorithm 1: SAM Interpreter

```
begin
    CurrentState ⟵ sBegin
    while CurrentState ≠ sEND do
        temp ⟵ nextStateValue()
        if temp ≠ currentState then
            removeProperty(currentState, stateValue)
            actionList ⟵ getActionList()
            if executeAction(actionList) then
            addProperty(currentState, temp)
            else addProperty(currentState, errorState)
end
```

Execution Stack

The behavior of an agent can be seen as program executed by a computer. In the same manner as for computer programs, agent behavior should be able call sub behaviors. We designed an execution stack to maintain the history of behavior calls, and the state of the behavior that issued the call. For example let us consider an agent currently at the state A, and its current behavior is *GetInfo*, called with the parameter *Bob*. If the next action is to load the behavior *SearchPicture* with the same parameter *Bob*, this behavior will become the current behavior and will be placed on top of the stack over *GetInfo*. The figure 1.4 depicts the stack before and after the transition. In the OWL implementation, the current behavior is set with the property *hasBehavior* on the individual *currentBehavior*.

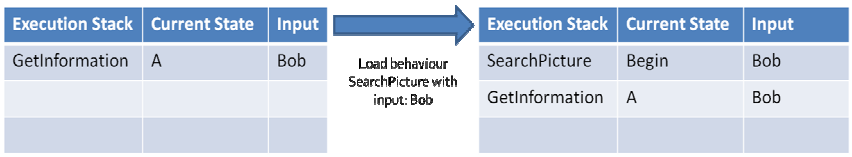

Fig. 1.3 Stack Evolution after loading of a behavior

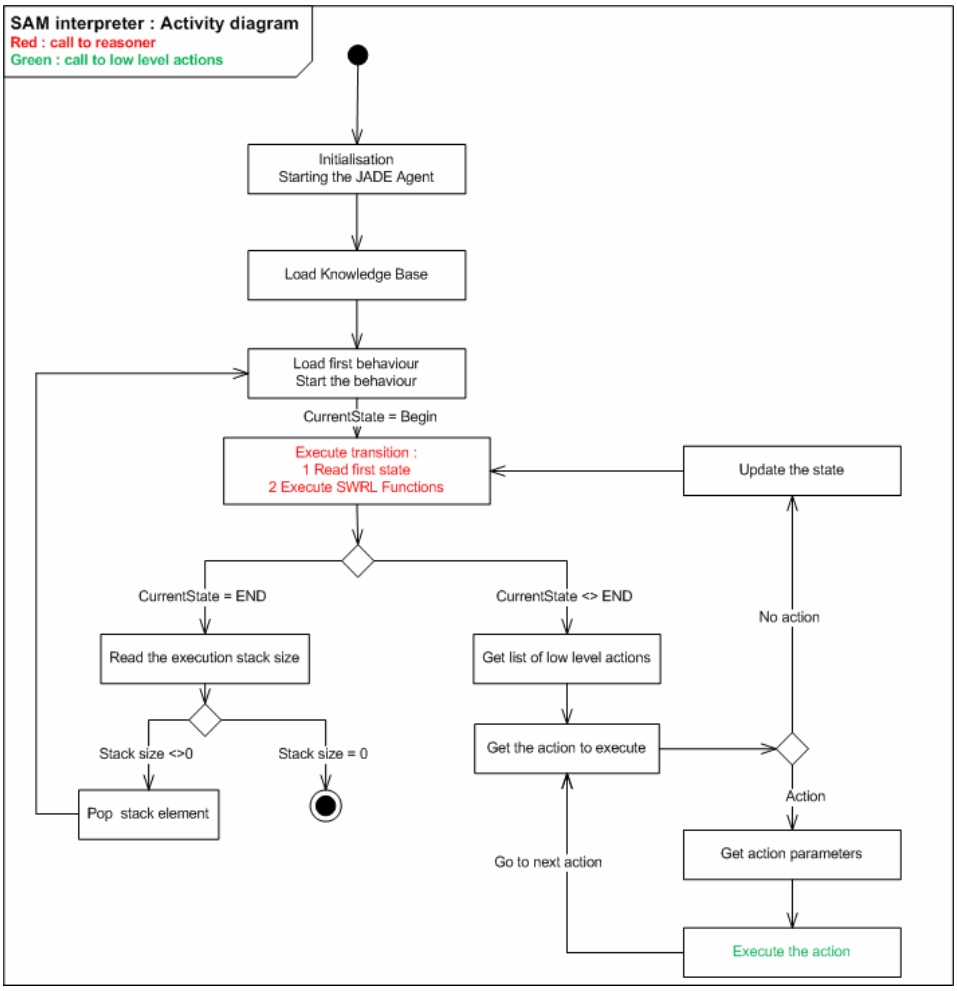

Fig. 1.4 : Interpreter activity diagram

Language Syntax

The syntax of the rule language that we designed (given in figure 1.2.5) is expressed in Extended Backus-Naur Form (EBNF). This syntax is based on the existing SWRL EBNF syntax as specified in [12]. SAM grammar is included in the SWRL grammar. In the antecedent of a SAM rule (*SAMantecedent*) it is mandatory to specify to which state the rule applies. This is set up by the *hasStateValue* property. The previous property, *currentState*, ensures that the rule will be fired

when the current state of the EFSM is the one to which the rule applies. The second part of the antecedent contains the triggering conditions. In this part, conditions under which the transition will be triggered are defined. The range of these conditions is the knowledge base of the agent. These conditions are represented by atom* which is not modified from the original SWRL specification. Conditions can test the validity of class belonging, property between classes or between individuals, including received messages.

The rule consequent term (*SAMconsequent*) specifies the destination state of the transition and the sequence of atomic actions to be executed. Each action has different parameters. Parameters are passed using two properties, *hasParameterName* and *hasParameterValue*. The first property applies to the action which is to be executed and it specifies the name of the parameter. Then property *hasParameterValue* is applied to the name of the parameter in order to specify its value.

```
SAMrule       ::= 'Implies(' [ URIreference ]
                  { annotation } SAMantecedent SAMconsequent ')'

SAMantecedent::= currentState('i-variable')'
                 hasStateValue'('i-variable')' atom*

SAMconsequent::= hasNextState'('i-variable')'
                 hasActionList'(a-list')' atom*

a-list        ::= hasValue(action) hasNext(a-list)
                | endlist

action        ::= URIreference hasParameterName(a-name)

a-name        ::= hasParameterValue(i-object)

atom          ::= description '(' i-object ')'
                | dataRange '(' d-object ')'
                | individualvaluedPropertyID '(' i-object i-object ')'
                | datavaluedPropertyID '(' i-object d-object ')'
                | sameAs '(' i-object i-object ')'
                | differentFrom '(' i-object i-object ')'
                | builtIn '(' builtinID { d-object } ')'

builtinID     ::= URIreference

endlist       ::= URIreference

i-object      ::= i-variable | individualID

d-object      ::= d-variable | dataLiteral

i-variable    ::= 'I-variable(' URIreference ')'

d-variable    ::= 'D-variable(' URIreference ')'
```

Fig. 1.6 EBNF interpreted by SAM

Semantic Agent Model

The architecture, the control structure and the language syntax we have just presented before enable us to elaborate the semantic agent model. Using the previous given architecture, we built an OWL representation of the agent with different components (Figure 1.7. In accordance with the previous section, the model holds a finite number of states and of atomic actions, as well as the parameters for the actions. We defined two specials states, *sBegin* and *sEnd* that specify the beginning and end states of the EFSM. Every agent's behavior must start with *sBegin* and end with *sEnd*. Environment interactions are described within the received messages queue.

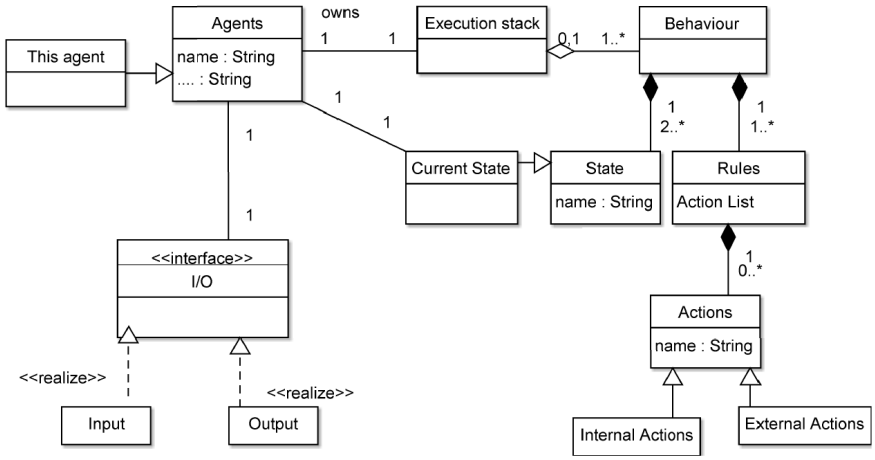

Fig. 1.7: The Semantic Agent Model

As mentioned, possible actions that are not SWRL built-ins are divided into two categories: internal and external actions. Here we detail the different atomic actions that are required in both categories (Figure 1.8 presents the different actions by layer).

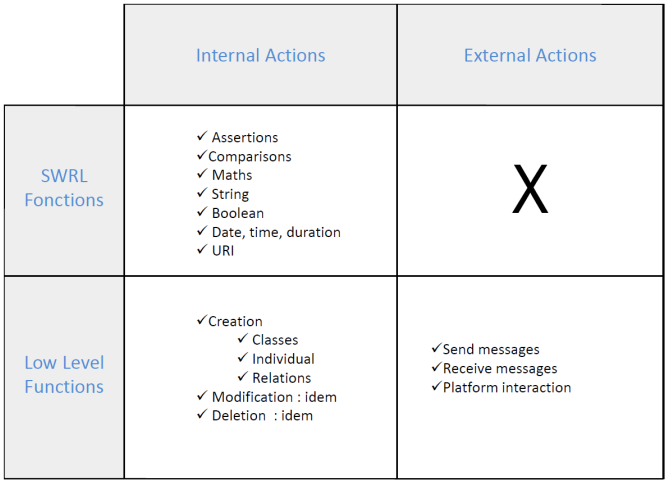

Fig. 1.8 : Actions by layer

Internal Actions: agent knowledge is expressed using OWL concepts: classes, properties, individuals and data value. For each concept, three basic operations are needed: creation, modification, deletion. Unfortunately only the first one is supported by SWRL built in. SWRL supports assertion but does not support negation. In practical terms, it is possible to assert that properties apply to individuals or classes in the rule consequent. The following example is taken from the SWRL proposal document and shows the assertion of the uncle property by composing parent and brother properties:

$$parent(?x, ?y) \wedge brother(?y, ?z) \Rightarrow uncle(?x, ?z) \tag{1.1}$$

However the following rules (2,3) are not possible since SWRL neither supports negation as a failure (2) nor non-monotonicity (3). Hence it is not possible to withdraw information using the rule consequent.

$$\neg Person(?x) \Rightarrow NonHuman(?x) \tag{1.2}$$

$$parent(?x, ?y) \wedge brother(?y, ?z) \Rightarrow \neg aunt(?x, ?z) \tag{1.3}$$

As only creation is possible using SWRL (at a higher level), we define additional actions at lower level:

- modify/remove property
- modify/remove class belonging from a resource
- modify/delete individual

- modify/delete datarange property

Internal actions, belonging to SWRL built-ins are executed by the rule engine. Other internal actions, the low level actions are called by the agent interpreter.

External Actions refer to the agents' interactions with their environment. We restrict our scope to software agents that evolve in an electronic environment. Interactions are then limited to message exchanges between agents. We rely on the FIPA ACL specification for the message structures. Received messages are stored in the message list. In the agent's KB, messages are put in a list *ReceivedMessages* that is an instance of OWLList. Eventually there are two basic external actions, *sendMessage* and *receiveMessage*. Following the ACL specification, forging a message requires several parameters; among them we can cite sender, receiver, ontology used, performative and so on. From those simple actions, it is possible to build complex interactions between actions. For instance FIPA ACL specifies an extensive communicative act library including query-answer, contracting, proposal, subscribing. Different fields of the message are represented in the OWL knowledge Base using properties, i.e. *hasPerformative*, *hasContent*, *hasSender*.

Defining new actions

The agent model contains a finite list of basic actions for communication and knowledge base management purpose. In SAM there are two approaches to define new actions. The first is to extend the set of available of low level actions. The second one is to define new actions by combining the existing ones.

Defining new atomic acti
guage. This approach is then
thors. It should be applied o
approach consists in defining
noted these actions as compo
a kind of composed action si
by transition. To define new
for agents' behaviors. Compo
These rules should only be a
these rules are not stored as
they are instances of the class
rule (in Manchester Syntax

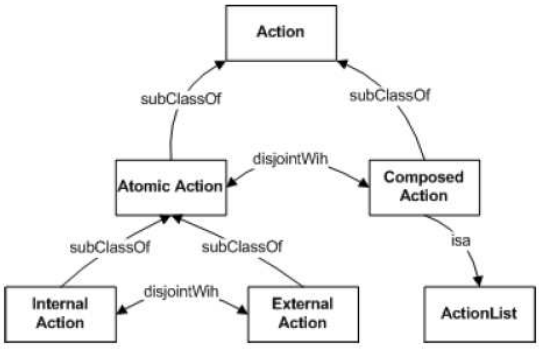

Fig. 1.9 Ontology of the actions in SAM

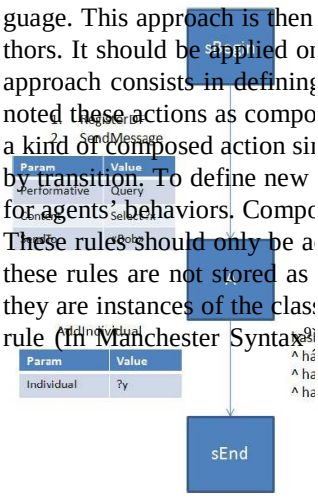

Fig. 1.10 Illustrative example (a), Scenario

the following. Let us assume that the agent is firing a transition between state A and B. During this transition a com- posed action called comp is to be executed. First the engine removes the rules of the current behavior from the knowledge base and stores them using a string representation. The engine also keeps tracks of the current state and transition sequence that was executed.

The engine sets the current state of the agent to an intermediate state *sBegin*. Then it extracts the string representation of the rules from comp and adds them to the knowledge base. The composed action is then executed following the same way as an agent behavior. Once the action finished, the engine removes the rules and sets back the agent's behavior context. Note that this process is recursive and a composed action can call another composed action.

Example

To illustrate the mechanism behind semantic agents, we take a simple example and process the several steps of the execution. The example has following content: start the agent Alice, register it with the directory facilitator of the framework, and send a query to the agent *Bob*. If a received message is from *Bob* and if this message has the performative answer then Alice adds the content of the answer into its knowledge base. The resulting EFSM is depicted in figure 1.10. The left column of the figure describes the low level atomic actions executed during a transition. Triggering conditions, contained in the antecedent of the rule, are on the right side of the picture. The first transition is conditions free. If *Alice* is in the *sBegin* state then the transition to the state *A* will occur. Actions related to the transition are executed as a sequence. The next actions are executed only if the previous succeeded. The action *registerDF* is executed first. If it is successful (returns true) a message with the query performative and containing a query is sent to agent Bob. The rule used to describe this transition is presented below in a human readable syntax :

$$
\begin{aligned}
&CurrentState(?x)\\
&\wedge hasStateValue(x, sBegin)\\
&\wedge NextState(?y)\\
&\Rightarrow hasStateValue(y, A)\\
&\wedge hasContents(ActionSequence, registerDF)\\
&\wedge hasNext(ActionSequence, item)\\
&\wedge hasContents(item, SendMessage)\\
&\wedge hasParameterName(SendMessage, Sender)\\
&\wedge hasParameterValue(SendTo, Bob)\\
&\ldots \textit{same with other parameters}\\
&\wedge hasNext(item, endList)
\end{aligned}
$$

Within the architecture, the engine checks whether a transition occurs in requesting the *NextState* value to the knowledge base . If this value is different from the *CurrentState* then a transition is enabled. Then the engine retrieves the values of *ActionSequence*, with the respectives parameters. *ActionSequence* is a linked-list (Fig.1.11) in which each item has the *hasParameterName* property. The value of the parameter is specified in the *hasParameterValue* property. The structure of the list of actions follows the OWL model depicted in figure 1.7. The *Sendmessage* instruction is linked to its parameters using properties as described in figure 1.11. The second transition contains triggering conditions regarding the received message. As *Alice* sent a query to *Bob*, the next step of *Alice's* behavior may be to handle the answer from *Bob*. Thus, we specify a condition on received messages to ensure that *Bob* is the sender and that the message is of type *Answer:*

$$
\begin{aligned}
&CurrentState(?x)\\
&\wedge hasStateValue(x, A)\\
&\wedge NextState(?y)\\
&\wedge hasReceived(?z)\\
&\wedge hasPerformative(z, Answer)\\
&\wedge hasSender(z, Bob)\\
&\wedge hasContent(z, ?w)\\
&\Rightarrow hasStateValue(y, sEnd)\\
&\wedge hasContents(ActionSequence, AddInvidual)\\
&\wedge hasNext(ActionSequence, endList)\\
&\wedge hasParameterName(AddInvidual, name)\\
&\wedge hasParameterValue(name, w)
\end{aligned}
$$

We will now detail the interactions between the different layers in the architecture during the execution of the first transition.

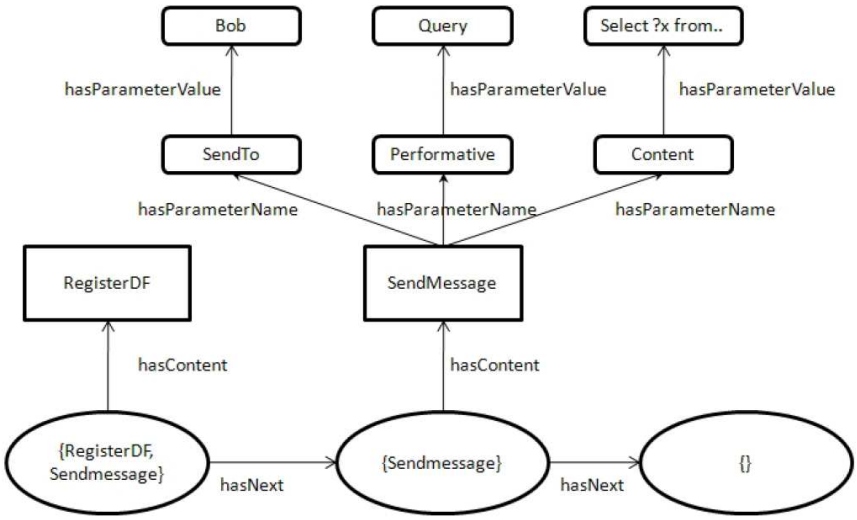

Fig. 1.11 Illustrative example (b), ActionList data structure

Execution Phase

Representing the action execution on a timeline following the architecture as in Section 1.2.1 is represented in Figure1.12. It follows Algorithm 1. The SAM engine firstly enquires of a rule triggering, in this case, the knowledge Base query returns *NextState* = A. As A != *sBegin*, the engine retrieves the current list of actions containing *RegisterDF* and *SendMessage*. Actions are performed sequentially. First *RegisterDF* is executed and if it returns true then *SendMessage* is be executed. When both actions succeeded, the current state of the agent is updated to *NextState* value; in this case it is state A.

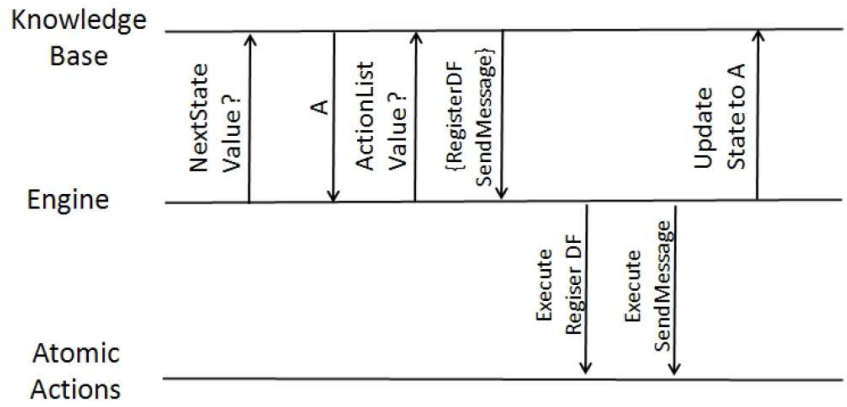

Fig. 1.12 Illustrative example (c), Flow chart of the transition from Begin to A

Implementation

We have developed a JAVA interpreter that communicates with the knowledge Base using the Protege-OWL API 10 Pellet is used in combination with Jena 11 as a SWRL reasoner. The JADE framework is used for the low level external actions. The framework handles agent registration, service discovery and message passing. It also provides an environment that is FIPA-ACL compliant. One implementation issue we encountered was that OWL does not support RDF lists. An OWL equivalent called OWLList has been developed and is used to represent action sequences and the queue of received messages. A first version of the open-source prototype

is available online 12. Besides the validation of our model, the implementation prototype presents some limitations. Nowadays the status of SWRL reasoners is not satisfying because none of them fully support the SWRL specification. We have used Pellet as a SWRL reasoner, since it is currently the most advanced open- source implementation of SWRL. As developments stands at the moment, several important features are not supported by Pellet, for instance some SWRL built-ins are not yet available. The implementation results show the feasibility of the proposal and we intend to further develop the prototype to make it fully suitable for the development of applications. Semantic Web technologies is a field where advances take place. Current restrictions on SWRL support should no longer be an issue since advances in the field of the Semantic Web technologies occur very rapidly and regularly. Finally, this implementation of the prototype allowed us to validate our approach and to identify the limitations.

Perspectives

One of the primary advantages of agent based knowledge management over the classical centralized approach is the proactiveness of the agents [22]. Proactiveness is the ability of agents to initiate changes and to take initiatives. It is opposite to the reactive approach where agents react to stimulus or changes in their environment. Concretely, in the agent design, the proactiveness is implemented in different agent behaviors. The benefit of semantic agent programming is to enable the semantic description and exchange of the agents' behaviors. Thus, agents evolving in cooperative environments are able to learn behaviors from other agents. In common knowledge management frameworks the nature of knowledge exchanges between agents is limited to static knowledge. Semantic agent programming our proposal allows agents to share not only static knowledge but also dynamic knowledge insofar as agents are able to exchange their own behaviors.

This ability opens a broad scope of applications and questions. In the same way as for static knowledge exchanges, behaviors exchanges are subject to trust and security issues. Moreover, behaviors are executed by the agents and the execution of a malicious behavior could lead to serious security is- sues. We believe that existing cryptographic and trust mechanisms can easily be adapted to the exchanges of semantic behaviors.

From a larger point of view, this approach takes the opposite of the current software as a service trend. In the service approach, providers share (or sell) the use of their services, but not the implementation of the service. With the semantic

behavior exchange approach, agents share the implementation of the service. Clearly this approach is valuable in cooperative environment, for example when agents belong to the same organizations. Several studies have shown that reactive and proactive agents lead to the same performance among organizations [16, 24, 5].

The consequence of semantic behavior sharing on proactive agents is important. Agents are now able to learn behaviors from other agents and to recombine, evaluate, modify these behaviors to enhance their proactive capabilities. These missing abilities were not taken into account in former studies and we believe it can greatly improve the performance of proactive agents.

Conclusion

In this chapter, we presented how the next generation of Semantic Web technologies can be applied in MAS programming. We discussed the limitations of current semantic approach and noticed the lack of semantic programming for agents' behaviors. State of the art frameworks are limited in terms of interoperability. To bridge this gap, we designed agent architecture to support behavior programming with semantic rules using a finite number of actions, identical for each agent. This approach allows the sharing of behaviors between agents, without relying on a specific lower level language. We detailed the three layer architecture, the language syntax and the interpreter. Afterwards we discussed the advantages of semantic agent programming in terms of knowledge management especially in the field of proactive cooperative agents.

References

1. V. Dignum A. Abecker, L. van Elst, editor. *Proceedings of the ECAI-2004 Workshop on Agent-mediated Knowledge Management (AMKM-2004)*, 2004.
2. Yariv Aridor, David Carmel, Ronny Lempel, Aya Soffer, and Yoëlle S. Maarek. Knowledge agents on the web. In *CIA '00: Proceedings of the 4th International Workshop on Cooperative Information Agents IV, The Future of Information Agents in Cyberspace*, pages 15–26, London, UK, 2000. Springer-Verlag.
3. M. Bonifacio, P. Bouquet, and P. Traverso. Enabling distributed knowledge management: Managerial and technological implications. *RMA TIQUE*, page 23.
4. R.H. Bordini and J.F. Hubner. BDI agent programming in AgentSpeak using Jason. In *Proceedings of 6th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VI)*, volume 3900, pages 143–164. Springer.
5. K.M. Carley, M.J. Prietula, and Z. Lin. Design versus cognition: The interaction of agent cognition and organizational design on organizational performance. *Journal of Artificial Societies and Social Simulation*, 1(3):1–19, 1998.
6. K.T. Cheng and AS Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th international conference on Design automation*, pages 86–91. ACM New York, NY, USA, 1993.
7. K. Clark, P.J. Robinson, and R. Hagen. Multi-threading and message communication in Qu-Prolog. *Theory and Practice of Logic Programming*, 1(03):283–301, 2001.
8. O. Corby, R. Dieng-Kuntz, and C. Faron-Zucker. Querying the semantic web with corese search engine. In *ECAI*, volume 16, page 705, 2004.
9. F. Gandon, R. Dieng, O. Corby, and A. Giboin. A multi-agent system to support exploiting an XML-based corporate memory. In *Proceedings PAKM'00, Basel*, 2000.
10. K.V. Hindriks, F.S. De Boer, W. Van der Hoek, and J.J.C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401,
11. K.V. Hindriks, F.S. De Boer, W. Van Der Hoek, and J.J.C. Meyer. Control structures of rule-based agent languages. In *Intelligent Agents V: Agent Theories, Architectures, and Languages: 5th International Workshop, Atal'98: Paris, France, July 1998: Proceedings*, page 384. Springer, 1999.
12. I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. *W3C Member Submission*, 21, 2004.
13. Ludger van Elst Juriaan van Diggelen, Virginia Dignum and Andreas Abeckerm, editors. *Proceedings of AAMAS 2005 Workshop AMKM-2005*, July 2005.
14. A. Katasonov and V. Terziyan. Semantic agent programming language (S-APL): A middleware platform for the Semantic web. In *Proc. 2nd IEEE International Conference on Semantic Computing*, pages 504–511, 2008.
15. Michal Laclavik, Zoltan Balogh, Marian Babik, and Ladislav Hluchý. Agentowl: Semantic knowledge model and agent architecture. *Computers and Artificial Intelligence*, 25(5), 2006.
16. Z. Lin and K. Carley. Proactive or reactive: An analysis of the effect of agent style on organizational decision-making performance. *Intelligent Systems in Accounting, Finance and Management*, 2:271–289, 1993.

17. M. Luck, P. McBurney, and C. Preist. *Agent technology: Enabling next generation computing.* AgentLink II, 2003.
18. Ralf Mller, Volker Haarslev, and Bernd Neumann. Expressive description logics for agent-based information retrieval. In *Treur (Eds.), Knowledge Engineering and Agent Technology, IOS.* Press, 2000.
19. B. Parsia, E. Sirin, B.C. Grau, E. Ruckhaus, and D. Hewlett. Cautiously Approaching SWRL. Technical report, Technical report, University of Maryland, 2005.
20. A.S. Rao. AgentSpeak (L): BDI agents speak out in a logical computable language. *Lecture Notes in Computer Science*, 1038:42–55, 1996.
21. Y. Shoham. AGENT0: A simple agent language and its interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 2, pages 704–709, 1991.
22. L. van Elst, V. Dignum, and A. Abecker. *Towards Agent-Mediated Knowledge Management*, pages 1–30. Springer Berlin / Heidelberg, 2003.
23. Ludger van Elst, Virginia Dignum, and Andreas Abecker, editors. *Agent Mediated Knowledge Management, International Symposium AMKM 2003, Stanford, CA, USA, March 24-26, 2003, Revised and Invited Papers*, volume 2926 of *Lecture Notes in Computer Science*. Springer, 2004.
24. J. Xiao, R. Catrambone, and J. Stasko. Be quiet? evaluating proactive and reactive user interface assistants. In *Human-computer interaction: INTERACT'03; IFIP TC13 International Conference on Human-Computer Interaction, 1st-5th September 2003, Zurich, Switzerland*, page 383. Ios Pr Inc, 2003.
25. Youyong Zou, Tim Finin, Li Ding, Harry Chen, and Rong Pan. Using semantic web technology in multi-agent systems: a case study in the taga trading agent environment. In *ICEC '03: Proceedings of the 5th international conference on Electronic commerce*, pages 95–101, New York, NY, USA, 2003. ACM.