SmartResource Platform and Semantic Agent Programming Language (S-APL)

Artem Katasonov and Vagan Terziyan

Agora Center, University of Jyväskylä
P.O. Box 35, FIN-40014, Jyväskylä, Finland
artem.katasonov@jyu.fi, vagan@it.jyu.fi

Abstract. Although the flexibility of agent interactions has many advantages when it comes to engineering a complex system, the downside is that it leads to certain unpredictability of the run-time system. Literature sketches two major directions for search for a solution: social-level characterization of agent systems and ontological approaches to inter-agent coordination. Especially the latter direction is not yet studied much by the scientific community. This paper describes our vision and the present state of the SmartResource Platform. The main distinctive features of the platform are externalization of behavior prescriptions, i.e. agents access them from organizational repositories, and utilization of the RDF-based Semantic Agent Programming Language (S-APL), instead of common Prolog-like languages.

1 Introduction

When it comes to developing complex, distributed software-based systems, the agent-based approach was advocated to be a well suited one [1]. From the implementation point of view, agents are a next step in the evolution of software engineering approaches and programming languages, the step following the trend towards increasing degrees of localization and encapsulation in the basic building blocks of the programming models [2]. After the structures, e.g., in C (localizing data), and objects, e.g., in C++ and Java (localizing, in addition, code, i.e. an entity's behavior), agents follow by localizing their *purpose*, the thread of control and action selection.

The actual benefit of the agent-oriented approach arises from the fact that the notion of an agent is also appropriate as a basis for the analysis of the problem to be solved by the system. Many processes in the world can be conceptualized using an agent metaphor; the result is either a single agent (or cognitive) description or a multi-agent (or social) description [3]. Jennings [1] 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.

The problem of crossing the boundary from the domain (problem) world to the machine (solution) world is widely recognized as a major issue in software and systems

P. Petta et al. (Eds.): MATES 2007, LNAI 4687, pp. 25–36, 2007.

engineering. Therefore, when it comes to designing software, the most powerful abstractions are those that minimize the semantic distance between the units of analysis that are intuitively used to conceptualize the problem and the constructs present in the solution paradigm [2]. A possibility to have the same concept, i.e. agent, as the central one in both the problem analysis and the solution design and implementation can make it much easier to design a good solution and to handle complexity. In contrast, e.g. the object-oriented approach has its conceptual basis determined by the underlying machine architecture, i.e. it is founded on implementation-level ontological primitives such as object, method, invocation, etc. Given that the early stages of software development are necessarily based on intentional concepts such as stakeholders, goals, plans, etc, there is an unavoidable gap that needs to be bridged. [4] even claimed that the agent-oriented programming paradigm is *the only* programming paradigm that can gracefully and seamlessly integrate the intentional models of early development phases with implementation and run-time phases. In a sense, agent-oriented approach postpones the transition from the domain concepts to the machine concepts until the stage of the design and implementation of individual agents (given that those are still to be implemented in an object-oriented programming language).

Although the flexibility of agent interactions has many advantages when it comes to *engineering* complex systems, the downside is that it leads to *unpredictability in the run time* system; as agents are autonomous, the patterns and the effects of their interactions are uncertain [2]. This raises a need for effective coordination, cooperation, and negotiation mechanisms. (Those are in principle distinct, but the word "coordination" is often used as a general one encompassing all three; so for the sake of brevity we will use it like that too.) Jennings [2] discussed that it is common in specific systems and applications to circumvent these difficulties, i.e. to reduce the system's unpredictability, by using interaction protocols whose properties can be formally analyzed, by adopting rigid and preset organizational structures, and/or by limiting the nature and the scope of the agent interplay. However, Jennings asserted that these restrictions also limit the power of the agent-based approach; thus, *in order to realize its full potential some longer term solutions are required*. Emergence of such a longer term solution that would allow flexible yet predictable operation of agent systems seems to be a prerequisite for wide-scale adoption of the agent-oriented approach.

The available literature sketches two major directions of search for such a solution:

- D1: *Social level* characterization of agent-based systems. E.g. [2] stressed the need for a better understanding of the impact of sociality and organizational context on an individual's behavior and of the symbiotic link between the behavior of the individual agents and that of the overall system.
- D2: *Ontological* approaches to coordination. E.g. [5] asserted a need for common vocabulary for coordination, with a precise semantics, to enable agents to communicate their intentions with respect to future activities and resource utilization and get them to reason about coordination at run time. Also [6] put 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.

Recently, some progress has been made with respect to D1, resulting, e.g., in elaboration of the concept of a *role* that an agent can play in an organization (see Sect. 2).

However, with respect to D2 very little has been done. Bosse and Treur [3] discussed that the agent perspective entails a distinction between the following different types of ontologies: an ontology for internal mental properties of the agent A, MentOnt(A), for properties of the agent's (physical) body, BodyOnt(A), for properties of the (sensory or communication) input, InOnt(A), for properties of the (action or communication) output, OutOnt(A), of the agent, and for properties of the external world, ExtOnt(A). 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(A), i.e. the *domain ontologies*. MentOnt(A) comes for free when adopting a certain agent's internal architecture, such as Beliefs-Desires-Intentions (BDI) [7]. Also, the communication parts of InOnt(A) and OutOnt(A) come for free when adopting a certain communication language, such as FIPA's ACL. However, BodyOnt(A), i.e. the perceptors and actuators the agent has, sensory part of InOnt(A), i.e. the agent's perception patterns, and action part of OutOnt(A), e.g. the agent's acting patterns, are not usually treated. However, sharing these ontologies is a necessary precondition for agents' awareness of each other's actions, i.e. for D2. Already referred to article by [5] 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.

In our work, we attempt to provide a solution advancing into both D1 and (especially) D2 and somewhat integrating both. This paper describes the present state of our SmartResource Platform and the central to it Semantic Agent Programming Language (S-APL). The rest of the paper is structured as follows. Section 2 presents some basic thinking leading to our approach and comments on the related work. Section 3 describes the architecture of the SmartResource Platform, while Sect. 4 describes S-APL. Finally, Sect. 5 concludes the paper.

2 Motivation and Related Work

On the landscape of research in agent-based systems, we can identify two somewhat independent streams of research, each with its own limitations. The first stream is the research in multi-agent systems (MAS); the second stream is the research in agents' internal architectures and approaches to implementation.

Researchers in MAS have contributed with, among others, various methodologies for designing MAS, such as Gaia [8], TROPOS [4], and OMNI [9]. For example, OMNI (which seems to be the most advanced with respect to D1) elaborates on the organizational context of a MAS, defines the relationship between organizational roles and agents enacting those roles, discusses how organizational norms, values and rules are supposed to govern the organization's behavior and thus to put restrictions on individual agents' behaviors. However, OMNI touches only on a very abstract level the question about how the individual agents will be implemented or even function; the agents are treated as rather atoms. One reason is that it is (reasonably) assumed that the agent organization's designer may have no direct control over the design of individual agents. The organization designer develops the rules to be followed and enforcing policies and entities, such as "police" agents, while development of other agents is done by external people or companies. One of few concrete implementation requirements mentioned in

OMNI is that a rule interpreter must be created that any agent entering the organization will incorporate, somehow. The OMNI framework also includes explicitly the ontological dimension, which is restricted, however, to a domain ontology only (see Sect. 1), and thus does not provide much new with respect to D2.

The other stream of research, on individual agents, has contributed e.g. with well-known BDI architecture, and introduced *agent-oriented programming* [10] along with several *agent programming languages* (APL) such as AGENT-0 [10], AgentSpeak(L) [11], 3APL [12] and ALPHA [13]. All of those are declarative languages and based on the first-order logic of n-ary predicates. For example, an agent program in ALPHA consists of declarations of the beliefs and goals of that agent and declaration of a set of rules, including belief rules (generating new beliefs based on existing ones), reactive rules (invoking some actions immediately) and commitment rules (adopting a commitment to invoke an action). Perceptors (perceiving environment and generating new beliefs) and actuators (implementing the actions to be invoked) are then pieces of external code, in Java. As discussed in Sect. 1, agent-oriented approach postpones the transition from the domain concepts to the machine concepts until the stage of the design and implementation of individual agents. The advantage of using an APL is that the transition is postponed even further, until the implementation of particular perceptors and actuators.

This advantage seems to be, however, the only one that is considered. We did not encounter in literature approaches that would extend the role of APL code beyond the development stage. APL code is assumed to be written by the developer of an agent and either compiled into an executable program or interpreted in run-time but remaining an agent's intrinsic and static property. APL code is not assumed to ever come from outside of the agent in run-time, neither shared with other agents in any way.

Such export and sharing of APL code would, however, probably make sense in the light of findings from the field of MAS, and also in the light of D2. Methodologies like OMNI describe an organizational role with a set of rules, and an APL is a rule-based language. So, using an APL for specifying a role sounds as a natural way to proceed. The difference is that APL code corresponding to a role should naturally be a property of and controlled by the organization, and accessed by the agents' enacting the role potentially even in the run-time. Run-time access would also enable the organization to update the role code if needed.

The second natural idea is that the agents may access a role's APL code not only in order to enact that role, but also in order to coordinate with the agents playing that role. As one option, an agent can send to another agent a part of its APL code to communicate its intentions with respect to future activities (so there is no need for a separate content language). As another option, if a role's code is made public inside the organization, the agents may access it in order to understand how to interact with, or what to expect from, an agent playing that role.

However, when thinking about using the existing APLs in such a manner, there are at least two issues:

- The code in an APL is, roughly speaking, a text. However in complex systems, a description of a role may need to include a huge number of rules and also a great number of beliefs representing the knowledge needed for playing the role. Also,

in a case of access of the code by agents that are not going to enact this role, it is likely that they may wish to receive only a relevant part of it, not the whole thing. Therefore, a more efficient, e.g. a database-centric, solution is probably required.
- When APL code is provided by an organization to an agent, or shared between agents, mutual understanding of the meaning of the code is obviously required. While using first-order logic as the basis for an APL assures understanding of the semantics of the rules, the meaning of predicates used in those rules still needs to be consistently understood by all the parties involved. On the other hand, we are unaware of tools allowing unambiguous description of the precise semantics of n-ary predicates.

As a solution to these two issues, we see creating an APL based on the W3C's Resource Description Framework (RDF). RDF uses binary predicates only, i.e. triples (n-ary predicates can be represented nevertheless, of course, using several approaches). For RDF, tools are available for efficient database storage and querying, and also for explicit description of semantics, e.g. using OWL. Our proposition for such an RDF-based APL is the *Semantic Agent Programming Language (S-APL)* that will be described in Sect. 4.

3 SmartResource Platform

The SmartResource Platform is a development framework for creating multi-agent systems. It is built on the top of the Java Agent Development Framework (JADE, see http://jade.tilab.com/), which is a Java implementation of IEEE FIPA specifications. The name of the platform comes from the name of the research project, in which it was developed. In the SmartResource project (see http://www.cs.jyu.fi/ai/OntoGroup/SmartResource_details.htm), a multi-agent system was seen, first of all, as a middleware providing interoperability of heterogeneous (industrial) resources and making them proactive and in a way smart.

The central to the SmartResource Platform is the architecture of a SmartResource agent depicted in Fig. 1. It can be seen as consisting of three layers: Reusable Atomic Behaviors (RABs), Behavior Models corresponding to different roles the agent plays, and the Behavior Engine. An additional element is the storage for agent's beliefs and goals. The SmartResource Platform uses the RDF data model, i.e. any belief or goal is a subject-predicate-object triple, e.g. "John Loves Mary".

A *reusable atomic behavior (RAB)* is a piece of Java code implementing a reasonably atomic function. RABs can be seen as the agent's perceptors and actuators. As the name implies, RABs are assumed to be reusable across different applications, different agents, different roles and different interaction scenarios. Obviously, RABs need to be parameterizable.

In the SmartResource Platform, the behavior of an agent is defined by the roles it plays in one or several organizations. Some examples of the possible roles: operator's agent, feeder agent, agent of the feeder N3056, fault localization service agent, ABB's fault localization service agent, etc. Obviously, a general role can be played by several agents. On the other hand, one agent can (and usually does) play several roles.

A *behavior model* is a document that is supposed to specify a certain organizational role, and, therefore, there is one-to-one relation between roles and behavior models. In

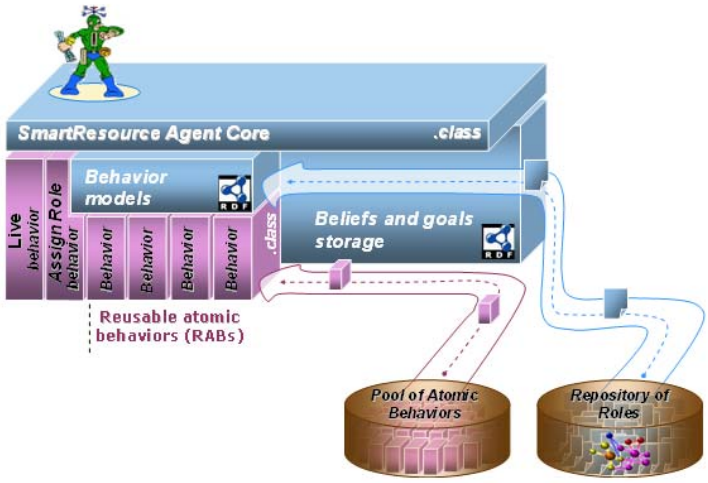

Fig. 1. SmartResource Platform

the SmartResource Platform, behavior models are encoded in a high-level rule-based language, Semantic Agent Programming Language (S-APL). S-APL is based on the RDF data model, i.e. the whole document can be seen as a set of subject-predicate-object triples. A behavior model consists of a set of beliefs representing the knowledge needed for playing the role and a set of behavior rules. Roughly speaking, a behavior rule specifies conditions of (and parameters for) execution of various RABs.

The *behavior engine* is the same for all the SmartResource agents (this of course means that each agent has a copy of it). The behavior engine consists of the agent core and the two core behaviors that we named “Assign Role” and “Live”.

The AssignRole behavior processes an S-APL document, loads specified initial beliefs and goals into the beliefs and goals storage, and parses the behavior rules. In addition, it registers the new role with the system Directory Facilitator agent. It is recommended that if a behavior model is to specify the need of interaction with another agent, that agent should be specified by its role, not the name or another unique identifier of a particular agent. Then, DFLookup atomic behavior, provided with the platform, can find with DirectoryFacilitator names of agents playing a particular role. If several agents play the role needed, the behavior model is supposed to include some rules specifying a mechanism of resolving such a situation, e.g. random select, auction, etc. Different such mechanisms can of course be assigned to resolving conflicts with respect to different roles. When an agent is created it has to be given at least one behavior model to start working. Therefore, the agent’s core needs to directly engage the AssignRole behavior for the model(s) specified. However, all the later invocations of AssignRole, i.e. adding new roles, are to be specified in some behavior models. Therefore, AssignRole has the duality of being a part of the behavior engine and a RAB in the same time.

The Live behavior implements the run-time cycle of an agent. Roughly speaking, it iterates through all the behavior rules, checks them against existing beliefs and goals, and executes the appropriate rules. Execution of a rule normally includes execution of a RAB and performing a set of additional mental actions, i.e. adding and removing some beliefs. At least at the present stage, if there are several rules that are executable, all of them are executed.

As can be seen from Fig. 1, the SmartResource Platform allows agents to access behavior models from an external repository, which is assumed to be managed by the organization which "hires" the agents to enact those roles. It is done either upon startup of an agent, or later on. Such externalization of behavior models has several advantages:

- Increased flexibility for control and coordination. Namely, the organization can remotely affect the behavior of the agents through modifying the behavior models. Another advantage is that the models can always be kept up-to-date.
- An agent may "learn" how to play a new role in run-time; it does not need to be pre-programmed to do it.
- Inter-agent behavior awareness. How is discussed in Sect. 2, the agents not enacting a particular role can still make some use of the information encoded in its behavior model. One reason is to understand how to interact with, or what to expect from, an agent playing that role.

As can also be seen from Fig. 1, the SmartResource Platform allows agent on-demand access even of RABs. If an agent plays a role, and that role prescribes it to execute an atomic behavior that the agent is missing, the agent can download it from the repository of the organization. In a sense, the organization is able to provide not only instructions what to do, but also the tools enabling doing that. Physically, a RAB is delivered as either .class file or a .zip file (in case when the RAB has several .class files). The obvious additional advantages are:

- An agent may "learn" new behaviors and so enact in a completely new role.
- Agents may have a "light start" with on-demand extension of functionality.

The present version SmartResource Platform provides the behavior model OntologyAgent.rdf along with used in it RAB OntologyLookupBehavior. By starting an agent based on this model, one creates an agent that provides access to both repository of roles and pool of atomic behaviors.

The SmartResource Platform provides the behavior model startup.rdf, which has to be loaded by an agent at startup in order to enable it to remotely access behavior models from an OntologyAgent. For roles specified on startup of the agent, the agent's core takes care of it. In addition, startup.rdf includes the rule for engaging DFLookupBehavior. This rule is obviously needed for resolving the OntologyAgent role. However, it is also enough for any future needs of resolving roles, just a specific convention is to be followed. Figure 2 shows the common process of starting up an agent, when the behavior models are accessed from an OntologyAgent.

The SmartResource Platform provides also the behavior model RABLoader.rdf, which has to be loaded by an agent in order to enable it to remotely access atomic behaviors from an OntologyAgent. It includes rules for requesting, receiving, and (if needed) unzipping the behavior files.

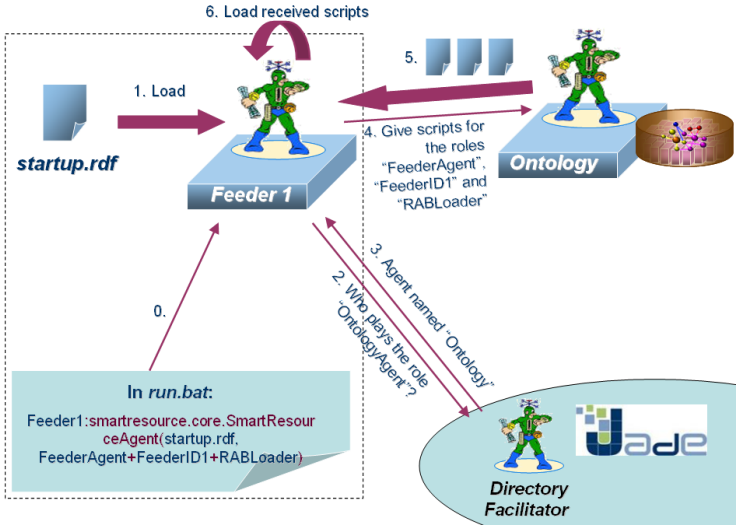

Fig. 2. An agent's start-up

Figure 3 depicts a more complex scenario of auction for selection of a service provider, in this case a fault localization service. The operator's agent behavior model prescribes that in case of several localization services, an auction has to be performed (for other roles, e.g. random select is done). The agent first sends to both localization agents a special request to load the role "AuctionSeller", and then a request to make an offer on, say, price of the service. The agent "Ls1" has loaded the role "AuctionSeller" from the beginning, but the agent "Ls2" did not. So, "Ls2" contacts the OntologyAgent and requests the needed behavior model now. This scenario demonstrates that roles can be loaded also dynamically.

4 Semantic Agent Programming Language (S-APL)

This chapter describes the RDF/XML syntax of the Semantic Agent Programming Language (S-APL) through its three constructs: Belief, Goal and Behavior.

As can be noticed, in defining/referring to beliefs and goals, S-APL does not use the RDF syntax, but has them as literals of the form "subject predicate object". The main reason for this is inability of RDF to restrict the scope of a statement. In RDF, every statement is treated as a global truth. But for describing behavior rules, the statements are specifications of IF and THEN conditions, not facts. Additional reason is a wish to keep S-APL documents concise and human-readable/editable.

In S-APL, all beliefs/goals must be triples. However, at least at present stage, we do not enforce the RDF rule that only the object can be a literal while the subject and the predicate must be URIs. In other words, in S-APL beliefs/goals, the subject and the predicate can be literals as well. When using URIs, a convenient way is to utilize XML's ENTITY construct to simulate the namespaces mechanism (see example below).

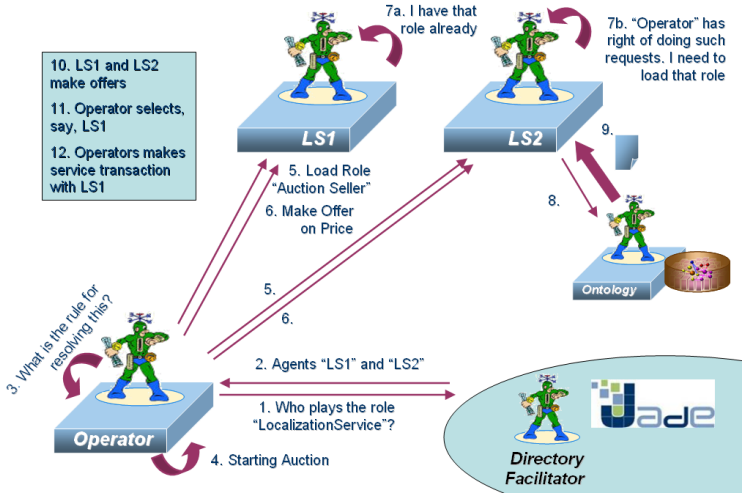

Fig. 3. Scenario: Auction for selection of the service provider

Element <gb:Belief> specifies a belief, with which the agent (in a role) will be initialized. The statement of the belief is given in the <gb:statement> field in the format (whitespace-separated) "subject predicate object". Element <gb:Goal> specifies a goal, with which the agent (in a role) will be initialized. The statement of the belief is also given in the <gb:statement> field.

Element <gb:Behavior> specifies a behavioral rule. The fields trueIf, falseIf, trueIfGoalAchieved, achievesGoal, and event describe the left side (IF) of the rule, while the rest describe the right side (THEN) of the rule. None of the fields is mandatory. All can appear more than once, with exception of gb:class and gb:event.

- *<gb:trueIf>* Specifies a precondition, i.e. a belief that must be found in the set of the agent's beliefs to make the rule applicable. If several trueIf is given, they all must be found, i.e. they can be seen as connected with AND.
- *<gb:falseIf>* Specifies a negative condition, i.e. such a belief that, when found in the set of the agent's beliefs, makes the rule not applicable. If several falseIf is given, any of them is enough, i.e. they can be seen as connected with OR.
- *<gb:achievesGoal>* Specifies a goal that must be found in the set of the agent's goals to make the rule applicable. In a sense, it specifies an expected rational effect of the rule execution and puts a need for it as a precondition. If several achievesGoal is given, any of them is enough, i.e. they can be seen as connected with OR.
- *<gb:event>* Specifies an interface event (either from GUI or from HTTP server) that must be the current event to make the rule applicable. If a rule has event specified, it will never be executed from the normal Live cycle of the agent, but only from the interface event handling routine.
- *<gb:trueIfGoalAchieved>* Specifies a sub-goal that must be achieved before an applicable rule can be executed. If according to all the other conditions the rule is applicable and only one or more trueIfGoalAchieved is not fulfilled, the agent will

add those to the set of its goals. In other words, the agent will try to eventually execute a rule that is applicable. If several trueIfGoalAchieved is given, they all need to be present in the beliefs to make the rule executable, so they can be seen as connected with AND. Also, all of them that are not present in the beliefs, will be added to the set of goals at once.

- <*gb:addOn*<*X*>> Specifies a belief that has to be added in the phase <X>. Possible values for <X> are: *Start* – add the belief when the rule is executed, before invoking the actual behavior i.e. RAB, *End* – add the belief when the behavior has ended the execution, *Success* – add when and if the behavior has ended the execution in success, *Fail* – add when and if the behavior has ended the execution in failure.
- <*gb:removeOn*<*X*>> Specifies a belief that has to be removed in the phase <X>. Possible values for <X> are the same as above. If the specified belief contains "*" and matches several existing beliefs, all matching beliefs will be removed.
- <*gb:class*> Specifies the Java class implementing the behavior.
- <*x:*<*anything*>> A parameter that is to be passed to the instance of behavior, and has some meaning in the context of that behavior.

Note that explicit adding or removing of goals is not supported. A goal can only be added if it appears in trueIfGoalAchieved of an applicable rule (and not yet in the set of goals), and removed only when either (1) a rule is executed having it among its achievesGoal or (2) in the beginning or a Live cycle, it is found in the set of beliefs.

Note also that an additional implicit condition for whether a rule is executable is presence of the specified Java class. If it is not found, the rule is considered as not executable, so neither beliefs are modified nor goals are removed.

In the statements, "*" can be used with the meaning of "anything", and "*<var>*" can be used as variable. If variables are used, the rule is applicable/executable if the beliefs and goals of the agent provide at least one possible binding of the values. If several bindings are possible, the first found is taken. The left part of the rule is processed in the following order: event, trueIf, achievesGoal, trueIfGoalAchieved, falseIf. This defines the order in which the variables are bind. The possible set of values for a variable is searched when the variable is first encountered. After that, values from this set can only get filtered out but no new ones can be added.

All the fields of a rule are passed as the start parameters to the instance of the behavior, not only <x:<anything>>. Therefore, if needed, the behavior can have access to the context of its execution, e.g. trueIf, addOnStart, etc. Note though that, in all the start parameters, the variables are substituted with their values.

An example of usage follows. Given that the date is 8 of March, if the agent knows a woman and knows something that she likes, start GiftingBehavior to gift her that thing. A sub-goal of this is to buy the needed thing (handled in the rule behavior2). FalseIf statements are used to prevent the behavior to be executed twice. The belief "I Gifting <X>" is added as soon as the rule is executed (note, this happens after the sub-goal is achieved), and removed as soon as GiftingBehavior ends (regardless of the result). If the result is success, belief "I Gifted <X>" is added. This example has only one belief statement using namespaces and thus referring to an ontology. This is done for the sake of simplicity.

An Example of the RDF/XML Syntax of the Semantic Agent Programming Language

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
<!ENTITY  ex "http://www.example.com/ontology#">
]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:gb="http://www.smartresource.com/rgbdf#"
             xmlns:x="http://www.smartresource.com/atomic_behaviors#">
<gb:Belief rdf:about="belief1">
    <gb:statement>I Know Alice</gb:statement>
</gb:Belief>
...
<gb:Behavior rdf:about="behavior1">
    <gb:class>smartresource.non_existing.GiftingBehavior</gb:class>
    <gb:trueIf>&ex;Date &ex;Is 08.03</gb:trueIf>
    <gb:trueIf>I Know *X*</gb:trueIf>
    <gb:trueIf>*X* Is Woman</gb:trueIf>
    <gb:trueIf>*X* Likes *thing*</gb:trueIf>
    <gb:falseIf>I Gifting *X*</gb:falseIf>
    <gb:falseIf>I Gifted *X*</gb:falseIf>
    <gb:trueIfGoalAchieved>I Bought *thing*</gb:trueIfGoalAchieved>
    <gb:addOnStart>I Gifting *X*</gb:addOnStart>
    <x:receiver>*X*</x:receiver>
    <x:object>*thing*</x:object>
    <gb:removeOnEnd>I Gifting *X*</gb:removeOnEnd>
    <gb:addOnSuccess>I Gifted *X*</gb:addOnSuccess>
</gb:Behavior>
<gb:Behavior rdf:about="behavior2">
    <gb:class>smartresource.non_existing.BuyingBehavior</gb:class>
    <gb:achievesGoal>I Bought *thing*</gb:achievesGoal>
    <x:object>*thing*</x:object>
    <gb:addOnSuccess>I Bought *thing*</gb:addOnSuccess>
</gb:Behavior>
</rdf:RDF>
```

5 Conclusions and Future Work

Although the flexibility of agent interactions has many advantages when it comes to engineering a complex system, the downside is that it leads to certain unpredictability of the run-time system. Emergence of a solution that would allow flexible yet predictable operation of agent systems seems to be a prerequisite for wide-scale adoption of the agent-oriented approach. Literature sketches two major directions for search for a solution: social-level characterization of agent systems (more or less studied) and ontological approaches to inter-agent coordination (not yet studied much).

This paper described our vision and the present state of the SmartResource Platform. In the architecture of the platform, we attempt to provide a solution advancing into both directions mentioned and somewhat integrating both. The main distinctive features of the platform are externalization of behavior prescriptions, i.e. agents access them from organizational repositories, and utilization of the RDF-based Semantic Agent Programming Language (S-APL), instead of common Prolog-like languages.

In the follow-up project called Smart Semantic Middleware for Ubiquitous Computing (UBIWARE) 2007-2010, we are going to continue our work on the platform. At least the following important research questions have to be yet answered:

– Is it important and, if yes, how to implement the separation between a role's capabilities (individual functionality), and the business processes in which this role can be involved (complex functionality)?

- How to realize an agent's roles as higher-level commitments of the agent that restrict its behavior, still leaving freedom for learning and adaptation on lower-levels, instead of totally and rigidly prescribing the behavior?
- What mechanisms are needed for flexibly treating the potential (and likely) conflicts among the roles played by one agent?
- What would be concrete benefits of and what mechanisms are needed for accessing and using a role's script by agents who are not playing that role but wish to coordinate or interact with an agent that does?

Acknowledgments

This work was performed in the SmartResource project, which was financially supported by the National Technology Agency of Finland (TEKES) and industrial partners ABB, Metso Automation, TeliaSonera, TietoEnator, and Jyväskylä Science Park.

References

1. Jennings, N.: An agent-based approach for building complex software systems. Communications of the ACM 44(4), 35–41 (2001)
2. Jennings, N.: On agent-based software engineering. Artificial Intelligence 117(2), 277–296 (2000)
3. Bosse, T., Treur, J.: Formal interpretation of a multi-agent society as a single agent. Journal of Artificial Societies and Social Simulation 9(2) (2000)
4. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
5. Tamma, V., Aart, C., Moyaux, T., Paurobally, S., Lithgow-Smith, B., Wooldridge, M.: An ontological framework for dynamic coordination. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 638–652. Springer, Heidelberg (2005)
6. Jennings, N., Sycara, K.P., Wooldridge, M.: A roadmap of agent research and development. Autonomous Agents and Multi-Agent Systems 1(1), 7–38 (1998)
7. Rao, A., Georgeff, M.: Modeling rational agents within a BDI architecture. In: KR'91. Proc. 2nd International Conference on Principles of Knowledge Representation and Reasoning, pp. 473–484 (1991)
8. Wooldridge, M., Jennings, N., Kinny, D.: The Gaia methodology for agent-oriented analysis and design. Autonomous Agents and Multi-Agent Systems 3(3), 285–312 (2000)
9. Vazquez-Salceda, J., Dignum, V., Dignum, F.: Organizing multiagent systems. Autonomous Agents and Multi-Agent Systems 11(3), 307–360 (2005)
10. Shoham, Y.: Agent-oriented programming. Artificial Intelligence 60(1), 51–92 (1993)
11. Rao, A.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
12. Dastani, M., van Riemsdijk, B., Dignum, F., Meyer, J.J.: A programming language for cognitive agents: Goal directed 3APL. In: Dastani, M., Dix, J., El Fallah-Seghrouchni, A. (eds.) PROMAS 2003. LNCS (LNAI), vol. 3067, pp. 111–130. Springer, Heidelberg (2004)
13. Collier, R., Ross, R., O'Hare, G.: Realising reusable agent behaviours with ALPHA. In: Eymann, T., Klügl, F., Lamersdorf, W., Klusch, M., Huhns, M.N. (eds.) MATES 2005. LNCS (LNAI), vol. 3550, pp. 210–215. Springer, Heidelberg (2005)