Understanding and Defining the Behaviour of Systems of Agents with Use Case Maps

R.J.A. Buhr[†], M. Elammari[†], T. Gray[‡], S. Mankovski[‡], D. Pinard[‡]
Carleton University[†] and Mitel Corporation[‡]
Ottawa, Canada

Abstract

Agents are appropriate for defining, creating, maintaining, and operating the software of distributed systems in a flexible manner, independent of service location and technology. However, we humans have difficulty understanding or defining how a system of agents works as a whole to accomplish some application purpose when the only models we have to work with are provided by a programming language or a CASE tool that supports a set of relatively low level software 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. Use case maps provide such a model. This paper outlines principles of applying use case maps to help with the process of discovering agents and their relationships, defining abstract system models of sets of collaborating agents, constructing implementable system definitions from the models, arranging for the agents to instantiate the systems from the definitions, and evolving the system models and definitions based on experience. A novel aspect of this approach is its ability to represent, in a manageable way, behaviour patterns involving many agents, as first-class entities that may be plugged in according to circumstances at run time. The principles are illustrated by applying them to an agent-based multimedia communications example.

1.0 Introduction

Systems of agents are complex in part because both the structural form and the behaviour patterns of the system change over time, with changing circumstances. By *structural form*, we mean the set of active agents and interagent relationships at a particular time. This form changes over time as a result of interagent negotiations that determine how to deal with new circumstances or events. We call such changing structural form *morphing*, by analogy with morphing in computer animation. By *behaviour patterns*, we mean the collaborative behaviour of a set of active agents in achieving some overall purpose. In this sense, behaviour patterns are properties of the whole system, above the level of the internal agent detail or of pairwise, interagent interactions. Descriptions of whole system behaviour patterns need to be above this level of detail to avoid becoming lost in the detail, because agents are, in general, large-grained system components with lots of internal detail, and because agents may engage in detailed sequences of interactions that easily obscure the big picture. In agent systems, behaviour patterns and morphing are inseparable, because they both occur on the same time scale, as part of normal operation.

Use case maps (UCMs) [5][6][7][8][9] are descriptions of large grained behaviour patterns in systems of collaborating large grained components. They not only rise above the kind of details mentioned above but also express morphing directly in a natural, and easily understandable manner.

Therefore, combining agents and use case maps is appropriate; in fact, they seem to be made for each other.

We are in the process of developing an agent framework which is intended to allow software interoperability across enterprise applications. The framework includes definitions of the structure of an agent and an agency for realizing interoperability. The framework provides for a dynamic matching of overall enterprise goals with enterprise resources through a negotiation process. This allows the decoupling of goals from and resources and hence allows services to be defined independently of the enterprise's specific resources which are used to accomplish the goals. This provides for a system which is evolvable for both new services and new technology. To accomplish this we have extended the well known BDI agent model [14] with some new concepts [1][20][21][22]. However, the manner in which we employ use case maps to define and understand our agent systems is not strongly dependent on these new concepts, so they will not be described in this paper. Although the examples of use case maps that we shall present will be implemented with these new concepts, to explain the use case maps we shall rely only on a few concepts from the standard BDI model.

1.1 Use Case Maps

Agents are large-scale causal entities in agent systems. Use case maps describe large scale causal sequences in systems of any kind. We exploit this strong match of properties to provide high level definitions and understanding of agent systems. This section summarizes the features of use case maps that make them suitable for this purpose.

1.1.1 Overview

We need diagrams to understand complex systems; indeed agent systems are no exception. By definition, an agent system is a set of collaborating agents, overall behaviour of which is not centered in any one agent. So, to understand or define how an agent system works as a whole, we need to keep the agents in the picture but rise above the level of agent-centric details. Use case maps help humans do this.

The diagrams that are seen in the agent literature (see Section 4.0) and that we ourselves use (in addition to use case maps) are *agent-centric*. By this we mean that relationships expressed in such diagrams are pairwise between agents, with the consequence that larger relationships involving many agents must be pieced together from pairwise relationships and from knowledge of the internal workings of agents. Examples of pairwise relationships are “negotiates with”, “inherits from”, and “has contract with”. Diagrams of pairwise relationships do not directly show the composite effect of combinations of pairwise relationships that extend over many agents, because the collaborative behaviour of the agents. The larger relationships are properties of sets of agents or of the system as a whole and are the *raison d’etre* for the system; leaving them to the detail, as agent-centric descriptions do, leaves only prose as the means of getting the big picture, which is unsatisfactory. Use case maps provide an independent way of standing back from such agent-centric views to get a diagrammatic view of the properties of an agent system as a whole.

Use case maps give a high level view of causal sequences in a system as a whole, as paths. The term “causal” simply means that the sequences are understood by *humans* to be caused by the stimulus, not necessarily that individual agents have any knowledge of this causality (although they may). Imagine tracing a path with your finger (see Figure 1) over a diagram of the structure of an agent system (a diagram showing the agents as peer and/or nested

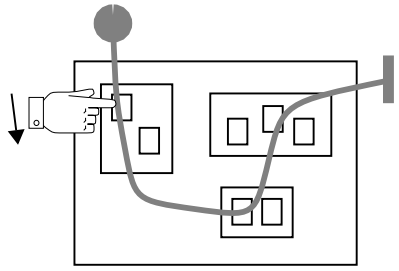


Figure 1: Example of a use case map.

boxes), to explain a causal sequence of events that might occur. A use case map is basically a record of paths traced by one or more such finger-pointing sequences. A filled circle indicates a start point of a path, meaning the point where stimuli occur that cause activity to start progressing along the path. A bar indicates an end point, meaning the point where the effect of stimuli are felt. Paths trace causal sequences between start and end points. Think of the agents and other components along the paths in a use case map as performers: they perform responsibilities at points where the paths touch the agents. In general, responsibilities are indicated by named points along paths, left out here. The basic assumption of use case maps 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. In Figure 1, if one replaced the pointing finger with a highlighter pen, the result would be analogous to a road map on which possible journeys have been highlighted.

The term *use case map* was inspired by Jacobson’s popularization of the concept of *use cases* for object-oriented software engineering [12]. A use case is a prose description of one or more related scenarios, which explains how a system works from a user perspective. The term is not restricted to human users interacting with system externals—in general, the term includes the possibility of scenarios involving many systems or subsystems that are “users” of each other. Use cases may, in general, have many related scenarios. Every use case has at least one. We often use the term scenario instead of use case, understanding that we mean one of the scenarios associated with a use case. If there is only one, the terms mean the same thing. In general, we use the term scenario to mean a description of either

required behaviour for a system not yet constructed, or of observed activity along a use case path for an actual running system.

1.1.2 Plug-ins

Morphing may be shown in use case maps by two means. The first means uses *stubs* and *plug-ins.* Plug-ins describe subpaths that are shown only as stubs in main UCMs. Plug-ins are particularly powerful for indicating morphing, as suggested by the diagram in Figure 2. This diagram shows the existence of variant plug-ins with different paths and agents, for the same stub. The concept is that the choice of a plug-in is determined by the system state at the time a scenario -- along the main path -- enters the stub. Plug-ins may themselves have stubs, allowing UCMs to be recursively decomposed (see Section 3.2 for examples). In relation to BDI models of agent systems, stubs and plug-ins may be thought of as extending the idea of *desires* (stubs) and *intentions* (plug-ins).

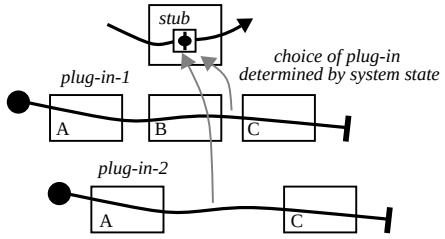


Figure 2: Morphing by means of stubs and plug-ins.

1.1.3 Causal Sequence

That UCMs show causal sequences, not control sequences, and are particularly powerful for expressing morphing with plug-ins because they enable separation of the description of a causal sequence started by an agent from the description of how the agent controls the sequence. That the agent controls a plug-in sequence is indicated on the UCM, by positioning the stub inside the agent. The details of how the control is achieved are deferred. Diagrams of control structures are component-centric ones that are needed eventually but that commit to too much detail for the level of abstraction we seek here.

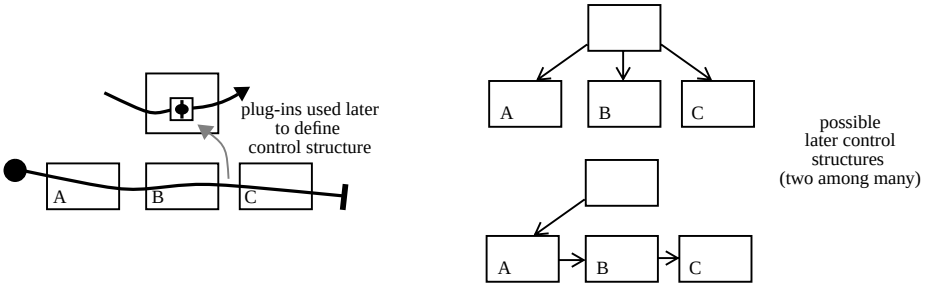


Figure 3: UCMs show causal sequence.

Plug-ins can be kept simple because the paths describing them do not have to be shown continually returning to the controlling agent to indicate return of control (Figure 4). Instead, causal sequences are shown independently of the controlling agent (Figure 3) and only later used to define control structures for the controlling agent.

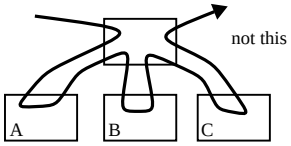


Figure 4: Causal sequences are not control sequences.

1.1.4 Composing Behaviour with Plug-ins and Slots

The second means of showing morphing is with *slots*. Slots are organizational places that components may enter dynamically (like positions to be filled by people in human organizations). Slots are represented diagrammatically by component boxes with dashed outlines. Slots and plug-ins may be used together as shown in Figure 5. This diagram shows a resource proxy object moving from a pool in a service-provider agent to a slot in some other agent that needs access to the resource. In an alternative plug-in, the service provider moves the resource proxy object to a local slot (in itself), which the user may access only indirectly. The different plug-ins may be selected under different circumstances, for example, to achieve a certain quality or capacity of service, dictated by the supplying agent's *beliefs* (another BDI term) about how to satisfy requirements for service.

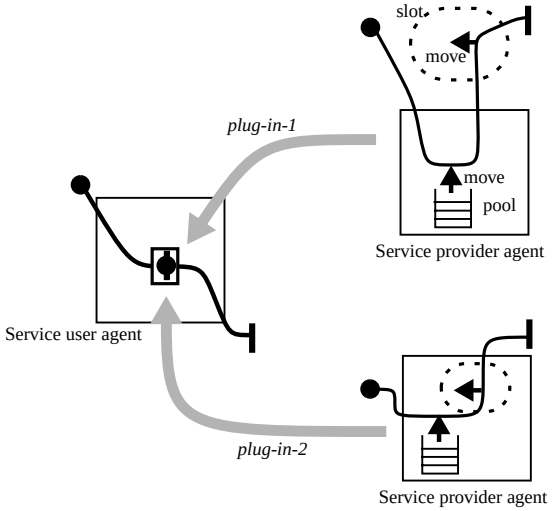


Figure 5: An agent may compose its behaviour from plug-ins (typically many).

1.1.5 Agents and UCMs

We can now see how the agent concept of autonomy is strongly related to the use of causal modelling in UCMs. In agent terms, autonomy means that an agent is solely responsible for fulfilling its role. Agents through their autonomy are required to generate appropriate behaviour to fulfil their responsibilities. This requirement can be shown in UCMs by an number of means: ordinary responsibilities along paths (see Section 3.2); and by means of stubs and plug-ins (illustrated in Figure 5). A stub (loaded with both a required capacity and quality of service) indicates that an agent will be responsible for generating the behaviour. In our system, agents will negotiate with each other to match the stub (extension of the concept of desire) with an appropriate plug-in (extension of the concept of intention) to realize it. Plug-ins represent alternative behaviours which can stably fulfil the requirements of the stub. There may be a variety of plug-ins, each of which will be more or less suited to matching the quality and capacity of service indicated by the stub.

2.0 Principles of Applying Use Case Maps to Agent Systems

We are developing an approach to applying use case maps to agent systems that has the following elements:

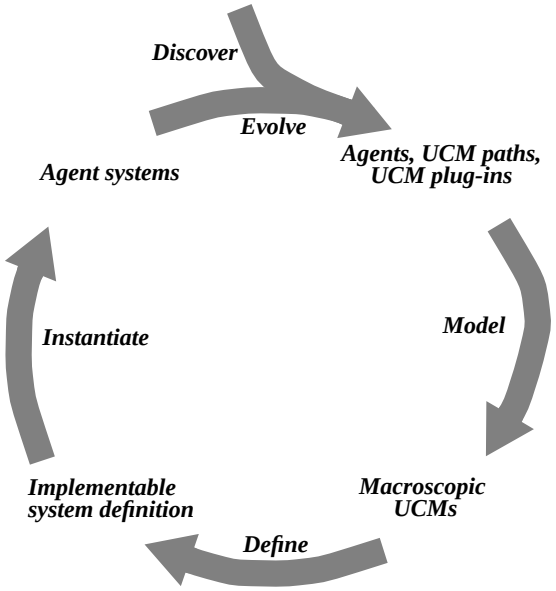


Figure 6: Phases of our approach.

Discover

Discover is an exploratory phase that precedes UCM model definition. It is accomplished by tracing application scenarios that describe functional behaviour as UCM paths through the system, discovering agents and plug-ins along the way. Generally, one starts with some black-box 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 quite sketchy ideas about both use cases and agents. For example, UCMs may be used to discover agents to realize paths that represent black-box use cases, or to discover paths that have no expression yet as black-box use cases (which amounts to discovering new use cases or variations of existing ones). In general, discovering black-box use cases and discovering agents and UCM paths inside the system black box should be interrelated activities, because this leads to the most helpful set of use cases; otherwise, many use cases may be developed that give little additional insight into how the system should be organized.

Model

The ultimate result of the discovery process is a UCM model of the system that, in diagrammatic form, superimposes causal paths for scenarios on a structural substrate of agents. This model has the following aspects:

- Associated documentation defines aspects of the model such as preconditions and postconditions of scenarios along paths, responsibilities of agents along paths, interresponsibility data flow along paths, and system state changes caused by the performance of responsibilities.
- The model describes macroscopic behaviour at the level of collaborating user agents that achieve some overall system purpose (e.g., internal telephone call, external telephone call, in the example of Section 3.0). UCMs at this level include interagent negotiations required to reach mutual agreement for completing some portion of the overall purpose, but to defer variant details to plug-ins.
- The model includes a catalogue of diagrams of plug-ins with associated documentation of where and under what system conditions they may be plugged in.
- Because the UCM model is a causal model, it is above the level of details such as interfaces, methods, and messages. The UCM model is formal in its diagrammatic structure but, as presented in this paper, informal in its associated documentation.

Define

The UCM model, supplemented by other information, is used to generate an implementable system definition. This is where our extensions to the BDI model enter the picture in an important way. In our agent system, such a matching is represented by an understanding which is a prelude and guide to negotiation. Through negotiation a contract is established in which the objects required to fulfil the understanding are supplied to the motivated agent by an agent with appropriate authority. The details are beyond the scope of this paper.

Instantiate

Instantiation may occur at different times:

- At *system initialization*, agents use the system definition to instantiate an initial system with some fixed agents, appropriately initialized. For example, pools of "resource proxies" will be created within fixed service-provider agents.
- When a human user *logs in*, the initial agents use the system definition to instantiate the user's agent structure. This agent structure includes a user agent for the human user, but may also include other agents.
- At any time, agents may use the definitions derived from the plug-ins to negotiate interagent relationships to instantiate appropriate localized behaviour patterns.

Evolve

Operational experience may lead to discovery by humans of new agents, new plug-ins, and new ways of using existing plug-ins, and to consequential revision by humans of the UCM model and by machines of the consequential system definitions.

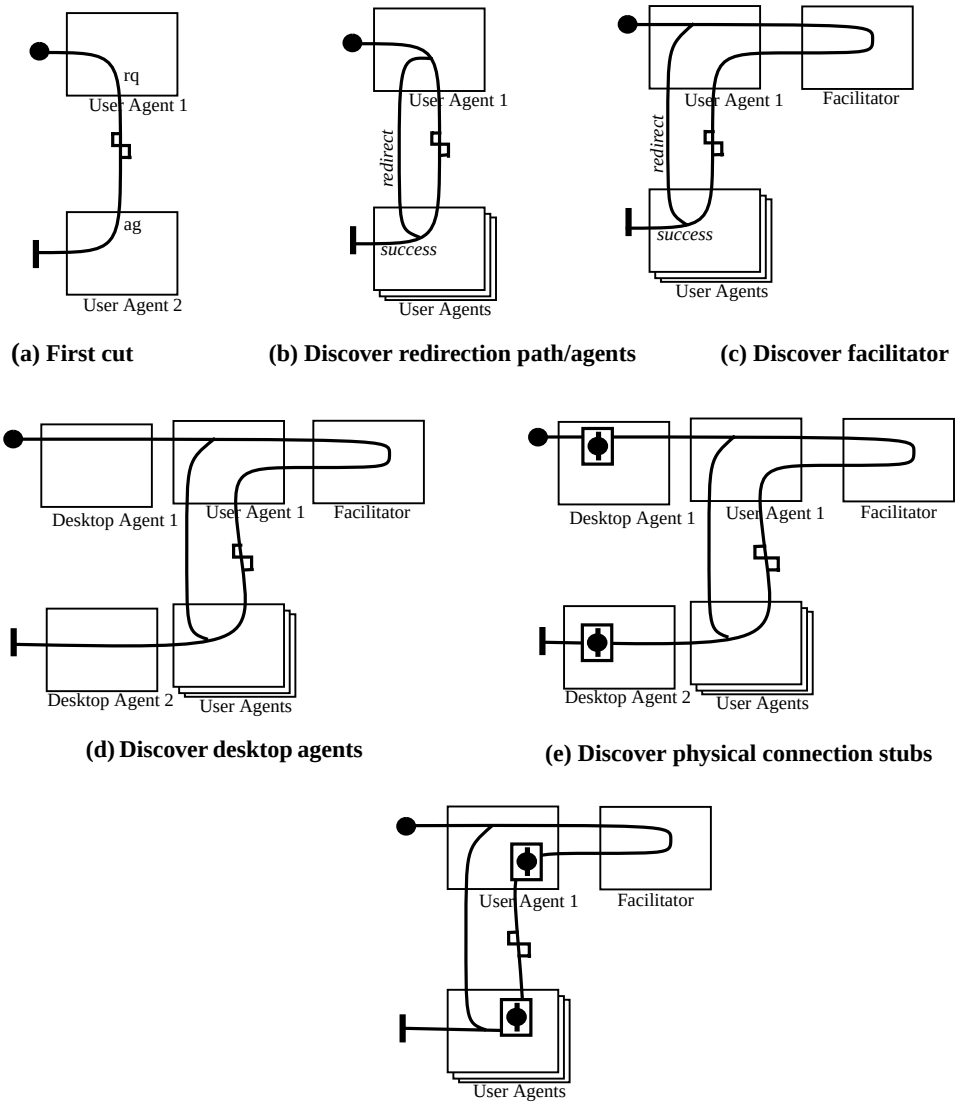
3.0 A Multimedia Communications Application

This section applies the above approach to a multimedia application for an intranet, which combines voice/data and interactive/noninteractive communication. This application has several different types of agents. User agents are the internal representatives of external users in the application environment. Other types of agents enable the user agents to do their work, for example, resource agents that provide access to available physical resources, enable abstract device agents that coordinate actions of physical devices. From this application, examples will now be provided of the discover and model elements of the approach. Details of the define, instantiate, and evolve elements are outside the scope of this paper.

3.1 Discover

We start by drawing UCMs that describe broad system behaviours. We keep discovering new agents and UCM features until we have discovered a good enough UCM model of the system as a whole (we say "good enough" because, by definition, UCM models can only be "complete" in the sense that they describe all the paths and agents we need to know about at the scale and level of abstraction of the model). Figure 7 illustrates an example of a discovery sequence (a)-(f) for our multimedia application. What we are trying to discover here is the means by which a suitable set of agents may establish a physical communications capability between two human users of the agent system. We begin only with the idea that a user agent for each human user will be central to the process.

- (a) identifies the fact that the two user agents must negotiate to accomplish the objective. A precondition of the path as a whole is that a communication capability is required, a postcondition is that the capability is established. The responsibilities *rq (request)* and *ag (agree)* are two halves of a shared responsibility to agree on the details of a request originating in the first agent. The symbol on the path between the responsibilities is a UCM notation to indicate that negotiation of the details may involve back and forth interactions -- not shown in the map -- which are interleaved with the performance of the responsibilities, and so are inseparable from them. In other words, the responsibilities are shared through detailed interactions not shown. In general, UCMs do not show detailed interactions, so the symbol may seem redundant. However, it is not, because it distinguishes the case where the interactions are interleaved with the responsibilities from the case where they are just needed to communicate between agents. For simplicity, *rq* and *ag* are left out of the remaining parts of the figure.



(a) First cut **(b) Discover redirection path/agents** **(c) Discover facilitator**

(d) Discover desktop agents **(e) Discover physical connection stubs**

(f) Hide desktops in resource allocation stubs

Figure 7: Discovering paths and components jointly.

- (b) takes into account the possibility of redirecting communication to a third user agent, for example, for call forwarding. The italicized label *redirect* identifies path a path segment, not a responsibility. This diagram shows a stack of agents occupying the position formerly occupied only by agent 2, indicating the existence of a set of possible "redirected-to" user agents. Showing user agents (or any component) in a stack like this means that each is distinct, all are operationally identical from the perspective of a traversing path, but only one is selected by path context. In this particular diagram, the implication is that the source user agent starts negotiation all over again with a "redirected-to" user agent (how it discovers where to find this agent is left as a detail).
- (c) adds a facilitator agent to help user agents discover the "redirected-to" agents by storing and providing the addresses of other user agents.
- (d) adds desktop agents to be responsible for controlling and manipulating attached devices such as a personal computer or a USB (Universal Serial Bus) phone—a phone that may be attached to a personal computer—thus keeping these responsibilities separate from user agents. Desktop agents provide for different modes of working, for example, working at home versus working in the office.

- (e) adds stubs where finer grained, possibly state-dependent, behaviour patterns may be plugged in to handle variant modes of physical communication (such as calling through a view window on the desktop or doing it through a phone). Appropriate plug-ins are described in Section 3.2.
- (f) hides everything associated with desktop agents in a stub in the main path traversing the user agent, to provide for the possibility that an internal agent with no corresponding external human user may participate in telephone calls in a way that does not need a desktop agent, for example, as in a speech-enabled expert system coordinated by an agent.

This is enough to give a sense of the discovery process. We shall now follow up in more detail the concept, in part (f) of the above figure, of the main decisions being made in resource allocation stubs in the user agents on the caller and answerer sides, leaving out the facilitator and redirect paths as peripheral matters in relation to resource allocation, and pushing down desktop agents and other agents that handle more detailed matters into stubs. In doing so, we shall complete the discovery process by discovering more agents and more stubs.

3.2 Models

The following figures, taken together, provide an example of a UCM model to describe successful connection of a telephone call. This is a specific example of establishing a physical communications capability between two human users of an agent system, as in Section 3.1. Here we omit some nuances that were developed in Section 3.1 in favour of focusing on some other aspects that were deferred there.

The following guidelines were used in developing the paths of this model. Position the start point of a path where the ultimate starting event occurs (outside the boxes the path traverses, unless one of the boxes is itself the ultimate source of the start event), position a responsibility in the component that performs it, position a stub in the component that controls the stub sequence, and position the end point of a path where the postconditions are felt (outside the boxes that the path traverses, unless the only postcondition is an internal change of state in some component that will only be sensed by a different path). The choice between identifying functionality at points along paths by responsibilities or stubs is determined by whether or not details can be deferred to the level of messages, or should be expanded further in UCM form. When starting from a blank sheet, these are matters to be determined by factors such as overall consistency of abstraction across the model, ease of understanding of the model, and distinguishing variant from fixed behaviour.

Figure 8 shows paths for two scenarios, CALL and ANSWER, each initiated by a different human user at a different physical location. In this figure, the focus is placed on the high level activities by using stubs to identify fine-grained activities and by indicating places where possible alternate state-dependent behaviour patterns may be plugged in to handle details of physical communication.

The precondition of the CALL scenario is that a human wants to place a call. The scenario's path begins with the OH1 (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 caller's user agent negotiates with the answerer's user agent to establish call parameters, shown by the pair of shared responsibilities, *rq (request)* and *ag (agree)*. The path leads to the AR (allocate resources) stub which allocates the required resources for the call, e.g. reserves a virtual communication channel. The resource allocation activities are put in a plug-in to allow for the possibility of different AR plug-ins for different conditions (for example, calling via a video display window may require allocation of video resources). The path is then AND-forked into two paths (the bar with two paths emerging). This one-to-many forking of the path is called an AND fork because it indicates that the emerging paths are logically concurrent. Generally, we shall refer to any one path emerging from an AND fork, simply as a fork. The first fork leads to the RNG (ring) stub which notifies the call recipient, for example by ringing a phone device. The second fork leads to the STAT stub at the other end by means of which the caller is notified of call progress. The postcondition of this scenario is that the answerer's phone is ringing and caller hears ringback.

The precondition of the ANSWER scenario, which starts at the answerer side, is that the phone is ringing and the postcondition is that the call is connected. This scenario has two stubs. The first is the OH2 (answerer offhook) stub, which establishes its end of the connection, and the second is the same STAT stub as before, which notifies the caller of call progress.

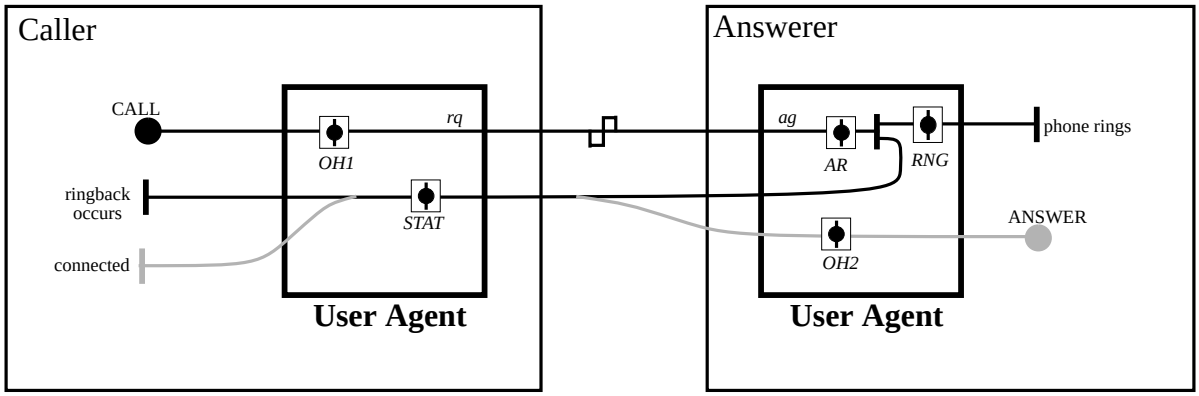


Figure 8: Call connection UCM.

Precondition:

CALL. Some human wants to place a call.

ANSWER. Answerer off-hook.

Postcondition:

CALL. At answerer end, phone rings

CALL. At caller end, ringback occurs

ANSWER. Call is connected.

Responsibilities:

rq. request

ag. agree

Stubs:

OH1. do offhook processing at caller end (different plug-ins if via a phone or a call view on a desktop)

AR. allocate resources (e.g. virtual communication channel)

RNG. ring attached devices

STAT. inform caller of call status

OH2. do offhook processing at answerer end (different plug-ins if via a phone or a call view on a desktop)

In the following figures, the phone agent detects phone hook signals and dialling digits, and provides a dial tone and ringing signals. The digits validation agent ensures the correctness of the dialled number. The connect agent is responsible for the allocation of virtual communication channels and establishes user-to-user connections. The view agent creates and maintains call views. The desktop agent is an abstract device agent which coordinates all of the resource agents.

Figure 9 shows plug-ins for the stubs of the main path, namely OH1, AR, RNG and STAT. In these plug-ins, the end point where the main path continues is labelled *continue*. The plug-ins are as follows:

- The OH1 plug-in shows the details of off-hook processing at the caller's end. The plug-in begins with the phone agent detecting the phone is off-hook and then AND-forking into two paths. One fork causes the view agent to create an active call view (a window) to display call progress information, shown in figure by a plus sign and small arrow into the path; the newly created view is moved into the active call view slot, shown by the arrow leaving the path into the active call view slot. The other fork begins by performing the *digtone* responsibility. The *digtone* responsibility generates dial tone to indicate the phone is ready to receive messages, turns off dial tone when the first digit is received, and collects digits. After the dialled number is collected, it is validated by the digit validation agent. If the number is valid, the path AND-forks into two paths, one updates the call view by displaying the dialled number, the other continues to the connect agent. The connect agent reserves a virtual channel which will be used to establish communication with the call recipient. Observe that, because there are several AND forks, only one of which is labelled *continue*, this means the main path can continue before all the forks complete.
- The STAT plug-in reports the status of call. It is shared by the CALL and ANSWER scenarios. At the beginning of the STAT plug-in, three concurrent paths are AND-forked. The top two forks are followed by the CALL scenario. All three forks are followed by the ANSWER scenario. The top fork turns on or off the ring back tone

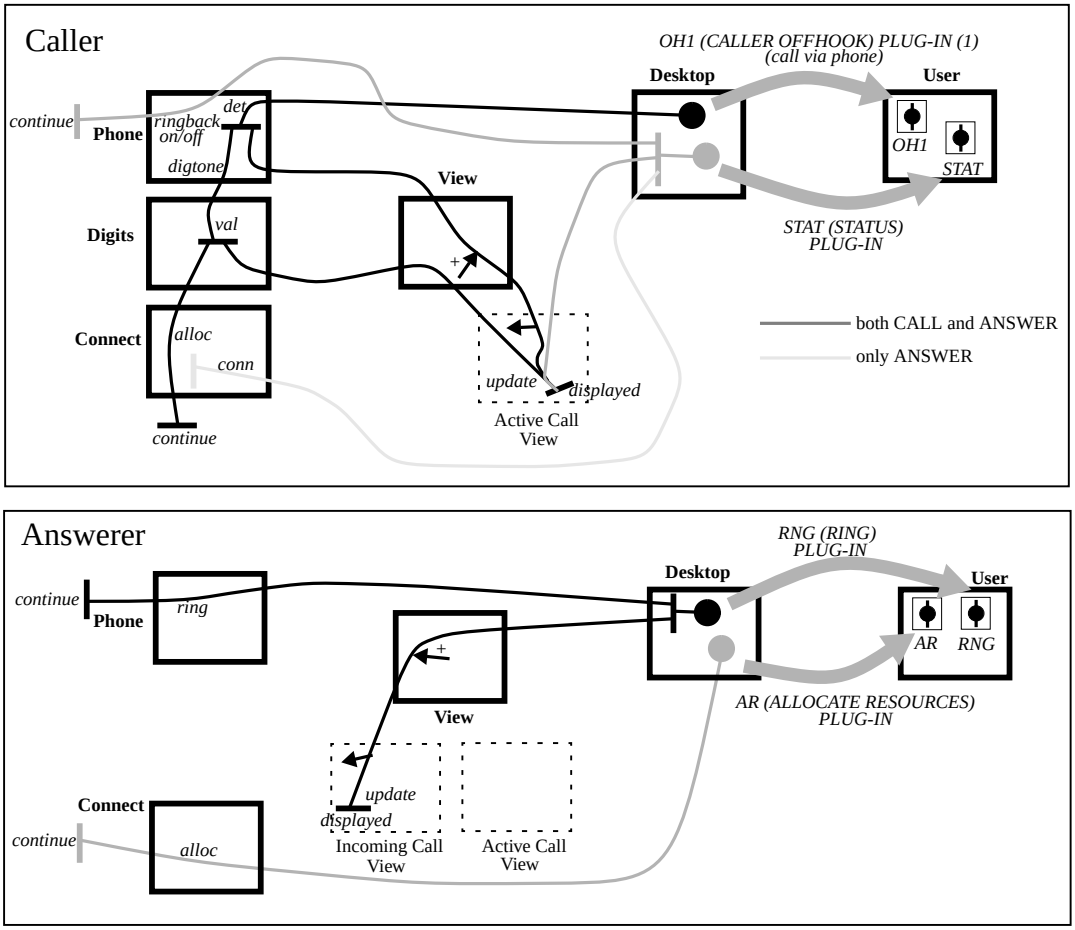


Figure 9: Caller and answerer plug-ins.

Precondition:	
OH1. Some human wants to place a call	STAT. There is call status to report
AR. A call has been requested	RNG. Resources have been allocated at answerer end
Postcondition:	
OH1. A phone number is collected	STAT. Caller notified of call status
AR. Virtual channel allocated	RNG. Phone is ringing
Responsibilities:	
det. detect phone is off hook	*digtone.* collect digits and manipulate phone tone
update. display information about current call status	*val.* validate phone number
alloc. allocate virtual channel	*ringback on/off.* turn on/off ringback signals
conn. connect virtual channel	*ring.* ring phone

depending on the status, either ringing or offhook, of the answerer's phone. The middle fork updates the active call view by displaying information about the current call status. The bottom fork connects the previously allocated virtual channel, to enable physical communication to start.

- The AR plug-in, at the answerer's end, performs resource allocation. In this case, the allocation is quite simple—a virtual communication channel is reserved by the connect agent.

- The RNG plug-in notifies the answerer by ringing the phone and creating an incoming call view. This process begins by forking two concurrent paths. The first fork travels through the view agent, creating an incoming call view which is used to populate the incoming call view slot. The incoming call view is then updated with appropriate call information. The second fork simply causes the phone agent to ring the physical phone.

In Figure 10, the OH2 plug-in shows the details of off-hook processing at the answerer's end. The plug-in begins with the phone agent detecting phone is off-hook. As a result, the path is forked into two concurrent paths. The activities along the first fork are as follows: the view agent creates an active call view; the incoming call view is destroyed, shown in figure by a minus sign and small arrow into the path; and the newly created component is moved into the active call view slot. The activities along the second fork are as follows: the phone agent stops the ringing of the physical phone and the connect agent establishes user-to-user connection.

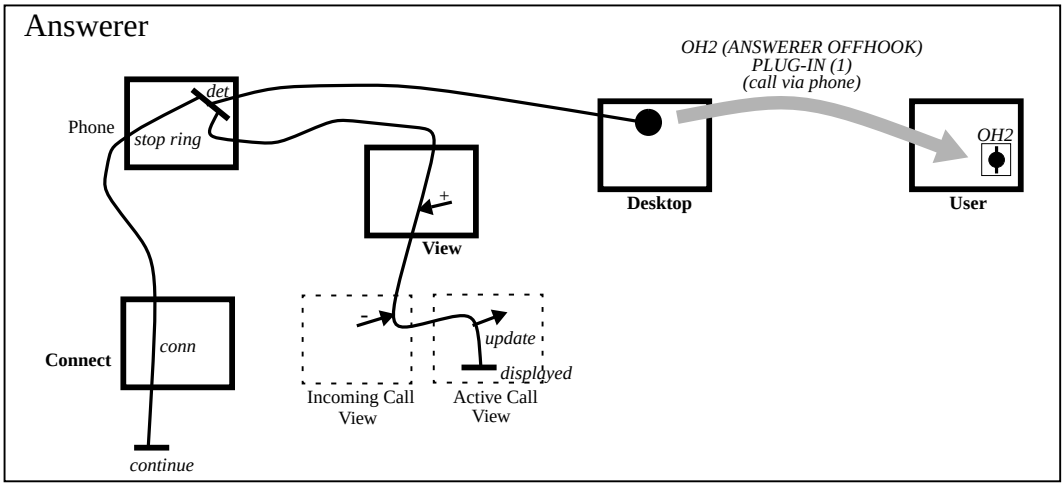


Figure 10: Answerer offhook plug-in.

Precondition: A human wants to answer call.

Postcondition: Answerer is connected

Responsibilities:

det. detect phone is off hook

stop ring. stop ringing

conn. connect virtual channel

update. display information about current call status.

Figure 11 shows two alternate plug-ins for the OH1 and OH2 stubs. Both plug-ins are assumed to be started via a view. The responsibilities of these alternate plug-ins are similar to those in the above figures with the exception of the *hands free* responsibility. The phone agent performs this responsibility in both plug-ins to put the physical phone in hands free mode.

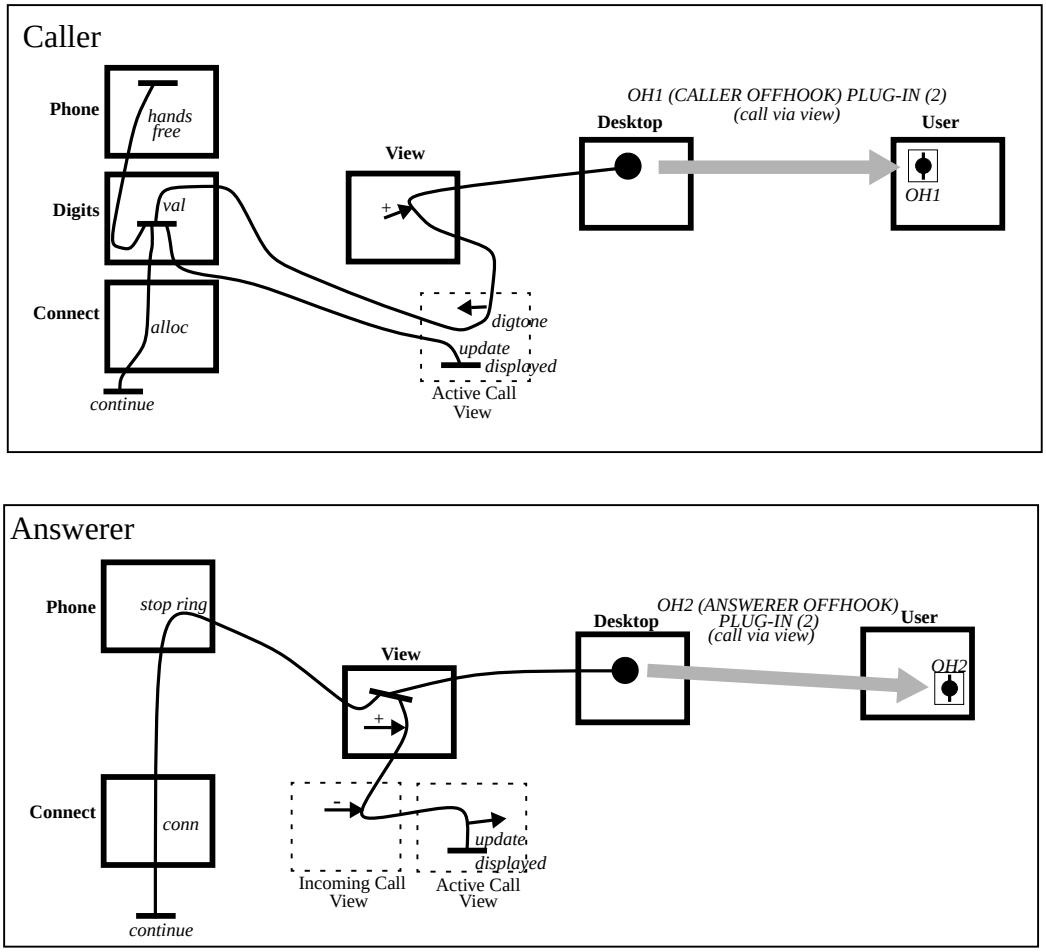


Figure 11: Caller and answerer alternate offhook plug-ins.

Precondition:	
OH1. Some human wants to place a call.	OH2. A human wants to answer call.
Postcondition:	
OH1. A phone number is collected	OH2. Answerer is connected
Responsibilities:	
digtone. collect digits and manipulate phone tone	*val.* validate phone number
update. display information about current call status	*hands free.* put phone in hands free mode
alloc. allocate virtual channel	*stop ring.* stop ringing and put in hands free mode
conn. connect virtual channel	

4.0 Discussion

The visual technique we have used in this paper (use case maps) is, to the authors' knowledge, the only one that exists for system description at this level of abstraction. In both the agent community and other communities, such as the object-oriented community, people use more detailed visual techniques to describe systems. For example, Kendall [13] models agent systems using object-oriented-style diagrams that require commitment to agent-centric details (e.g., pairwise interactions via messages, pairwise inheritance relationships). There are many tools in the object-oriented

community that support modelling of systems at this level of detail (for example, [4][17]). In the agent community, the Clearlake tool [11] specifically supports design of agent systems at a similar level of detail. From the perspective of this paper, workflow models (e.g., as discussed and applied by Kendall [13]) are a variation of data flow models. As Kendall points out, there are nuances that make them more useful for agent systems, but the following two basic properties of data flow models remain: they do not provide any easy means of describing one of the central features of agent systems, morphing; and their dependence on interagent workflows puts them in the category of agent-centric descriptions, a type of description that we have criticized earlier as being too committed to detail to give the big picture adequately. Otherwise, techniques in the agent community, such as COOL [2] and Shoham's AOP [19], represent agents formally with logic, with no visual representation. Visual representation is well known to be needed for human understanding. A suitable high level visual representation, such as use case maps, can be used as a starting point for generating more detailed visual descriptions and/or formal descriptions, as we propose here.

The path-centric view of systems provided by use case maps is helpful for high-level understanding and design. Component-centric views are still necessary for detailed design and implementation; use case maps enable us to stand back from such details. Figure 12 identifies these views as a part of making necessary paradigm shifts at increasing system scales. The horizontal steps indicate domains in which certain description techniques apply that are suitable for a particular scale. The vertical risers indicate paradigm shifts when the scale gets large enough that understanding in a particular domain is hindered by too much detail. Moving up a step in this diagram does not mean that the models of lower steps are replaced. Just as drawing DD diagrams does not replace code, but only gives a more abstract view of it, so drawing HLD diagrams like use case maps does not replace DD diagrams. In moving from the DD level to the HLD level, we are rising above the component-centric view that is necessary for detailed design and implementation.

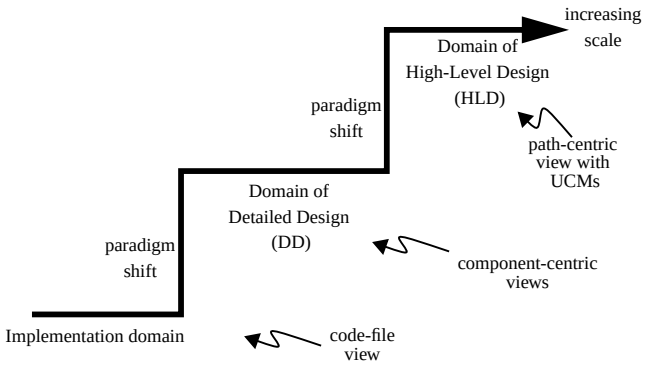


Figure 12: Paradigm shifts at different scales.

The properties of the UCM model that make the paradigm shift worthwhile are as follows:

1. *Has the primary objective of aiding human reasoning at a high level of abstraction, as opposed to entering details into a computer tool.* Use case maps are the only diagramming technique known to the authors that was shaped solely by the need for this property. Others popular techniques, e.g., [3][4][10][12][15][16][17], are at the DD level and are shaped primarily by the need for machine-executability of design models and/or machine-translatability into code, thus forcing a commitment to details at the level of methods, functions, messages, interprocess communication, interfaces, internal state machines of components, etc., that get in the way of reasoning at a high level of abstraction.

2. *Is first-class at the macroscopic level, meaning not dependent on details of components or code.* There is only one other notation that has this property, the so-called "high level message sequence charts" under development by the Z120 community [10] (this reference covers only detailed message sequence charts, but examples of proposed high level ones are given in [15]). However, this notation does not possess Property 3 and it clouds the mind's eye with boxes in the separate behaviour diagrams that look like components but are not, excacerbating the problem of mentally superimposing behaviour on structure.

3. *Combines system behaviour and system structure into a single coherent view.* To the authors's knowledge, only use case maps possess this property at a high level of abstraction. Other diagramming techniques at the DD level may attempt to do it by superimposing sequence numbers on connections in structural diagrams to indicate, say,

interobject or interprocess message sequences, but this requires many diagrams to present the big picture, thus clouding the mind's eye with details. Approaches that use separate diagrams, such as detailed or high level message sequence charts [10][12][15] cloud the big picture by forcing humans to combine diagrams in the mind's eye.

4. *Expresses "morphing" compactly, without requiring sequences of snapshot diagrams of changing structural forms.* To the authors' knowledge, only use case maps possess the property of representing morphing without sequences of snapshots. They do it by means of slots and plug-ins. Slots identify places where actual components may appear (to play local roles) or disappear, and by using paths as the loci for movement of actual components to or from slots. Plug-ins describe subpaths that are shown only as stubs in main UCMs. Plug-ins are powerful for indicating morphing. They are used to show the existence of variant plug-ins with different paths and agents, for the same stub. The concept is that the choice of plug-in is determined by the system state at the time a scenario along the main path enters the stub.
5. *Has diagrams that are easily grasped as visual patterns for a system as a whole.* Use case maps can combine many behaviour patterns in single diagram in a way that enables the mind's eye to sort them out. Recognizing behaviour patterns in DD diagrams by means of superimposed sequence numbers or separate detailed message sequence charts is much more difficult, particularly because many diagrams must be viewed.
6. *Provides a macroscopic system view for forward engineering, reverse engineering, maintenance, evolution, and reengineering.* Only use case maps and high level message sequence charts provide reference views that are independent of details and so can be used to guide decisions about details. Use case maps do it more compactly and simply (Properties 2-5).
7. *Can be saved for documentation and maintained without unreasonable effort.* The avoidance of commitment to details and the compactness of use case maps contributes to this property. However, tool support is desirable (a use case map editor is currently being developed for this purpose at Carleton).

A unique aspect of UCMs—arising from Properties 2-5—is the twofold bridge they provide between HLD structure and HLD behaviour on the one hand and between requirements and DD solutions on the other hand (Figure 13). This distinguishes UCMs from other scenario notations [10][12][15] and makes them particularly appropriate for our purpose here. This twofold bridge is possible because UCMs have two forms, bound and unbound. The

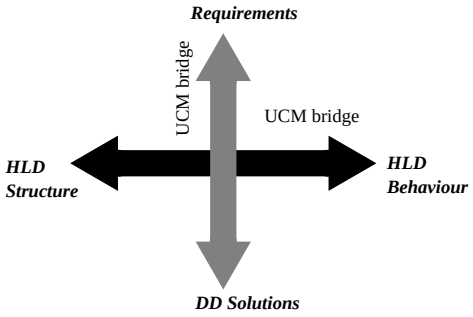


Figure 13: Bridging two gaps with UCMs.

bound form shows paths over structure, with labelled responsibility points along paths bound to components by visual superposition. This form combines structure and behaviour in compact visual form; we say that this form provides a seamless bridge between structure and behaviour at the HLD level. The unbound form shows only responsibilities, not components. This form provides a means of expressing behaviour requirements without necessarily committing to HLD solutions. Because the same paths may be used for both requirements and HLD solutions, we say that this form provides a seamless bridge between requirements and DD solutions.

5.0 Conclusions

We have described how use case maps may be employed to describe agent systems in a way that aligns well with the idea of agents as high-level causal entities. We have identified and illustrated discovery and modelling phases of agent systems with use case maps. We have identified further phases in the development of agent systems, namely

define and instantiate, that follow from our use case map descriptions, using extensions to the BDI model (not described in this paper). The approach described is original and unique. It contributes to the development of agent-oriented programming as a discipline.

References

[1] S. Abu Hakima, I. Ferguson, N. Stonelake, E. Bijam, R. Deadman, *A Help Desk Application for Sharing Resources Across High Speed Networks Using a Multi-agent network architecture*, Workshop of Distributed Information Networks, IJCAI 95, Montreal.

[2] 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.

[3] G. Booch, Object-Oriented Design, Benjamin/Cummings, 1994.

[4] G. Booch, J. Rumbaugh, *Unified Method for Object-Oriented Development*, Documentation Set, Version 0.8, Rational Software Corporation, 1995.

[5] R.J.A. Buhr, *Use Case Maps for Attributing Behaviour to Architecture*, SCE-96-2: November 5, 1996, Contribution to the Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii, http://ftp.sce.carleton.ca/UseCaseMaps/attributing.ps.

[6] R.J.A. Buhr, R.S. Casselman, T.W. Pearce, *Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework*, SCE 95-17, http://ftp.sce.carleton.ca/UseCaseMaps/dpwucm.ps.

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

[8] R.J.A. Buhr, *Design Patterns at Different Scales*, presented at PLoP96, Allerton Park Illinois, Sep 96. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/plop.ps.

[9] R.J.A. Buhr, A. Hubbard, *Use Case Maps for Engineering Real Time and Distributed Computer Systems: A Case Study of an ACE-Framework Application*, Hawaii International Conference on System Sciences, Jan 7-10, 1997, Wailea, Hawaii, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hicss-final-public.ps

[10] CCITT Recommendation Z120: Message Sequence Charts (MSC), undated document.

[11] Guideware Corporation, *The Implementation of Business Processes with Mobile Agents*, Technical Paper, Guideware Corporation, Mountain View, California, 1995.

[12] I. Jacobson et. al., *Object-Oriented Software Engineering (A Use Case Driven Approach)*. ACM Press, Addison-Wesley, 1992.

[13] 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.

[14] A. Rao, M. Georgeff, *BDI Agents from Theory to Practice*, Technical Note 56, AAII, April 1995.

[15] B. Regnell, M. Andersson, J. Bergstrand, *A Hierarchical Use Case Model with Graphical Representation*, Proc. ECBS96, IEEE Second International Symposium and Workshop on Engineering of Computer Based Systems, March 1996.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modelling and Design*, Prentice Hall 1991.

[17] B. Selic, G. Gullickson and P.T. Ward, *Real-time Object-Oriented Modeling*, Wiley, 1994.

[18] Shaw and Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

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

[20] M. Weiss, T. Gray, A. Diaz, *A Middleware for Developing Distributed Multimedia Applications*, Proceedings of the ISCA International Conference on Parallel and Disrupted Computing, Orlando 1995.

[21] M. Weiss, T. Gray, A. Diaz, *An Agent Based Distributed Multimedia Service Environment*, International Conference on Tools with AI, Herndon VA, 1995.

[22] M. Weiss, T. Gray, A. Diaz, *A Service Environment for Distributed Multimedia Applications*, Workshop of Distributed Information Networks, IJCAI 95, Montreal.