Semantic Agent Programming Language (S-APL): A Middleware Platform for the Semantic Web

Artem Katasonov and Vagan Terziyan
University of Jyväskylä, Finland
artem.katasonov@jyu.fi, vagan@jyu.fi

Abstract

The agent-based approach is an effective one for building middleware interconnecting distributed heterogeneous resources and providing semantic interoperability among them. On the other hand, agents need the Semantic Web technologies for flexible yet effective coordination among them with a particular issue of enabling agents to communicate not only about the domain but also about their own abilities, goals, and present and intended actions. This paper describes Semantic Agent Programming Language (S-APL) intended to be a core middleware language for the Semantic Web. S-APL integrates the semantic description of the domain resources with the semantic prescription of the agents’ behaviors. Additionally, S-APL can be used as the content language in the inter-agent communications, both in querying for data and in requesting for action.

1 Introduction

When it comes to developing complex, distributed software-based systems, the agent-based approach was advocated to be a well suited one [10]. Jennings [10] argued that agent-oriented decompositions (according to the purpose of elements) are an effective way of partitioning the problem space of a complex system, that the key abstractions of the agent-oriented mindset are a natural means of modeling complex systems, and that the agent-oriented philosophy for modeling and managing organizational relationships is appropriate for dealing with the dependencies and interactions that exist in complex systems.

We believe therefore that the agent-based approach is an effective one for building middleware platforms whose goal is to interconnect distributed heterogeneous resources and to provide interoperability among them. In [13], we presented a vision of such an agent-based semantic middleware. In this vision, every resource to be integrated has a representative, a software agent. This agent has to, among other tasks, provide the adaptation between the semantic data and any proprietary data formats and protocols that the resource uses. The semantic technologies have a two-fold value in this vision. First, they are the basis for the discovery of heterogeneous resources and data integration across multiple domains (a well-known advantage). Second, they are to be used for behavioral control and coordination of the agents representing those resources (a novel use). Therefore, semantic technologies are to be used both for descriptive specification of the services delivered by the resources and for prescriptive specification of the expected behavior of the resources as well as the integrated system. Such two-fold application of semantics can, in ideal case, lead to a “global understanding” between the resources. This means that a resource A can understand all of (1) the properties and the state of a resource B, (2) the potential and actual behaviors of B, and (3) the business processes in which A and B, and maybe other resources, are jointly involved.

The latter point above comes from the fact that flexible yet effective coordination is an important yet largely unresolved problem in the agents research field. According to available literature, one of the major directions of search for a solution is ontological approaches to coordination. E.g. [17] asserted a need for common vocabulary for coordination, with a precise semantics, to enable agents to communicate their intentions about future activities and resource utilization and get them to reason about coordination at run time. Also [11] puts as an issue to resolve the question about how to enable individual agents to represent and reason about the actions, plans, and knowledge of other agents to coordinate with them.

So far, very little has been done in this direction, however. Bosse and Treur [5] discussed that the ontological understanding among agents requires sharing the following different types of ontologies: an ontology for internal mental properties of the agent, MentOnt, for properties of the agent’s (physical) body, BodyOnt, for properties of the (sensory or communication) input, InOnt, for properties of the (action or communication) output, OutOnt, of the agent, and for properties of the external world, ExtOnt.

Using this distinction, we could describe the state of the
art as following. The work on explicitly described ontologies was almost exclusively concerned with ExtOnt, i.e. the domain ontologies. MentOnt comes for free when adopting a certain agent’s internal architecture, such as Beliefs-Desires-Intentions (BDI) model [16]. Also, the communication parts of InOnt and OutOnt come for free when adopting certain communication languages, such as FIPA’s ACL and SL [8]. However, BodyOnt, i.e. the vocabulary for describing preceptors and actuators that the agent has available, the sensory part of InOnt, i.e. the agent’s perception vocabulary, and the action part of OutOnt, e.g. the agent’s acting vocabulary, are not usually treated. However, sharing these ontologies is a necessary precondition for agents’ awareness of and understanding each other’s actions, i.e. for coordination. The article by Tamma et al. [17] is one of the first endeavors into this direction, which however only introduced and analyzed some of the relevant concepts, such as resource, activity, etc.

Consider the following example. Two robots (say, one of the U.S. and one of the E.U.) are performing exploration of the same area on Mars, and let’s assume they are in situation where they need to cooperate or at least to coordinate. Assuming that both follow FIPA ACL, they can exchange messages, understand performatives INFORM, REQUEST, REJECT, etc. Assuming that both follow BDI architecture, they share meaning of ”I intend”, ”My goal is”. Because they both designed for Mars, they hopefully have similar domain ontology concepts like ”surface”, ”rock”. Now assume that the US robot has a ”jet pack” and thus can ”fly”, while the EU robot has no idea about possibility of moving in the third dimension. How the US robot can communicate its movement intentions, so that two can coordinate?

In other words, it is not only that the Semantic Web needs agents as active software components to serve heterogeneous Web resources, but also agents need the Semantic Web technologies to enable flexible coordination among them. An issue, however, is to enable agents to understand and communicate about not only the external world, i.e. the domain, but also each other’s abilities, goals, and present and intended actions. As the set of possible abilities and actions is not finite, we could not hope for creating a single standard ontology for those. Rather, we should enable open ontological modeling of actions and abilities, so that, e.g., after reading an annotation of the ability ”to fly” the EU robot would understand what this particular ability entails.

Important related work with respect to this challenge is the research on semantic description of Web Services, with technologies like WSMO [7] and SAWSDL [18]. However, the Web Services research is concerned mainly with semantically describing the inputs and outputs of a service, while it is of no interest how the service actually functions, how it is implemented, etc. Therefore, the applicability of the Semantic Web Services technologies is rather restricted to domains where the components of distributed systems do not share resources and do not have a physical component, e.g. do not control physical processes, and thus do not have other types of interaction beyond using each other. In contrast, we aim for solutions which would be more general and cover also the domains of the Internet of Things, e.g. industrial (see [13]). In these domains, the components of distributed systems may have to understand the behavior of each other beyond the usage interfaces – to avoid negative interactions over shared resources (the classic example is two robots trying to pass through a door at the same time) or to exploit potential positive interactions (like robots passing through a door one immediately after the other, so there is a need to open and close the door only once, not twice).

In this paper, we describe Semantic Agent Programming Language (S-APL) developed to support our vision from [13] and to be the core language of our middleware platform. The goal of S-APL is to provide a basis for sharing all 5 ontologies (Ext, Ment, Body, In and Out; see above), and thus enable better understanding among agents, and ultimately effective and flexible coordination among them.

S-APL is an RDF-based language that is a hybrid of semantic rule-based reasoning frameworks such as N3Logic [4] (implemented in the CWM reasoner [2]) and agent programming languages (APLs) such as e.g. AgentSpeak(L) [15] and ALPHE [6]. From the CWM point of view, S-APL is an extension of it with common APL features such as the BDI architecture, i.e. ability to describe goals and commitments – data items presence of which leads to some executable behavior, and ability to link to sensors and actuators implemented in a procedural language, namely Java. From the APL point of view, S-APL is a language that provides all the features (and more) of existing APLs, while being RDF-based and thus providing advantages of semantic data model and reasoning. In a nutshell, our proposition is: Let’s treat agent programs as data; data that can be stored into a database, queried for, merged, shared between agents, and so on. As with any data in a distributed computer system, there are problems of other-party understanding the meaning of the data and of machine-processibility. Therefore, the utilization of the Semantic Web technology is a natural approach.

S-APL is, first of all, a language for prescribing agents’ behaviors, i.e. semantic programming them. We also show, however, that S-APL can be used as the content language in the inter-agent communications, both in querying for data and in requesting for action, i.e. in place of FIPA’s SL and W3C’s SPARQL. In [14], we described an old version of S-APL. Although being based on same basic ideas, that version used RDF/XML notation while now we use Notation3, and was rather primitive as compared to one described here. That paper can be consulted, however, for a more elaborate motivation for such an RDF-based APL. More technical de-
tails on S-APL can be found in [12].

The rest of the paper is structured as follows. Section 2 presents the agent architecture we use and the place of S-APL in it, while Section 3 describes S-APL itself. Section 4 describes how the agents communicate with S-APL as the content language. Finally, Section 5 concludes the paper.

2 Agent Architecture

The architecture of an S-APL agent is depicted in Figure 1. The basic 3-layer agent structure is common for the APL approach, see e.g. ALPHA [6]. There is the behavior engine implemented in Java, a declarative middle-layer, and a set of sensors and actuators which are again Java components. The latter we refer to as Reusable Atomic Behaviors (RABs). We do not restrict RABs to be only sensors or actuators, i.e. components concerned with the agent’s environment. A RAB can also be a reasoner (data-processor) if some of the logic needed is impossible or is not efficient to realize with S-APL, or if one wants to enable an agent to do some other kind of reasoning beyond the rule-based one.

As Figure 1 stresses, an S-APL agent can obtain the needed data and rules not only from local or network documents, but also through querying S-APL repositories. Such a repository, for example, can be maintained by some organization and include prescriptions (lists of duties) corresponding to the organizational roles that the agents are supposed to play. In our implementation, such querying is performed as inter-agent action with FIPA ACL messaging but does not involve any query or content languages beyond S-APL itself (see Section 4). As can be seen from Figure 1, agents also can load RABs remotely. This is done as an exception mechanism triggered when a rule prescribes engaging a RAB while the agent does not have it available. Thus, organizations are able to provide not only the rules to follow but also the tools needed for that.

We also equip each agent with a blackboard, through which RABs can exchange arbitrary Java objects. Similar solution can be found e.g. in the Cougaar framework [9].

Figure 1. The agent architecture

The middle layer is the beliefs storage. What differentiates S-APL from traditional APLs is that S-APL is RDF-based. This provides the advantages of the semantic data model and reasoning. An additional advantage is that in S-APL the difference between the data and the program code is only logical but not any principal. Data and code use the same storage, not two separate ones. This also means that: a rule upon its execution can add or remove another rule, the existence or absence of a rule can be used as a premise of another rule, and so on. None of these is normally possible in traditional APLs treating rules as special data structures principally different from normal beliefs which are n-ary predicates. S-APL is very symmetric with respect to this – anything that can be done to a simple statement can also be done to any belief structure of any complexity.

As Figure 1 stresses, an S-APL agent can obtain the needed data and rules not only from local or network documents, but also through querying S-APL repositories. Such a repository, for example, can be maintained by some organization and include prescriptions (lists of duties) corresponding to the organizational roles that the agents are supposed to play. In our implementation, such querying is performed as inter-agent action with FIPA ACL messaging but does not involve any query or content languages beyond S-APL itself (see Section 4). As can be seen from Figure 1, agents also can load RABs remotely. This is done as an exception mechanism triggered when a rule prescribes engaging a RAB while the agent does not have it available. Thus, organizations are able to provide not only the rules to follow but also the tools needed for that.

We also equip each agent with a blackboard, through which RABs can exchange arbitrary Java objects. Similar solution can be found e.g. in the Cougaar framework [9]. The reason for that is not to unnecessarily restrict the range of applications that could be realized with S-APL. Without such a blackboard, RABs would be always forced to translate all data into RDF (even when the S-APL code of the agent is not concerned with the content of data, or could not process it) or at least serialize it as text string to put the as object of a statement. This could restrict the performance and, more importantly, significantly reduce the wish to use S-APL. Blackboard is also necessary to accommodate objects like Socket, HttpServletResponse or similar to enable an agent to process and respond to HTTP requests, which may be needed in many applications.

With the blackboard extension, S-APL platform can be used in different ways:

- **Semantic Reasoning.** S-APL rules operating on S-APL data.
- **Semantic Data.** RABs (i.e. Java components) operating on S-APL semantic data.
- **Workflow management.** RABs operating on Java blackboard objects, with S-APL used only as workflow management tool, specifying what RABs are engaged and when.
- **Any combination of the three options above.**

The architecture of our platform implies that a particular application utilizing it will consist of a set of S-APL documents (data and behavior models) and a set of atomic behaviors needed for this particular application. Since reusability is an important concern, it is reasonable for the platform to provide some of those ready-made. Therefore, our platform as such can be seen as consisting of the following three elements: (1) the behavior engine, (2) a set of "standard" S-APL models, (3) a set of "standard" RABs. The set of currently provided standard S-APL models includes, e.g., the
communication models described in Section 4. The set of currently provided standard RABs includes RABs for such operations like downloading a web document, sending an email, loading and transforming into an S-APL presentation a text table document, loading an XML document, etc (see details in [12]).

Technically, our implementation is built on the top of the Java Agent Development Framework (JADE) [1], which is a Java implementation of IEEE FIPA specifications. JADE provides communication infrastructure, agent life-cycle management, agent directory-based discovery and other standard services.

3 Semantic Agent Programming Language

S-APL has as an axiom that anything inside an agent’s mind is a belief. All other mental attitudes such as goals, commitments, behavioral rules are just compound beliefs. Thus, an S-APL document is basically a statement of some agent’s current or expected (by an organization) beliefs.

S-APL is based on Notation3 (N3) [3] and utilizes the syntax for rules very similar to that of N3Logic [4]. N3 was proposed as a more compact, better readable and more expressive alternative to the dominant notation for RDF, which is RDF/XML. One special feature of N3 is the concept of formula that allows RDF graphs to be quoted within RDF graphs, e.g. \{:room1 :hasTemperature 25\} :measuredBy :sensor1. An important convention is that a statement inside a formula is not considered as asserted, i.e., as a general truth. In a sense, it is a truth inside a context defined by the statement about the formula and the outer formulas. In S-APL, we refer to formulas as context containers. The top level of the S-APL document, i.e. of what is the general truth for the agent, we refer to as general context or just G.

Below, we describe the main constructs of S-APL. We use three namespaces: "sapl:" for S-APL constructs, "java:" for RABs, and "p:" for RAB parameters. The empty namespace "":" is used for resources that are assumed to be defined elsewhere.

The two constructs below are equivalent and define a simple belief. The latter is introduced for syntactic reasons.

\{:room1 :hasTemperature 25\}
\{:room1 :hasTemperature 25\} sapl:is sapl:is true

The next two constructs add context information:

\{:room1 :hasTemperature 25\} :measuredBy :sensor1
\{:room1 :hasTemperature 25\} sapl:is sapl:is true ;
:measuredBy :sensor1

The former states that "sensor1 measured the temperature to be 25" without stating that "the agent believes that the temperature is 25". In contrast, the latter states both. This demonstrates a specific convention of S-APL: rather than doing several statements about one container, "[... ] P O; P O" leads to linking the statements inside the formula to two different containers. Then, using sapl:is true it is also possible to link some statements to a container and to one of its nested containers.

The goals of the agent and the things that the agent believes to be false are defined, correspondingly, as:

sapl:I sapl:want \{:room1 :hasTemperature 25\}
\{:room1 :hasTemperature 25\} sapl:is sapl:is false

sapl:I is a special resource that is defined inside the beliefs of an agent owl:sameAs the URI of that agent. A specific convention of S-APL is that e.g. "sapl:I sapl:want \{A B C\}. sapl:I sapl:want \{D E F\}" is the same as "sapl:I sapl:want \{A B C, D E F\}". In other words, the context containers are joined if they are defined through statements with the same two non-container resources.

The commitment to an action is specified as follows:

\{sapl:I sapl:do java:ubibware.shared.MessageSenderBehavior\}
\{sapl:I \}
\{ configureAs \{
  \{p:receiver sapl:is :John.\}
  \{p:content sapl:is \{ :get :temperatureIn :room1\} .\}
  \{ sapl:Success sapl:is add \{ :John :informs :temperature\} \}
\}

The "java:" namespace indicates that the action is a RAB. Otherwise, it would be an abstract action (capability) that a rule was supposed to translate into a plan. When the behavior engine finds such a belief in G, it executes the RAB and removes the commitment. In the configuration part, one may use special statements to add or remove beliefs. The subject can be sapl:Start, sapl:End, sapl:Success, and sapl:Fail. The predicate is either sapl:is or sapl:is not.

Using such statements, one can easily specify sequential plans: \{sapl:I sapl:do ... \} configureAs \{ ... \} sapl:Success sapl:is add \{sapl:I sapl:do ... \} configureAs \{ ... \} \}.

The commitments to mental actions are as follows:

\{sapl:I sapl:remove \{ :John :informs :temperature\} \}
\{sapl:I sapl:is \is \{ :John :informs :temperature\} \}

sapl:remove uses its object as a pattern that is matched with G and removes all beliefs that match. sapl:is does not need to be normally used, since just stating something is the same as adding it to beliefs. This construct is needed when one wants to postpone the creating of the belief until the stage of the agent run-time cycle iteration when commitments are treated, or when one uses as the object a variable holding the ID of a statement or a container (see below).

The conditional commitment is specified as:

\{ \{ ?room :hasTemperature ?temp :measuredBy * \}
?temp > 30 \} => \{ ... \}
?room and ?temp are variables. => and > are shorthands for sapl:implies and sapl:gt. * means “anything”. The object of sapl:gt and other filtering predicates (>=, <, <=, =, !=) is an expression that can utilize arithmetic operations, functions like abs, floor, random, etc. and string-processing functions like length, startsWith, substring, etc. When the behavior engine finds in G a belief as above and finds out that all the conditions in the subject context container are met, it copies to G, substituting variables with their values, all the beliefs from the object container. Those can be plain beliefs and/or commitments, unconditional or conditional. S-APL allows a variable value to substitute a part of a resource, e.g. "logs/?today/received". Such a liberty is in contrast with, e.g., N3Logic [4] approach where a variable value can only be a substitute for the whole resource; however, it was shown to greatly simplify the programming.

As with any commitments, the conditional commitment is removed after successful execution. In order to create a persistent rule, the => statement has to be wrapped as:

{ [...] => [...] } sapl:is sapl:Rule

It is possible to define a guard for a conditional commitment so it is dropped if the guard becomes false:

{ [...] => [...] } sapl:is sapl:true ;
sapl:existsWhile [...]  

We introduce sapl:existsWhile as a way of creating commitment guards because APLs normally require the ability of defining commitments that are dropped as unachievable or not relevant anymore. However, in S-APL sapl:existsWhile can of course be used with any type of beliefs. There are also a couple of alternatives to =>:

{ [...] -> [...] ; sapl:else [...] }  
{sapl:1 sapl:want [...] } >> [...]  
-> and >> are the shorthands for sapl:impliesNow and sapl:achievedBy. -> specifies a conditional action rather than a commitment: it is checked only once and removed even if it was false. One can also combine it with sapl:else to specify the beliefs that have to be added if the condition was false. >> works the same as => with the only difference that if the left side of it refers to some goals, commitments or interface (GUI or HTTP) events, those are removed automatically when the rule fires. Thus, >> has the meaning of logical transition rather than pure inference.

A specific convention of S-APL is that if there are several possible solutions to the query in the left side of =>, -> or >>, the right side is copied by default for the first-found solution only. One can use sapl:All wrappings to define that the right part has to be copied several times: for every unique value of some variable of every unique combination of the values of some variables. These wrapping can be used in either the left or the right side:

{ { ... } sapl:All ?x } sapl:All ?y } => {...}  
{ {...} => { {...} sapl:All ?x } sapl:All ?y }  

sapl:All on the right side is allowed to define different wrappings for different (top-level) resulting statements, e.g. "{...} => {X Y Z. {{?x L ?y } sapl:All ?y. A ?x} sapl:All ?x}". On the left side of =>, sapl:All must always wrap the whole contents of the container.

Other solutions set modifier wrappings are also available, namely sapl:OrderBy, sapl:OrderByDescending, sapl:Limit, and sapl:Offset. The meaning of those are the same as of their equivalents in SPARQL. One can also wrap a condition in the left side of => with sapl:Optional to have the same effect as SPARQL's OPTIONAL, and connect two conditions with sapl:or to have the same effect as SPARQL's UNION. It is also possible to specify exclusive conditions, i.e. ones that must not be known to be true, by using the wrapping sapl:1 sapl:doNotBelieve [...].

One can also define new calculated variables:

{?person :height ?h. ?feet sapl:expression "/?h/0.3048".  
?m in sapl:min ?feet } => {...}

sapl:expression gives to the new variable the value coming from evaluating an expression. sapl:min is a special predicate operating on the set of matching solutions rather than on a particular solution. The other predicates from the same group are sapl:max, sapl:sum, sapl:count (number of groups when grouped by values of some variables) and sapl:countGroupedBy (number of members in each group).

Variable can also refer to IDs of statements and context containers, and one can use the predicates sapl:hasMember, sapl:memberOf, rdf:subject, rdf:predicate and rdf:object. After obtaining the ID of the container with "?x :accordingTo :Bill", one can do the following things:

{ [...] } sapl:is sapl:true  
sapl:1 sapl:add ?x  
sapl:1 sapl:remove ?x  
?sap:1 sapl:erase ?x  
?sap:1 sapl:hasMember {?room1 :hasTemperature 25}  

The first construct defines a query that is evaluated as true iff any belief that is found in the context container ?x has a match in the context container {}. sapl:1 sapl:doNotBelieve :accordingTo :John } => {...}  

The second one links the statements from ?x to G, while the third copies them to G. The fourth uses the contents of ?x as the pattern for removing beliefs from G, while the fifth removes the container ?x itself. Finally, the sixth add to the container ?x a new statement.

There are several ways to create a variable holding IDs of some statements (we hope that the meanings are clear):

{ {?x :hasTemperature 25} sapl:ID ?x } :accordingTo :Bill  
{?x rdf:predicate :hasTemperature } :accordingTo :Bill  
{?x sapl:is sapl:true } :accordingTo :Bill  
?c :accordingTo :Bill. ?c sapl:hasMember ?x
One can use a query like "\{ {?x sapl:is sapl:true} :accordingTo :Bill, {?x sapl:is sapl:true} :accordingTo John } \Rightarrow (...)\}", which is evaluated as true if there is at least one belief from the first container that has a match in the second container. One can also use sapl:true, sapl:add, sapl:remove, sapl:erase, sapl:hasMember and sapl:memberOf to do the same things as explained above, but for one statement only.

4 Agent communication in S-APL

IEEE FIPA developed a set of standard specifications for agent communication including ACL for message envelopes and SL for contents [8]. While the standard position of ACL is unquestionable, the value of SL is less certain. Although meaning "semantic language", SL is not based on W3C's RDF semantic data model. Rather, SL follows the traditional agent design approaches where the agents' beliefs and thus also the atoms of their communications are n-ary predicates. However, N-ary predicates do not make the meaning of data as explicit as RDF triples do. Also, only the whole message can be linked to an ontology, as compared to the ability of RDF to link every individual resource to its own ontology, if needed.

In this section, we describe how we use S-APL as the content language in agent communications. Since one of the important communicative actions is querying for information, this role of S-APL overlaps with that of SPARQL. The problem with SPAQRL is that while being a language for querying RDF, it is not RDF itself. Also obviously, a content language for agent communication must support other types of communicative actions, for example, request for action. For these reasons we did not consider using SPARQL as such. Rather, when designing S-APL we included into it features analogous to most of the SPARQL’s ones (see Section 3).

The beliefs storage of an S-APL agent can be queried externally by other agents, of course subject to security and other policies. The core of a query is the same as if the agent itself would query its beliefs to check the premises of a rule, i.e. it can use all the constructs allowed for the left side of \( \Rightarrow \) (see Section 3). The core of the query has to be wrapped with sapl:I \{ sapl:I \}:want \{ \{ sapl:You sapl:answer [...] \} \}. The use of "sapl:I \{ sapl:I \}:want" may look unnecessary. However, this allows distinguishing between sapl:I \{ sapl:I \}:want [...] and e.g. :Boss sapl:want [...], i.e. mediating a wish of another agent. Both cases may require exactly the same action to be taken, however, may affect differently on whether the agent will comply or not.

As the response, the agent is to send the matching part of its belief storage, or, if no match, the query itself wrapped with sapl:I sapl:doNotBelieve [...]. Below, we list two small S-APL programs that an agent has to load in order to be able to be queried this way. The first one, Listeners.sapl,
For example, let’s assume that there is an agent with Listener and Informer programs loaded and having the following beliefs in its storage:

```plaintext
:factory1 :hasSpace :room3, :room1, :room2.
:room1 :hasTemperature 25; :hasHumidity 80.
:room2 :hasTemperature 20; :hasHumidity 90.
{:room :hasTemperature 30} =>
  {sapl:I ex:sendAlarm {:source sapl:is ?room}}
```

Then, the following queries (only the core is shown) will get the corresponding responses:

- (Q) :factory1 :hasSpace :room1
  (R) :factory1 :hasSpace :room1
- (Q) [{:factory1 :hasSpace ?room} sapl:All ?room] sapl:OrderBy ?room
  (R) :factory1 :hasSpace :room1 :factory1 :hasSpace :room2 :factory1 :hasSpace :room3
- (Q) :factory2 :hasSpace ?room
  (R) sapl:I sapl:doNotBelieve {:factory2 :hasSpace ?room}
- (Q) {:factory1 :hasSpace ?room} [:?room :hasTemperature ?temp] sapl:is sapl:Optional sapl:All ?room
  (R) :factory1 :hasSpace :room2 :room2 :hasTemperature 20 :factory1 :hasSpace :room3 :factory1 :hasSpace :room1.
  :room1 :hasTemperature 25
- (Q) [{:?r1 :hasTemperature ?temp. ?temp > 24} sapl:or {:?r2 :hasHumidity ?hum. ?hum > 85}] sapl:All ?r1 } sapl:All ?r2
  (R) :room2 :hasHumidity 90. :room1 :hasTemperature 25
- (Q) {?left => ?right. ?left sapl:hasMember {:room1 :hasTemperature ?t}
  (R) {?room :hasTemperature 30} =>
    {sapl:I ex:sendAlarm {:source sapl:is ?room}}
```

The last of the example queries above demonstrates an important feature of S-APL – ability of agents to exchange rules. Therefore, an agent can ask another agent how that will react if some situation occurs. Alternatively, an agent may query another agent if that knows a plan leading to achieving a goal. The above example also demonstrates a specific convention of S-APL – the data against which a query is evaluated can be implicitly universally quantified through the use of variables. This is why the query for a specific room 

```plaintext
[{:room1 :hasTemperature ?t} => {]
```

yields a universal rule

```plaintext
[{:room :hasTemperature 30} => {]
```

The next simple program below, Believer.sapl, instructs the agent to add to its beliefs everything that it receives in messages with the “inform” performative. This program can, for example, be used by an employee agent which is committed to comply with any of its boss’ statements. Some similar code is also needed for an agent that queried some information from an Informer (see above) to add the response to its beliefs. Note that substituting the line “?content sapl:is sapl:true” with something like “?content sapl:is sapl:true”. accordingTo :John” will lead to another effect: instead of believing into the information provided, the agent will record it wrapped as "John thinks that ... ."

```
/*Believer.sapl*/

```plaintext
{?requestID p:received {p:content sapl:is ?content.  
  p:performative sapl:is inform}}

```plaintext
} =>

```plaintext
  {sapl:I sapl:remove {?requestID p:received *}. 
    ?content sapl:is sapl:true
  }} sapl:is sapl:Rule
```

Due to the fact that in S-APL there is no principal difference between data and program code, the content of the message to a Believer agent could include commitments, unconditional or conditional, and the Believer would comply and perform those actions. In this sense, the Believer with the code as above is more like a slave fully controlled by its master. In less dependent settings, agents can request other agents to perform some actions, corresponding either to a RAB or an abstract capability. The core of the request is the same as in a normal unconditional commitment (see Section 3), only with sapl:You in place of sapl:I, and in addition wrapped with sapl:I sapl:want: sapl:I sapl:want { sapl:You sapl:do ... } sapl:configuredAs {...}. The small program below, Follower.sapl, is to be used then by an agent to perform actions requested in this way (again, only the basic logic is included without access control or other checks).

```
/*Follower.sapl*/

```plaintext
{?requestID p:received {p:content sapl:is 
  {sapl:I sapl:do ?action 
    sapl:configuredAs ?params}}

```plaintext
} =>

```plaintext
  {sapl:I sapl:remove {?requestID p:received *}. 
    {sapl:I sapl:do ?action} sapl:configuredAs ?params
  }} sapl:is sapl:Rule
```

5 Conclusions

We intend Semantic Agent Programming Language (S-APL) to be a core middleware language for the Semantic Web. S-APL integrates the semantic description of the domain resources with the semantic prescription of the agents’ behaviors. Additionally, S-APL can be used as the content language in the inter-agent communications, both in querying for data and in requesting for action.

Unlike traditional means for specifying agent-based systems, S-APL operates on semantic RDF data and thus can utilize the advantages of semantic inference. Moreover, in S-APL the behavior prescription is done using semantic predicates as well and also atomic behaviors and their parameters are also semantic resources that can be ontologically modeled. Use of S-APL as a communication language is quite natural because the communication over S-APL is easily organized with S-APL programming alone. An obvious additional benefit is the level of integration that in no-effort is then achieved between the communication and the agent behavior prescriptions. For example, an agent
can query another agent for behavior rules – either to understand how that will react if a certain situation occurs or to learn itself how to achieve a certain goal. Similarly, agents can exchange commitments, plans, or other belief structures of any complexity. To mention, in our current implementation the S-APL repositories (see Section 2) are managed by agents with Listener and Informer programs loaded and answering queries of the type “?x sapl:belongs <some_sapl_program>”.

We believe that the S-APL, both as the behavior prescription means and as the communication content language, provides a proper basis for middleware platforms for the Semantic Web. Elaborating on formal semantics of S-APL is an important direction of future work.

Acknowledgments

This work is performed in UBIWARE project, which supported by Tekes (Finnish National Agency for Technology and Innovation) and industrial partners ABB, Metso, Fingrid and Inno-W.

References