ORIGINAL RESEARCH

A comparison of organization-centered and agent-centered multi-agent systems

Andreas Schmidt Jensen, Jørgen Villadsen

Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kongens Lyngby, Denmark

Correspondence: Jørgen Villadsen. Address: Department of Applied Mathematics and Computer Science, Technical University of Denmark, Matematiktorvet, Building 303B, DK-2800 Kongens Lyngby, Denmark. Email: jovi@dtu.dk.

Received: December 2, 2012 **Accepted:** March 24, 2013 **Online Published:** April 23, 2013
DOI: 10.5430/air.v2n3p59 **URL:** http://dx.doi.org/10.5430/air.v2n3p59

Abstract

Whereas most classical multi-agent systems have the agent in center, there has recently been a development towards focusing more on the organization of the system, thereby allowing the designer to focus on what the system goals are, without considering how the goals should be fulfilled.

We have developed and evaluated two teams of agents for a variant of the well-known Bomberman computer game. One team is based on the basic Jason system, which is an implementation in Java of an extension of the logic-based agent-oriented programming language AgentSpeak. The other team is based on the organizational model Moise+, which is combined with Jason in the middleware called J-Moise+.

We have investigated whether taking the organization-oriented approach had any clear advantages to the classical way of implementing multi-agent systems. Although not decisive the investigation did indicate that the agent-oriented approach has a number of advantages when it comes to game-like scenarios with just a few different character types.

Key words

Multi-Agent Systems, Agent-Oriented Programming, Organizations

1 Introduction

Within the area of multi-agent systems there has recently been a development towards making the organization of such systems explicit [3-5]. However, while drawbacks of classical (agent-centered) multi-agent systems have been listed [3], the actual advantages of making the organization explicit have not been thoroughly investigated and the approaches compared.

This paper summarizes our work with such investigation of the organization of multi-agent systems. The investigation was conducted by implementing two systems: a classical agent-centered multi-agent system (ACMAS) and an organization-centered multi-agent system (OCMAS).

In an ACMAS the agent is in focus whereas in an OCMAS we are more concerned with the organization; i.e. the structure of the multi-agent system. Naturally all multi-agent systems have a structure, but it is most often implicitly defined by the agents and their relations. By explicitly defining the organization it is possible to focus on what the agents should do without at the same time deciding how they should do so. In other words, the organization makes it possible to create the structure of the system without specifying details about the implementation.

The two types of systems have been compared using a team-based version of the well-known game Bomberman. Since the two approaches are quite different in many ways we base our comparison of the ACMAS and OCMAS on the following measures:

- Structure of the source code
- Development speed
- Performance
- Error handling
- Debugging
- Complexity of the scenario
- Number of intelligent agents

The original Bomberman game consists of five key elements: bombs, boxes, solid obstacles, exitways and power-up panels. Whereas boxes are destructible, solid obstacles are not. This means that Bomberman will always be able to take cover behind solid obstacles. Most boxes must be destroyed since they hide both power-ups and exitways. Exitways are what Bomberman must find to be able to complete a level. A power-up can be used to enhance Bomberman's abilities and bombs. In the beginning, his bombs are weak, but by using power-ups he will be able to drop several stronger bombs at a time.

We want to be able to experiment with the cooperative aspects of intelligent agents so we propose an altered version of Bomberman in which two teams attempt to eliminate each other:

Team-based Bomberman

The multi-agent system is similar to the Bomberman game. It consists of two teams fighting against each other. Each team consists of at least two "bombermen" (or agents). The teams are situated in a maze-like environment consisting of solid obstacles and boxes. An agent can place bombs which at some point will explode. An agent dies when it is hit by an explosion. Explosions will also destroy boxes. A team wins when all players from the other team have been eliminated.

This version of Bomberman consists of some of the same key elements as the original game: a maze, destructible and indestructible obstacles, and bombs. This should allow the agents to employ the strategies intended for the game, while at the same time competing in teams.

Exitways have been removed, since the overall goal of "getting to the surface" is not relevant. Power-ups are not included to avoid making the overall system too complex.

Still the team-based version of Bomberman is quite complicated and for this reason we have chosen to consider just two implementations using the Jason and Moise+ platforms, rather than implementing a series of simpler systems on a larger set of platforms like in the extensive comparison of agent-based simulation platforms [10].

Jason

The implementation of the ACMAS is done using Jason. We provide just a very brief overview of the interpreter, however, we will not go into details. A thorough description of Jason is available [2].

The language of Jason, AgentSpeak, is a Prolog-like logic programming language. AgentSpeak allows the developer to create a plan library for the agent. A plan in AgentSpeak is basically of the following form:

```
+triggering_event : context <- body.
```

Roughly speaking, if an event matches a trigger, the context is matched with the current state of the agent. If the context matches the current state, the body is executed; otherwise the engine continues to match contexts of plans with the same trigger. If no plan is applicable, the event fails.

The plans are matched using a top-down approach; the first plan which can be matched with the current state is chosen, so the different orderings can result in different agents.

Moise+

The implementation of the OCMAS is based on the Moise+ organizational model [5, 7, 8], in which it is possible to create a structural, functional and deontic specification of an organization. The organizational model has been combined with Jason in the middleware called J-Moise+ [6].

Moise+ is an organizational model for multi-agent systems which makes it possible to specify the organization in a MAS structurally, functionally and deontically. The model takes an organizations-centered approach, meaning that an organization will exist a priori (created at design-time) and the agents ought to follow it [5].

Structural Specification: Moise+ uses the concepts of roles, role relations and groups in the structural specification of an organization. Each agent plays one or more roles. The roles are related by links, which specify how agents are acquainted and can communicate. In order to further structure the organization, the agents can join different groups depending on the roles they play.

Functional Specification: The functional specification consists of a goal decomposition tree, known as a Social Scheme (SCH), where the root is the goal of the SCH and each node is a sub-goal that can be delegated to different agents. Three operators are defined for decomposing a goal into sub-goals: sequence, choice and parallelism [5]. These operators allow us to create complex schemes in which the agents can commit to advanced missions.

Deontic Specification: The relation between the structural and functional specification is made explicit by the deontic specification. Using it, we can constrain the agents further by specifying what missions an agent ought to follow and what missions an agent is allowed to follow when playing certain roles.

2 Design and implementation

We provide a brief description of some of the details of the design and implementation of the agent-centered and organization-centered versions of the Bomberman scenario. In section 3 we discuss the issues encountered during these phases and compare the approaches.

2.1 ACMAS

We use Prometheus methodology [9] as a guideline for designing the ACMAS team. The first step is to create a system specification in which we identify the goals of the system. Figure 1 shows the initial goals and how they relate (by subgoal relation). We write “goal”, when the sub-goals of that goal can be pursued in parallel, “*goal*”, when just one of the sub-goals need to be achieved and simply “goal”, when the sub-goals must be completed in a sequence. For example, while the agent can choose how to avoid getting hit by a bomb, it will need to first find an enemy and then move, to be able to get to that enemy.

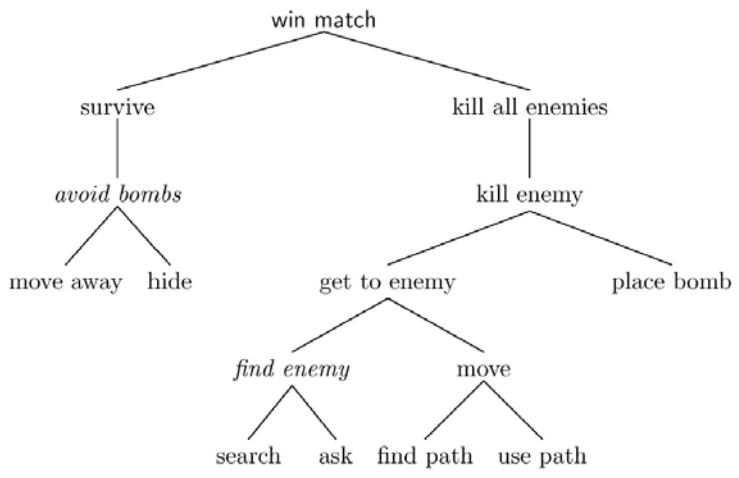

Figure 1. Initial goal specification

Functionality

The goals can be grouped into functionalities which show how behaviors of the system can be achieved. It is a combination of goals, percepts and actions which are relevant to that behavior. Figure 2 gives an overview of the functionality of this system. Rectangles are functionalities, circles are goals, stars are percepts and rectangles extended with a triangle are actions.

For example, in order to kill an enemy the agent must conduct the behavior of the functionality “kill enemy”. This means perceiving an enemy, moving to that enemy, placing a bomb and avoiding getting killed by the bomb (part of the “surviving” goal).

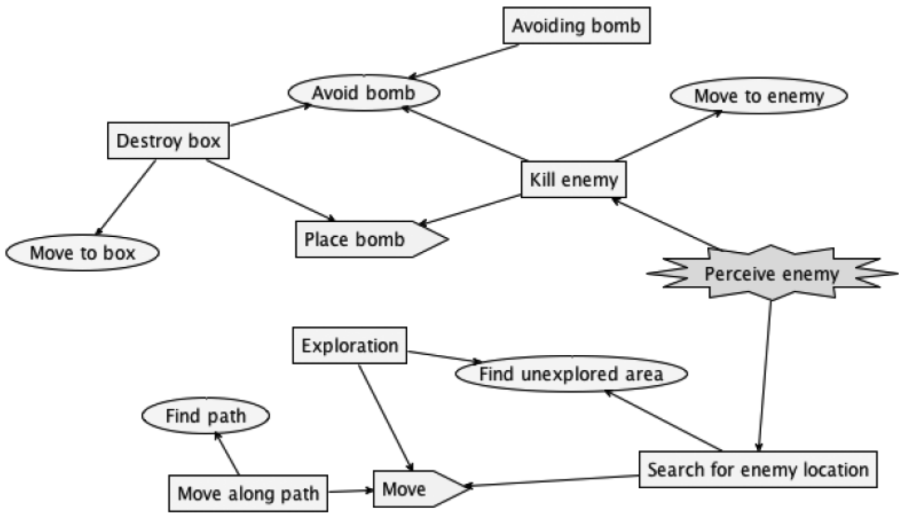

Figure 2. Functionalities for the system

Using the identified goals, functionality, percepts and actions, it is possible to identify a set of possible scenarios of the system, i.e. things that may happen during a run of the system. From the game definition we can deduce the following main scenarios:

- Position perceived
- Enemy perceived
- Ally perceived
- Bomb perceived
- Explosion perceived
- Obstacle perceived
- Target acquired
- The agent is out of bombs
- The agent is dead

The scenarios can then in turn be implemented in Jason, by creating plans triggered by certain events, such as committing to goals or perceiving certain objects in the environment.

Implementation

Plans in Jason are quite simple as seen in the example below. The code describes how the agent should move to a target location, when it is possible to take shortcuts by blowing up some boxes. The shortcuts are taken by means of intermediate targets, which are boxes that should be removed before the shortcut is clear.

```
+!move(X,Y)
  :   pos(AgX,AgY) &
      .get_intermediate_target(AgX,AgY,AgX,AgY) &
      bombs(N) & N > 0
  <-  do(bomb).
+!move(X,Y)
  :   pos(AgX,AgY) &
      .get_intermediate_target(AgX,AgY,AgX,AgY)
  <-  do(skip).
+!move(X,Y)
  :   pos(AgX,AgY) &
      .get_intermediate_target(AgX,AgY,TX,TY)
  <-  .get_path(AgX,AgY,TX,TY,Act);
      do(Act).
+!move(X,Y)
  :   pos(AgX,AgY)
  <-  .get_path(AgX,AgY,X,Y,Act);
      do(Act).
```

The implementation of the move goal consists of four plans. The first three have to do with intermediate targets (taking a shortcut), while the last simply decides the next step towards the intended location. In plan 1 the agent is at the intermediate target and has at least one bomb, which allows it to remove the box at this location – do(bomb). Plan 2 handles a similar situation, where the agent does not have an available bomb. It therefore chooses to wait another cycle, such that a bomb may become available – do(skip). Plan 3 considers the situation where the agent is not at the intermediate target, so it decides the next step towards this location. Finally, the last plan considers the situation where no intermediate target exists, so the agent decides the next step to take towards the target location.

Note the importance of the ordering of the plans; had the last plan been at the top, the agent would never consider intermediate targets, since it would always be at a location (pos(AgX,AgY)) and be able to decide the next step towards another location (.get_path(...)). Furthermore, if plan 2 was considered before plan 1, the agent would always skip at an intermediate target, since plan 2 can be matched whenever plan 1 can (but not vice versa).

2.2 OCMAS

We use the Gaia methodology [12] as a guideline for designing the organization. It is a methodology for designing multi-agent systems, however an agent-oriented one. A part of it considers the organization of the system by defining roles, their permissions, obligations and interactions. Though Gaia takes a static approach to the organization, deciding upon what roles the system consists of will also apply to the dynamic approach of Moise+.

Gaia will primarily be used for the part of the structural specification concerning roles and interaction between roles. By identifying the groups they can then be related to the roles so that the organization is usable in Moise+. Finally the functional aspect of the system should be identified and related to the structural specification using deontic relations.

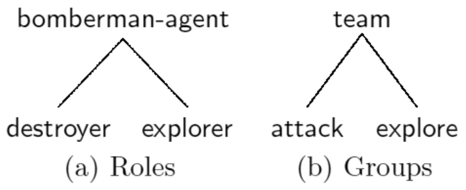

Figure 3. Roles and groups

Structural Specification

The structural specification defines the available roles and groups, and relations between these. We have defined an abstract role, the bomberman-agent. All other roles inherit this role. Similarly to the types of agents in the ACMAS, there are two basic roles: explorer and destroyer. Figure 3(a) shows how the roles are related.

The design of roles and groups is quite simple. Figure 3(b) shows the groups and their subgroup relations. We considered letting the attack group have two sub-groups: one for agents knowing the location of an enemy, and one with no such knowledge. This would enable the agents to focus on either attacking the enemy or searching for it. However as we describe below, the deontic specification makes this distinction possible within a group.

Functional Specification

The functional specification defines a set of goals and missions the agents can commit to. Figure 4 gives an overview of the goals of the organization shown as decomposition trees or SCH's.

The overall goal is to win the match. This is done by killing all enemies. Killing an enemy is done by first finding an enemy and then killing it. A combination of the schemes (b) and (c) thus leads to the following decomposition for finding and killing an enemy (where a vertical bar (|) denotes or, and a comma (,) denotes and):

Eliminate enemy = (Explore | Receive location), Move to enemy, Attack.

The structural and functional specifications are related by roles and missions. Therefore the goals of the system must be organized in missions that agents can commit to. This will allow the agents to fulfill a set of goals by completing a mission. The following three missions cover all of the goals of the social schemes in figure 4:

explore = {explore map, find unexplored area, move to unexplored area}

find enemy = {find enemy, explore, receive location}

kill enemy = {move to enemy, attack, place bomb, avoid bomb}

For example, when an agent commits to the mission explore it will automatically commit to the goals of finding unexplored areas and moving to those areas.

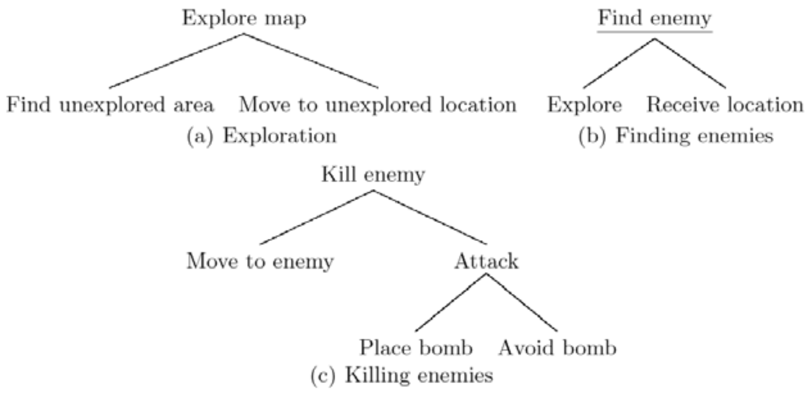

Figure 4. Goal Decomposition Tree or SCH's for (a) exploring the map, (b) finding enemies and (c) killing an enemy. An underlined node indicates a choice between the child nodes.

Deontic Relationship

The final step is to relate the structural and functional specifications by obligations and permissions, i.e. a deontic relationship. The idea is to force agents to commit to certain missions when they choose to play a role. As described above, the relation is quite clear. However, it remains to be explicitly defined exactly what obligations and permissions a role has.

The fact that roles inherit a deontic relation makes it much easier to decide on how a role is related to the missions. Using obl and per for obligations and permissions, respectively, we propose the following deontic relations between roles and missions:

obl(explorer, explore, Any)

obl(destroyer, kill enemy, Any)

per(destroyer, find enemy, Any)

Having these deontic relations means that an agent playing the explorer role must commit to the exploration mission. This ensures that the team will always have an agent committed to exploring the environment. We could also permit the explorer to commit to the kill enemy mission, which makes the OCMAS very flexible; several agents can commit to the same missions just by creating a deontic relation (of course, the agent should be capable of completing the mission for it to be a success).

The constraints of the destroyer role make it possible to distinguish between the two deontic constraints. The agent is allowed to search for enemies and will do so, when no enemy location is known. This distinction is more elegant than having subgroups since it allows an agent to stay in a group even though its mission has changed.

Committing to a mission in J-Moise+

When the agents commit to a mission in a scheme the J-Moise+ engine will generate goal achievement events for the goals that are currently available. For instance, when an explorer commits itself to the mission of exploration, it will automatically generate the goal achievement event of finding an unexplored area. Whenever a goal is completed, an event for the next goal of the plan is generated. In this case the next available goal will be to move to the unexplored area. This allows us to specify a very simple implementation for the explorer:

```
+!exploreMap[scheme(Sch)]
  <-  jmoise.set_goal_state(Sch, exploreMap, satisfied).
+!findUnexploredArea[scheme(Sch)]
  :   <context>
  <-  <plan to find unexplored area>;
      jmoise.set_goal_state(Sch, findUnexploredArea, satisfied).
+!moveToUnexploredArea
  :   <context>
  <-  <plan to move to unexplored area>.
+near(_,_)
  <-  ?scheme(exploration, Sch);
      jmoise.set_goal_state(Sch, moveToUnexploredArea, satisfied).
```

The organizational specification shows exactly what general steps are required to explore the map, so it is only necessary to create plans for each goal event. When the plan is successfully executed, the agent informs J-Moise+ that the goal has been satisfied. It is then the responsibility of J-Moise+ to generate the next goal event.

Notice that the goal state of !moveToUnexploredArea is not satisfied when the plan has been executed, since the agent may not be able to move to the unexplored area in a single step. Instead it is possible to let the agent react to percepts, such as near, which tells the agent that it has moved to the desired area, so that the goal is satisfied at that state instead.

3 Experiments and results

We now present the main results of our experiments. Naturally, the results of our experiments will be biased by the fact that it was conducted using a single tool for each of the two paradigms. However since the design of the systems are independent of the tool used for implementation it is still relevant to discuss pros and cons of the two paradigms.

3.1 Jason

The ACMAS is built from the ground. Having to build everything from the ground gives a lot of freedom with regards to the structure of the implementation; there are no constraints as to where specific details must be implemented. This has led to a solution where plans for achieving sub-goals and reacting to percepts can be implemented concisely, while still doing as intended.

The resulting agents are therefore reacting quite fast to changes in the environment; with short code and only few precisely defined responsibilities, the agents are easily able to prioritize during a game, if it, for instance, is necessary to take cover from a bomb. But the freedom one has with regards to structure has also been the biggest issue during the implementation. Ensuring successful transition between goals has caused some trouble during the implementation. The debugging functionality can be quite tricky to master, and the only way to test the transition from one goal to another is by executing the system. This can render the process quite slow.

3.2 Moise+

The relation between goals is handled by Moise+, which makes the Jason code quite concise and clear. The code is often quite verbose, because of the statements required to setup and manage the organizational structure (by use of actions such as jmoise.create_group(...)) and the extensive use of annotations (+!goal[scheme(Sch)] etc.). While this in general makes the code quite clear, it also can result in situations where one need to include a plan for a goal event in which the goal is simply set to be satisfied.

We saw that agents enacting roles with only few responsibilities risk waiting after achieving a sub-goal until the special J-Moise+ agent determines the next goal to pursue. For example, when the exploreMap goal is available, it is immediately satisfied. When this happens, the scheme for exploring is finished and a new scheme must be created in order to continue the exploration. Compared to the explorer of the ACMAS, the performance differences are quite clear; in the ACMAS, the agent immediately chooses a new spot to move to, while in the OCMAS, the agent waits for the generation of an appropriate goal event.

Organizational knowledge

When a group or scheme is created, the agents will perceive certain events so that they are able to react accordingly. In order to be able to distinguish between similar events, annotations are added that among other things include which agent created the organizational object.

This can be used to let an agent decide not to join a group if a specific agent has created it, or only committing to a mission it is permitted to commit to, if it is related to a specific group. This is a great use of the Jason annotations, as it is perfectly clear how to use them. Furthermore, because it is annotations they will not be shown if the programmer chooses not to use them.

However the agent does not know whether it is allowed to join a group before it attempts to join it. This means that if it is not permitted to join a group or play a role and it attempts to do so, an error event is created. The agent cannot reason in detail about the error so it will not know why it could not join the group.

3.3 Agent-Centered Multi-Agent Systems

There are well-defined methodologies for designing an ACMAS such as the Prometheus methodology which we used as part of our design phase. This makes it possible to design the general structure of the system before implementing it, by specifying possible goals, functionality and scenarios. Note that it is not required to use an existing methodology when building a multi-agent system, but our experience is that by identifying certain crucial parts of the system early, the implementation will be more stable.

Even though the Prometheus methodology helps in identifying agent types and setting up interaction protocols (we have not described these issues in this paper), the agents are still free to act in any way they want. There is no expectation of how the agents act, or rather, the developer expects the agents to act in certain ways, but has only the implementation as a tool for ensuring this. An agent could intent to bring about a certain state of affairs which was not expected, and even though it might be possible to help the agent back on the right track, measures for doing so are not well-defined.

When designing and implementing an ACMAS one has full control over everything (up to the constraints of the choice of implementation). This means that certain issues may be dealt with easier. For instance, consider an agent which can save another agent by moving it away from a heavily trafficked road. If the agents are constrained by an organization that dictates that only agents enacting the role of “rescuers” may move other agents, the agent will have to enter the organization and choose to enact this role before being able to save the other agent. If the agent is not constrained it can immediately move the other agent away from the road.

3.4 Organization-Centered Multi-Agent Systems

Building an OCMAS is a more well-structured process than that of building an ACMAS, since it consists of two well-defined parts, (1) specifying organization and (2) implementing the details of the organization, where the latter depends on the completion of the first. While this does not automatically result in a more structured program, it does force the user to think more about what, why and how. When specifying the organization the focus is on what the overall goals are. This leads to considering why these are the goals and in that way it allows us to justify the choices made, even before they are implemented. When the plans are implemented the focus is on how the agents are supposed to complete their goals.

By specifying role relations and constraints in an organizational structure rather than in the actual implementation, issues such as interaction between agents and cooperation is dealt with more easily, since the expected behavior of the agents is presented in a well-defined manner. The developer responsible for implementing the agents enacting the roles in the organization will then enable the agents to reason about their obligations so that they conform to the organizational expectations.

In this setting we have not considered agents violating their obligations, since the system is a closed system with a predetermined overall goal and a known number of agents. In open societies, where agents can be implemented in different systems by different people, this is an issue that should of course be dealt with.

4 Conclusion

We have investigated the so-called agent-centered and organization-centered approaches to designing and implementing multi-agent systems. By leaning on existing methodologies for designing the systems we have been able to determine how structured the approaches are and what constraints are put on the developer.

In an agent-centered multi-agent system, the agents are generally not constrained and can as such easily act according to the expectation of (implicit) roles without having to join an organization and decide which roles to enact. On one hand this makes the agents able to quickly adapt to environmental changes that would otherwise require permission from an organization, while it on the other hand complicates the system because the agents can act unexpectedly.

The design phase of an organization-centered multi-agent system is well-defined, since there are exact requirements for specifying the organizational structure, function and obligations. Therefore, expected behavior is more easily specified in the organizational approach. It is defined using role obligations and permissions and allows the designer to specify bounds for the agents.

By taking both the agent-centered and organization-centered view on the same strategy we have gained insights about both the advantages and disadvantages of each approach. The focus has been on a single scenario, which means that not all corners of the approaches have been investigated. Even so, the results have made several differences of the approaches clear, differences that in some situations make one approach highly advantageous compared to the other.

We have chosen to consider just the Jason and Moise+ platforms and a quite complicated team-based version of Bomberman. Besides other scenarios it would be interesting to consider other organization-based models like the Agent/Group/Role (AGR) or Holonic approaches.

References

[1] Tristan M. Behrens, Jürgen Dix, Jomi Hübner, and Michael Köster. Multi Agent Programming Contest. http://www.multiagentcontest.org/, June 2012.

[2] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons Ltd, 2007.

[3] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From Agents to Organizations: an Organizational View of Multi-Agent Systems. Agent-Oriented Software Engineering (AOSE) IV. 2004: 214–230.

[4] Mahdi Hannoun, Olivier Boissier, Jaime Simão Sichman, and Claudette Sayettat. Moise+: An Organizational Model for Multi-agent Systems. Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI, 2000.

[5] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. Proceedings of the 16th Brazilian Symposium on Artificial Intelligence, 2002.

[6] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. S-Moise+: A Middleware for developing Organised Multi-Agent Systems. Proceedings of the International Workshop on Organizations in Multi-Agent Systems, from Organizations to Organization Oriented Programming in MAS, 2005.

[7] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Developing Organised Multi-Agent Systems Using the Moise+ Model: Programming Issues at the System and Agent Levels. International Journal of Agent-Oriented Software Engineering, 2007.

[8] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Moise+ tutorial. Available from: http://moise.sourceforge.net/, 2008.

[9] Lin Padgham and Michael Winikoff. Developing Intelligent Agent Systems. John Wiley & Sons Ltd, 2004. http://dx.doi.org/10.1002/0470861223

[10] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson. Agent-based Simulation Platforms: Review and Development Recommendations. SIMULATION 82: 609. Available from: http://sim.sagepub.com/content/82/9/609, 2006. http://dx.doi.org/10.1177/0037549706073695

[11] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, 2nd edition, 2003.

[12] Michael Wooldridge, Nicholas R. Jennings and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems. 2000: 285-312.

[13] Michael Wooldridge. An Introduction to Multi Agent Systems. John Wiley & Sons Ltd, 2nd edition, 2009.