An Agent-Oriented Methodology: High-Level and Intermediate Models

M. Elammari, W. Lalonde

School of Computer Science, Carleton University
Ottawa, Canada

email: {elammari,lalonde}@scs.carleton.ca

Abstract
The recent introduction of many agent-based systems and the increase in agent publications have made the area of agent-based systems a popular area. However, a number of challenges must be met before agent-based system development can claim to be fully mature. One of the most important challenges is the use of a comprehensive design approach whereby the design models fully support agent aspects at the design level. This paper describes a design methodology which allows the development of agent based systems from user requirements. The methodology models the external and internal behaviour of agents. It provides a means of both visualizing the behaviour of systems of agents and defining how the behaviour is achieved. It provides a systematic approach for generating from high-level designs implementable system definitions. Humans, with machine assistance, can manipulate in a systematic way models at one level of abstraction into models at the next lower level.

1.0 Introduction

The agent paradigm represents a significant shift in approach to the development of complex software systems. The need to model and understand agent system complexity is well recognized [11][12][15][20]. However, a comprehensive method for designing agent systems is lacking. There is little that has been done in the area of analysis and design of multi-agent systems. The current modelling techniques are too complex and are often modifications of object-oriented techniques without taking into considerations the first class attributes of multi-agent systems.

Kendall et al. [11] model agent systems using workflow models and Kinney et al. [12] use extentions of object-oriented techniques. Other techniques in the agent community, such as COOL [1] and Shoham’s AOP [16], represent agents formally with logic, with no visual representation. This is important since visual representations are well known to be needed for better human understanding.

The lack of agent-oriented methodologies is attributed to the fact that developers often forget that they are actually developing software [20]. Developers have the tendency to work on issues such as agent architectures and agent coordinations, and completely neglect software engineering issues.

Wooldridge et al. [20] state that “in the absence of agent-oriented techniques, object-oriented techniques may be used to great effect. They may not be ideal, but they are certainly better than nothing.” Several researchers chose the path of extending object-oriented design techniques, rather than developing new ones, mainly because of their familiarity. Even though OO techniques are mature and effective in capturing complex systems, they were developed with objects in mind. We developed our design process with agents in mind. Our process captures through its models: the agents making up the system, their goals, plans, beliefs, and relationships. The overall purpose of our models is similar in spirit to those generated using OO techniques. Just as OO techniques capture objects in the system, their attributes, structure, and relationships, our models similarly capture agents, their attributes, structure, and relationships. The difference between the two is the way these details are captured and presented.

Some design techniques that extend OO techniques mainly change terminology. For instance, a method is called a plan and an attribute is called a belief. There is a mismatch between the concepts used in OO techniques and the agent oriented paradigm [19][20]. The major difference between agents and objects is their internal structure. Objects encapsulate methods and attribute while agents encapsulate goals, plans, beliefs, and commitments. They also have different types of relationships and communication patterns. Agents are required to have active and proactive behaviour while objects are not. Due to these differences, agent systems are often more complex than OO systems and hence OO design techniques generally fail to capture the complexity of agent systems.
We believe that any design process that is tailored to agents should provide support for the following:

• **System view**: Understanding agent systems requires a high-level visual view of how the system works as a whole to accomplish some application related purpose. This understanding would be difficult to convey if the only models available were low level design diagrams, such as object interaction diagrams, and class inheritance hierarchies. We need a macroscopic, system-oriented model to provide a means of both visualizing the behaviour of systems of agents and defining how the behaviour will be achieved, at a level above such details.

• **Structure**: Agents’ internal structure is expressed by aspects such as goals, plans and beliefs. A process should facilitate the discovery of agents needed along with their internal structure.

• **Relationships**: An agent has dependency and jurisdictional relationships with other agents. An agent might be dependent on another agent to achieve a goal, perform a task or supply a resource. A process should capture the different inter-agent dependencies and jurisdictional relationships.

• **Conversations**: Agents must cooperate and negotiate with each other. When agents communicate, they engage in conversations. A process should capture the conversational messages exchanged and facilitate the identification of conversational protocols used in communication.

• **Commitments**: Agents have obligations and authorizations about services they provide to each other. A process should capture the commitments between agents and any conditions or terms associated with them.

• **Systematic transitions**: A good design process should provide guidelines for model derivations and define traceability between the models.

In this paper, we describe a process for designing agent systems in which a visual technique is used to give a bird’s eye view of the system as a whole and to provide a starting point for developing the details of agent metamodels and software implementations to satisfy the requirements. A novel aspect of our technique is that it encourages a constructive approach in which systems are developed through a series of levels of abstraction in which humans, with machine assistance, can manipulate abstractions at one level into abstractions at the next lower level. Our development approach has two phases that we term the discovery and definition phases. The discovery phase guides the discovery of agents and their high-level behaviour. The ultimate goal of this phase, apart from discovering the agents and their relationships, is to produce models that capture the high-level structure and behaviour of the system. The definition phase produces implementable definitions. The goal is to have clear understanding of the behaviours, the entities that participate in exhibiting these behaviours and their interrelations, as well as inter-agent conversations and commitments.

This rest of the paper is organized as follows. We begin in Section 2.0 by providing an overview of the models produced by the methodology. Section 3.0 explains the example used to illustrate the process. The methodology is demonstrated by means of a real-world agent based telecommunication application. The first phase of the methodology, the discovery phase, is discussed in Section 4.0. The definition phase and its associated models are discussed in Section 5.0. Section 6.0 provides a conclusion and identifies work in progress.

### 2.0 The Models

Five types of models are generated by our approach. Figure 1 shows these models and the traceability between them. The **High-level model** identifies agents and their high-level behaviour. It gives a high-level view of the system and provides a starting point for developing the details of the other models. It is generated by tracing application scenarios that describe functional behaviour, discovering agents and behaviour patterns along the way. The **internal agent model** describes the agents in the system in terms of their internal structure and behaviour. It captures agent aspects such as goals, plans and beliefs. The internal agent model is derived directly from the high-level model. The **relationship model** describes agent relationships: dependency and jurisdictional. The **conversational model** describes the coordinations among the agents. The **contract model** defines a structure that captures commitments between agents. Contracts can be created when agents are instantiated or during execution as they are needed.

The high-level model is the only model that is associated with the discovery phase. The rest of the models are associated with the definition phase. Before we describe the different models of integral to our approach, we describe in the next section the case study used to illustrate our design process.

### 3.0 Case Study: An Intranet Telephony Application

We applied our approach to an intranet telephony application which is being developed at Mitel Corporation. The application combines voice/data and interactive/noninteractive communications [5][6][8]. Users of this application can communicate with each other using voice or data. The mode of communication can be interactive or noninterac-
The interactive mode requires that at least two users be active. An obvious example of this is when two users are talking to each other. The noninteractive mode requires only one active user. An example of this is when a user wants to leave a text message (email) or a voice message to another user. Users are allowed to login to any desktop and their incoming calls are forwarded to them at that desktop. A desktop can have a USB (Universal Serial Bus) phone connected to it which can be used as a regular phone. A user can subscribe to several calling features (e.g., call waiting and call forwarding) and can assume different roles (work in different jobs) in an organization. In the case study, the organization is assumed to be responsible for the development of several products and for providing help support. For each product supported, there is a specific division responsible for that product. The division has two subgroups: the development group and the help desk team. The development group is run by a manager who oversees several developers. The help desk team has a number of attendants who are responsible for responding to customer queries. The team must provide, at all times, an adequate response to each customer's requests. The help desk team must gather the state of the calls to the attendants and decide if more attendants are needed or if existing attendants should be retired. When the help desk needs more attendants, a request is made to the product development team to spare some of its developers.

The rules under which a user operates may change depending on the role he plays in the organization. For example, a user may play the role of a developer or a help desk attendant depending. Depending on his role, the user may inherit fundamentally different rules restricting behaviour and privileges. For example, the help desk team may restrict its attendants so they cannot make long distance calls.

### 4.0 The Discovery Phase

We use use-case maps (UCMs), which are suitable for high level visual representations, as a starting point for generating more detailed visual descriptions. A novel aspect of our approach is that it encourages constructive program design in which systems are built up via the decomposition of high level designs. Use-case maps are used to model the high-level activities because of their ability to simply and successfully depict the design of complex systems, and provide a powerful visual notation for a review and detailed critique of the design.

#### 4.1 Use case maps (UCMs)

UCMs [2][3][4] are precise structural entities that contain enough information in highly condensed form to
enable a person to visualize system behaviour. UCMs (as shown in Figure 2) provide a high level view of causal sequences in the system as a whole, in the form of *paths*. The causal sequences are called scenarios. In general, UCMs may have many paths (the figure only shows one, for simplicity). The causality expressed by the paths is understood by *humans*, not necessarily by individual components of the system.

**Figure 2: Example of a UCM.**

A filled circle indicates a *start point* of a path, the point where stimuli occur causing activity to start progressing along the path. A bar indicates an *end point*, the point where the effect of stimuli are felt. Paths trace causal sequences between start and end points. The causal sequences connect *responsibilities*, indicated by named points along paths (e.g., r1, r2 and r3). Paths are superimposed on rectangular boxes representing operational components of the system (e.g., C1, C2 and C3), to indicate where components participate in the causal sequences. Individual paths may cross many components and components may have many paths crossing them.

The basic assumption is that stimulus-response behaviour can be represented in a simple way with paths. This is a very common characteristic of the types of systems with which we are concerned. The result is a path-centric view of a system, rather than a conventional component-centric view.

UCMs may be decomposed using a generalization of responsibilities called *stubs* (e.g., S in Figure 3). Stubs may be positioned along paths like responsibilities but are more general than responsibilities in two ways: They identify the existence of sub-UCMs and they may span multiple paths (not shown). Stubs enable us to draw UCMs that give a high level overview of the general trend of paths, while leaving localized meanderings that might obscure the big picture to sub-UCMs shown in separate diagrams. A plug-in may involve additional system components not shown in the main UCM.

A key feature of stubs for agent systems is the ability to represent dynamically pluggable behaviour patterns. A stub may have alternative plug-ins that may be selected according to different system conditions at the time a scenario reaches the stub. Stubs of this kind are shown in dashed outline to distinguish them from stubs that are only used for static path decomposition.

There are other UCM features that are useful for agent systems (see [7] for more details).

### 4.2 The High-Level Model

The discovery phase is an exploratory phase that leads to the high-level model definition. In this phase, the agents are discovered and their high-level behaviour identified. The ultimate result is UCMs of the system that, in diagrammatic form, superimposes causal paths for scenarios on a structural substrate of agents. This model includes these major sources of information:

- Documentation defining operational aspects of the model such as preconditions and postconditions of scenarios along paths, responsibilities of agents along paths, and system state changes caused by the performance of these responsibilities.
- The macroscopic behaviour of the system at the level of collaborating agents achieving some specific system purpose. UCMs at this level capture interagent collaborations required for major tasks, but defer the details to other models.
- A catalogue of plug-ins (diagrams with associated documentation) desribing where and when they may be used.

The high-level model is derived by tracing application scenarios describing functional behaviour as UCM paths through the system. This leads to discovering agents, responsibilities, and plug-ins along the way. Generally, one starts with some use cases and some knowledge of the agents required to realize them. However, there is no requirement that all agents or all use cases be known beforehand. One may start from very general ideas about both use cases and agents. For example, UCMs may be used to discover agents to realize paths that represent use cases, or to dis-
cover new paths that traverse known system components.

Initial agents can be extracted from the nouns that exist in the problem domain. These agents must be selected carefully. Only entities that are essential and that are active throughout the application should be chosen. Entities that are passive (i.e. that don’t need to react to the environment changes) should not be viewed as agents. We classify agents into three types to simplify the process of identifying them and to make it easier to see the differences between them. Each of these types has a special purpose and models one specific aspect of the system:

- **Functional** types represent physical entities that include humans (represented by user agents) and resources (represented by resource agents).

- **Management** types are responsible for a portion of the overall goal of the system. They are usually responsible for managing or controlling groups of functional agents. For example, a printing service agent that controls a group of printers is a management agent.

- **Role** types represent roles that agents can play much like positions that can be filled by people in human organizations. Each role has one or more agents (functional or management) capable of fulfilling the role.

The steps involved in the discovery phase can be summarized as follows:

- Identify scenarios and major components involved.
- Flush out the scenario by identifying more components which include identifying roles.
- For each scenario, identify preconditions and postconditions.
- For each component in a scenario, identify responsibilities and constraints.
- Identify responsibilities that can be achieved by different subscenarios and replace them with stubs.

### 4.3 High-Level Model for the Example

Figure 4 shows a UCM for a basic call scenario between two user agents in some larger system of agents (see [9] for a more general call scenario). User agents are the representatives inside the system of human users in the application environment. Among other responsibilities, the user agents store user information and preferences.

The precondition of the CALL scenario is that a human wants to place a call. The scenario’s path begins with the OH (caller offhook) stub which hides the details of offhook processing at the caller’s end. After all responsibilities associated with offhook processing are performed, the path leads to the POC (process outgoing call) stub which processes the dialed number and generates a call request to be sent to the answerer. The POC stub has two outgoing ports, b and c. Port b is followed when the caller is allowed to connect to the answerer and port c is followed in case of call denial. Port c leads to the STAT stub by means of which the caller is notified of call denial. In case the call is allowed (port b is followed), the path leads to the PIC (process incoming call) stub which processes the incoming call request. The PIC stub has three outgoing ports, b, c and d. Port b is followed only if the answerer accepts the call request. Port c is followed to notify the caller of the call status. The call status notification informs the caller if the call is accepted or refused. An example of situations when an agent refuses connection is when the user is busy. Port d is the means by which the two agents negotiate. If the call request is accepted, the path leads to the RNG (ring) stub which notifies the call recipient, for example, by ringing a phone device. The postcondition of this scenario is that the answerer’s phone is ringing and the caller hears ringback.

Due to the lack of space, we only provide plug-ins for the POC and PIC stubs. See [9] for plug-ins for the rest of the stubs. Figure 5 illustrates two alternate plug-ins for each of the POC and PIC stubs. In these plug-ins, the end points from which the main path continues are labeled. The plug-ins can be summarized as follows:

- The default plug-in for the POC describes the default behavior when the caller has not subscribed to any telephony feature. The plug-in performs the \textit{snd-req} responsibility which causes the caller to send a request for a call connection to the answerer user agent.

- The OCS plug-in for the POC stub is selected when the caller subscribes to the originating call screening (OCS) feature. The path begins by checking the OCS list. If the dialed number is in the list, the connection is refused. This is shown by the fork in the path that follows the \textit{check} responsibility. The simple fork in the path immediately after the \textit{check} responsibility is called an or-fork, and indicates alternative scenario paths. Otherwise the caller is allowed to connect to the dialed number.
The default plug-in for the PIC stub describes the default behavior when the answerer has not subscribed to any feature. The plug-in starts with an or-fork. If the user is busy, the path labeled busy is followed and the caller is notified that the answerer is busy. Otherwise the path is forked into two concurrent paths. The fork, with the bar across it, is called an and-fork, and indicates that the scenario proceeds concurrently along two paths. One fork allows the agent to notify the answerer of the incoming call. The other notifies the caller of call progress.

The CF plug-in for the PIC stub is selected when the answerer subscribes to the call forwarding feature (CF), and system conditions at the time of entry to this stub select this feature. The CF feature performs the fwd-req responsibility which causes the incoming call to be forwarded to another user agent.

4.4 Management Scenarios

In the previous section, we described how agents cooperate to establish a call connection. In this section, we describe a scenario that captures the agent organization and management decisions. We assign a management agent to each group (or subgroup) in the organization. Each of these agents is responsible for monitoring events that are of interest to the group and enforcing any policies that the group may have.

The whole organization is represented by one agent, called Enterprise, responsible for the development of all products and their help support. A division is presented by a Division agent responsible for developing and supporting a single product. Such an agent manages two groups: a product development group (represented by a ProdDevelopment agent) and a help desk team (represented by a HelpDesk agent). The manager of the product development group is represented by a PDManager agent and a developer in the group is represented by a Developer agent. The help desk team has a number of attendants who are represented by a number of HDAttendant agents. The HelpDesk agent is responsible for ensuring adequate response to a customer’s requests. It is also responsible for gathering the state of the calls to the attendants and deciding whether it should request more agents or retire agents. When the HelpDesk needs more attendants, it requests new agents from its superior (a Division agent) which has to decide, based on a policy, whether the ProdDevelopment group can spare any of its developers. A designer may want to assign managers to the division and help desk groups, if there are human managers who manage those groups. We decided not to assign managers to those groups. We made the assumption that there are no human managers for those groups and hence each agent that represents any of those group is responsible for managing it.

Figure 6 illustrates a scenario in which the product development manager requests a new developer to be assigned to its team. This request is processed by the product development agent which notifies its superior, a Divi-
sion agent, of the request. The Division agent evaluates the request and it has three choices for proceeding. The three forks after the “eval req.” responsibility represent the three possibilities. The first fork denies the request. The second fork causes the help desk to retire one of its attendants and this attendant is assigned to the developers team. The third fork goes to a higher authority, the enterprise agent. The enterprise agent, in our design, is the highest authority in the organization. The enterprise agent allocates a new employee for the division and the employee’s user agent is moved to the developers team. In the UCM, roles are represented by *slots* (boxes with dashed lines). Slots are organizational places that may be entered dynamically by components. The movement of components along a path is implied by (1) a move arrow pointing at a path and (2) a subsequent move arrow pointing away from the path into a slot. The UCM shows user agents as a stack. The stack notation implies that each component of the stack is distinct, all operationally identical from the perspective of a traversed path, but only one is selected by the path context.

This management UCM captures the relationships between agents and describes how requests are processed in the organization. Most of the path segments that link agents in management UCMs are links between superiors and subordinates. Those path segments are the basis for reasoning about the agent relationships and will used at a later stage to create the organization.

5.0 Definition Phase

In the previous section, we described the high-level model that captures the high level behaviour of agent systems. In this section, we describe the definition phase which produces intermediate models that facilitate the implementation of agent systems. The high-level model, supplemented by other information, is used to generate these models. These models express the full functional behaviour of an agent system by identifying aspects of agents such as goals, beliefs, plans, jurisdictional and dependency relationships, contracts, and conversations.
Our goal is to have clear understanding of the behaviours, the entities that participate in exhibiting these behaviours and their interrelationships. We define four types of models to describe an agent system. The internal agent model defines the agents in terms of their internal structure and behaviour. The relationship model describes inter-agent dependencies and jurisdictional relationships. The conversational model presents the coordinations among the agents. The contract model defines a structure that translates one agent requirements to another. Contracts can be created when agents are instantiated or later on as needed.

Agents identified in the discovery phase act as starting point and do not represent the final agents that will be instantiated. As the models are refined, new agents may be discovered.

5.1 Internal Agent Model

The internal agent model is directly derived from the high-level model. It describes the internal structure of the agents discovered in the high-level model. The agents are described in terms of their goals, beliefs and tasks. We use tables to describe the internal structure (a table for each agent).

The mapping from UCMs to the internal agent model is almost straightforward. The relationships among UCMs, agent terminology, and the internal agent model is summarized in Table 1. Path segments that traverse an agent represent goals, dynamic stubs along paths represent sub-goals, static stubs represent sets of agent tasks, path preconditions and postconditions help in forming the belief set, and responsibilities along the path constitute the agent’s high-level tasks. Also the model captures, if needed, the causality relationship in UCMs. This is done by converting path segments connecting two agents in a UCM to a task in the internal agent model. Each of these tasks is basically responsible for causing tasks in other agents to be activated.

The internal agent model table has four columns. The goal column lists the goals an agent may adopt to reach a desired state. The precondition column lists the beliefs that should hold in order for the goal to be executed. The post-condition column lists the effects of executing a successful goal on an agent’s beliefs. The task column lists all the agent tasks, including subgoals, that are required to fulfill each goal. A goal may be decomposed into subgoals which provide detailed or alternate ways of achieving that goal. These subgoals are shown in the tasks column as well as in the goal column. Each row in the internal agent model table represents a plan that can be instantiated at runtime to fulfill a goal. A goal may have different plans that can fulfill it, and hence a goal may have more than one entry in the table. If needed, a finite state machine can be associated with each plan to explain its logic. The comment column contains a textual explanation of each plan.

Note that as a general rule, if a path segment has exactly one dynamic stub, then only the mapping of the dynamic stub is needed. In other words, no goal is created to represent the path segment. The reason is that it does not make sense to have a goal with only one sub-goal. If a path segment has responsibilities or more than one stub, then the path-segment should be mapped to a goal in the internal agent model.
ERROR: syntaxerror
OFFENDING COMMAND: --nostringval--

STACK:
true