Representing Agents and their Systems: A Challenge for Current Modeling Languages

Renato Levy
Intelligent Automation, Inc.
7519 Standish Place, Suite 200, Rockville, MD 20855 USA
rlevy@i-a-i.com

James Odell
James Odell Associates
3646 W. Huron River Drive
Ann Arbor, MI 48103 USA
email@jamesodell.com

Keywords: Systemic, Agents, Multi-agent systems, UML, AUML, model, modeling languages, modeling notation

Received: July 15, 2003

Leading-edge organizations are now developing systems that employ autonomous, interactive entities, or agents. [1; 2] Compared to its predecessors, the agent-based approach is evolutionary. However, its usages could be revolutionary. This paper begins by presenting some of the differences and similarities between agents and previous approaches. We then discuss some of the challenges for using current modeling approaches to represent agent-based systems. Our position is two folded: many of the evolutionary aspects of agent modeling can accomplished by extending current modeling languages such as UML 2.0; while the revolutionary aspects, however, will probably require new approaches.

1 Introduction

Advances on technology and on system's theory (non-linearity, complexity and chaos theory) has led to engineers to challenge problems which had been deemed intractable for a number of years. These problems are usually NP-hard in high order, which makes even the development of efficient heuristics a very complex challenge. Observation of how nature deals with problems of such complexity led to a different approach to software development, known as agent-based software, which has been successful in developing solutions for such problems. The agent-based software paradigm has established itself as viable approach for developing software directed towards control and simulation of complex systems.

Figure 1 illustrates one way of thinking about the evolution of programming paradigms. Originally, the basic unit of software was the complete program where the programmer had full control. The program's state was the responsibility of the programmer and its invocation determined by the system operator. The term modular did not apply because the behavior could not be invoked as a reusable unit in a variety of circumstances.

As programs became more complex and memory space became larger, programmers needed to introduce some degree of organization to their code. The modular programming approach employed smaller units of code that could be reused under a variety of situations. Here, structured loops and subroutines were designed to have a high degree of local integrity. While each subroutine's code was encapsulated, its state was determined by

	Monolithic Programming	Modular Programming	Object-Oriented Programming	Agent Programming
Unit Behavior	Nonmodular	Modular	Modular	Modular
Unit State	External	External	Internal	Internal
Unit Invocation	External	External (CALLed)	External (message)	Internal (rules, goals)

Figure 1: Evolution of programming approaches [3].

externally supplied arguments and it gained control only when invoked externally by a CALL statement. This was the era of procedures as the primary unit of decomposition.

In contrast, object orientation added to the modular approach by maintaining its segments of code (or methods) as well as by gaining local control over the variables manipulated by its methods. However in traditional OO, objects are considered passive because their methods are invoked only when some external entity sends them a message.

Software agents have their own logical thread of control, localizing not only code and state but their invocation as well. Such agents can also have individual rules and goals, making them appear like "active objects

with initiative." In other words, when and how an agent acts is determined by the agent.

At each evolutionary step, then, various modeling languages were created to aid system developers. The latest and most popular graphical language is the Unified Modeling Language (UML) developed by the Object Management Group (OMG). As agent based systems starts their transition from university and research labs into mainstream engineering, grows the necessity for appropriate graphical languages and tools to support it. Since agent technology can be viewed as an evolution on previous technologies, it would be reasonable to believe that agent-based languages can be based on previous approaches — at least in part. However, the way in which agents can be used for application systems is far richer than earlier approaches. Here, we may also need to develop new languages to accommodate the agent-based approach, in addition to adopting and modifying pre-agent languages.

The rest of this paper is organized as follows: In section 2 we present the philosophical differences between agent systems and their predecessor software engineering paradigms. Section 3 demonstrates how these philosophical differences impact our ability to represent such systems in current modeling languages, and specifically in UML. In section 4, we proposed a set of alternative representations that are able to solve some of the previous modeling limitations and in section 5 we present a study case in which some of the challenges and proposed solutions are debated. Section 6 concludes this paper with an invitation for an open debate about the issues raised.

2 Philosophical Differences

Agents are commonly regarded as autonomous entities, because they can watch out for their own set of internal responsibilities. Furthermore, agents are interactive entities that are capable of using rich forms of messages. These messages can support method invocation—as well as informing the agents of particular events, asking something of the agent, or receiving a response to an earlier query. Lastly, because agents are autonomous they can initiate interaction and respond to a message in any way they choose. In other words, agents can be thought of as objects that can say "No"—as well as "Go." Due to the interactive and autonomous nature of agents, little or no iteration is required to physically launch an application. Van Parunak summarizes it well: "In the ultimate agent vision, the application developer simply identifies the agents desired in the final application, and the agents organize themselves to perform the required functionality." [3] No centralized thread or top-down organization is necessary since agent systems can organize themselves.

However, several other key areas exist that differentiate the agent-based approach from traditional approaches such as OO. The list below describes some underlying concepts that agent-based systems can employ. None are universally used by agents: active object systems may use them as well. Furthermore, no agent system is required to use all of them. This list merely provides a "menu" of features that agent systems can —and often do — employ.

Decentralization: Objects can be thought of as centrally organized, because an object's methods are invoked under the control of other components in the system. Yet, some situations require techniques that are decentralized and self-organized. For example, classical ballet requires a high degree of centralization called choreography, while at the other extreme the processes of nature involve a high degree of individual direction. However, most businesses require a balance of standardized procedures and individual initiative: one extreme or the other would be detrimental to the business.

Supply-chain systems can be planned and centrally organized when the business is basically stable and predictable. In unstable and unpredictable environments, supply chains should be decentralized and self-organized (an option not supported by commercial supply-chain systems today). Agent-based environments can employ both centralized and decentralized processing. While agents can certainly support centralized systems, they can also provide us with the ultimate in distributed computing.

Multiple and dynamic classification: In OO languages, objects are created by a class and, once created, may never change their class or become instances of multiple classes (except by inheritance). Agents can provide a more flexible approach. For example, a particular agent can be a person, employee, spouse, landowner, customer, and seller all at the same time or at different times. When the agent is an employee, that agent has all the state and procedural elements consistent with being an employee. If the agent is terminated from his or her job, the employment-related state and procedural elements are now longer available to the agent. Whether employed or not, the agent is still the same entity—it just has a different set of features. The ability to express roles and role changes is not new to OO. However, most OO languages do not directly support this mechanism (even though UML does).

Furthermore, agents might play different roles in different domains. When you go to work, you play the employee role. When you return home, you change roles—for example, playing the spouse role. OO languages do not directly support such domain-dependent mechanisms that are necessary for agent-based environments. The single-class OO approach is efficient and reliable; the multiple and dynamic approach provides flexibility and more closely models our perception of the world. Agents can use either approach; the choice belongs to the system designer.

Instance-level features: The features possessed by each object are defined by the object's class—a benefit enjoyed by agents as well. However, each agent may also acquire or modify its own features, i.e., features that are not defined at the class level, but at the individual agent (or instance) level. In other words, if an individual agent has the ability to learn, it can change its own behavior— permitting it to act differently that any other

agent. If an agent can change itself, it can add (as well as subtract) features dynamically. For example, with genetic programming software, agents are created genetically. Here, each parent contributes some portion of an offspring agent's genetic string—much in the same way that occurs in nature. This approach is particularly popular in one area of agent-based systems known as artificial life. (Artificial life is the study of man-made systems that exhibit the behavioral characteristic of natural living systems. It models life-as-we-know-it within the larger picture of life-as-it-should-be.)

Emergence: The interaction of many individual agents can give rise to secondary effects where groups of agents behave as a single entity. For example, ant colonies, flocks of birds, and stock markets have emergent qualities. Each consists of individual agents acting according to their own rules and even cooperating to some extent. Yet, ants colonies thrive, birds flock, and markets achieve global allocations of resources—all without a central cause or an overall plan. Agents can possess just a few very simple rules to produce emergence. In fact, when constructing agent-based systems, starting out with simple agents is important, because emergence is then easier to understand and harness. More complexity can be added over time to avoid being overwhelmed.

Since traditional objects do not interact without a higher level thread of control, emergence does not usually occur. As more agents become decentralized, their interaction is subject to emergence—either positive or negative. This phenomenon is both the good news and bad news for large multiagent systems.

Analogies from nature: The autonomous and interactive character of agents more closely resembles natural systems than do objects. Since nature has long been very successful, identifying analogous situations to use in agent-based systems is sensible. For example, agents can die when they lack supportive resources. In supply-chain manufacturing, when a manufacturing-cell agent cannot operate profitably, it dies of "malnutrition." Furthermore, another manufacturing cell could come by and scavenge useful bits from the newly dead cell.

Agents can exhibit properties of parasitism, symbiosis, and mimicry. They can participate in "arms races" where agents can learn and outdo other agents. Agents can participate in sexual (and asexual) reproduction that can incorporate principles from Darwinian and Lamarckian evolution. Agent societies can exhibit political and organizational properties—whether they are organized, anarchic, or democratic. In short, nature can provide a rich trove of ideas for multiagent system design.

3 Current Notation Challenges

Representing automated systems with currently available notations is known to be problematic. The excessive need for English notes in the modeling notation is one primary indication of such inadequacies. Modeling languages that communicate to a narrow set of system developers and do not communicate to others is a problem for communication among developers in general. These limitations have already triggered a revision process in UML (known as UML 2.0), which tries to remove some of these current limitations. Furthermore, FIPA has recently launched a Modeling Technical Committee which will develop an agent-based notation called AUML (Agent-based Unified Modeling Language).

With agent-based systems, modeling languages are even more challenging because of the richness of representing agents and their systems. In this section, we discuss various aspects of agent-based systems and where graphical modeling languages might be useful to conceptualize and communicate about these systems. First, we begin by examining various aspects of intra-agent requirements. Second, we examine modeling language opportunities that represent agents interacting with other agents. Lastly, we consider the role of the environment in agent-based systems and potential areas for modeling languages.

3.1 Intra-Agent Modeling

Agents are autonomous entities and therefore must be able to manage their own thread of control. This management can consist of simple rules and procedures. More elaborate agents, however, can include belief-desire-intention (BDI) mechanisms and learning capabilities. Expressing some of these features graphically is already occurring.

Agent makeup: A common requirement for developers of agent-based systems is to specify the way in which an agent is composed. For instance, [4] suggest extensions to UML that expresses features, such as state attributes, actions, capabilities, perception, constraints, and available services.

However, agent might consist of other kinds of structures, such as classes, components, packages, as well as other agents. Here, UML class, component, and package diagrams can be employed to depict these notions.

Agent activities and goals: A new aspect that agents bring to modeling is that each agent can seek multiple goals and perform multiple tasks. These goals and tasks are pursued by the agent via the roles that the agent assumes when interacting with other agents. At first, this representation may look like no more than the equivalent to an aggregation pattern in a class diagram, which can be easily represented in UML. However, an agent's relationship with its goals and tasks is not as simple as an object aggregation. The autonomicity of an agent frequently promotes that such agents may not pursue a given goal or task, even though it might be included in its realm of specification.

Although one could extrapolate that it is easy enough to include zero as a valid quantity for a given goal/task, which would indicate that such goal/task might never be pursued, the semantics of the notation would have been changed from its original meaning.

Several existing diagrams could model some of these situations. For example, a UML activity and state diagram could depict an agent's activities flow of control or state-based nature [5]. Goals, goal hierarchies, and goal-task implications could be depicted using notations defined in MESSAGE [6]. However, these goal-related diagrams have not reached a great acceptance.

Dynamic adaptability: Different than objects, agents can have the ability to modify their own behavior. Goals and tasks can be added and removed, as new features are acquired, learned, or considered obsolete for the environment. Despite the actual methodology used to implement the learning process, the needed representation for this feature was not present on standard object-oriented modeling. Dynamic adaptability can also include when, and, where a role be acquired/learned.

Using analogy: Analogies from nature, including human social psychology can be useful to aid designing MAS. For example, modeling techniques would be useful for depicting notions such as single cell animal, the shared environment of cell structures within cell, the communication environment within a cell; a cell-to-internal-structure relation. The forthcoming section on Environmental Modeling will help with most of these concerns.

3.2 Inter-Agent Modeling

In a MAS, agents interact with other agents. Furthermore, to make multiagent systems scaleable, some form of agent grouping must be provided.

Agent interaction: Social systems consist of sets of interdependent role behaviors, providing a collective pattern in which agents play their parts, or roles. The limitations of the current notation become even more visible, when the need to represent inter-task relationships is present. To illustrate this argument, let's assume that an agent of type A can enroll as either, buyer, broker or seller in a particular negotiation, but it can only assume one of these tasks for a particular negotiation.

To further complicate the modeling, several negotiations may be active at any particular moment. Since these multiple tasks may need to access common information at the agent level, it is important to determine how access to common values is controlled and prioritized. Observe that in standard software engineering the modeler hardly ever reaches this level of detail, leaving to the implementer to guarantee correctness. In this case, however, the correctness is not at the implementer's level, but rather is an aspect of the system being modeled. UML sequence and activity diagrams [7] are one mechanism for depicting interactions using roles (See Fig. 2.). However, much still remains to be done in this area. For example, depicting role changes and role constraints still remains a challenge.

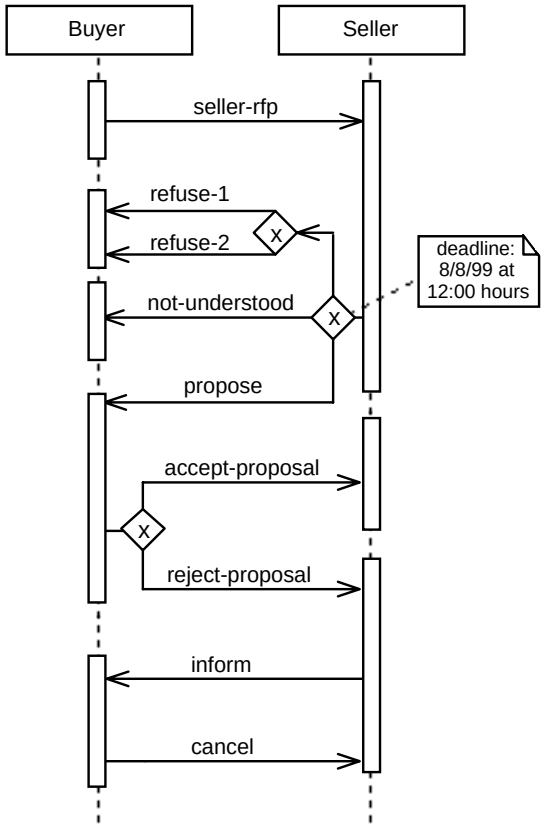

Figure 2: Interaction protocol involving buyer and seller agents.

Agent populations: Agent-based systems are no longer contained within the boundaries of single, small-agent groups. A group is a set of agents that are related via their roles, where these relationships must form a connected graph within the group. Groups can range from small "work cells" to large organizations and institutions. To meet the demands of large-scale system implementations, groups of agent must interact with other agent groups, as well as affect individual agents.

Representing groups, roles, and agent dependencies would be useful in developing MAS. Castelfranchi [8] has defined several forms of agent dependency that can be expressed graphically using a UML-based dependency diagram. Ferber [9] presents graphical approach of his AALAADIN software to represent groups, as well as their membership and interface points. However, much still remains to be done in this area. For example, a way of defining the mechanisms and environment for a group is still not very well developed. However, the forthcoming Environment Modeling section might shed some light on this.

Other: The shared environment of agents with groups, the communication environment between groups, and group-to-agent relations, is also an area for examination. It will be address in the next section on Environmental Modeling.

3.3 Environmental modeling

Another issue in which agent based systems differ from traditional OO object is in the way the agents interact with each other. Agents don't have direct access to other agents; instead they use the environment in which they are immersed to transmit messages to other agents. As an agent executes, it modifies its environment either directly (sending messages that other agents can listen) or indirectly (by altering some of the environment aspects which other agents can sense).

In this fashion the environment plays the role of a Petri dish, setting the rules with which those agents will interact. Due to its vital role, it is important to describe precisely such environment since a slight change could impact the results of the agent system in unpredictable ways. Currently there are no standardized ways to describe this important feature, and to differentiate it from the agent code itself.

Without an environment, an agent is effectively useless. Cut off from the rest of its world, the agent can neither sense nor act. An environment provides the conditions under which an entity (agent or object) can exist. It defines the properties of the world in which an agent will function. Designing effective agents requires careful consideration of both the physical and communicational aspects of their environment.

Physical Environment: The particular kind of environment that biological agents (animals and plants) require for survival is referred to as their ecological niche. While artificial agents can have different requirements for survival, they still require an ecological niche, or physical environment, to support them. The physical environment provides those principles and processes that govern and support a population of entities.

Principles: For agents, principles of the physical environment can be thought of as laws, rules, constraints, and policies that govern and support the physical existence of agents and objects. However, currently there are no modeling languages that can express the basic characteristics for an agent environment [10; 11]: accessibility, determinism, diversity, controllability, volatility, temporality, locality, and medium. Perhaps, no graphical techniques can adequately express any of these characteristics. However, some thought should go into whether or not modeling languages might be useful to the MAS developer.

Processes: In an agent environment, a primary purpose of processes is to implement the environmental principle. For example, the gravitational field is a principle that can be implemented with a process that attracts entities in a prescribed manner. In other words, the falling of an apple to earth can be regarded as the process of gravity in action. Different physical environments will be required for different kinds of agents—and vice versa. With artificial agents, much more than physics is happening because much of the environment is information intensive. In many defense-related agent systems, the information-intense environment includes satellite telemetry, body- and vehicle-based communications technology, and geographic positioning grids. In agent-based supply chains, information about orders and resources is a major component of the system.

To support the varied information requirements of such agent-based systems, a common processing platform would be useful and would consist of: application support, communication and transportation, physical linkage, agent management system, agent platform security manager, agent platform communication channel. Indeed several agent platforms have been develop to support the implementation of such agent systems (OpenCybele, JADE, Zeus, Voyager, aglets just to name a few) each with its own strengths and weaknesses.

In order to detail which features are more relevant for the MAS under development and assist implementers in selecting the correct tools, it is fundamental for the developer to be able to express the relationship of the agents with their environment as well as the structure of each agent. Again, few graphical techniques can adequately express many of these requirements. Yet, some thought should go into whether or not modeling languages might be useful to express these requirements to a MAS developer. For example, the UML deployment, component, and class diagrams might be useful here.

Communication Environment: While an agent can operate by alone, the increasing interconnections and networking require a different kind of agent—one that can communicate effectively with other agents. A communication environment provides two things. First, it provides the principles and processes that govern and support the exchange of ideas, knowledge, information, and data. Second, it provides those functions and structures that are commonly employed to enhance communication, such as roles, groups, and the interaction protocols between roles and groups. In short: The communication environment provides those principles, processes, and structures that enable an infrastructure for agents to convey information.

In rich multiagent societies (MAS), several principles are required to facilitate the communication environment. These would include: communication language, interaction protocols, coordination strategies, social policies, and culture.

An agent's communication environment provides processes that enable agents to interact productively. In particular, it must provide: interaction management, language processing and policing, coordination strategy services, Directory service, mediation services, policy enforcement service, social differentiation, and social order[1].

Providing techniques for modeling both communication principles and processes are highly important to the functioning success of any large-scale MAS. As mentioned earlier, UML sequence and activity

[1] The agent communication *channels* are defined as part of the physical environment. The communication *environment* uses those channels to convey information.

diagrams are two mechanisms for depicting interactions using roles.

4 Notation Proposition

4.1 Intra-agent modeling

In this paper, we propose the modeling of agents as classes, with a new set of associations towards their roles, which in turn can be defined as classes or components. Figure 3 shows a possible diagram to represent the relationship between an agent and its roles. In this diagram, the agent uses the UML implements association on a different manner then the original way intended by OO. Our proposed agent-modeling notion of classes has no parallel with actual implementation but rather the concept of independent structure. Hence the notion of an implementation association is somewhat different in which it qualifies the agent as capable of assuming the target role.

The diagram below has other notation propositions, which can be observed as the relationships between the roles themselves. One may observe two proposed standard associations between roles. The «prevents» association means that while an agent is performing a given role, within a context (i.e., a specific interaction between agents), it becomes illegal for such an agent to perform the other role in the same context. These associations are unilateral, which forces us to indicate twice when the association is mutual exclusive.

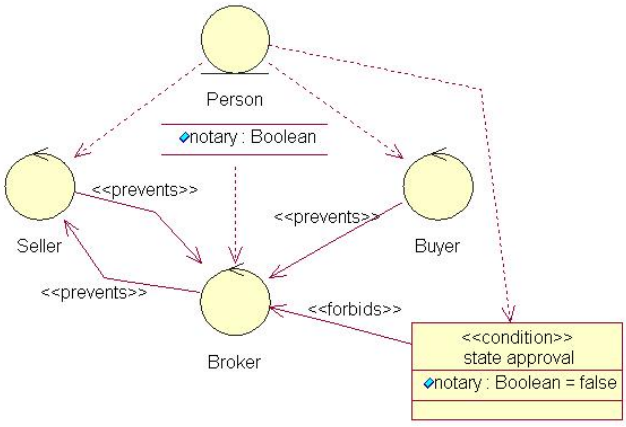

Figure 3: Proposed Class Diagram for Agents

The diagram above also demonstrates two new concepts that are important for multi-agent descriptions. The first concept is the presence of a variable. This variable does not represent a real variable in the implementation sense but rather an agent feature that is observable by its roles. The second concept is a concept of condition. A condition is a clause that holds relationships between an agent and one of its possible roles. In the example above the condition will hold true, when the agent's notary feature is false. The consequence of the condition becoming true is the associations with the roles, which in the case shown forbids the agent to assume the broker role.

There is a slight but significant difference between the «prevents» and the «forbids» association. The «forbids» association impedes the execution of a role in any context, which has a much broader effect then the former one. The dual for the «prevents» and «forbids» associations would be the «permits» and «allows» associations respectively. One can certainly anticipate the needs of other standard associations such as: obtain, reset, removes, and others, which are yet to be explored.

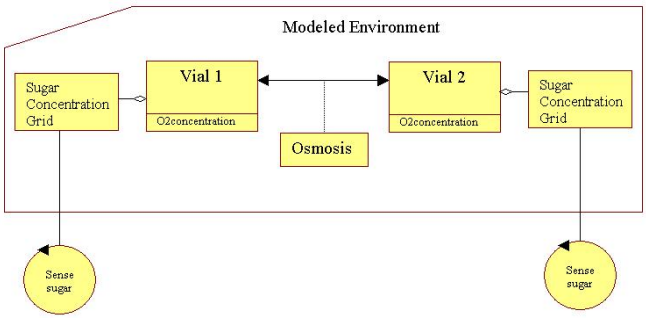

Figure 4: Class Diagram with Environment description

4.2 Environment Modeling

Our proposal for environment modeling is also based on the UML class diagram. Once more the modeling makes no inference on the implementation implication of classes but rather the encapsulation concept that they assume. In our proposed modeling the global environment is represented as wrapper around local environments. Figure 4 demonstrates a simplistic environment to simulate bacteria growth. In this environment, two sugary solutions are placed in vials that share an osmotic membrane. The relationship that describes the osmosis process between the two sub-environments is clearly defined as dependant on the mechanics of the osmosis class. Each sub-environment has its own grid that controls the amount of sugar available in a certain coordinate.

The model environment indicates that an agent has to perform a "sense sugar" role in order to receive information about the current concentration of sugar in its location. In contrast any agent in this environment immediately knows the concentration of O2 without the need to an interaction. From the aggregate symbol in the diagram above one can conclude that the grid is actually a part of the vial sub-environment, but it has encapsulated some unique behavior, as it is in this case the way the sugar diffuses in the syrup.

5 Example

The purpose of this chapter is to demonstrate how even a simple example real example can become a challenge for notion languages when the richness of the system is to be fully described such as needed when describing agent systems.

5.1 Case Study Description

The case study demonstrated is based in the United Nations Security Council resolution process and was used as a debate example in the FIPA Modeling Technical Committee.

Description: The UN Security Council (UN-SC) consists of 15 members, where 5 are permanent members and the others are rotated from the members of the United Nations according with the rules of the organization. Members become the Chair of the Security Council in turn monthly.

To pass a UN-SC resolution, the following procedure would be followed:

(1) At least one member of UN-SC submits a proposal to the current Chair;

(2) The Chair distributes the proposal to all members of UN-SC and set a date for a vote on the proposal.

(3) At a given date that the Chair set, a vote from the members is made;

(4) Each member of the Security Council can vote either FOR or AGAINST or ABSTAIN;

(5) The proposal becomes a UN-SC resolution, if at least nine members voted FOR, and no permanent member voted AGAINST (veto power).

(6) The members vote one at a time.

(7) The Chair calls the order to vote, and it is always the last one to vote.

(8) The vote is open (in other words, when one votes, all the other members know the vote)

(9) The proposing member(s) can withdraw the proposal before the vote starts and in that case no vote on the proposal will take place.

(10) All representatives vote on the same day, one after another, so the chair cannot change within the vote call; but it is possible for the chair to change between a proposal is submitted until it goes into vote, in this case the earlier chair has to forward the proposal to the new one.

(11) A vote is always finished in one day and no chair change happens on that day. The chair sets the date of the vote.

(12) There is no change in the composition of the Security Council during the entire voting process. Proposals that cannot be voted in time are automatically withdrawn and should be resubmitted (or not) when the new composition of the Security Council is reestablished.

One must observe that the procedure above was defined for a case study of agent-oriented modeling, and it does NOT necessary represents the reality.

5.2 Notation Challenges

Even in this simple system, one can identify several notions that can be problematic in modeling language representations.

The first notation challenge is to clearly represent the group organization within the Security Council amongst the several agents, (i.e., permanent/temporary members, chair) and how agents (members) join or leave their groups.

The second problem is how to demonstrate the cyclical nature of the voting process without creating a lifeline for each member and even more how to describe the temporary attributions of a member while it is occupying the “chair” role.

Other notation challenges are due to the possible combinations of allowed/disallowed membership/chair change during different moments in the process. The multitude of combinations forces us to create a modeling format that supports this flexibility and yet clearly defines which paths of execution are possible.

5.3 Proposed Diagrams

The diagrams presented in this section were our proposed solution to this study case as presented in the FIPA modeling Technical Committee forum.

Our solution for the case study presented was composed of four diagrams. The first diagram [Figure 5] presents the Security Council (SC) environment with its two groups and indicates each member by name (members were current when the solution was crafted).

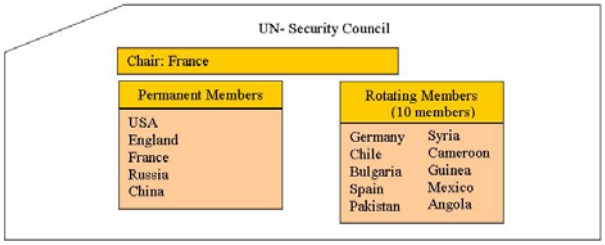

Figure 5: UN-Security Council Environment

One of the drawbacks pointed out in our solution was the lack of a process description by which temporary members are rotated (or even that this rotation is a necessary feature of the system). In order to introduce this notion, the SC environment has to be defined as a sub-environment of the whole United Nations environment. Other solutions presented in the forum, which have modeled the environment with a group membership focus, were able to express this process in a clearer fashion.

The intra-agent representation of our solution was entirely based on the functional perspective of the member agent. For a full description of the agent’s internal structure other perspectives are necessary such as goal orientation (how the agent would use the available roles to pursue a given goal), social relationship (how the instantiation of role varies the membership in the defined groups of the system) and even in case of software systems, the implementation perspective which describes each of the classes used to implement the agent and the relationship between these classes on a software engineering view.

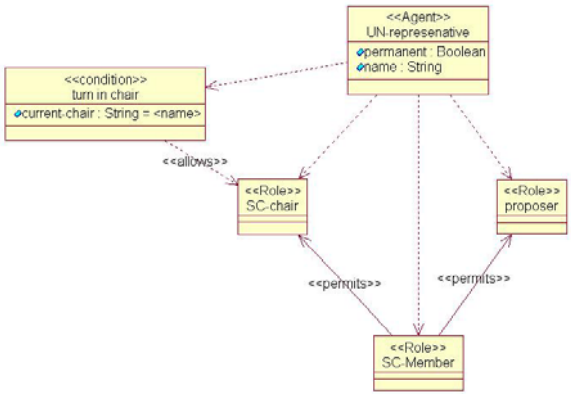

Figure 6: Intra-Agent functional description

In object-oriented systems, typically only the implementation perspective is used and notions of the functional perspective are merged into the diagram. Due to the complexity of agent systems (and its use to explain and predict model behaviors in non-software oriented domains) a clear separation and indication of the perspective of the diagram becomes quintessential. To our knowledge this kind of diagram (with small nomenclature and notation changes) seems to be the most homogenous between the ones used to describe agents systems.

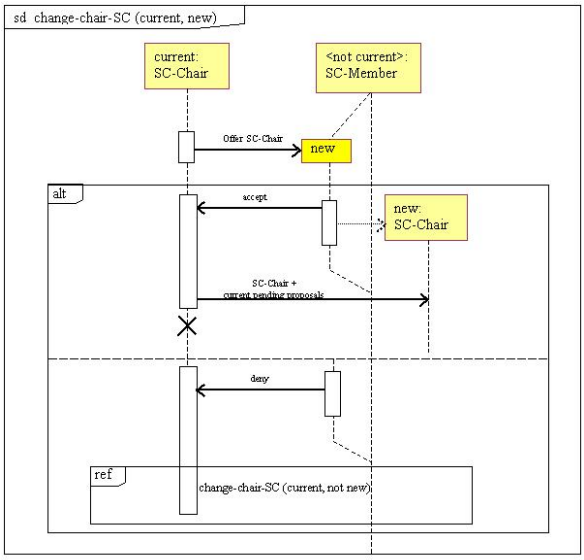

Figure 7: Chair rotation interaction diagram

Figure 7 and Figure 8 show the chair rotation process and the proposal voting process in an interaction diagram format (sequence diagram in UML). In our proposal we have tried to keep the notation as close as possible with the newer version of UML (2.0), altering and extending only when necessary.

One of the extensions was the usage of parameters to define a specific individual in a lifeline that represents a group in which the individual is member. The usage of agent conditions (current chair) or message-defined values allows the representation of the group as a whole in the lifeline, and at the same time isolates the addressed individual in the group, promoting a temporary bifurcation of the lifeline.

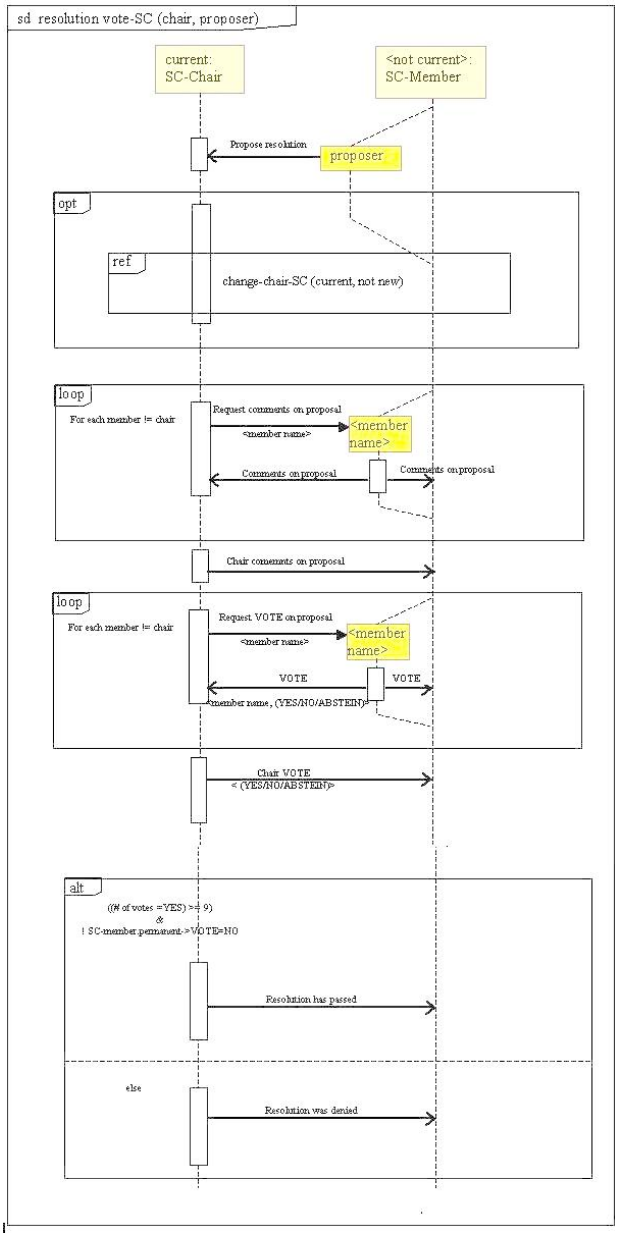

Figure 8: interaction diagram for proposal voting

The lifeline bifurcation (present in UML 2.0 without parameters) has been criticized as being visually cumbersome when several blocks (alt, loops, …) are involved.

The second extension is expressed in Figure 7, to indicate the change/add of role in which a SC-member becomes the new chair of the Security Council.

The final extension is only to create the optional block representation (marked by an opt label in the block construction). This type of block, which does not exist in UML 2.0, indicates that actions within the block may or not happen (as a block). This simple extension allows the consolidation of two very similar interaction paths and hence the simplification of the overall interaction diagram.

Discussions with the FIPA modeling technical committee have raised the concern that the relationships between different interaction diagrams are not clear in our solution. Other authors in the forum have presented Workflow/Activity based diagrams that were developed to present the overall scheme between these diagrams.

6 Conclusion

In this paper we have presented some of the challenges of modeling and notation of agent based systems and how they differ from standard object oriented systems. We have also proposed a notation format for the presented challenges that are compliant with an extended view of UML.

This paper has no intention to try to determine the best notation for agent systems. The intention is rather to present the need and stir the debate on this issue that is currently active in the Agentlink and FIPA forums.

Acknowledgement

We would like to acknowledge NASA for funding this effort in standardization of AUML notation as part of the project under contract NAS2-02003 and the collaboration of NASA's technical representative Ms. Michelle Eshow in our efforts.

Our thanks to Radovan Cervenka, Hong Zhu and Misty Nodine, for their collaboration in the definition and discussion of the study case presented which was extracted from the discussion in the FIPA Modeling TC forum, where each researcher presented their own solution.

References

[1] HPLabs, http://www.hpl.hp.com/agents/

[2] BritishTelecom- http://more.btexact.com/projects/agents.htm

[3] Parunak, H. Van Dyke, "'Go to the Ant': Engineering Principles from Natural Agent Systems," Annals of Operations Research, 75, 1997, pp. 69-101.

[4] Huget, Marc-Philippe, "Agent UML Class Diagrams Revisited," proceedings of the AgeS 2002 Workshop, Bolognia, 2002.

[5] Odell, J., H.V.D. Parunak, and B. Bauer, Representing Agent Interaction Protocols in UML, in Agent-Oriented Software Engineering, P. Ciancarini and M. Wooldridge, Editors. 2001, Springer: Berlin. p. 121-140.

[6] Evans, R., et al., MESSAGE: Methodology for Agent-Oriented Software Engineering. 2001, EURESCOM Project P907, Deliverable 3.

[7] Odell, J., H.V.D. Paranak, and B. Bauer, "Extending UML for Agents," in Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence, G.W. Yves Lesperance, and Eric Yu, Editor. 2000, workshop proceedings: Austin, TX. p. 3-17.

[8] Castelfranchi, C., "Engineering Social Order," Nordic Journal of Philosophical Logic, 2002. (to appear).

[9] Ferber, J. and O. Gutknecht, "A meta-model for the analysis and design of organizations in multi-agent systems," in Third International Conference on Multi-Agent Systems (ICMAS'98). 1998. Paris, IEEE Computer Society.

[10] Weiss, G., ed. Multiagent Systems: A Modern Approach to Distributed artificial Intelligence. 1999, MIT Press: Cambridge, MA.

[11] Russell, S. and P. Norvig, Artificial Intelligence: A Modern Approach. 1995, NJ: Prentice-Hall