The BDIP Software Architecture and Running Mechanism for Self-Organizing MAS

Yi Guo, Xinjun Mao, Fu Hou, Cuiyun Hu, Jianming Zhao

The BDIP Software Architecture and Running Mechanism for Self-Organizing MAS

Yi Guo[1] , Xinjun Mao[1], Fu Hou[1], Cuiyun Hu[1], Jianming Zhao[2],

[1] Department of Computer Science and Technology,
School of Computer, Nation University of Defence Tehnology
[2]Department of Computer Science and Technology, Zhejiang Normal University
Berniegy@gmail.com

Abstract. As there are huge gaps between the local micro interactions among agents and the global macro emergence of self-organizing system, it is a great challenge to design self-organizing mechanism and develop self-organizing multi-agent system to obtain expected emergence. Policy-based self-organization approach is helpful to deal with the issue, in which policy is the abstraction of self-organizing mechanism and acts as the bridge between the local micro interactions and global macro emergence. This paper focuses on how to develop software agents in policy-based self-organizing multi-agent system and proposes a BDIP architecture of software agent. In our approach, policy is an internal component that encapsulates the self-organizing information and integrates with BDI components. BDIP agent decides its behaviors by complying with the policies and respecting BDI specifications. An implementation model and the running mechanism as well as corresponding decision algorithms for BDIP agents are studied. A case of self-organizing system is studied to illustrate our proposed approach and show its effectiveness.

Keywords. multi-agent system, self-organization, agent architecture

1 Introduction

Self-organization refers to a process where a system changes its internal structure without explicit external and central control. It often results in emergent behavior in the global system [1].With the pervasiveness of distributed information systems, self-organizing systems become more and more attractive to researchers from different application areas [2]. Agent technology is considered as an appropriate and powerful paradigm to develop large-scale complex systems applications. As a kind of such complex systems, self-organizing systems are usually engineered with agent metaphor, which views the whole system as MAS (Multi-Agent Systems) and using software agents as basic components to construct the systems [4].

However, developing self-organizing MAS in an iterative and effective way is still a great challenge in the literature of software engineering [5]. The obstacle is how to obtain desirable global system characteristic through the local interactions among agents. In self-organizing MAS, there is an absence of centralized control node in the system and the agents only interact with their local environment. This leaves a significant gap between the local interaction and global system characteristic, and brings obstacles which are put in evidence during the development of the systems. To obtain the desirable global characteristic, the developer therefore must adjust the behaviors of agents iteratively. However, self-organizing MAS often consist of large numbers of agents and are deployed in complex environment. Designing and deploying such systems in an iterative way is difficult. How to effectively support the development of such systems is still an open issue [5].

Against the background, we have proposed a policy-based self-organization approach in our previous work [6] which affects the emergences of multi-agent systems by restricting or guiding agents' behaviors in terms of policy. In order to support the development of the PSOMAS (Policy-based Self-Organizing Multi-Agent Systems), this paper proposes a BDIP software architecture in which policies are viewed as component of the agent. Based on the architecture, an implementation model and running mechanisms as well as agent behavior decision algorithms are also provided. The rest of this paper is organized as follows: Section 2 gives a brief introduction of policy-based self-organizing MAS. Section 3 proposes the agent architecture, policy representation, and running mechanism of the software agents. Section 4 discusses the policy-based agent decision algorithms and a case study is illustrated in section 5. Section 6 discusses the related works as well as conclusions and future works are discussed in section 7.

2 Policy based Approach to Self-Organizing Multi-agent Systems

To intuitively understand the challenges in self-organizing MAS, we firstly introduce an example of a group of self-organizing robots whose aims are to explore and carry ore in a strange environment. Each robot provides functions to randomly walk in the environment to find ore resource, and carry ore back to the base. Furthermore, each robot can broadcast its position to other robots. The system searches and takes ore by using the self-organization of these robots. When the users search for ore depending on these robots in strange environments, they perhaps need to cope with different circumstances, such as various landforms and ore distributing and etc. However, it is impossible that the fixed behaviors of robots can satisfy all scenarios. On the other hand, it is not always feasible that the users redesign and redeploy the robots after they have acquired the new requirements, for example the robots are exploring on the mars. Then we need a new approach to effectively change the behaviors of agent during runtime to meet the variety environment and requirements.

In human society, policies are often used to restrict and guide the behaviors of people. With these policies, human society often presents a self-organizing process

and results in different emergent phenomena. For example, the economic policies often result in the changes of macro economic index, which is owing to the people's economic behaviors like stock transaction are affected by such policies. In the literature of self-organizing MAS, in order to facilitate the solving of the issues discussed above, the policies in human society can be used for self-organizing MAS. By this approach, policies give a presentation of the behaviors of agents and the agents must to comply with them at runtime. On one hand, developers need to design the policies as well as the agents in the design phase of such systems. On the other hand, the behaviors of agents are affected by these policies at runtime, and developers can evaluate the system's macro characteristics whether satisfies the requirements or not. If it does not satisfy the requirements, developers can change the policies, which can cause changes of the agents' behaviors and result in the changes of the self-organizing process of the whole system.

3 BDIP Architecture and Running Mechanism

3.1 BDIP Architecture of Software Agent

The BDI architecture of software agents had been accepted for a long time both in academe and industry. The architecture has three components: *Belief*, *Desire*, and *Intention*. *Belief* means the cognitions of an agent about its environment and internal state. *Desire* means the goals that an agent wants to pursue, and *Intention* means the commitment plans of the agent which is useful for the accomplishment of the goals in *Desire* [7]. We consider that the BDI architecture is useful for analyzing the autonomous behaviors of rational agents and easier to accept the *policy* as a new decision component than other architectures (e.g. reactive architecture). This paper proposes a BDIP (Belief, Desire, Intention and Policy) agent architecture by extending the BDI architecture (see Fig.1).

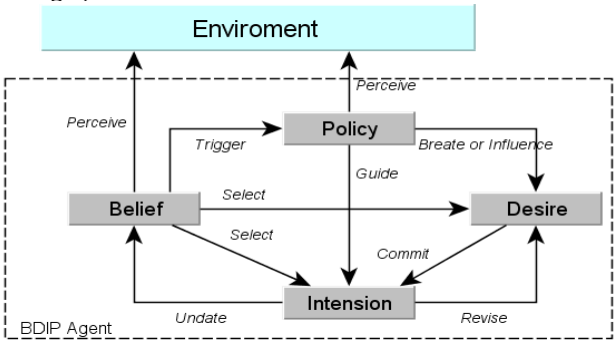


Fig. 1. BDIP architecture of Agent

In PSOMAS, the environment of a BDIP agent consists of policies and other agents [6], and the BDIP agent can perceive the policies from its local environment. The perceived policies are deposited in the *Policy* component, and are triggered by

the *policy conditions* (introduced in section 3.2) which are specified in the *Belief* component. Policies can affect both *Desire* component and *Intention* component. For the *Desire* component, a new goal will be created by the policy or some of goals are prohibited. For the *Intention* component, the execution of committed plan will be guided by the policy, for example some actions are forbidden to be executed and some actions are preferential (corresponding algorithms will be introduced in section 4). The relationship among *Belief*, *Desire* and *Intention* is same as [7].

3.2 Representation and Realization of Policy

Policies can be viewed as a set of rules which restrict the behaviors or states of the agents in the system. This paper distinguishes two kinds of policies: *Obligation* and *Prohibition*. The *Obligation* represents the action that an agent need to perform or the state the agent need to keep as well as the *Prohibition* represents the action that an agent must not to perform or the state not to appear. On the other hand, a policy consists of the conditions to be satisfied and the action (state) to be performed (kept) by agents. Formally, it can be described by EBNF as follows:

Policy**:=** ***Obligation*** (***IF*** Self-condition ***WHEN*** Env-event ***DO*** (Action ⏐ State))
⏐ ***Prohibition***(***IF*** Self-condition ***WHEN*** Env-event ***DO*** (Action ⏐ State))

Self-condition specifies the internal state of an agent to be satisfied and *Env-event* represents the happened event of the environment of the agent. ***"Obligation(IF*** Self-condition ***WHEN*** Env-event ***DO*** (Action ⏐ State***)"*** means that when the *Self-condition* of the agent is satisfied and *Env-event* is happening in the environment, the agent need to perform the *Action* or keep the *State*, ***Prohibition*** means the opposite semantics. The *Self-condition* and *Env-event* can be viewed as *policy-condition*.

A policy at the run-time can be in four states: *Deployed*, *Deactived*, *Executed*, or *Deleted*. The initial state of a newly deployed policy is *Deployed*, what means that the policy exists in the system but is not yet perceived by any agents. More interesting is the policy inside an agent can has three states. *Deactive* means either the policy conditions are not satisfied or the behaviors of the agent are complying with the policy i.e. the agent does not need to adjust its *Desire* or *Intention* component. When the policy conditions are satisfied, the policy is transited to *Executed*. During this state, agent will adjust the *Desire* and (or) *Intention* component according to the policy. If there are some goals or plans prohibited by the policies, the goals and plans will be suppressed and saved. If a user deletes a policy from the system, the agents will be informed by the system. Before the policy is deleted from *Policy* component, the policy needs to be transited to the *Deleted* state in which the agent will check the *Desire* and *Intention* components whether there are goals or plans are suppressed by this policy and resume them respectively if existing.

3.3 Implementation Model and Running Mechanism of BDIP Software Agent

Fig. 2 depicts the running mechanism of BDIP agents. *B*, *D*, *I* and *P* represent Belief set, Goal set, Plan set and the Policy set of the agent respectively. Agenda can be seen as a queue of actions. All executions of plans and other actions must on the agenda.

DE (Deliberation and Execution) is responsible for executing actions on the agenda and adjusting *B*, *D*, *I*, and *P*. In the DE component, AE (Action Execution) always executes the first action of the agenda. After the execution of an action, there may be a new action need to be added on the agenda. For example, an agent needs to execute action *CreateGoal* to create a sub-goal in the execution of a plan, and AE can directly add these new actions on agenda (direct effects). On the other hand, the execution of actions maybe change the state of belief set, DE will inform the CE (Condition Evaluation) about these belief changes. CE then checks whether to trigger a goal, a plan, or a policy as well as add corresponding actions on the agenda (side effects). Moreover, the event in the environment of an agent may also add actions to the agenda, e.g. messages that have been received from other agents and messages need to be processed (External effects).

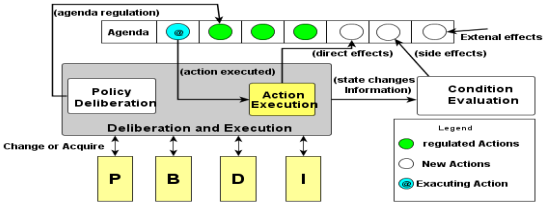


Fig. 2. Implementation Model and Running Mechanism of BDIP Agent

PD (Policy Deliberation) is the component which is used to adjust the *D* and *I* component. When the *policy conditions* of a policy are satisfied, CE will add the *TiggerPolicy* on the agenda, the policy will be transited from *Deactived*" to *Executed* state. Moreover AE will add *ExecutePolicy* on the agenda after *TiggerPolicy* executed, and the *D* and *I* components will be adjusted by PD after *ExecutePolicy* executed. In the adjusting process, perhaps the actions on the agenda also be regulated by PD (agenda regulation), for example add new actions, delete action, and adjust the sequence of the actions. On the other hand, when a user changes the policies in the system, CE will acquire this message and add *PerceivePolicy* or *DropPolicy* on the agenda.

4 Behavior Decision Algorithms of BDIP Agent

In the agent behaviors adjusting process, PD use different algorithms to cope with the different policy type. Table 1 shows the different algorithms for different policy type, the italic are the names of algorithms and "*Null*" means the regulation of this component is needless. The details of these algorithms are listed in Table 2. Moreover we consider the goal in *D* component as discussed in [8] is consists of different states at runtime: "Active" state means the goal is currently pursued by agent. "Suspended" state and "Options" state represent the goal is inactive. Therefore regulation of *D* component is the transition between goal states.

In the algorithm *Obligation_State_for_Goal*, the *goal(state)* means to make the obligation state as a goal. PD searches this goal in *D* component, and upgrades the prior-

ity of this goal as soon as the goal is found. If the goal dose not exist in *D* component, PD will create this goal in *D*. *Achieve goal(state)* means to pursuer this goal right now. In the Algorithm *Obligation_Actiion_for_Plan* PD creates a plan which obtains the execution of the obligated action and adds corresponding actions on the agenda. The plan will be deleted as soon as its execution finished. When a certain state of an agent is prohibited by a **Prohibition** policy, PD will search this state both in *D* and in *I* component. In the algorithm *Prohibition_State_for_Goal*, the state will be treated as a goal. PD will suppress this goal and add *DeliberateNewOption* which is used for selecting of another goal to pursue if the goal is "Active" state (line 1-4).

Table 1. Behavior decision Algorithm for Different Policies Type

Policy Type	Object	Algorithm for Intention	Algorithm for Desire
Obligation	State	Null	*Obligation_State_for_Goal*
	Action	*Obligation_Action_for_Plan*	Null
Prohibition	State	*Prohibition_State_for_Plan*	*Prohibition_State_for_Goal*
	Action	*Prohibition_Action_for_Plan*	Null

When the prohibited goal is being "Suspended" or "Option" state, PD will suppress it and executes *UpdateGoal* by updating the *D* component. Algorithm *Prohibition_State_for_Planl* is designed to search the prohibited state in *I* component. When the prohibited action belonging to the executing plan, PD will suppress this plan and add *ScheduleCondidates* on the agenda to select another plan. If the state can be achieved by the "Suspended" or "Options" plans, PD will suppress the plan and update *I* component. If the user prohibits a certain action to be executed in terms of **Prohibition** policy, PD will search this action on agenda and in *I* component. When the action is on the agenda, PD will suppress it and call another plan on the agenda (line 1-4). If the action is not executing currently but is contained in some plans in *I* component, PD will suppresses these plans and update the *I* component (line 5-7).

Table 2. Behavior Decision Algorithms

Name: Obligation_State_for_Goal

Input: Obligated action; goal of *Desire*
Output: goal
01 if (goal(state) is active) {
02 Rise goal(state) Priority in Agenda;
03 } else
04 if(goal(state) is Option){
05 Add action *SuppressContents* in Agenda;
06 Acheve goal(state);
07 } else
08 if(goal(state) is Suspended) {
09 waitfor (satisfyCondition);
10 Achieve goal(state);
11 }else {
12 Add action CreatGoal in Agenda;
13 Achieve goal(state);
14 Add action DropGoal in Agenda;}

Name: Prohibition_State_for_Goal

Input: Prohibited state; goal of *Desire*
Output: goal
01 if (goal(state)is one of achieving goals) {

Name: Prohibition_State_for_Plan

Input: Obligated state; plan of *Intention*
Output: plan
01 if (goal(state) is executing Plan) {

02 Add action *SuppressContents* in Agenda; 03 Add action *DeliberationNewOption* in Agenda; 04 }else 05 if(goal(state) is Suspended or Option) 06 Add action *SuppressContents* in Agenda; 07 Add action *UpdateGoal* in Agenda;	02 Add action *SuppressContents* in Agenda; 03 Add action *ScheduleCandidates* in Agenda; 04 }else 05 if(state is the state of the plan in *Intention* component) 06 Add action *SuppressContents* in Agenda; 07 Add action *UpdateGoal* in Agenda;
Name: Prohibition_Action_for_Plan	**Name**: **Obligation_Action_for_Plan**
Input: Prohibited action; plan of *Intention* Output: plan 01 if (Action is executing Plan) { 02 Add action *SuppressContents* in Agenda; 03 Add action *ScheduleCandidates* in Agenda; 04 }else 05 if(Action in Plan component) 06 Add action *SuppressContents* in Agenda; 07 Add action *UpdateGoal* in Agenda;	Input: Obligated action; plan of *Intention* Output: plan 01 Add action *CreatPlan* into Agenda; 02 Add action *ExecutePlanStep* in Agenda; 03 Add action *TerminatePlan* in Agenda;

5 Case Study

In this section, we will analyze the case that we introduced in section 2. In this case, the actions of robots should be designed in the design phases. These actions include: random searching, taking ore, sending message, responding message etc. On the other hand, we assume that there are two scenarios need to be considered. The first scenario is that there is only one ore source in the environment. To find resource and carry ore more quickly, it is appropriate to make each robot performs behaviors as follows: 1) randomly works in the environment to find ore resource; 2) carries ore back to base if finds the ore resource; 3) broadcasts position of ore resource if finds it; 4) goes to the position and carries ore to base if having received position information from other robot. To realize these behaviors, the user can deploy the policies as follows:

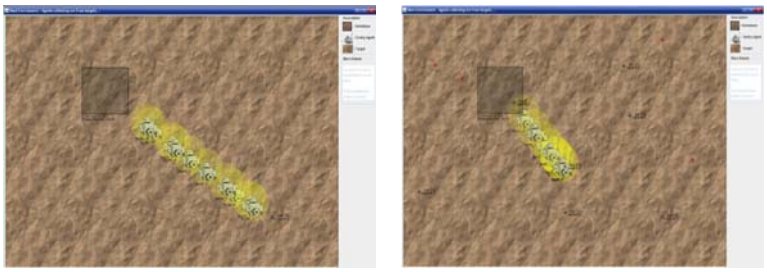
Obligation(***IF*** Searching ***WHEN*** Others_Send_Message ***DO*** TakeOreFromReceived-Position)
Obligation(***IF*** Find_Ore_at_Some_Position ***WHEN*** $ ***DO*** Broadcast(Position))

In the second scenario, robots will be deployed in an environment which has more than one ore resource. Moreover, the user needs all robots to carry the found ore from near to far. According to this requirement, robots should firstly collect some ore resource positions whatever the position is found by itself or received from others. Then select the nearest one from the base to carry ore. To realize this requirement, user should add some new policies:

Obligation (***IF*** Searching ***WHEN*** Others_Send_Message ***DO*** storage_message)

Obligation (***IF*** Message_number=MAX ***WHEN*** $ ***DO*** TakeOre_Nearest)
Prohibition (***IF*** Message_number <MAX]***WHEN*** $ ***DO*** RespondMessage)

We have implemented the BDIP robots in a simulation way by using the Jadex platform [10]. The basic actions of agents are implemented as *Plan* of the agent. Some important information such as ore resource positions are implemented as *Belief*, the interaction among agents this case is implemented as the message events of the agent. The *Intention* of the robots are to find the ore resource and carry ore back to base. On the other hand, the *policy conditions* are implanted as either a part of *belief* (e.g. ore resource) or implemented as the message events. When the system is running, if the *policy conditions* are satisfied, the agent will perform its behaviors comply with these policies, the trigger condition of the plans are implemented same as the *policy conditions*.



(a) Scenario 1 (b) Scenario 2

Fig. 3. Running snapshot of the Case

Fig. 3 shows the running snap shot of the case. The gray transparent pane is the base. The number below the base is the amount of ore in the base which has been carried by robots. The yellow transparent cycle of a robot presents the scope that the robot can explore for ore. The red points in environment mean the ore resources which have not been found by robots. The gray points mean the ore resource which has been found by robots and the number below the gray points mean the remained reserves of the ore resource. Fig. 3(a) shows the first scenario in which only one ore resource in the environment. From the figure we can see all robots are carrying ore from this resource when one of them has found it. Fig. 3 (b) shows the second scenario which has many ore resources in the environment. In Fig. 3(b), the robots have collected enough positions of the ore resources and carry the ore from the resource which is the nearest to the base among the collected positions.

6 Related Works

Recently, many research efforts have been made on the agent modeling based on norms and policies. However majority of them are engaged in the design of norm-based MAS organization, which propose norms to have an effective influence on agent and agent role, for example organization regimentation [11] and enforcement mechanisms [13][14]. In the agent architecture aspect, the main contributions of norm

acceptance of agents are focus on the theory of BDI agent, e.g. [3][9]. These works focus on the theory frameworks e.g. logical expressions to explain how to represent the norms or policies in the agent and how to influence the reasoning process of the agent. The implement of architecture and running mechanisms of BDI or norm-based BDI agent are relatively few. [12] proposes a multi-level agent architecture, in which norms can communicated, adopted and used as meta-goals. [8] proposes a BDI Interpreter architecture for the running mechanisms of BDI agent, which our work can be seen as an extension of it.

7 Conclusion and Future works

In the development of self-organizing MAS, the great challenge is how to bridge the huge gap between the agent local interaction behaviors and the system macro emergence characteristics. Against this open issue, this paper proposes a BDIP architecture of software agent. In the BDIP architecture, agent can perceive the policies of the system and adjusts its behaviors according to policies. A flexible running mechanism is also proposed based on the architecture, which executing meta-operations from a dynamic agenda structure. The architecture and its running mechanism are flexible enough to support the adjustments of system policies so that the users could control the self-organizing process and result by changed policies in the system. Moreover, the details of policy-based decision algorithms of agents are designed, and a case study of policy-based self-organizing robot system is implemented in a simulation way with the Jadex platform. Through the case study we can see that the changes of policies of the system can alter the self-organizing emergence result, and users can satisfy different system requirements by adjusting the policies.

In future works we still focus on the development of the policy-based self-organizing multi-agent systems. A PSOMAS developing environment named PSOMASDE is in our current works. The developing environment include agent design platform and running environment, as well as the policies can be represented by the XML files that can be loaded by the system. Besides the implementation of such systems, the development methodology is also in consideration, which is based our previous work named ODAM [15].

Acknowledgement

The research acknowledges financial support from Natural Science Foundation of China under granted No 61070034, Program for New Century Excellent Talents in University, and Opening Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University.

References:

1. Giovanna D.M.S., Marie P. G. and Anthony K.: Self-Organizsation in MAS. Technology

report, AgentLink III Technical Forum Group (2005).
2. Mamei M., Ronaldo M. and Zambonelli,F.: Case Studies for Self-Organization in Computer Science. Journal of Systems Architecture. 52(8), 443-460, (2006)
3. D. Frank, K. David, and S. Liz. :Motivational Attitudes of Agents: On Desires, Obligations and Norms.: In: Barbara D. K. and Edward N.(edt.) LNCS, vol. 2296, pp, 61-70. Springer, Heidelberg (2001)
4. Yi Guo, Xinjun Mao, Cuiyun Hu.: A Survey of Engineering for Self-Organization Systems. In: 23th Software Engineering & Knowledge Engineering, pp. 527-531. Knowledge Systems Institute Press. USA. (2011)
5. Parunak, H. V. and Sevn A. B.: Software Engineering for Self-Organizing Systems. In: 12th International Workshop on Agent-Oriented Software engineering. AAMAS2011 (2011).
6. Yi Guo, Xinjun Mao, Cuiyun Hu.: Design Pattern for Self-Organization Multi-agent Systems based on Policy. In: 6th International conference on Frontier of Computer Science and Technology. pp. 1572-1577. IEEE Press USA(2011)
7. A. S. Rao, M. P. Georgeff.: Modeling Rational Agents within a BDI-Architecture. In: 2nd International Conference on Principles of Knowledge Representation and Reasoning. pp. 473-484. Kaufmann Press, USA(1991)
8. A. Pokahr, L. Braubach, and W. Lamersdorf.: A Flexible BDI Architecture Supporting Extensibility. In: 2005 IEEE/ WIC/ ACM International Conference on Intelligent Agent Technology. pp. 379-385.IEEE Press. USA(2005)
9. N. Crida, E. Argente, P. Noriega, and V. Botti.: Towards a Normative BDI Architecture for Norm Compliance. In: 2010 Multi-agent Logics, Languages, and Organisations Federated Worshops pp, 65-81. (2010)
10. L. Braubach, A.Pokahr, and W. Lamersdorf.: Jadex: A BDI Agent System Combining Middleware and Reasoning. In: Rainer U., Matthias K., Monique C.,(eds.) Software Agent-Based Applications, Platforms and Development Kits. pp.143-168. Birkhauser Press(2005)
11. N. Criado, E. Argente, and V. Botti.: Thomas: An Agent Platform for Supporting Normative Multi-agent Systems. Journal of Logic and Computation. doi: 10.1093/logcom/exr025, (2011).
12. C. Castelfranchi, D. Frank, M.J. Catholijin, and T. Jan. Deliberative Normative Agents: Principles and Architecture. In: Jennings and Y. Lesperance, (eds.) Intelligent Agents VI, pp. 364-378. Springer Heidelberg,(2000)
13. S. Modgil, N. Faci, F. Meneguzzi, N. Oren. S. Miles, and M. Luck.: A Framework for Monitoring Agent-based Normative Systems.In: 8th International Conference on Autonomous Agents and Multi-agent System. pp. 153-160. ACM Press USA(2009)
14. D. Grossi, H. Aldewereld, F. Dignum.: Designing Norm Enforcement in E-Institutions. In:Pablo N., Javier V. S. Guido B. etl.(eds.) Coordination, organizations, institutions, and norms in agent systems II. LNCS, vol. 4386, pp, 107-120. Springer, Heidelberg (2006)
15. Xinjun Mao, Cuiyun Hu, and Ji Wang: An Organization-based Approach to Developing Self-Adaptive Multi-Agent Systems. International Transactions on Systems, Science and Applications. 5(4), 297-317.(2009)