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 Kinny 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

Published in the proceedings of AOIS 1999 (Agent-Oriented Information Systems), Heidelberg (Germany), 14-15 June 1999.

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 interrelationships, 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 integral to our approach, we describe in the next section the case study used to illustrate our design process.

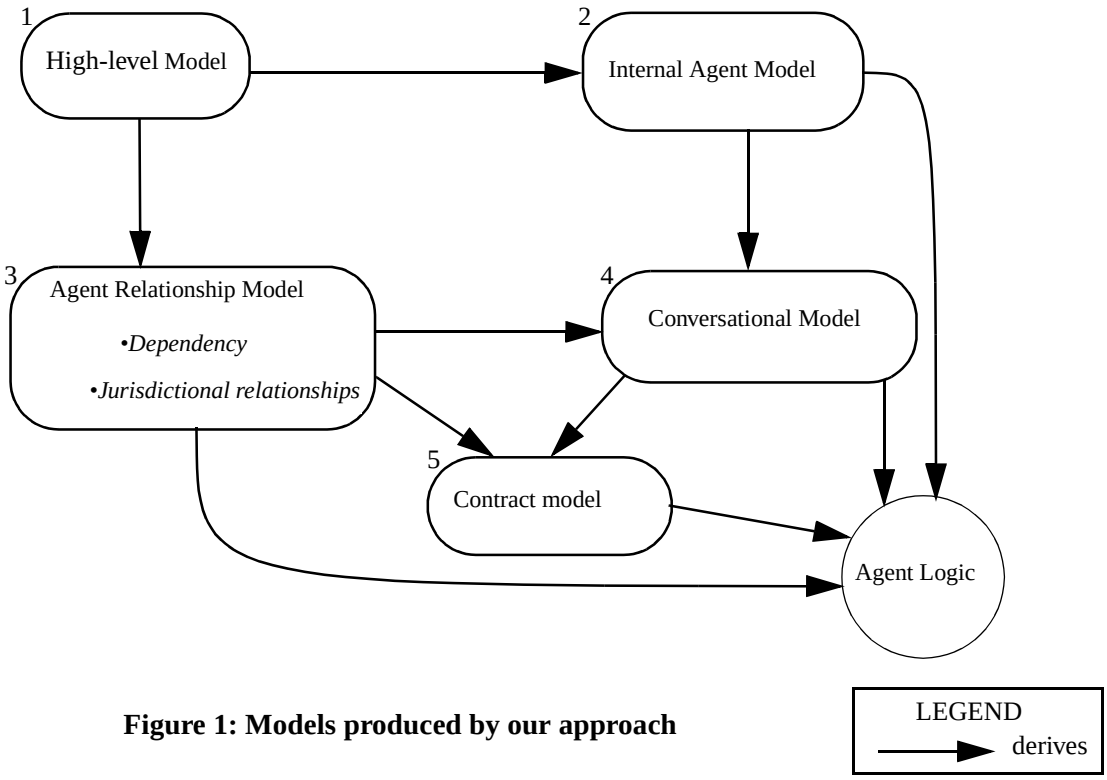

Figure 1: Models produced by our approach

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 noninteractive. 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 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 pro-

gram 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.

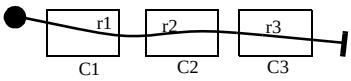

Figure 2: Example of a UCM.

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

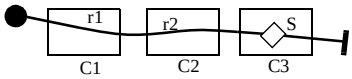

Figure 3: A UCM with a stub.

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) describing 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 discover 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:

- *Actor* type: represent physical entities that include humans (represented by user agents) and resources which can be hardware or informational (represented by resource agents).
- *Management* type: are responsible for managing or controlling groups of actor agents. For example, a printing service agent that controls a group of printers is a management agent.
- *Task* type: are responsible for a performing a specific task. For example, an agent that is responsible for generating telephone tones is a task agent.
- *Program* type: are responsible for a representing or interfacing to a program.
- *Role* type: represent roles that agents can play much like positions that can be filled by people in human organizations. A role can have one or more agents capable of fulfilling it.

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

- Identify scenarios and major components involved. Draw UCM paths that connect the identified components.
- 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 *snd-req* responsibility which causes the caller to send a request for a call connection to the answerer user agent.

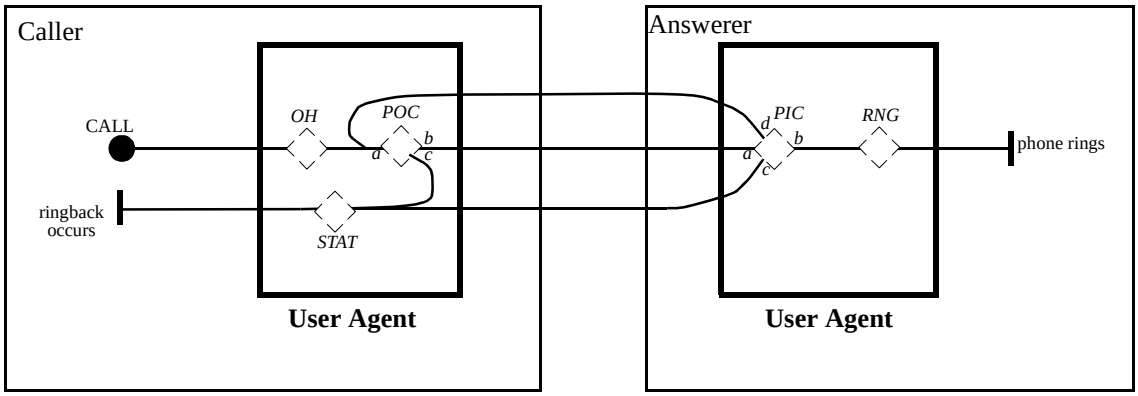

Figure 4: Call connection UCM.

Precondition:
Some human wants to place a call.
Postcondition:
At answerer end, phone rings At caller end, ringback occurs
Stubs:
OH. do offhook processing at caller end
POC. process outgoing call
PIC. process incoming call
RNG. ring attached devices
STAT. inform caller of call status

- 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 *check* responsibility. The simple fork in the path immediately after the *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

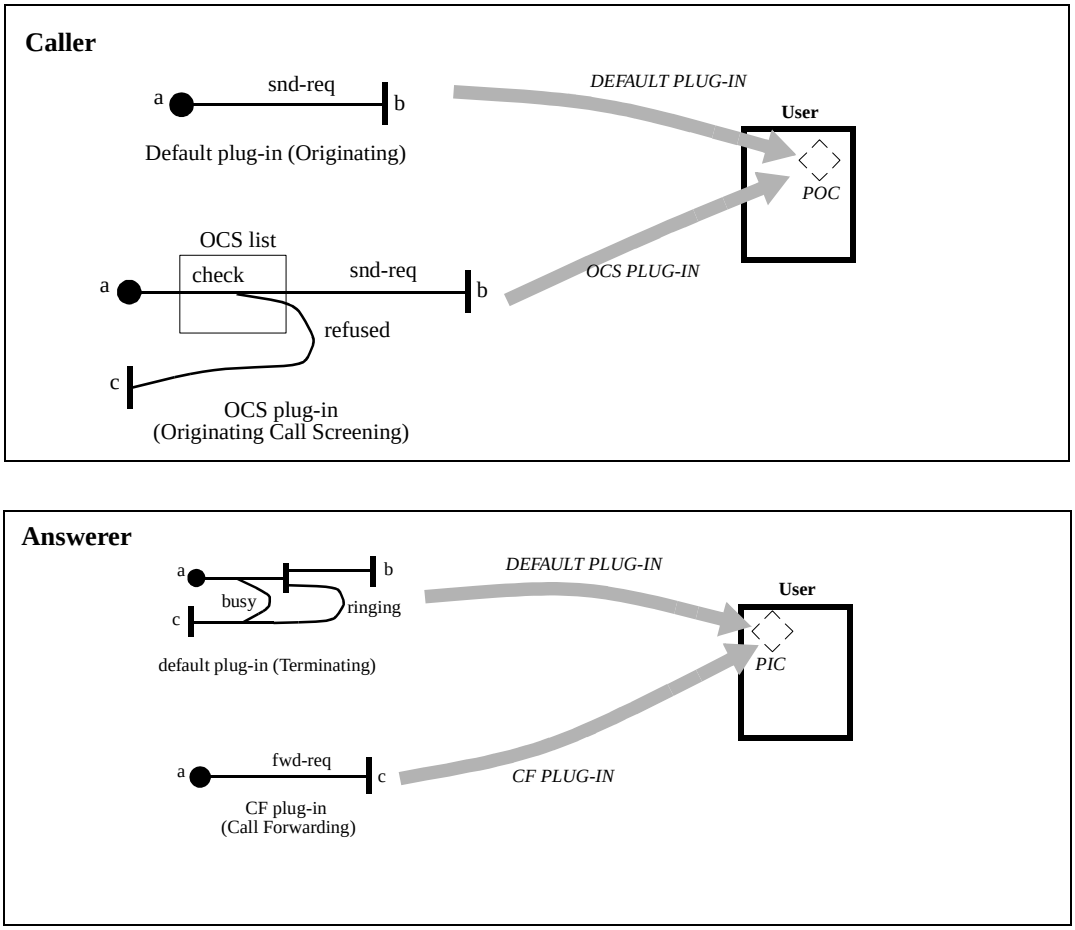

Figure 5: POC (process outgoing call) and PIC (process incoming call) plug-ins

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 Division 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.

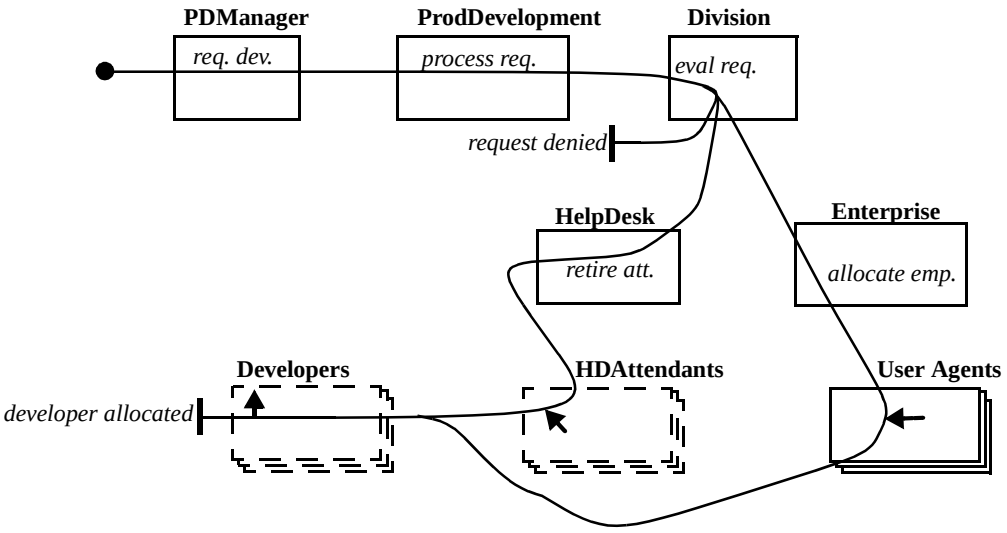

Figure 6: Product development manager requesting a developer

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, plans 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 tasks 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 of 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.
UCMs allow different scenarios to share a common path segment. Sometimes the only thing these scenarios have in common is the responsibilities along the common path segment. A designer must decide if a common path segment should be mapped to one goal for all scenarios or to a goal for each scenario.

The process of building the internal agent model from UCMs can be summarized as follows:

- Analyze each path segment that traverses an agent and associate a goal with it, subgoals with its stubs, and actions with its responsibilities.
- For each path segment, identify pre and post conditions and map them to beliefs.
- Analyze path segments that connect agents and identify agent tasks that are responsible for causing tasks in other agents to be activated.

Table 1: From UCMs to internal agent model

UCMs	Agent Terminology	Internal Agent Model
Path segments traverse an agent	Goals	Goal column
Dynamic stubs	Subgoals	Goal in the task column Goal in the goal column
Static stubs (decomposition)	Sets of agent tasks	Task in the task column
Path segment preconditions	Beliefs	Preconditions column
Path segment postconditions	Beliefs	Postconditions column
Responsibilities	Agent tasks	Task in the task column
Path segments connecting two agents	Agent coordination	Task in the task column

5.1.1 Deriving an Internal Agent Model for the Example.

In order to derive the internal agent model for our user agents, we start by examining the different UCM path segments that cross them. From the CALL scenario in Figure 4, we can see that there are two path segments that cross the user agents (one in the caller and the other in the answerer). Each of these segments is mapped to a goal and inserted into the internal agent model, as shown in rows 1 and 2 in Figure 7. The preconditions and postconditions of each segment are inserted in the corresponding row. The dynamic stubs along paths are inserted as sub-goals in the tasks column and a goal row is created for each. To illustrate the mapping of dynamic stubs into the internal agent model, we need to examine the POC and PIC stubs closely. Each of these stubs is mapped into a goal in the internal agent model. Since there are two plug-ins (see Figure 5) that can satisfy each stub, two rows (plans) for each goal are needed to represent all associated plug-ins (for a total of four rows). The preconditions, postconditions, and responsibilities for each plug-in are captured in the corresponding row (plan). Goals that represent the other stubs can be created in same way.

5.2 The Agent Relationship Model

The agent relationship model describes inter-agent dependencies and jurisdictional relationships. These relationships are described by two formal graphical notations. They are *dependency* and *jurisdictional* diagrams.

The relationship diagrams are derived from the coordination expressed in the UCMs and from responsibilities and preconditions of UCMs. Coordination is captured in UCMs by path segments that connect two agents. Also analyzing of responsibilities and preconditions may lead to the discovery of dependencies. For example, a responsibility labeled “allocate channel” may indicate that the agent that has this responsibility is dependent on another agent to provide it with a resource; in this case, a channel resource. A designer examines all inter-agent relationships captured

	Goal	Precondition	Postcondition	Task	Comment
1	Initiate call request	User off-hook	Request sent to answerer or call denied	Goal(process off-hook) Goal(originate call) Goal(notify caller)	Caller in main UCM
2	Process call request	There is an incoming call	Incoming call processed	Goal(terminate call) Goal(notify answerer)	Answerer in main UCM
3	Originate call	Number is collected	Request sent to answerer	send_request	Default plug-in for OC stub
4	Originate call	Number is collected	Request sent to answerer or call denied	check_list send_request notify_refuse	OCS plug-in for OC stub
5	Terminate call	There is an incoming call	Caller and/or answerer are notified	ring notify_caller	Default plug-in for TC stub
6	Terminate call	CF is on. There is an incoming call	Caller notified of a new destination	forward_req	CF plug-in for TC stub

Figure 7: The internal agent model for the user agent

in UCMs and classifies each as either a dependency or a jurisdictional relationship. We consider each case separately below.

5.2.1 Dependency Diagram

A dependency diagram relates an agent that provides a service to an agent that requires that service. A node in the dependency diagram represents an agent, and a directed link from a provider to a requester represents a dependency (the requester is the dependent). Examples of dependencies are goals to be achieved and tasks to be performed.

Four types of agent dependencies are identified: goal, task, resource, and negotiated dependencies. These dependencies are similar to the dependencies described by Yu et al. [18] for capturing numerous kinds of constraints and relationships that are frequently encountered in business processes.

Goal dependency indicates that an agent is dependent on another agent to achieve a certain goal or state. The dependent agent does not, however, specify how the other agent should fulfill the goal. *Task dependency* indicates that an agent requires a specific task to be performed. *Resource dependency* indicates that an agent is dependent on a supplying agent to provide it with a specific resource. A resource can be physical or informational. *Negotiated dependency* indicates that an inter-agent negotiation is required to fulfill the dependency. The identification of these dependency types helps in choosing the right inter-agent conversational protocol.

Figure 8 illustrates the different symbols used in dependency diagrams. A dependency is shown in the diagram as an arrow going from a dependee (supplier) to a dependent agent. Also the figure illustrates the four types of graphical symbols used to differentiate dependencies.

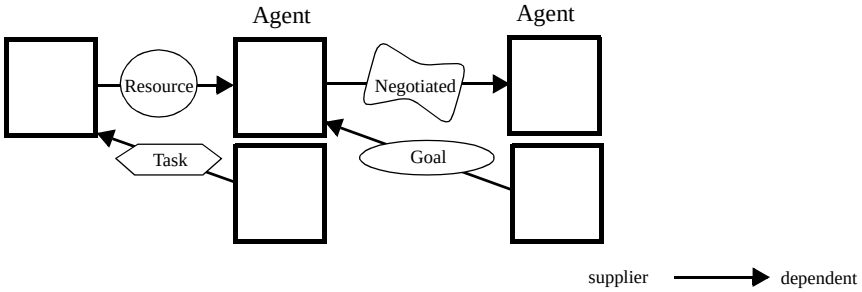

Figure 8: Symbols used in dependency diagram

Figure 9 shows an example of a dependency diagram. This diagram is based on the example discussed in

Section 3.0. Each path segment in the UCMs that connects two agents generates a dependency in the dependency diagram. For example, the two dependencies between the caller and the answerer are derived from the UCM in Figure 4 where there are three paths that connect two agents. The upper two paths constitute a *loop* indicating a requirement for negotiation and consequently, a negotiated dependency. The caller and the answerer need to negotiate to determine the parameters of the call. The lower path segment in the same figure is determined to be a resource dependency, because the caller is dependent on the answerer to notify it of the call status.

Note that since the caller and the answerer are two distinct user agent roles, only one node (labeled user agent) could have been chosen to represent both in the dependency diagram. However, we separated the two roles to make it easier to understand the dependencies.

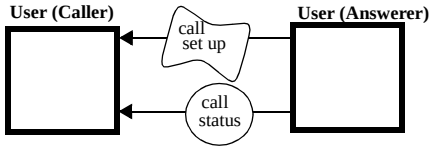

Figure 9: Dependency diagram

5.2.2 Jurisdictional Diagram

The jurisdictional diagram describes the organization of agents in terms of their authority status (relates superior and subordinate agents). The agents are placed in the jurisdictional hierarchy with the agent that has the highest jurisdiction at the top. Agents use their jurisdictional relationships to allocate authorities and delegate policies (permissions and restrictions). The jurisdictional hierarchy is derived from user requirements. User requirements include UCMs that capture management decisions and agent roles, as well as from the requirement prose. When a UCM path segment is identified as a jurisdictional relationship, both the superior and subordinate agents are also identified. The identification of the jurisdictional relationship effects how agents communicate with each other. For example, a subordinate agent should not reject a request from a superior agent.

Since UCMs, by definition, do not show interactions between components, a designer must identify if a path segment connecting two agents represents a direct or indirect relationship. In case of direct relationship, the two agents coordinate their activities while in case of indirect relationship, there is another agent that facilitates the coordination. If the relationship is identified to be indirect and the coordinating agent happens to be the superior of the two agents in the UCM, then the path segment is identified as a jurisdictional relationship and the three agents are placed in the jurisdictional model.

Agents play different roles in an organization. The rules under which an agent operates may change depending on the role it plays in an organization. For example, a superior agent may disallow call waiting for a subordinate agent when the subordinate agent is playing the role of a help desk attendant. Such roles are captured in UCMs by *slots*. The identification of roles and the agents that can fill them allows us to create a jurisdictional hierarchy that takes into consideration all the roles agents can play.

The jurisdictional hierarchy has a number of fundamental properties that are useful for agent systems. In particular, it:

- guides agents to find each other.
- helps the selection of strategies (a superior agent provides policies for lower agents).
- aids agents in resolving their conflicts. In case of conflict, an agent may try, using its authority status, to resolve a conflict.
- provides agents with an organizational mechanism for informing each other about modification, creation, or removal of system resources.

Figure 10 shows an example of an organization of agents. Usually the non-leaf nodes are management agents while the leaves are actor agents. The diagram was created from the UCM in Figure 6. Recall that a designer examines all inter-agent relationships captured in UCMs and classifies each as either a dependency or a jurisdictional relationship. When a jurisdictional relationship is identified, both the superior and subordinate agents are also identified.

For example, the path segments that connect PDManager, ProdDevelopment, Division, and Enterprise agents are identified as jurisdictional relationships. The agents that participated in these jurisdictional relationships are placed in the jurisdictional diagram with the agent that has the highest jurisdiction at the top. The jurisdictional rela-

tionships between the PDManager, ProdDevelopment, Division, and Enterprise agents are captured in the jurisdictional model by the hierarchy of agents of the extreme left side of the diagram shown in Figure 10.

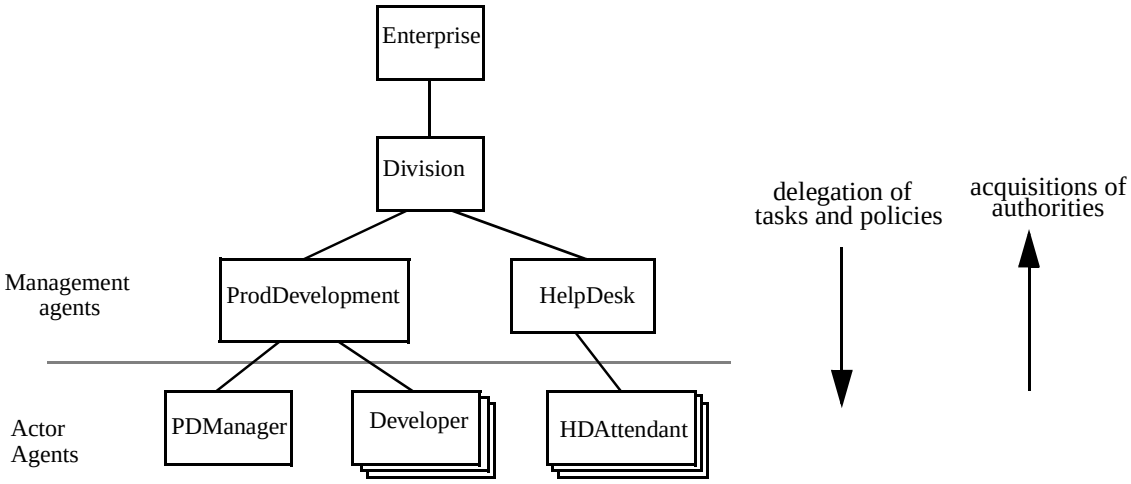

Figure 10: Jurisdictional model for an organization

5.3 The Conversational Model

The purpose of the conversational model is to identify what messages are exchanged in order for the agents to cooperate and negotiate with each other. The conversational model is derived from the agent relationship model and the internal agent model. The conversational model basically identifies what messages are exchanged to fulfill the dependencies and the jurisdictional relationships identified in the agent relationship model. The content of the conversational messages is determined by the plans that satisfy the dependencies which are captured in the internal agent model.

The model is described in a tabular format (a table for each agent). The table has three columns: *receive*, *send* and *comment*. The *receive* column lists the messages received by the agent. The *send* column list *all* possible responses to each received message. The *comment* column contains a textual explanation. If needed, the sender of the received message and the receiver of the sent message can be identified by prefixing the messages. Such prefixing need not to be included in the model if both the senders and receivers of messages are of the same type.

Agents must cooperate and negotiate with each other. When agents cooperate, they engage in conversations that can be represented by speech acts [14]. The identification of the relationship type helps in choosing the right conversational protocol. Each type of relationship has a set of predefined conversational messages associated with it:

- A negotiated dependency has four types of conversational messages associated with it: *Prop*, *CProp*, *ACCEPT*, and *REJECT*. Together, they implement a generic agent negotiation mechanism. In our system, an agent that wants to negotiate with another agent sends a proposal (Prop message) and waits for a response. The responses the agent can get for its proposal can be a counter proposal (CProp), proposal acceptance (ACCEPT), or proposal rejection (REJECT). If an agent gets a proposal or a counter proposal, then it needs to evaluate the proposal and send a response back.
- A resource dependency is associated with *Request* and *Inform* messages. The request message is sent when an agent wants to request a resource (informational or physical) and the inform message is sent when an agent wants to make a statement to another agent. For example, an inform message may contain a response to a previous request or just a status message.
- A goal dependency has an *Achieve* message associated with it. The achieve message takes a goal name or a condition to be achieved as one of its parameters.
- A task dependency has one *Execute* message associated with it. The message parameters specify what task to execute.
- A jurisdictional relationship has two messages associated with it: *Ask* and *Tell*. The ask message is sent from a subordinate to a superior and the tell message is sent from a superior to a subordinate. A subordinate agent cannot reject a request from a superior agent.

The conversational messages are summarized in Table 2.

Table 2: Conversational messages for the different relationships

Agent Relationship	Messages	Comment
Negotiated dependency	Prop CProp ACCEPT REJECT	A proposal message A counter proposal Proposal acceptance Proposal rejection
Task dependency	Execute	Execute a task
Resource dependency	Request Inform	Request a resource Supplier informs a dependent
Goal dependency	Achieve	Achieve a goal
Jurisdictional relationship	Tell Ask	From superior to subordinate. From subordinate to superior.

The list of the predefined messages is by no means a comprehensive one. A designer can add and associate new messages with the different relations as needed.

It may happen that two agents need to go through a mediator or a controller in order for them to communicate. This can be shown by explicitly including all the messages exchanged with the third agent in the conversational model.

5.3.1 Deriving a Conversational Model for the Example

In the dependency diagram, the dependency between the caller and answerer user agents is identified as a negotiated dependency. Recall that a negotiated dependency has four types of conversational messages associated with it: Prop, CProp, ACCEPT, and REJECT (explained in Section 5.3). The plans captured in the internal agent model (Figure 7), in conjunction with the dependency identification, helped us to construct the conversational messages in Table 3. Here we only show an example of discovering the conversational messages for the negotiated dependency. Conversational messages for other dependencies can be discovered in the same way.

Table 3: Interagent conversational model for the example

	Receive	Send	Comment
1		Prop(:connectFrom a :connectTo b)	Originating
2		Prop(:connectFrom a :connectTo b)	OCS
3	Prop(:connectFrom a :connectTo b)	ACCEPT | REJECT	Terminating
4	Prop(:connectFrom a :connectTo b)	CProp(:connectFrom a :connectTo f)	CF
5	CProp(:connectFrom a :connectTo f)	Prop(:connectFrom a :connectTo f)	Originating

Line 1 and line 2 of the conversational model show what is being sent if either the originating plan (default plan, line 3 of the internal agent model) or the OCS plan (line 4 of the internal agent model) was selected. Note that the message generated by the OCS plan is listed even though it is possible that the plan may not send any messages if the destination is in the OCS list.

Line 3 shows that the terminating plan (for an answerer) is to respond with either accept or reject messages to a connection proposal from the caller.

Line 4 of the conversational model illustrates an agent (an answerer), who has subscribed to call forwarding, responding to a connection proposal. The response is a counter proposal recommending to connect to user agent *f* instead. User agent *f* is the designated agent for handling the calls of the receiver of the original proposal. Line 5 shows the response of the user agent to the counter proposal in line 4. The user agent accepts the counter proposal by initiating a new connection proposal to *f*. Note that *a*, *b* and *f* are formal parameters in this table.

5.4 The Contract Model

The term contract has been used in several research areas. A *contract net*, by Smith [17], is a collection of

nodes that cooperate in achieving goals which, together, satisfy some high level goal or task. The term contract in design was first popularized by Meyer [13]. Meyer used preconditions and postconditions in routines to bind routines and their callers. Holland et al. [10] also used contracts in design to specify obligations of objects. We propose a contract structure similar in its purpose to Holland's and Meyer's approaches. Smith's approach can be used at runtime to establish contracts between agents.

A contract specifies, in a high-level language, obligations and authorizations between the different agents about the services provided to each other. Our aim is to create an organization of agents, and a network of commitments between them. This organization facilitates resource sharing, agent cooperation and conflict resolution. This is done by instantiating agents with contracts in which agents commit certain resources (or services) to each other. Commitment here means that an agent will at a later time give access to its resources. An agent that commits some of its resources must take into account this commitment before the resources can be accessed by the agent itself or other agents.

The purpose of the contract model is to define the expectations of how agents can fulfill the dependencies defined by the dependency model as well as the expectations of agents when they play the roles defined by the jurisdictional model. Conversations are used as guidelines for discovering those expectations.

***AGENT A* [(*idA*)]**	**CONTRACT**	***AGENT B* [(*idB*)]**
Authorizations		
serviceA1() [cost *function*] ... *serviceAn()* [cost *function*]		*serviceB1()* [cost *function*] ... *serviceBm()* [cost *function*]
Obligations		
provide: *serviceA()* [cost *function*] use: *service()* (when *condition* | insteadOf *service()*)		provide: *serviceB()* [cost *function*] use: *service()* (when *condition* | insteadOf *service()*)
Policies		
condition1 [desirable] [--> *failAction*] ... *conditionN* [desirable] [--> *failAction*]		*condition1* [desirable] [--> *failAction*] ... *conditionM* [desirable] [--> *failAction*]
Beliefs		
belief1 [shared] ... *beliefN* [shared]		*belief1* [shared] ... *beliefM* [shared]

Figure 11: A contract template

A contract has five parts, as shown in Figure 11: participants, authorizations, obligations, beliefs, and policies. The *participants* part lists the contract participants. An optional unique id can be associated with each agent name which can be used in the contract to refer to the agent. The *authorizations* part specifies the services that agents make available to each other. An optional *cost* clause can be associated with each service. The cost clause includes a function that determines the cost of the service.

The *obligations* part specifies which services an agent must provide and/or use. There are two primitives used in this part. They are *provide* and *use*. The *provide* primitive identifies which services an agent must provide. An optional cost clause can be associated with provided services. The *use* primitive identifies which services an agent must use and under what circumstances. The behaviour of the participants can be specialized and extended through the *use* primitive. There are two keywords associated with the use primitive: *when* and *insteadOf*. The *when* keyword specifies a condition under which the service must be used. The *insteadOf* keyword allows a contract participant to enforce the use of a replacement service should the defaults not be satisfactory.

Whether a service belongs to the authorizations or the obligations section is clear. A service that is required by a contract participant becomes an obligation for the other participant. A service owned by an agent belongs to the authorizations part if the agent permits or requires the other contract participant to use it. Services listed in the authorizations part can be changed or become unavailable at any time while services listed in the obligation part cannot be

changed or dropped without the consultation with the other contract participant. This distinction allows agents to effectively manage their resources. For example, an email delivery agent that provides to other agents a speech generator service as an authorized service, may decide at any time and for any reason to not continue providing this service. The email delivery agent would continue fulfilling its responsibilities (e.g. delivering email) even though there is no speech generator service available.

For resources and services in a contract, the *policies* part specifies quality and capacity of services, information about their importance to an application, and their usage policy. Policies in a contract are either mandatory or desirable. The default is assumed to be mandatory. Defining importance permits agents to optimize various aspects of their services. For example, an agent that provides a printing service could easily manage its resources if it knew that quality of printing was not important. A fail action can be associated with the policy clause. The action is triggered whenever the policy clause is violated. For example, fail actions can be used to penalize agents or report their failure to other agents.

The *beliefs* part defines what beliefs needed for the contract and which agent has to maintain them. Beliefs are private to an agent, however the beliefs part can force and agent to share (disclose) a belief to another agent by marking the belief *shared* in a contract.

5.4.1 Deriving Contracts for the Example

A designer builds the contract model by examining the different relationships captured in the agent relationship model and deciding whether a contract is needed to capture the relationship commitments. If a contract is needed, the designer decides what authorizations, obligations, policies, and beliefs have to be captured in the contract.

Figure 12 shows an example of a contract. The contract is between the HelpDesk and HDAttendant agents. In this contract, the HelpDesk agent authorizes the HDAttendant to use three services for reasons which will be explained later. The HDAttendant is obligated to provide a connect() services which allows it to connect incoming calls. It is also obligated to use the reportCallStart() and reportCallEnd() services when respectively a call starts and ends. One of the contract's policy requirements is that HDAttendant is forbidden to receive any other incoming calls when it is busy serving a customer. The other desirable policy requirement is that the HDAttendant responds to calls in less than five rings. The policy clause states that when the agent fails to respond in less than five rings, the reportDelay() action is triggered. This action reports to the HelpDesk the violation of the policy clauses. The HelpDesk keeps track of the performance of each HDAttendant and may discontinue the services of a HDAttendant if it keeps violating the contract. The belief part captures the fact that the two agents need to keep track of a starting time and a time period for the commitment.

HelpDesk (A)	**CONTRACT**	HDAttendant (B)
Authorizations		
reportDelay() reportCallStart() reportCallEnd()		
Obligations		
		provide: connect() use: reportCallStart() when call starts use: reportCallEnd() when call ends
Policies		
		Disallow incoming calls when busy Connect in less than 5 rings desirable --> reportDelay()
Beliefs		
starting time time period		starting time time period

Figure 12: A contract between HelpDesk and HDAttendant

6.0 Conclusion

We described a methodology which was specially developed for agent systems. The methodology provides a systematic approach for generating implementable system definitions from high-level designs. The methodology captures effectively the complexity of agent systems, agents' internal structure, relationships, conversations, and commitments. The methodology contributes to the development of agent-oriented programming as a discipline. Our goal in realizing this methodology is to establish the first steps to broader research in agent methodologies.

The practicality of the approach was confirmed by designing a prototype for a special case of the telephony feature interaction problem. Our contributions provided guidelines that helped telecom software engineers evolve an agent based design. The systematic approach provided by the methodology led to an executable prototype where telephony features were implemented as rule engines communicating via a blackboard. Plans captured in the internal agent model were implemented as competing rule engines. Messages captured by the conversational model were implemented as tuples and finally, agents communicated by posting such tuples to a blackboard.

We are currently developing a tool to support this methodology. A novel aspect of the tool is its support for traceability between models. The tool supports systematic movements between models. The designer can trace the chain of derivations backward and forward from any part of any model. Also a change in any part of a model causes all effected parts to be highlighted. This aids designers in identifying places where changes should be made and ensuring that models remain consistent.

We hope this approach will help clients and designers communicate better about requirements, provide a systematic process for transforming requirements into metalevel agent logic (and hence into software implementation), and help with system evolution by providing a high level reference for making detailed changes.

Acknowledgments

We would like to acknowledge the valued insights and contributions of Ray Buhr, Tom Gray, Serge Mankovski and Michael Weiss.

References

[1] M. Barbuceanu, M.S. Fox, *COOL: A Language for Describing Coordination in Multi-Agent Systems*, In Proceedings of the International Conference on Multi-Agent Systems, San Francisco, CA, 1995.

[2] R.J.A. Buhr, *Use Case Maps for Attributing Behaviour to Architecture*, SCE-96-2: Contribution to the Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii.

[3] R.J.A. Buhr, R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996.

[4] R.J.A. Buhr, *Design Patterns at Different Scales*, presented at PLoP96, Allerton Park Illinois, Sep 96.

[5] R.J.A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, S. Mankovski, *Feature-Interaction Visualization and Resolution in an Agent Environment*, in the Fifth International Workshop on Feature Interactions (FIW98), Sweden, September 1998.

[6] R.J.A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, S. Mankovski, *High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example*, Third Conference on Practical Application of Intelligent Agents and Multi-Agents (PAAM'98), London, UK, March 1998.

[7] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, *A High Level Visual Notation for Understanding and Designing Collaborative, Adaptive Behaviour in Multi-agent Systems*, Hawaii International Conference on System Sciences (HICSS'98), Hawaii, January 1998.

[8] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, *Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example*, Hawaii International Conference on System Sciences (HICSS'98), Hawaii, January 1998.

[9] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, D. Pinard, *Understanding and Defining the Behaviour of Systems of Agents with Use Case Maps*, poster at Practical Applications of Intelligent Agents and Multi-Agents Conference PAAM '97, April 1997.

[10] I.M. Holland, *Specifying Reusable Components Using Contracts*, ECOOP '92, Utrecht, June 1992.

[11] E. Kendall, *A Methodology for Developing Agent Based Systems for Enterprise Integration*, IFIP Working Conference of TC5 Special Interest Group on Architectures for Enterprise Integration, Queensland, Australia, November 1995.

[12] D. Kinny, M. Georgeff, A. Rao, *A Methodology and Modelling Technique for Systems of BDI Agents*, Seventh european Workshop on Modelling Autonomous Agents in a Multi-Agent Worlds, January 1996, Eindhoven, The Netherlands.

[13] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, N.J., 1988.

[14] C. Numaoka, M. Tokoro, *Conversation among Situated Agents*, Proc. of the 10th International Workshop on Distributed Artificial Intelligence, 1990.

[15] J. Pan, M. Tanenenbaum, *An Intelligent Agent Framework for Enterprise integration*, IEEE Transactions on Systems, Man and Cybernetics, Vol. 21, No. 6, 1991.

[16] Y. Shoham, *Agent-Oriented Programming*, Artificial Intelligence, 60(1), pp 51-92, 1993.

[17] R.G. Smith, *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*, IEEE Transactions on Computers, Vol. 29, No. 12, pp. 1104-113, 1980.

[18] E.S. Yu, J. Mylopoulos, *Understanding 'Why' in Software Process Modeling, Analysis, and Design*, In Proc. of the 16th IEEE International Conference on Software Engineering, Sorrento, Italy, pp. 159-168, 1994.

[19] M. J. Wooldridge, *Agent-based Software Engineering*, In IEE on Software Engineering, 144(1), 26-37, February 1997.

[20] M. J. Wooldridge, N. R. Jennings, *Pitfalls of Agent-Oriented Development*, In Proc. 2nd Int. Conf. on Autonomous Agents (Agents-98), Minneapolis, USA, 385-391, 1998.