SPLAW: A Computable Language for Agent-oriented Programming

FAN Xiaocong XU Dianxiang HOU Jianmin ZHENG Guoliang

Department of Computer Science and Technology

Nanjing University

NanJing 210093, P. R. China

zhenggl@nju.edu.cn

Abstract

Agent oriented programming (AOP), which is a special kind of object-oriented programming, has recently been discussed from several viewpoints. It can be worked out best for open systems and has the potential to become a very attractive technique in the future. In this paper, we describe a specification and programming language — SPLAW, for BDI agent. The syntax and operational semantics of SPLAW are presented, and by means of labeled transition system; the proof theory is also provided. SPLAW has two advantages. First, it is based on KQML, the standard inter-agent communication language, which makes it possible for agents written in SPLAW to interoperate with other agents obeying KQML. And second, it has the correspondent relationship between its operational semantics and proof theory. Owing to these, we hope that SPLAW will provide a feasible solution to bridge the gap between theory and practice.

Keywords Agent-oriented programming, KQML, inheritance, Agent-based computing

1 Introduction

Yoav Shoham has proposed a new programming paradigm[8] (AOP) based on a societal view of computation. The key idea is to build computer systems as societies of agents and the central features include (i)agents are reactive, autonomous, concurrently executing computer processes; (ii)agents are cognitive systems, programmed in terms of beliefs, goals, and so on; (iii)agents are reasoning (internally-motivated) entities, specified in terms of logic; (iv)agents communicate via speech acts.

Since the presentation of AOP, agent based computing has been hailed as "the new revolution in software"[6] because agent-based systems have advantages in dealing with openness, where components of the system are not known in advance, can change over time, and are highly heterogeneous (in that they are implemented by different people, at different times, using different problem solving paradigms). In addition, agent based systems have natural metaphor, can deal with problems such as distribution of data, control, expertise or resources and integrate legacy system by adding an agent wrapper, etc..

However, the construction of large-scale embedded software systems demands the use of design methodologies and modeling techniques that support abstraction, inheritance, modularity, and other mechanisms for reducing complexity and preventing error. Unfortunately, so far there has been few such researches in agent-oriented methodologies. If multi-agent systems are to become widely accepted as a basis for large-scale applications, adequate agent-oriented methodologies (AOM) will be essential[9].

Perhaps foremost amongst the methodologies that have been developed for the design, specification, and programming of conventional software systems are various Object-oriented approaches. They have achieved a considerable degree of maturity, and a large community of software developers familiar with their use now exists. At the same time, the OO design and development environment is well supported by diagram editors and visualization tools.

But OO methodologies are not directly applicable to agent systems—agents are usually significantly more complex than typical objects, both in their internal structures and in the behaviors they exhibit.

In order to construct a complete methodology for AOP, one of the essential things is to develop a programming language because agent-oriented language and the implemented architectures of agents decide about the usefulness of AOP in the applications. In this paper, we try to center upon the language problem, by providing a computable programming language—SPLAW, for BDI agent.

BDI agents are systems that are situated in a changing environment, receive continuous perceptual input, and take actions to affect their environment, all based on their internal mental state. Beliefs, desires, and intentions are the three primary attitudes and they capture the informational, motivational, and decision components of an agent, respectively[1,10].

SPLAW is a programming language based on a restricted first-order logic. The behavior of an agent is dictated by the programs written in SPLAW; the beliefs, desires, and intentions of agents are not explicitly represented as modal formulas, but

written in SPLAW. The current state of an agent, which is a model of itself, its environment, and other agents, can be viewed as its current belief state; states that an agent wants to bring about based on its external or internal stimuli can be viewed as desires; and the adoption of plans to satisfy such stimuli can be viewed as intentions. We just take a simple specification language as the execution model of an agent and then ascribe the mental attitudes of beliefs, desires, and intentions from an external viewpoint. In our opinion, this method is likely to have a better chance of unifying theory and practice.

2 The Syntax of SPLAW

An SPLAW agent program is composed of some *agent class* definitions and a main procedure which creates agent instances. The class framework of agent is something like the syntax of Eiffel. The feature part of *agent class* includes a set of base beliefs, a set of static plans, a set of predicate symbols, a set of function symbols, a set of primitive actions and their implementations(belief revision[3], services, etc.). The main difference lies in the representation of plan.

Definition 2.1 The *connectives* of SPLAW includes !(for achievement), ?(for test), @(for communication), &(for $\wedge$), ~(for $\neg$) and <-(for $\leftarrow$). Variables are expressed by strings started by capital letter, and constants by strings started by lowercase. $\forall$ is a global quantifier, $ is a blocking mark, and *null* is used to represent a void return value. Standard first-order definitions of terms, formulas, closed formulas, free and bound occurrences of variables are used.

Definition 2.2 If b is a predicate symbol, and $t=t_1, \ldots, t_n$ is a vector of terms, then $b(t)$ is a *belief atom.* If $b(t)$ and $c(s)$ are belief atoms, $b(t) \wedge c(s)$ and $\neg b(t)$ are *beliefs.* A belief atom or its negation will be referred to as a *belief literal.* A ground belief atom will be called a *base belief.* The base belief set of an agent is composed of all its base beliefs.

Definition 2.3 If g is a predicate symbol, and $t=t_1, \ldots, t_n$ is a vector of terms, then $!g(t)$ and $?g(t)$ are *goals*, where $!g(t)$ is an *achievement goal*, which states that the agent wants to achieve a state where $g(t)$ is a true belief, and $?g(t)$ is a *test goal*, which states that the agent wants to test whether the formula $g(t)$ is a true belief or not.

Definition 2.4 If a is an action symbol and $t=t_1, \ldots, t_n$ is a vector of terms, then $a(t)$ is an *action.*

Definition 2.5 There are three forms of *communicative actions* in SPLAW: *@q(Send,Rec,Self,Eve: cond(t), Retn)*, which expects a reply, *@q(Send,Rec,Self,Eve:cond(t), true)*, which expects no reply, and *@m(Rec,Id,Retn)* for reply messages, where *Send* denotes the message sender, *Rec* denotes the message receiver, *Id* denotes the intention identifier of the request, *Eve* denotes the cooperative request, *cond(t)* is the request constraints and *Retn* is a return parameter.

Definition 2.6 If $b(t)$ if a belief atom, $!g(t)$ and $?g(t)$ are goals, then $+b(t)$, $-b(t)$, $+!g(t)$, $-!g(t)$, $+?g(t)$ and $-?g(t)$ are called *general triggering events*, where +/- denote the addition and deletion of beliefs or goals respectively. *@q(Send,Id,Eve:cond(t),Retn)*, *@q(Send,Id,Eve:cond(t),true)*, and *@m(Rec,Id, Retn)* are called *communicative triggering events. Triggering events* includes general triggering events and communicative triggering events.

Definition 2.7 *Planning statements* include *simple statements*, *sequential statements*, *non-deterministic statements*, *condition statements*, and *loop statements*. Where, beliefs, goals, actions, communicative actions and *true* (empty) are called simple statements; A sequence of simple statements separated by ";" are called sequential statements; non-deterministic statements have the form #<cond>:<statements>#...#<cond>:<statements>#other:$; condition statements have the form IF <cond>: <statements> FI; and loop statements have the form ||<cond>:<statements>||.

Definition 2.8 If e is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n$ are planning statements, then $e{:}b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots ; h_n$ is a *static plan.* The expression to the left of the arrow is referred to as the *head* of the plan and the expression to the right of the arrow is referred as the *body* of the plan. The expression to the right of the colon in the head of the plan is referred to as the *context.* For convenience, we shall rewrite an empty body with the expression *true.*

Only when the context of a plan is a logical result of the base belief set, can this plan be adopted.

When an agent receives a request from user, if there is no static plan matching the request, the agent will create a *dynamic plan* to deal with it according to user's constraints. A dynamic plan has the form: *!true : <context> <-<plan body>.*

3 The Semantics of SPLAW

3.1 Agent Class Organization and the Inheritance Semantics of SPLAW

SPLAW provides the following basic agent classes: *BaseAgent, AloneAgent, CommuAgent, SocialAgent, WillAgent, CoopAgent* and *ActiveAgent*. The inheritance relations of these classes are shown in Figure 1. *BaseAgent* is the super class of all the other classes, which defines the basic behaviors an SPLAW agent should have, but the concrete definitions are deferred to the sub-classes. *AloneAgent* and *CommuAgent* are the direct sub-classes of *BaseAgent*. *AloneAgent* provides abstraction for the agents which can execute independently, while *CommuAgent* encapsulates the communicative behaviors and provides abstraction for the agents which can communicate with each other. *SocialAgent* inherits the communicative behaviors of *CommuAgent* and augment with the mechanism for cooperation, which makes the agents of this class have the capability to commit to cooperative problem solving. Although *WillAgent* inherits the communicative behaviors of its super class, it cannot support teamwork and has no responsibility for commitment to the external request. *CoopAgent*, which can facilitate inter-agent communications, is the bridge and coordinator for the agents of *ActiveAgent* to cooperate.

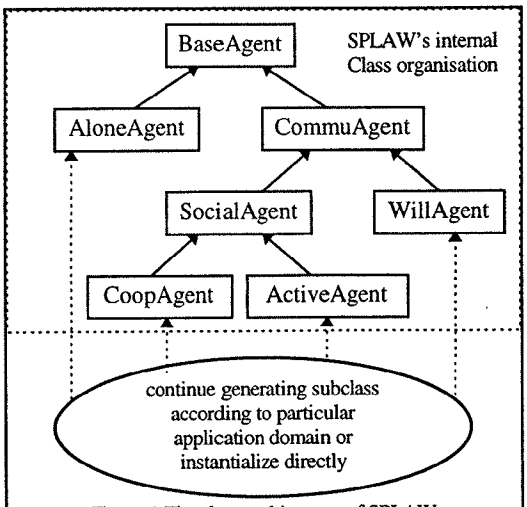

Figure 1 The class architecture of SPLAW

BaseAgent, CommuAgent and SocialAgent are all deferred classes and transparent to users. Users can define the sub-classes of *AloneAgent, WillAgent, CoopAgent* and *ActiveAgent*, override, or add new behaviors in order to implement systems of particular application domains. The frameworks of class definitions for *BaseAgent* and *CommuAgent* are shown as Figure 2.

```
deferred CLASS    BaseAgent
feature
        base_Beliefs_set={ ...};
        Plans_set={ ...};
        predicate_symbols={ ...};
        function_symbols={ ...};
        action_symbols={ ...};
        Select_Event() is deferred end;
        Select_Plan() is deferred end;
        Select_Intention() is deferred end;
        Belief_Revision() is deferred end;
        Unifier() is deferred end;
        Exec_Intention() is deferred end;
        Dynamic_planning() is deferred end;
end-CLASS BaseAgent
```

```
deferred CLASS    CommuAgent
export        KQML_Primitives
inherit       BaseAgent   redefine Dynamic_planning() ...
feature
        ...
        In_Queue;
        Out_Queue;
        KQML_Translate() is body end;
        In_Processing() is body end;
        Out_Processing() is body end;
        ...
end-CLASS CommuAgent
```

Figure 2 Sample frameworks for the definition of agent class

The operational semantics of inheritance is given by the plan searching algorithm of the SPLAW interpreter. After receiving a service request, the agent tries to match the plan in the interface of its own class, if not succeed, go up to its super class along the inheritance chain, and continue until success.

3.2 Communication Mechanism Between Agents

Agent communication languages (ACL) are concerned strictly with the communication between agents[5]. An ACL (such as KQML) is more than a protocol for exchange of data, because an *attitude* about what is exchanged by the agents is also communicated. An ACL may be thought as a communication protocol that supports many *message types*.

The coordination protocols such as CORBA ensure that applications can exchange data structures and methods across disparate platforms. Although the results of such standards will be useful in the development of software agents, they do not provide complete answers to the problems of agent communication. After all, software agents are more than collections of data structures and methods on them. For these reasons, in this paper we use KQML as the communication protocol of SPLAW agents.

As mentioned above, the communicative behaviors are encapsulated in class CommuAgent. Two message queues are defined for the agents of CommuAgent, i.e., *In_Queue* and *Out_Queue*, where Out_Queue is associated with the local communicative action set *CA*, which provide cache for message processing. The message processor (Figure 3) includes threads *Out_processing* and *In_processing*, which deal with the messages in the queues of Out_Queue and In_Queue respectively.

The thread In_processing selects a message from In_Queue, translates it from KQML format into internal pattern, gets rid of the *Rec* field, and sends it into the event set E. The thread Out_processing selects messages from Out_Queue one by one, replaces the field of *Self* by the identifier of sending intention's, translates the message into KQML standard format and sends it to the transport layer. The transport layer is supported by TCP/IP protocol and transparent to users.

3.3 The Operational Semantics of SPLAW Agent

Definition 3.1 An agent is given by a tuple $<E,A,CA,B,P,I,S_\varepsilon,S_\rho,S_\lambda>$, where E is a set of events, A is a set of actions, CA is a set of communicative actions, B is a set of base beliefs, P is a set of plans, I is a set of intentions, and the selection function S_ε selects an event from E; S_ρ selects an option from a set of applicable plans; S_λ selects an intention from the set I.

Definition 3.2 The set I is composed of all the intentions generated during the run-time of an agent. Each *intention* is a stack of partially instantiated plans (where some of the variables have been instantiated), and denoted by $[p_1\backslash p_2\backslash...\backslash p_n]$, where p_1 is the bottom and p_n is the top of the stack, and the elements of the stack are delimited by \. For convenience, the intention [+!true:true<-true] is referred to as the *true* intention and denoted by T. Each intention has a unique identifier, the i'th intention of agent Ag_l is denoted by $Token(Ag_l,i)$.

Definition 3.3 The set E is composed of all the events generated during the run-time of an agent. An *event* is a tuple $<e, i>$, where e is a triggering event and i is an intention. If i is T, the event is called an *external event*, if e has the form as $@m$ or $@q$, the event is called a *communicative event*, otherwise it is an *internal event*.

Definition 3.4 Let $S_\varepsilon(E)=\partial=<d, i>$ and let p be $e : b_1\wedge ... \wedge b_m <- h_1; ... ;h_n$. The plan p is a *relevant plan* with respect to an event ∂ iff there exists a most general unifier σ such that $d\sigma = e\sigma$, σ is called the *relevant unifier* for ∂. If there also exists a substitution θ such that $\forall(b1\wedge ... \wedge bm)\sigma\theta$ is a logical consequence of B, p is an *applicable plan* with respect to ∂ . The composition $\sigma\theta$ is referred to as the *applicable unifier* for ∂ and θ is referred to as the *solving substitution*.

3.3.1 Intention Generation

When an agent notices a change in the environment, a request from users, or an external cooperative request, an appropriate triggering event is generated and sent into E(shown in figure 3). The selection function S_ε selects an event e from set E (remove e from E at the same time) to unify with the triggering events of the plans in set P, which generates a set of applicable plans. The function S_ρ is used to choose one of these plans. Applying the applicable unifier to the *chosen plan* yields an *intention slice*, which is used to generate intentions according to the following methods:

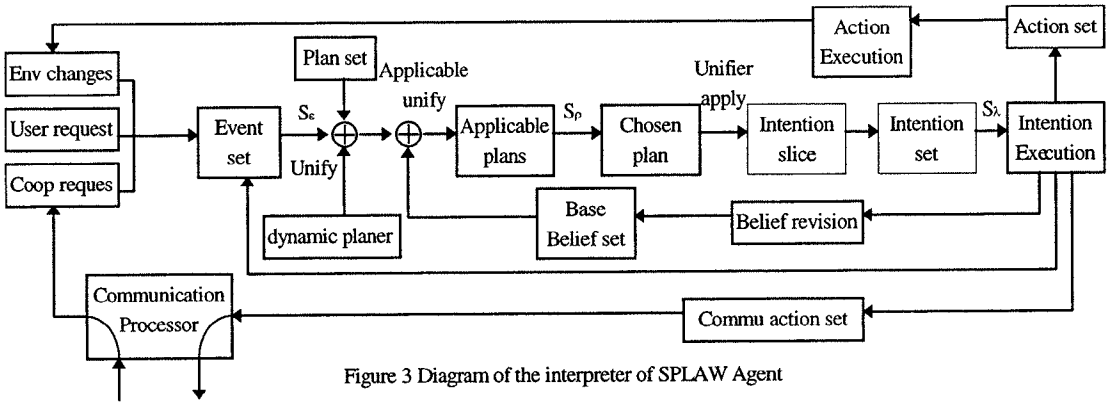

Figure 3 Diagram of the interpreter of SPLAW Agent

Suppose $S_\varepsilon(E)=\partial=<d, i>$.

If $\partial=<+!g(t), [p_1\backslash ... \backslash f : c_1\wedge ...\wedge c_y \leftarrow !g(t);h_2; ...;h_n]>$, let $S_\rho(O_\partial)$ = p, where O_∂ is the set of all the applicable plans of ∂, $p = +!g(s) : b_1\wedge ... \wedge b_m \leftarrow k_1; ... ;k_j$. The plan p is intended with respect to an event ∂ iff there exists an applicable unifier σ such that $[p_1\backslash ... \backslash f : c_1\wedge ...\wedge c_y \leftarrow !g(t);h_2; ...; h_n \backslash (+!g(s) : b_1\wedge ... \wedge b_m)\sigma \leftarrow (k_1; ... ;k_j)\sigma]> \in I$.

If $i = T$, a new intention i_{new} is generated and sent into I:

(i) In the case of a test goal $?g(t)$, $i_{new} = [T\backslash true : true <- ?g(t)]$;

(ii) In the case of beliefs $+b(t)$ or $-b(t)$, $i_{new} = [T\backslash true : true \leftarrow +b(t)]$ or $[T\backslash true : true \leftarrow -b(t)]$;

(iii) In the case of an achievement goal $+!g(t)$, let $S_\rho(O_\partial)$ = p, where O_∂ is the set of all the applicable plans of ∂, $p = e : b_1\wedge ... \wedge b_m <- h_1; ... ;h_n$. The plan p is intended with respect to ∂ iff there exists an applicable unifier σ such that $[T \backslash (e : b_1\wedge ... \wedge b_m <- h_1; ... ;h_n)\sigma] \in I$.

When d has the form $@q(Send,Id,Eve:cond(t),Retn)$. If there exists a plan $p=e : b_1\wedge ... \wedge b_m <- h_1; ... ;h_n$ and an unifier σ such that $(@q(Send,Id,Eve:cond(t),Retn))\sigma=(e)\sigma$, the plan p is intended with respect to ∂ iff there exists a solving substitution θ such that $[T\backslash true:true \leftarrow (@m(Send,Id, Retn))\sigma\theta(e:b_1\wedge...\wedge b_m<-h_1;...;h_n)\sigma\theta] \in I$; otherwise $[T\backslash true:true \leftarrow @m(Send,Id,null)] \in I$.

When d has the form $@q(Send,Id,Eve: cond(t),true)$. Let $S_\rho(O_\partial)$ = p, where O_∂ is all the applicable plans for ∂ and $p = e : b_1\wedge ... \wedge b_m <- h_1; ... ;h_n$. The plan p is intended with respect to ∂ iff there exists an applicable unifier σ such that $[T\backslash (e : b_1\wedge ... \wedge b_m <- h_1; ... ;h_n)\sigma] \in I$.

When d has the form $@m(Id,Retn)$, suppose the intention with identifier Id is $[p_1\backslash ...\backslash f:c_1\wedge ...\wedge c_y \leftarrow (Para);h_2; ...;h_n]$, if

retn≠null then *Id* is replaced by $[p_1\backslash...\forall:c_1\wedge...\wedge c_y\leftarrow(h_2;...;h_n)\{Para/retn\}]$; if *retn=null* then *Id* is blocked for ever.

3.3.2 The Execution of Intention

The function S_λ selects an intention from set *I* to execute. The first statement of the body of the top plan of an intention may be a simple statement (i.e. a goal, a belief, etc.) or a complex statement such as conditional statement, etc..

Definition 3.5 (simple statement) Suppose $S_\lambda(I) = i$. When $i = [p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow !g(t);h_2;...;h_n]$, the intention *i* is said to have been executed iff $<+!g(t), i>\in E$; When $i = [p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow ?g(t);h_2;...;h_n]$, *i* is said to have been executed iff there exists a substitution θ such that $\forall g(t)\theta$ is a logical consequence of *B* and *i* is replaced by $[p_1\backslash...\backslash(f:c_1\wedge...\wedge c_y)\theta\leftarrow h_2\theta;...;h_n\theta]$; When $i = [p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow a(t);h_2;...;h_n]$, *i* is said to have been executed iff $a(t)\in A$, and *i* is replaced by $[p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow h_2;...;h_n]$. In addition, according to the results of *a(t)*, corresponding external events such as $<*b(t),T>$ is generated to revise *B*, where $*\in\{+,-\}$; When $i = [p_1\backslash...\backslash p_{z-1}\backslash g(t):c_1\wedge...\wedge c_y\leftarrow true]$ and $p_{z-1} = e:b_1\wedge...\wedge b_x\leftarrow !g(s);h_2;...;h_n$, *i* is said to have been executed iff there exists a substitution θ such that $g(t)\theta = g(s)\theta$, and *i* is replaced by $[p_1\backslash..\backslash p_{z-2}\backslash(e:b_1\wedge...\wedge b_x)\theta\leftarrow(h_2;...;h_n)\theta]$; When $i = [p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow *b(t);h_2;...;h_n]$, *i* is said to have been executed iff *i* is replaced by $[p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow h_2;...;h_n]$, and *B* is revised such that **b(t)* holds.

Definition 3.6 (communicative statement) Suppose $S_\lambda(I) = i$. When $i = [p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow @q(Send,Rec,Self,Eve:cond(t),Retn);h_2;...;h_n]$, *i* is said to have been executed iff $@q(Send,Rec,Self,Eve:cond(t),Retn)\in CA_i$, and *i* is replaced by $[p_1\backslash...\forall:c_1\wedge...\wedge c_y\leftarrow\$(Retn);h_2;...;h_n]$, where $ means the intention is blocked; When $i = [p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow @q(Send,Rec,Self,Eve:cond(t),true);h_2;...;h_n]$, *i* is said to have been executed iff $@q(Send,Rec,Self,Eve:cond(t),true)\in CA_i$, and *i* is replaced by $[p_1\backslash...\forall:c_1\wedge...\wedge c_y\leftarrow h_2;...;h_n]$; When $i = [p_1\backslash...\forall:c_1\wedge...\wedge c_y\leftarrow @m(Rec,id,retn);h_2;...;h_n]$, *i* is said to have been executed iff $@m(Rec,id,retn)\in CA_i$, and *i* is replaced by $[p_1\backslash...\forall:c_1\wedge...\wedge c_y\leftarrow h_2;...;h_n]$.

Definition 3.7 (complex statement) Suppose $S_\lambda(I) = i$. When $i = [p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow IF\ ?c(t):s_1;...;s_v\ FI;h_2;...;h_n]$, *i* is said to have been executed iff, if $\exists\sigma$ such that $(c(t)\sigma)\in B$, then *i* is replaced by $[p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow s_1;...;s_v;h_2;...;h_n]$, otherwise *i* is replaced by $[p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow h_2;...;h_n]$; When $i = [p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow ||\ ?c(t):s_1;...;s_v||\ ;h_2;...;h_n]$, *i* is said to have been executed iff, if $\exists\sigma$ such that $(c(t)\sigma)\in B$, then *i* is replaced by $[p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow s_1;...;s_v\ ;||c(t):s_1;...;s_v||;h_2;...;h_n]$, otherwise *i* is replaced by $[p_1\backslash...\forall:c_1\wedge...\wedge c_y\leftarrow h_2;...;h_n]$; When $i = [p_1\backslash...\forall:c_1\wedge...\wedge c_y\leftarrow\#\ c_1(t):s_1\ \#...\#\ c_k(t):s_k\#\ others:\$;h_2;...;h_n]$, *i* said to have been executed iff, if $\exists\sigma$ such that $(c_w(t)\sigma)\in B\ (1\le w\le k)$, then *i* is replaced by $[p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow s_w;h_2;...;h_n]$, otherwise *i* is replaced by $[p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow\$;h_2;...;h_n]$, i.e., *i* is blocked for ever.

The algorithm of the interpreter of SPLAW can be written according to the above operational semantics.

4 Proof theory

Now, based on labeled transition systems, we briefly give its proof theory.

Definition 4.1 A BDI transition system is a pair $<\Gamma, \Rightarrow>$, where Γ is a set of BDI configurations and $\Rightarrow\subseteq\Gamma\times\Gamma$ is a binary transition relation.

Definition 4.2 A BDI configuration is a tuple $<E_i, A_i, CA_i, B_i, I_i, i>$, where $E_i\subseteq E$, $B_i\subseteq B$, $A_i\subseteq A$, $CA_i\in CA$, $I_i\subseteq I$, and *i* is the label of the transition.

Since P is a constant and goals appear as intentions, they are not taken into account.

Now we can write transition rules that take an agent from one configuration to its subsequent configuration.

Rule 4.1 $\dfrac{<\{...,<+!g(t),T>,...\},A_i,CA_i,B_i,I_i,i>}{<\{...\},A_i,CA_i,B_i,I_i\cup\{[p\sigma\theta]\},i+1>}$, where $p = +!g(s): b_1\wedge...\wedge b_m\leftarrow h_1;...;h_n\in P$, $S_\varepsilon(E) = <+!g(t), T>$, $g(t)\sigma = g(s)\sigma$ and $\forall(b_1\wedge...\wedge b_m)\theta$ is a logical consequence of B_i.

Rule 4.2 $\dfrac{<\{...,<\omega,T>,...\},A_i,CA_i,B_i,I_i,i>}{<\{...\},A_i,CA_i,B_i,I_i\cup\{[true:true\leftarrow\omega]\},i+1>}$, where $S_\varepsilon(E) = <\omega, T>$ and $\omega\in\{b(t), ?g(t)\}$.

Rule 4.3 $\dfrac{<\{...,<@\omega,T>,...\},A_i,CA_i,B_i,I_i,i>}{<\{...\},A_i,CA_i,B_i,I_i\cup\{[\delta\backslash\zeta]\},i+1>}$, where $S_\varepsilon(E) = <@\omega, T>$. If there exists $p = e: b_1\wedge...\wedge b_m\leftarrow h_1;...;h_n\in P$, $(@\omega)\sigma = (e)\sigma$, and $\forall(b_1\wedge...\wedge b_m)\theta$ is a logical consequence of B_i, then $\zeta = p\sigma\theta$ and when $\omega = @q(S,Id,Eve:cond(t),Retn)$, $\delta = true:true\leftarrow(@m(S,Id,Retn))\sigma\theta$; when $\omega = @q(S,Id,Eve:cond(t),true)$, $\delta = T$; otherwise $\delta = true:true\leftarrow@m(S,Id,null)$ and $\zeta = T$.

Rule 4.4 $\dfrac{<\{...,<@m(Id,return),T>,...\},A_i,CA_i,B_i,\{...,int,...\},i>}{<\{...\},A_i,CA_i,B_i,\{...,[p_1\backslash..\forall:c_1\wedge...\wedge c_y\leftarrow(h_2;...;h_n)\theta],...\},i+1>}$, where $S_\varepsilon(E) = <@m(Id, return),$

T>, and the intention with identifier Id is $int = [p_1 \backslash ... \backslash f : c_1 \wedge ... \wedge c_y \leftarrow \$(Para); h_2; ...; h_n]$, θ is substitution {Para/return}.

Rule 4.5 $$\frac{<\{..., <+!g(t), j>, ...\}, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash p_z], ...\}, i>}{<\{...\}, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash p_z \backslash p\sigma\theta], ...\}, i+1>}$$, where $S_\varepsilon(E) = <+!g(t), j>$, $j = [p_1 \backslash ... \backslash p_z]$, $p_z = f : c_1 \wedge ... \wedge c_y \leftarrow !g(t); h_2; ... ; h_n$, $p = +!g(s) : b_1 \wedge ... \wedge b_m \leftarrow k_1; ... ; k_x$, $g(t)\sigma = g(s)\sigma$ and $\forall(b_1 \wedge ... \wedge b_m)\theta$ is a logical consequence of B_i.

Rule 4.6 $$\frac{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash f: c_1 \wedge ... \wedge c_y \leftarrow !g(t); h_2; ...; h_n], ...\}, i>}{<E_i \cup \{<+!g(t), j>\}, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash p_z], ...\}, i+1>}$$, where $S_\lambda(I_i) = [p_1 \backslash ... \backslash p_z]$, $p_z = f : c_1 \wedge ... \wedge c_y \leftarrow !g(t); h_2; ... ; h_n$.

Rule 4.7 $$\frac{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash f: c_1 \wedge ... \wedge c_y \leftarrow ?g(t); h_2; ...; h_n], ...\}, i>}{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash (f: c_1 \wedge ... \wedge c_y)\theta \leftarrow h_2\theta; ...; h_n\theta], ...\}, i+1>}$$, where $S_\lambda(I_i) = [p_1 \backslash ... \backslash f : c_1 \wedge ... \wedge c_y \leftarrow ?g(t); h_2; ... ; h_n]$, $\forall g(t)\theta$ is a logical consequence of B_i.

Rule 4.8 $$\frac{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash f: c_1 \wedge ... \wedge c_y \leftarrow a(t); h_2; ...; h_n], ...\}, i>}{<E_i, A_i \cup \{a(t)\}, CA_i, B_i, \{..., [p_1 \backslash .. \backslash f: c_1 \wedge ... \wedge c_y \leftarrow h_2; ...; h_n], ...\}, i+1>}$$, where $S_\lambda(I_i) = [p_1 \backslash ... \backslash f : c_1 \wedge ... \wedge c_y \leftarrow a(t); h_2; ... ; h_n]$.

Rule 4.9 $$\frac{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash p_{z-1} \backslash !g(t): c_1 \wedge ... \wedge c_y \leftarrow true], ...\}, i>}{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash p_{z-2} \backslash (e: b_1 \wedge ... \wedge b_x \leftarrow h_2; ...; h_n)\theta], ...\}, i+1>}$$, where $S_\lambda(I_i) = [p_1 \backslash ... \backslash p_z]$, $p_z = !g(t) : c_1 \wedge ... \wedge c_y \leftarrow true$, $p_{z-1} = e : b_1 \wedge ... \wedge b_x \leftarrow !g(s); h_2; ... ; h_n$, $g(t)\theta = g(s)\theta$.

Rule 4.10 $$\frac{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash f: c_1 \wedge ... \wedge c_y \leftarrow \omega; h_2; ...; h_n], ...\}, i>}{<E_i, A_i, CA_i \cup \{\omega\}, B_i, \{..., [p_1 \backslash .. \backslash f: c_1 \wedge ... \wedge c_y \leftarrow \delta; h_2; ...; h_n], ...\}, i+1>}$$, where $S_\lambda(I_i) = [p_1 \backslash ... \backslash f : c_1 \wedge ... \wedge c_y \leftarrow \omega; h_2; ... ; h_n]$. If ω = @q(Send,Rec,self,Eve: cond(t) , Retn), δ = $(Retn); if ω = @q(Send,Rec,self,Eve: cond(t) , true) or ω = @m(Rec,id,retn), δ = true.

Rule 4.11 $$\frac{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash p_z], ...\}, i> \ \wedge \exists\sigma(c(t)\sigma \in B)}{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash f: c_1 \wedge ... \wedge c_y \leftarrow s_1; ...; s_k; h_2; ...; h_n], ...\}, i+1>},$$

$$\frac{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash p_z], ...\}, i> \ \wedge \neg\exists\sigma(c(t)\sigma \in B)}{<E_i, A_i, CA_i, B_i, \{..., [p_1 \backslash .. \backslash f: c_1 \wedge ... \wedge c_y \leftarrow h_2; ...; h_n], ...\}, i+1>},$$

where $S_\lambda(I_i) = [p_1 \backslash ... \backslash p_z]$, $p_z = f : c_1 \wedge ... \wedge c_y \leftarrow$ IF ?c(t) : s_1; ...; s_k FI ;h_2; ...; h_n .

The rules of the other cases can be written accordingly.

Definition 4.3 A BDI derivation is a finite or infinite sequence of BDI configurations, i.e., $\gamma_0, ..., \gamma_i, ...$

Using the above proof rules we can formally prove certain behavioral properties, such as safety and liveness of agent system.

5 An Example—Electronic Market Simulation System

5.1 Descriptions

Suppose on user *sweet*'s computer there is a service-performing agent, named *aide*, who provides services such as email-classifying, meeting-scheduling, and pastime-scheduling. By means of observing and learning from *sweet*'s everyday behaviors, *aide* gets accustomed to *sweet*'s habits little by little and accumulates them in its base belief set. On receiving *sweet*'s request, *aide* tries to decompose user's goal into subgoals, generates a dynamic plan, and sends the subgoals which need to cooperate with other agents to electronic market agent, named *market.* After receiving replies, *aide* recommends appropriate information to *sweet.*

Now, suppose *aide* receives a goal from user *sweet*: "go to see a film on Sunday, and then have a meal nearby". Also suppose part of the current base belief set of *aide* is as follows(separated by ;):

custom_dish_number(2); leisure(sunday); liketype(comedy); ~liketype(action);
like_eat(vegetable); ~like_eat(spicy_dish); like_drink(coffee);

aide decomposes *sweet*'s goal into two subgoals: *see a film* and *have a meal,* and then enters the electronic market to negotiate with other agents. *aide* selects an appropriate cinema first and tries to find a restaurant nearby which satisfies *sweet.* If no such restaurant exists, *aide* will try to select another cinema and continue the above procedure until success. According to *sweet*'s preferences (likes vegetable, hates spicy, likes comedy, hates action films, etc.) and the constraints (on Sunday, nearby), *aide* generates the dynamic plan *DP* as follows:

```
!true : true   <-
+~satisfied(sweet);                                                                                    (1)
|| ?satisfied:                                                                                         (2)
      @q(aide,market,Self,select_cinema: true , Cinema );                                              (3)
      @q(aide,Cinema,Self,select_film:leisure(sunday) & liketype(comedy) & ~liketype(action), Film );  (4)
      @q(aide,market,Self,select_rest: near(Cinema), Rest );                                           (5)
      IF ?~(Rest = null):                                                                              (6)
            ?custom_dish_number(Num);                                                                  (7)
            +index(Num);                                                                               (8)
            || ?(index(X) & X > 0) :                                                                   (9)
             @q(aide,Rest,Self,select_dish: like_eat(vegetable) & ~like_eat(spicy_dish), Dish);        (10)
             bookmark(Dish);  /*primitive action, write information into bookmark*/                   (11)
             IF ?index(Y): +index(Y-1) FI;                                                             (12)
            || ;
            @q(aide,Rest,Self,select_drink: like_drink(coffee), Drink );                               (13)
            bookmark(Drink) ;                                                                          (14)
            +satisfied(sweet);                                                                         (15)
      FI;
|| ; return.                                                                                           (16)
```

market provides facilities for cooperation between the active agents registered in it. Suppose part of *market's* current base belief set and plan set are as follows:

assistant(aide); cinema_proxy(redstar); cinema_proxy(dawn); restaurant_proxy(mallow); restaurant_proxy(nestle); nearby(redstar,nestle); nearby(dawn,mallow); ~restaurant_full(nestle); cinema_full(dawn); restaurant_full(mallow); ~cinema_full(redstar);

@q(A1,Id,select_cinema:true ,X) : ~cinema_full(X) <- true. (EPlan1)

@q(A2,Id,select_rest:near(X),Y): nearby(X,Y) & ~restaurant_full(Y) <- @q(market,Y,Self,decrease_room : 1, true). (EPlan2)

Cinema proxies provide the services of ticket booking according to customs' different requests. Suppose part of the cinema proxy *redstar's* current base belief set and plan set are as follows:

show_on(film1,sunday); show_on(film2,sunday); show_on(film3,saturday); show_on(film4,saturday); is(film2,comedy); is(film3,comedy); is(film4,action);

@q(A,Id,select_film:leisure(Time) & liketype(X) & ~liketype(Y),Film):
show_On(Film,Time) & is(Film,X) & ~is(Film,Y) <- take_a_ticket(Film). (CPlan1)

According to different users' appetites, restaurant agents are responsible for meal-booking, dish-selecting, etc.. Suppose part of *nestle*'s base belief set and plan set are shown as follows:

vacancy(5); have_drinks(beer); have_drinks(coffee); is(dish1,spicy_dish); is(dish2,vegetable); is(dish3,vegetable); is(dish1,vegetable);

@q(A2,Id,decrease_room:X, true): vacancy(Y) & Y>X <- !decrease(X). (RPlan1)

!decrease(X): vacancy(Y) <- +vacancy(Y-X). (RPlan2)

@q(A,Id,select_dish:like_eat(X) & ~like_eat(Y), Dish):is(Dish,X) & ~is(Dish,Y)<-true. (RPlan3)

@q(A,Id,select_drink:like_drink(Drink), Drink): have_drinks(Drink) <-true. (RPlan4)

5.2 Execution

On starting, *aide* executes the intention *[T\DP]*. The first statement of the dynamic plan *DP* is a belief assertion, which is used to add a new belief, *~satisfied(sweet)*, into its belief set. A loop is followed, which is ended until the belief *satisfied(sweet)* is satisfied. In the loop body, Statement(3) is used to ask *market* to select an appropriate cinema(with no constraints) and ascribe it to the parameter *Cinema*. Statement(4) is used by *aide* to ask *Cinema* to select a film(with the constraints: shown on Sunday, comedy and not action film) and ascribe it to the parameter *Film*. Statement(5) is used to ask *market* to select a restaurant which is near the selected cinema. If there exists such restaurant(Rest ≠ Null), then a loop is entered to select dishes for *sweet*. From previous experiences, *aide* learned that *sweet* was used to select just two dishes, like vegetable and hate spicy. Based on these, *aide* selects two dishes for *sweet* and records them into bookmark. And voluntarily, *aide* will try to select a kind of drink for *sweet*. Another belief assertion is followed to end the outer loop. At last, *aide* returns its bookmark to *sweet*.

On receiving the message *@q(aide,market,id,select_cinema: true,Cinema)* from *aide*, *market* generates a triggering event *<@q(aide,id,select_cinema: true,Cinema),T>*. Suppose *market* selects *Eplan1* to unify with this event, after unification *Eplan1* is changed into *@q(aide,id,select_cinema: true, Cinema) : ~cinema_full(Cinema) <-true. An substitution, {Cinema/redstar}*, which satisfies *market*'s base belief set, is selected and a new intention *[T\true:true<-@m(aide,id,redstar)\@q(aide, id,select_cinema:true,redstar): ~cinema_full(redstar) <- true]* is generated. Since the body of the top plan of this intention is *true*, after execution, the intention is changed into *[T\true:true<-@m(aide,id, redstar)]*. Now, *market* will send message *@m(aide,id, redstar)* and return *"redstar"* to *aide*.

On receiving the message *@q(aide,market,id,select_rest: near(redstar), Rest)* from *aide*, a similar procedure is adopted by *market* and the answer *"nestle"* is returned.

On receiving the message *@q(aide,redstar,id,select_film:leisure(sunday) & liketype(comedy) & ~liketype(action), Film)*

from *aide*, *redstar* generates a triggering event *<@q(aide,id,select_film: leisure(sunday) & liketype(comedy) & ~liketype(action), Film), T>*. Suppose *redstar* selects *Cplan1* to unify with this event. After unification, *Cplan1* is changed into *@q(aide,id,select_film: leisure(sunday)&liketype(comedy)&~liketype(action),Film):show_On(Film, sunday)&is(Film,comedy) & ~is(Film,action) <- take_a_ticket(Film)*. An substitution, *{Film/film2}*, which satisfies *redstar*'s base belief set, is selected and a new intention *[T\true:true<-@m(aide,id,film2)\@q(aide,id, select_film: leisure(sunday)&liketype(comedy)& liketype(action),film2): show_On(film2,sunday) & is(film2,comedy) & ~is(film2, action) <- take_a_ticket(film2)]* is generated. Since the body of the top plan of this intention is an action *take_a_ticket*, the action is added into set *A*. After execution of this action, the intention is changed into *[T\true:true<-@m(aide,id,film2)]*. Now *market* will send message *@m(aide,id,film2)* to return answer "*film2*".

On receiving the message *@q(market,nestle,id,decrease_room: 1,true)* from *market*, *nestle* generates a triggering event *<@q(market,id,decrease_room: 1,true), T>*. Suppose *Rplan1* is used to unify with this event, which generates a new intention i_{new}. Since the body of the top plan of i_{new} is an achievement goal *!decrease(1)*, an internal triggering event, *+!decrease(1)*, is generated. At this point, nestle will divert its attention by selecting *Rplan2* to unify with *+!decrease(1)* and generate another intention i_{new}'. The execution of i_{new}' will revise the belief *vacancy(5)* to *vacancy(4)*.

For the message *@q(aide,nestle,id,select_dish: like_eat(vegetable) & ~like_eat(spicy_dish), Dish) and @q(aide, nestle,id,select_drink: like_drink(coffee), Drink), nestle* can process in the same way.

In this example, *aide, redstar/dawn, mallow/nestle* are the instances of the subclasses of *ActiveAgent*, market is an instance of the subclass of *CoopAgent*. Service-performing agent *aide* reflects many features of agents: it can determine how to achieve a top level task (autonomy); it can cooperate with other service-performing agent (social ability); if a restaurant is full, it will try another one (responsibility); it can voluntarily check with *nestle* to book a cup of drink (proactiveness). In addition, mobility can also be added, where an agent can move from the local computer to the computer where *market* lies, in order to lessen the network cost.

6 Comparisons

6.1 Comparisons Between SPLAW and Pure Logic Program

An agent specification of SPLAW includes a base belief set and a set of plans, which is similar to a logic programming specification of facts and rules. However, some of the major differences between a logic program and an agent program are as follows:

In a pure logic program there is no difference between a goal in the body of a rule and the head of a rule. In an agent program the head consists of a triggering event rather than a goal. This allows the plans to be invoked not only by data-directed (using addition/deletion of beliefs), but also goal-directed (using addition/deletion of goal) stimuli.

Rules in a pure logic program are not context-sensitive as plans of SPLAW.

Execution of Rules successfully returns a binding for unbound variables; In addition to this, execution of plans also generates a sequence of primitive actions that affect the environment.

In SPLAW, a goal is called indirectly by generating an event. This gives the agent better real-time control as it can change its focus of attention, if needed, by adopting and executing a different intention. Thus, one can view agent programs as multi-threaded, interruptable logic programs. While, in a logic program, the goal being queried cannot be interrupted.

6.2 Comparisons with Related Work

Shoham has provided a new paradigm—agent-oriented programming and designed a simple agent language AGENT0[8]. In AGENT0, the mental states of an agent are composed of belief, capability and commitment. However, the request between agents can only include primitive actions, and complex actions must be decomposed into primitive actions and sent one by one, which lavishes the computational resources of agent and network bandwidth. PLACA[9] is an extended version of AGENT0. A new attitude, intention, is added and the planning capability for top level goals is provided. But it is based on modal logic, no proof theory is provided, and it is not clear how the data structures to capture the model theoretical semantics of belief, capability, and intention. While, in our work, we have discussed the correspondence between the proof theory and the operational semantics of SPLAW.

Although AgentSpeak[7] has provided the operational semantics and proof theory of BDI agent, it provides no communicative mechanism, and has not provided semantics for non-deterministic statements, conditional statements and loop statements. While, in SPLAW, not only the semantics of all kinds of components are provided, but has two other advantages: (i) based on standard inter-agent communication language KQML; and (ii) inheritance facilities.

Agent_K[2] also adopts the standard of KQML, which makes it possible for the Agent_K agents to cooperate with other agents that abide by the same standard. But it is also lacks a proof theory and explicit semantics, just as PLACA.

CONGOLOG[4] is a single thread programming language, and its semantics is based on situated calculus. While, SPLAW is a multi-thread system with the feature of inherence.

6 Conclusion

In SPLAW, the mental states of agent are expressed by data structures, which avoids the complexity of the proof theory of multi-agent logic. The semantics of agent system is described not from model theory, but from operation. The proof theory of SPLAW is given by means of labeled transition system, and the correspondence between the operational semantics and its proof theory is guaranteed, which makes it possible to unify the theory and practice.

Although the inheritance of agent has been presented for a long time[8], it has not been paid much attention. In this paper, from social behaviors of agents, we try to organize and define some basic agent classes for SPLAW. By inheriting these basic agent classes, or augmenting with new agent classes, user can design their agent systems of different application domains smoothly. By this way, the relation between OOP and AOP can be established naturally.

References

[1] Cohen P.R. and Levesque H.J. Intention is choice with commitment. *AI 42(3),* 1990.

[2] Davies W.H. and Edwards P. Agent-K: An Integration of AOP and KQML.*The Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94),* ACM Press, November 1994.

[3] Gardenfors P. and Rott H. Belief Revision. In Gabbay D. M., Hogger C.J. and Robinson J.A., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 4, Epistemic and Temporal Reasoning.* Clarendon Press, Oxford, 1995, 35-132.

[4] Lesperance Y., Levisque H.J., Lin F. and Marcu D. Foundations of a Logical Approach to Agent Programming. In Wooldridge M., Muller J.P. and Tambe M., editors, *Intelligent Agents II --IJCAI'95 Workshop (ATAL) Montreal, Canada, August 19-20.* LNAI 1037, Springer-Verlag, 1996, 331-346.

[5] Mayfield J., Labrou Y. and Finin T. Evaluation of KQML as an Agent Communication Language. In Wooldridge M., Muller J.P. and Tambe M., editors, *Intelligent Agents II --IJCAI'95 Workshop(ATAL) Montreal, Canada, August 19-20.* LNAI 1037, Springer-Verlag, 1996, 347-360.

[6] Ovum Report. Intelligent agent: the new revolution in software. 1994.

[7] Rao A.S. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In Van de Vlde W. and Perram J.W., editors, *Agents Breaking Away, MAAMAW'96,* Eindhoven, The Netherlands, January 1996. LNAI 1038, Springer-Verlag, 1996, 42-55.

[8] Shoham Y. Agent -oriented programming. *Artificial Intelligence, 60(1):*51-92, 1993.

[9] Thomas S.R. The PLACA Agent Programming Language. In Wooldridge M. and Jennings N. R., editors, *Intelligent Agents -- Proceedings of the 1994 Workshop on Agent Theories, Architectures, and Languages (ATAL-94).* LNAI 890,Springer-Verlag, 1995, 355-370.

[10] M. Wooldridge. Coherent Social Action. In *Proceedings of the Eleventh European Conference on AI (ECAI-94),* Amsterdam, The Netherlands, August 1994.