Designing and Specifying Mobility within the Multiagent Systems Engineering Methodology

Athie L. Self
Air Force Personnel Center
Randolph Air Force Base
San Antonio, TX
ajself@juno.com

Scott A. DeLoach
Department of Computing and Information Sciences
Kansas State University
234 Nichols Hall, Manhattan, KS 66506
sdeloach@cis.ksu.edu

ABSTRACT
Recently, researchers have created many platforms and applications for mobile agents; however, current Agent-Oriented Software Engineering (AOSE) methodologies have yet not fully integrated the unique properties of these mobile agents. This paper attempts to bridge the gap between current AOSE methodologies and mobile agent systems by incorporating mobility into the established Multiagent Systems Engineering (MaSE) methodology. We accomplished this by adding a move command to the MaSE analysis models and then defined the required transformations to incorporate the required functionality into the design. Finally, we translated the design models into Java-based agents that operate within a mobile agent environment.

1. Introduction
Dynamic agent systems have shown promise in solving certain problems such as finding services and information on the Internet and supporting mobile clients. These systems have shown an advantage in robustness and functionality over other client-server solutions, such as Remote Procedure Calls (RPC), messaging and sockets [3]. We define *dynamic agents* as agents that possess one of the following properties [7]:

- Cloning – the ability of an agent to create another instance of itself at the same or a different location
- Instantiation – the ability of an agent to create instances of another class of agent other than itself
- Mobility – the ability of an agent to move from machine to machine in a network

These three properties, or traits, have been the focus of new research in the distributed artificial intelligence arena with the property of mobility receiving the most attention.

While software engineering aspects of using multiagent systems are being addressed [16], [12] and many frameworks and applications have been developed [11], [2], [8], popular multiagent systems methodologies, such as GAIA [15], MESSAGE [9], and MaSE [6], do not currently provide support for mobile agents. The purpose of this paper is to describe how we incorporated mobility into the MaSE methodology [6], which is a general-purpose methodology for developing multiagent systems. To incorporate mobility into MaSE, we gave the designer the ability to use mobility to fulfill system goals.

SAC 2003, Melbourne, Florida, USA

The focus in this paper is strictly on the analysis and design phases of mobile multiagent systems. We assume that the agent determines when it is necessary to move. While other agents, or the agent platform itself, may advise the agent to move, the autonomous nature of agents allows the agent to determine whether it will actually move. We do not address issues such as system-generated moves that the agent cannot control due to shutdown, load balancing, etc. We also assume that an appropriate mobile agent platform handles the actual movement of the agent using a protocol similar to FIPA's Simple Migration Protocol [7]. In this protocol, an agent sends a request to its mobile agent platform, which terminates the agent and sends it to the destination platform where it is restarted. While the platform is responsible for movement, the agent is responsible for ensuring it restarts in the appropriate state.

The questions of determining where to move, representation of locations, determining which tasks and communications should be retained and which should not, etc. are also not explicitly discussed. The MaSE methodology provides models to allow the designer to specify these and other mobility related questions as he or she desires.

All examples in this paper were generated using the agentTool system [4], which is an automated development environment that supports the MaSE methodology. Each MASE model is specified graphically and semi-automated transformations allow agentTool to automatically generate designs based on role models and task diagrams [14], [13].

The remainder of this paper is structured as follows. Section 2 provides a quick overview of the MaSE methodology with the major focus being on the analysis and design phase models. Section 3 covers mobility specifications in the analysis phase. Section 4 presents how we capture mobility in the design phase. Finally, Section 5 summarizes the paper and provides possible areas for future research.

2. MaSE Methodology
Figure 1 presents a graphical overview of MaSE. MaSE consists of two phases and seven steps. For the purposes of this paper, we discuss only the output models of the analysis phase: role models and concurrent tasks.

The result of the MaSE analysis phase is a set of roles that agents will play, a set of tasks that define the behavior of specific roles, and a set of coordination protocols between those roles. A *role* is an abstract description of a function that an agent in the system must perform to meet system level goals [6]. Roles must exhibit certain behaviors to accomplish the goals assigned to them. We model these behaviors using a set of concurrent tasks. An example Role Diagram is shown in Figure 2. There are three roles (*Search Manager, Search Bidder* and *Searcher*), each with one associated task (*Fulfill Search Requests, Bid* and *Search)*. The *Contract Net* protocol is defined between the *Fulfill Search Requests* and *Bid* tasks, while the *SearchRequest* protocol is defined between the *Bid* and *Search* tasks. The Search Manager sends out bids for a search request. The winning Search Bidder then passes the request to the Searcher who returns the results of the search

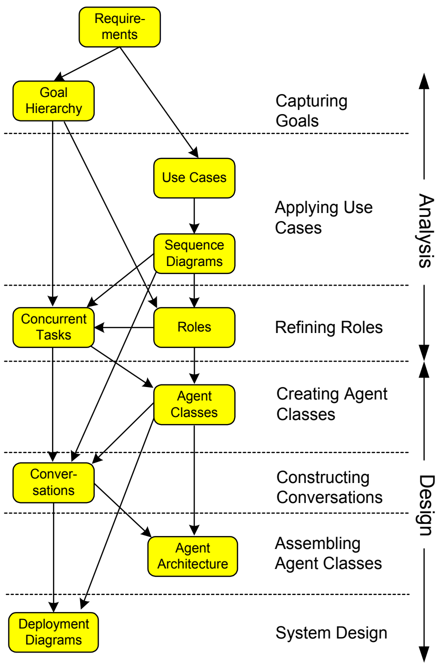


Figure 1. MaSE Methodology

Concurrent tasks are specified graphically using *Concurrent Task Diagrams* that are based on finite state automata [5]. Figure 3 describes the Search task from Figure 2. States within a concurrent task diagram represent the internal processing while transitions model the communication within the system and define valid state transformations. Activities within states model the internal processing that the roles perform. Thus, concurrent tasks define the behavior of agents by tying together the internal reasoning processes, interactions with other internal processes and external interactions with other agents.

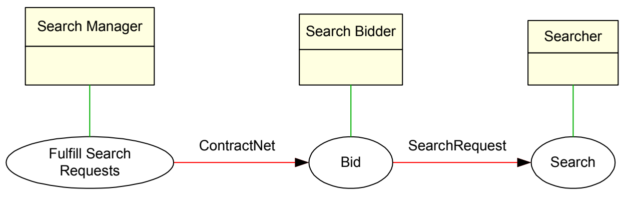


Figure 2. Role Diagram for a Mobile Search System (MSS)

Concurrent tasks are categorized by their *life span* and *responsiveness*. Task life spans are either persistent or transient. Persistent tasks always have a *null* transition from the start state to the first state. These tasks are started when the agent is created and run until either the task or agent terminates. Transient tasks, however, always have a trigger event on the transition from the start state. These tasks are not started upon agent creation but only when the agent receives its trigger event. Transient tasks enable multiple concurrently executing tasks of the same type.

Figure 3 shows the concurrent task diagram for the Bid task in Figure 2. The task starts in the *Idle* state and once an *announce(task)* message is received from the Search Manager, the *costToPerform* and *acceptability* activities in the *Prepare Bid State* determine whether or not to bid on that task. A bid is sent with a *aBid(task,cost)* message, and the task waits until it receives an *acknowledge* message from the Manager. If the bid is rejected, the task receives a *sorry(task)* message. If the bid is accepted, the task sends a *search(request)* message to the Searcher and waits for the results to send back to the Manager.

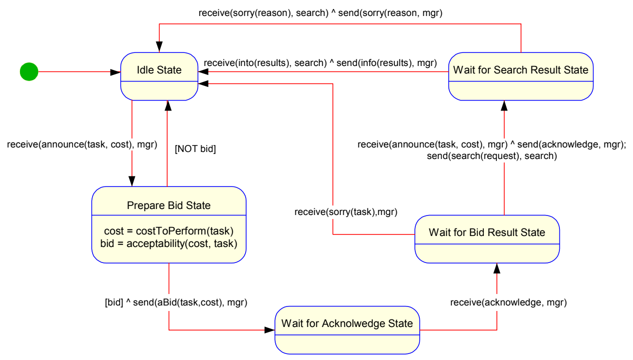

receive(sorry(reason), search) ^ send(sorry(reason, mgr)
Idle State
receive(into(results), search) ^ send(info(results), mgr)
Wait for Search Result State
receive(announce(task, cost), mgr)
[NOT bid]
receive(announce(task, cost), mgr) ^ send(acknowledge, mgr); send(search(request), search)
Prepare Bid State
cost = costToPerform(task)
bid = acceptability(cost, task)
receive(sorry(task),mgr)
Wait for Bid Result State
[bid] ^ send(aBid(task,cost), mgr)
receive(acknowledge, mgr)
Wait for Acknolwedge State

Figure 3. Bid Task from MSS

Figure 4 shows the concurrent task diagram for the Search task. This is a transient task and is started by the receipt of a *search(request)* message from the Bidder role, which causes the task state to transition to the *Search* state. This state contains the *find(request)* internal activity, which is executed upon entering the state. Depending on the value of the parameter *results* a message is sent containing either the results or a sorry message.

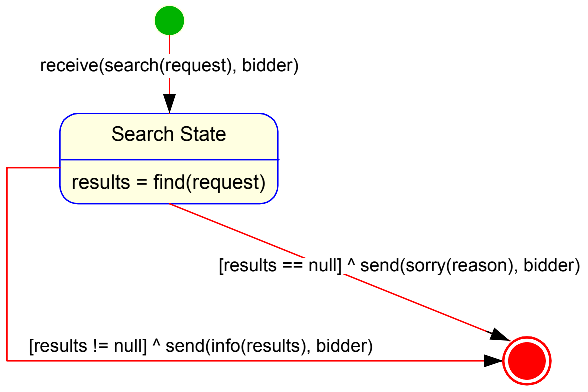

receive(search(request), bidder)
Search State
results = find(request)
[results == null] ^ send(sorry(reason), bidder)
[results != null] ^ send(info(results), bidder)

Figure 4. Search Task from MSS

3. Analysis Phase

We model mobility in the analysis phase in a straightforward manner – a simple *move* activity within a state in a concurrent task diagram. This move activity returns two values: a Boolean value and a *reason* value. The Boolean variable represents the results of the move (either success or failure). The reason value provides an explanation why the move failed, which provides the agent with knowledge to successfully recover from moving failures. The reason value is currently only useful if the selected agent platform on which the agent execute supports it. In the future, this functionality could be added to a standard such as MASIF [10]. Reasons for move failures might include the fact that the move destination address is not operational, the agent's current machine is isolated from the network, or the host denies the move due to security reasons, etc. The syntax for the move activity is shown below.

```
<Boolean, Reason> = move(location)
```

Figure 5 shows how mobility might be added to the Search task of Figure 4. In the *Move Needed State*, the three activities *searchDestination*, *getLocation* and *compare* are used to determine whether the requested information is at a different location. If a move is needed, the task transitions to the *Try Move State*. If the move is successful, the task begins searching. If the move is unsuccessful, a *sorry(reason)* message is sent to the Bidder.

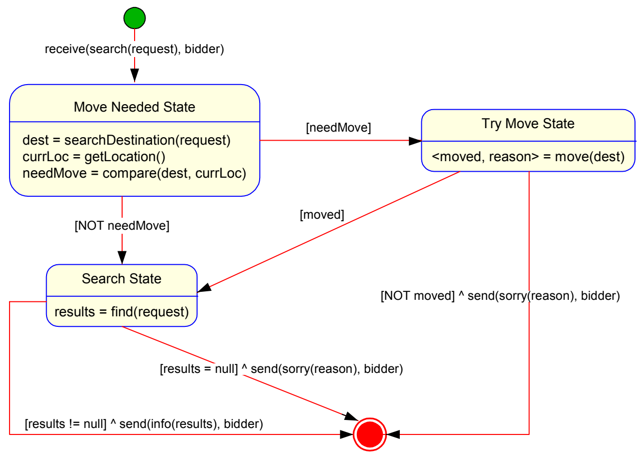

receive(search(request), bidder)
Move Needed State
dest = searchDestination(request)
currLoc = getLocation()
needMove = compare(dest, currLoc)
[needMove]
Try Move State
<moved, reason> = move(dest)
[NOT needMove]
[moved]
Search State
results = find(request)
[NOT moved] ^ send(sorry(reason), bidder)
[results = null] ^ send(sorry(reason), bidder)
[results != null] ^ send(info(results), bidder)

Figure 5. Search Task with Mobility

4. Design Phase

The MaSE design phase models consist of agent classes, the communications defined between those classes and the components that comprise those classes. Typically, tasks from the analysis phase are transformed into components in the design phase. These, possibly multiple, components define the internal agent architecture for each agent defined by the designer. Besides requesting moves, each component in a mobile agent must be able to respond to a pending move by saving its internal state and being able to restart at the new location. An *Agent Component* is created for each agent that oversees the operation and the interaction between components. In a mobile agent, the Agent Component must be transformed to handle shutdown and re-initialization of all of the agent's components. A brief overview of agent class and conversations is given in Sections 4.1 and 4.2 followed by a detailed discussion of the requirements for components in Section 4.3.

4.1 Agent Classes

An *Agent Class* is a model for the types of agents that will exist in the system and is similar to an object class from the object-oriented paradigm except that an agent class is defined by the roles it plays in the system and not by attributes and methods. Every task that is associated with a role becomes a component of the agent class playing that role in the system. Figure 6 shows the Agent Class Diagram for the MSS system. The *Search Manager* role was assigned to the *Search Manager* agent class while the *Search Bidder* and *Searcher* roles were assigned to the *Mobile Searcher* agent class.

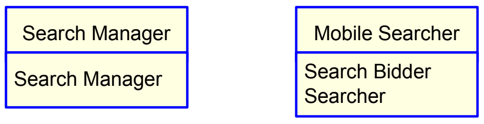


Figure 6. MSS Agent Classes

4.2 Conversations

States and transitions that describe external message passing protocols are extracted from the component state diagrams and used to create conversations. Conversations describe communication between agents and are modeled as coordinating finite state automaton (i.e., messages sent from one side of the conversation are received by the other side). Once conversations are extracted from component state diagrams, they are replaced by an action that instantiates the conversation [14].

4.3 Components

Components are independent processing modules (possibly implemented as objects) that comprise an agent. Components provide a set of functions and may or may not have internal state. Components are dynamic in nature, although certain components may endure for the entire lifetime of its agent. Using MaSE and its supporting tool agentTool [4], agent components are derived directly from the tasks in the roles assigned to each agent class. Therefore, each component can be classified as transient or persistent, based on their parent task. To coordinate the execution of these derived components, a standard *Agent Component* is also synthesize for each agent via agentTool transforms. The Agent Component controls the initiation of the other components, handles conversation initiation messages from other agents, and terminates the agent once its goals have been accomplished. Each derived components is either mobile or non-mobile. A *mobile component* contains at least one move activity while a *non-mobile component* does not contain any move activities. Figure 7 shows the default agent architecture generated by our semi-automated transformation system, agentTool.

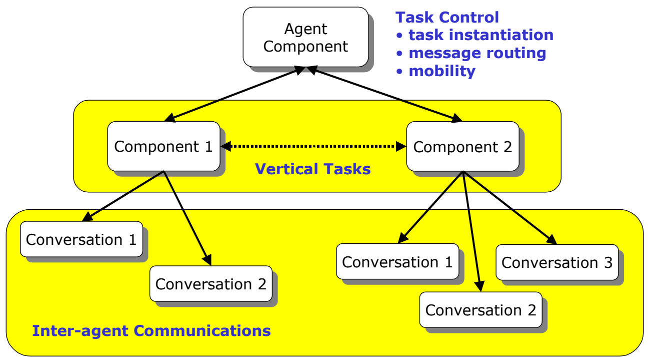


Figure 7. MaSE/agentTool Agent Architecture

4.3.1 Agent Component

There are three versions of the Agent Component – transient, persistent and heterogeneous – that are determined by the types of components that it must control. (A *heterogeneous* Agent Component handles both transient and persistent tasks.) As mentioned above, the Agent Component controls the initiation of the other components and maintains the list of all active components.

The Agent Component also handles the conversation initiation messages sent from other agents in the system. This allows the Agent Component to initiate transient components (they are not started until needed) that are activated by messages from other agents. For conversation initiation messages sent to existing components, the Agent Component simply routs the message to the appropriate component to handle the message.

If an agent class contains only persistent components (which may terminate if they accomplish their goals), the Agent Component is charged with terminating the agent once its goals have been accomplished. In this case, the persistent components may eventually fulfill all their goals and terminate, leaving only the Agent Component running [14]. Thus, the Agent Component is tasked with checking the status of the components regularly and terminating the agent and when they are no active components.

Figure 8 shows the Mobile Searcher (Figure 5) Agent Component, which contains both persistent and transient components. The Agent Component is started at agent creation and transitions automatically to the *Start Persistent Comps State*. If there is a problem starting any of these components, the Agent Component terminates the agent. If there are no problems then the Agent Component transitions to an *Idle State*, waiting to receive conversation initiation messages from other agents.

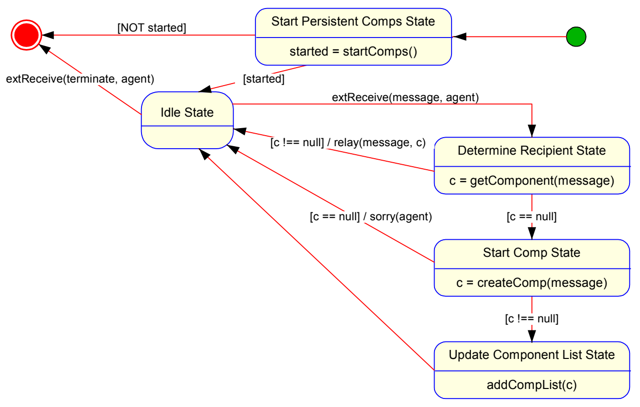

[NOT started]
Start Persistent Comps State
started = startComps()
extReceive(terminate, agent)
[started]
Idle State
extReceive(message, agent)
[c !== null] / relay(message, c)
Determine Recipient State
c = getComponent(message)
[c == null] / sorry(agent)
[c == null]
Start Comp State
c = createComp(message)
[c !== null]
Update Component List State
addCompList(c)

Figure 8. Agent Component for Mobile Searcher Agent Class

Once the agent receives a message, the *getComponent* activity in the *Determine Recipient State* checks to see if an active component is the recipient of the message. If one of the active components is the recipient, the *relay* activity delivers the message to the component. If none of the active components is the recipient, then the *createComp* activity in the *Start Comp State* checks to see if the agent contains a transient component can be initiated by the message. If the message does not initiate a component then the agent has received a stray message and a *sorry* conversation is started with the external agent. If the message is the trigger for a transient component, that component is started and the *addCompList* activity in the *Update Component List State* adds the component to the active component list.

The Agent Component is also responsible for fulfilling much of the agent mobility functions. It takes move requests from mobile components and determines whether the agent should move. The designer is responsible for adding the decision logic. If no logic is added, move requests are automatically approved. The mobile components send move request messages to the Agent Component instead of directly to the agent platform to ensure that the work being done by other components is not lost.

If a request for a move is accepted, the Agent Component informs all components of the move by sending *move required* internal messages according to the following rules:

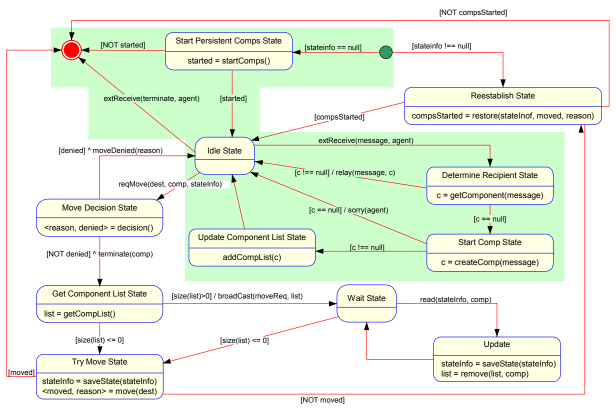
1. If there is only one mobile component and that component is a persistent component, then only the non-mobile components need to receive a message
2. If there is only mobile component and that component is a transient component, then all components need to receive a message
3. If there are two or more mobile components in an agent class, then all the components need to receive a message

The Agent Component also gathers the state information from all components. Only after all the components have terminated will the Agent Component request a move from the agent platform. Thus, the Agent Component is the repository for all component state information. After the agent has moved, the Agent Component has the duty of restarting all components at the new location that were active at the former location. Upon creation, the components are sent their state information that was saved at the agents' previous location.

Figure 9 shows the Agent Component from Figure 8 transformed to handle the mobility requirements described above (existing states and transitions from Figure 8 are shown in the shaded area). If the component is being started for the first time, the Agent Component transitions from the *start* state to the *State Persistent Comps State* as before. If the agent has moved and is being restarted at the new location then the Agent Component transitions to the *Reestablish State* where the *restore* activity restarts all the components that were active at the previous location. When a *reqMove* message is received from a mobile component, the Agent Component transitions to the *Move Decision State*. The *decision* activity is the place where the designer can insert logic as to how the agent decides to move. If the move is denied then the *moveDenied(reason)* message is sent to the mobile component and the Agent Component transitions back to an *Idle State*. If the move is approved then the mobile component that requested the move is sent a *terminate* message and the Agent Component transitions to the *Get Component List State*. If the agent class contains only one component then the Agent Component transitions to the *Try Move State*. If not, a *moveReq* message is broadcast to every active component. Once all the replies have been received the Agent Component transitions to the *Try Move State.* All the state information for the agent and components is saved and the *move(dest)* activity represents the call to the agent platform to request the move. If the move is successful, the agent terminates and is restarted at the new location. If the move fails, the agent is restarted on the current machine and the components are all restarted by the *restore* activity in the *Reestablish State*.

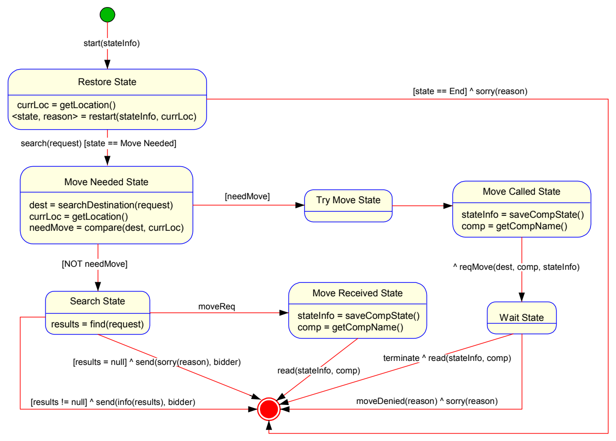
4.3.2 Mobile Components

A mobile component has to be transformed to include the necessary mobility functions. First, the *move* activity from the analysis phase is replaced with the ability to send internal move request messages to the Agent Component. After generating a move request, the mobile component also needs to save its current state so it can be restarted in the proper state after the move is completed. Finally, the mobile component must be able to respond to a *move required* messages from the Agent Component. Figure 10 shows the Search mobile component, automatically derived from the Search concurrent task in Figure 5 by agentTool.



[NOT compsStarted]
[NOT started]
Start Persistent Comps State
started = startComps()
[stateinfo == null]
[stateinfo !== null]
Reestablish State
compsStarted = restore(stateInof, moved, reason)
extReceive(terminate, agent)
[started]
[compsStarted]
[denied] ^ moveDenied(reason)
Idle State
extReceive(message, agent)
[c !== null] / relay(message, c)
Determine Recipient State
c = getComponent(message)
reqMove(dest, comp, stateInfo)
Move Decision State
<reason, denied> = decision()
[c == null] / sorry(agent)
[c == null]
Update Component List State
addCompList(c)
[c !== null]
Start Comp State
c = createComp(message)
[NOT denied] ^ terminate(comp)
Get Component List State
list = getCompList()
[size(list)>0] / broadCast(moveReq, list)
Wait State
read(stateInfo, comp)
Update
stateInfo = saveState(stateInfo)
list = remove(list, comp)
[size(list) <= 0]
[size(list) <= 0]
Try Move State
stateInfo = saveState(stateInfo)
<moved, reason> = move(dest)
[moved]
[NOT moved]

Figure 9. Agent Component for Mobile Searcher Agent Class with Mobility Functionality



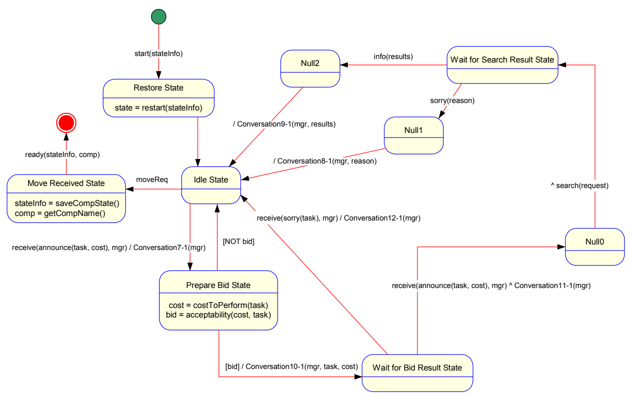
start(stateInfo)
Restore State
currLoc = getLocation()
<state, reason> = restart(stateInfo, currLoc)
[state == End] ^ sorry(reason)
search(request) [state == Move Needed]
Move Needed State
dest = searchDestination(request)
currLoc = getLocation()
needMove = compare(dest, currLoc)
[needMove]
Try Move State
Move Called State
stateInfo = saveCompState()
comp = getCompName()
[NOT needMove]
^ reqMove(dest, comp, stateInfo)
Search State
results = find(request)
moveReq
Move Received State
stateInfo = saveCompState()
comp = getCompName()
Wait State
[results = null] ^ send(sorry(reason), bidder)
read(stateInfo, comp)
terminate ^ read(stateInfo, comp)
[results != null] ^ send(info(results), bidder)
moveDenied(reason) ^ sorry(reason)

Figure 10. Mobile Search Component in MSS

The Search component is a mobile component and is started by the internal message *start(stateInfo)*. The *getLocation* and *restart* activities in the *Restore State* determine the state in which the component should begin executing. If the agent has successfully moved as part of the search process and is restarting at the new location, the component transitions directly to the *Search State*. If the move was unsuccessful, the component transitions directly to the end state and sends an internal *sorry(reason)* message to the Bidder component. If there is a new search request then the component transitions to the *Move Needed State*. The component transitions to the *Try Move State* if a move is needed. The move activity that was present in the task in Figure 4 has been removed and replaced by two states and three transitions. The *saveCompState* and *getCompName* activities in the *Move Called State* prepare the *reqMove* message that is sent to the Agent Component. The component then waits for a response. If a *terminate* message is received the component sends its state information to the Agent Component and terminates. If a *moveDenied(reason)* message is received, then the component sends an internal *sorry(reason)* message to the Bidder component.

4.3.3 Non-Mobile Components

A non-mobile component also has to be transformed to add the ability to respond to a *move required* messages from the Agent Component (including saving its current state) and restarting in the proper state after a move. In actuality, this ability could exist in any, or all states in the component state table. However, the designer is given the choice of states in which *move required* messages may be received. Figure 11 shows the Bid non-mobile component automatically derived from the Bid task in Figure 3 by agentTool.



start(stateInfo)
Restore State
state = restart(stateInfo)
Null2
info(results)
Wait for Search Result State
sorry(reason)
/ Conversation9-1(mgr, results)
Null1
ready(stateInfo, comp)
/ Conversation8-1(mgr, reason)
Move Received State
stateInfo = saveCompState()
comp = getCompName()
moveReq
Idle State
^ search(request)
receive(sorry(task), mgr) / Conversation12-1(mgr)
receive(announce(task, cost), mgr) / Conversation7-1(mgr)
[NOT bid]
Null0
Prepare Bid State
cost = costToPerform(task)
bid = acceptability(cost, task)
receive(announce(task, cost), mgr) ^ Conversation11-1(mgr)
[bid] / Conversation10-1(mgr, task, cost)
Wait for Bid Result State

Figure 11. Bid Component with Mobility Functionality

The Bid component is a non-mobile component and is started by the internal message *start(stateInfo)*. The *restart* activity in the *Restore State* determines the state in which the component should begin executing. In this case, the component transitions directly to the *Idle State* whether the agent had successfully moved or had just been created. Since, the Bid component is part of the Mobile Searcher agent class it must check for *moveReq* messages from at least one state. In this case, the designer chose only the *Idle State*. Once a *moveReq* message has been received, the component transitions to the *Move Received State* where the state information is saved and then sent to the Agent Component using a *ready(stateInfo,comp)* message.

For more detailed explanations of the functionality for non-mobile, mobile and agent components that includes the definition of formal transformations that convert the analysis models to generic design models and then from generic design models to mobile design models see the theses by Sparkman [14] and Self [13].

5. Conclusions

Incorporating mobility into the analysis phase of the MaSE methodology was accomplished with minimal impact. Adding mobility to the design phase entailed defining requirements and mapping those requirements to the different types of components, including the agent component, that comprise a mobile agent class. A semi-automated transformation system was integrated into our agentTool environment [4] as a proof of concept. This transformation system implemented all the transformations discussed in this paper. These transformations are capable of producing all the conversations between agents as well as the internal design of the agent components. Several mobile multiagent systems were developed using these transformations [13].

Incorporating the other two dynamic agent properties namely cloning and instantiation, is the main area for expanding this work. Cloning and instantiation add to the power of a dynamic agent system. Mobility is just a special case of cloning. Full cloning capability would bring the advantages of parallel computing and optimal distribution of tasks to a multiagent system in order to solve performance bottlenecks. The decision to clone should probably be the responsibility of the components, as is the case with mobility, with the agent component carrying out the details.

Mobile agent technology is a relatively new and exciting field in the area of artificial intelligence and software engineering. This research starts to bridge the gap between agent-oriented software engineering methodologies and mobile agent systems. Merging these two areas provides more power to solve the complex problems in this new era of increasing mobile, distributed computing.

6. Acknowledgements

The Air Force Office of Scientific Research sponsored this research while both authors were at the Air Force Institute of Technology. The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

References

[1] Baumann, J., K. Rothermel, *The Shadow Approach: An Orphan Detection Protocol for Mobile Agents*, Mobile Agents. Second International Workshop, MA'98, 1998. Stuttgart, Germany: Springer-Verlag.

[2] Bianchini, C. Fontes, D.S., do Prado, A.F. *A Distributed Software Agents Platform Framework.* 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems. May 29, 2002, Orlando, Florida.

[3] Chess, D., C. Harrison, and A. Kershenbaum, *Mobile Agents: Are They a Good Idea*? Mobile Object Systems: Towards the Programmable Internet. Second International Workshop, MOS '9, 1996. Linz, Austria: Springer-Verlag.

[4] DeLoach, S. A. and Wood, M. *Developing Multiagent Systems with agentTool*, in Y. Lesperance and C. Castelfranchi, editors, Intelligent Agents VII - Proceedings of the 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL'2000). Springer Lecture Notes in AI, Springer Verlag, Berlin, 2001.

[5] DeLoach, S. A. *Specifying Agent Behavior as Concurrent Tasks: Defining the Behavior of Social Agents*. Proceedings of the Fifth Annual Conference on Autonomous Agents, Montreal Canada, May 28 - June 1, 2001, ACM Press, pp. 102-103.

[6] DeLoach, S. A., Wood, M. F. and Sparkman, C. H., "Multiagent Systems Engineering", *The International Journal of Software Engineering and Knowledge Engineering*, Volume 11 no. 3, pp. 231-258, June 2001.

[7] FIPA, Agent Management Specification, 2000: **www.fipa.org**.

[8] Mamei, M. and Mahan, M. *Engineering Mobility in Large Multi Agent Systems: a case study in Urban Traffic Management.* 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems. May 29, 2002, Orlando, Florida.

[9] MESSAGE: *Methodology for Engineering Systems of Software Agents. Deliverable 1. Initial Methodology*. July 2000. EURESCOM Project P907-GI.

[10] Milojicic, D., et al, *MASIF: The OMG Mobile Agent System Interoperability Facility, Mobile Agents*. Second International Workshop, MA'98, 1998. Stuttgart, Germany: Springer-Verlag.

[11] Picco, G., A. Murphy and G.-C. Roman, *Lime: Linda Meets Mobility*, in: D. Garlan, editor, Proc. of the 21 st Int. Conference on Software Engineering (ICSE'99) (1999), pp. 368-377.

[12] Reyes, A. A. *Introducing the MArSHLAnd Design Optimization Tool for Mobile Multi-Agent Systems.* 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems. May 29, 2002, Orlando, Florida.

[13] Self, Athie, *Design and Specification of Dynamic, Mobile, and Reconfigurable Multiagent Systems*, MS thesis, AFIT/GCS/ENG/01M-11. School of Engineering, Air Force Institute of Technology, Wright Patterson Air Force Base, OH, 2001.

[14] Sparkman, Clint, *Transforming Analysis Models into Design Models for the Multiagent Systems Engineering (MaSE) Methodology*, MS thesis, AFIT/GCS/ENG/01M-12. School of Engineering, Air Force Institute of Technology, Wright Patterson Air Force Base, OH, 2001.

[15] Wooldridge, M., Jennings, N.R., & Kinny, D. "The Gaia Methodology for Agent-Oriented Analysis and Design." *Journal of Autonomous Agents and Multi-Agent Systems*. 3 (3), 2000.

[16] Zambonelli, F., Jennings, N.R., Omicini, A., and Wooldridge M.J. *Agent-Oriented Software Engineering for Internet Applications*. Coordination of Internet Agents: Models, Technologies, and Applications, Chapter 13. Springer-Verlag, March 2001.