Proceedings of the Second International Workshop On Agent-Oriented Software Engineering (AOSE-2001), Montreal, Canada, May 29th 2001.

Automated Derivation of Complex Agent Architectures from Analysis Specifications

Clint H. Sparkman[1], Scott A. DeLoach[2] and Athie L. Self[1]

[1]Department of Electrical and Computer Engineering, Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433-7765
clint.sparkman@lackland.af.mil, athie.self@afpc.randolph.af.mil

[2]Department of Computing and Information Sciences, Kansas State University
212 Nichols Hall, Manhattan, KS 66506
sdeloach@cis.ksu.edu

Abstract. Multiagent systems have been touted as a way to meet the need for distributed software systems that must operate in dynamic and complex environments. However, in order for multiagent systems to be effective, they must be reliable and robust. Engineering multiagent systems is a non-trivial task, providing ample opportunity for even experts to make mistakes. Formal transformation systems can provide automated support for synthesizing multiagent systems, which can greatly improve their correctness and reliability. This paper describes a semi-automated transformation system that generates an agent's internal architecture from an analysis specification in the MaSE methodology.

1 Introduction

In the last few years, agent technology has come to the forefront in the software industry because of advantages that multiagent systems have in complex, distributed environments. As agent technology has matured and become more accepted, agent-oriented software engineering (AOSE) has become an important topic for software developers who wish to develop reliable and robust agent-based systems [10, 11, 19]. Methodologies for AOSE attempt to provide a method for engineering practical multiagent systems. However, there are currently only a few AOSE methodologies for multiagent systems, and many of those are still under development [1, 12, 13, 18, 20]. Additionally, most existing methodologies lack specific guidance on how to transform the specification of a system into the corresponding design and implementation. This lack of guidance is not unique to engineering multiagent systems and plagues most software engineering methodologies. Unfortunately, it leaves the designer questioning if the resulting design correctly fulfills the initial system requirements.

The Agent Research Group at the Air Force Institute of Technology (AFIT), and now Kansas State University, has developed and continues to mature an AOSE methodology, called Multiagent Systems Engineering (MaSE) [4, 17], which covers the complete life cycle of a multiagent system. Recent work has focused on applying

formal methods to develop a transformation system to support agent system synthesis. Formal transformation systems [8, 9] provide automated support to system development, giving the designer increased confidence that the resulting system will operate correctly, despite its complexity. While formal transformation systems, and formal methods in general, cannot *a priori* guarantee correctness [2], if each transform preserves correctness, then the designer can be sure that the resulting design is at least correct with respect to the initial system specification.

Given a sufficient level of automated support, a transformation approach allows the designer to make only high-level design decisions, while the low-level details of the transformations are carried out automatically by the system. Transformation systems also provide traceability from the system requirements through the development process to the final executable code. Furthermore, if the system engineer is able to adequately decompose the problem and capture the system behavior in the analysis phase, then there is hope that the undesirable system behavior, to which multiagent systems are prone, can be avoided.

This level of automation will be required if such a system is to ever be truly useful. As we all know, initial system specifications are rarely, if ever, complete and consistent. Therefore, the ability to change the system specification and re-derive the design and implementation is a necessity. In effect, the long-term goal of this research to move the maintenance of software from the implementation level to the specification level. While there has been much work on general-purpose software specification languages, there has also been considerable work in specifying agent systems as well [3, 21].

In this paper, we present a semi-automated formal transformation system that generates MaSE design models based on the analysis models [16], which is the first step in formal agent system synthesis. Specifically, we explain how our transformation system generates an agent's internal design based on an initial analysis specification.

2 Background – Multiagent Systems Engineering

The MaSE methodology consists of the seven steps depicted in Fig. 1. The boxes represent the different models used in the steps, and arrows indicate the flow of information between the models. While similar to the waterfall approach, we have designed MaSE to be applied iteratively. The first three steps represent the Analysis phase of the methodology, while the last four steps represent the Design phase.

2.1 Analysis Models

The Role Model is the end result of the MaSE analysis phase. Role Models graphically depict the roles in the system, the goals they are responsible for, the tasks that each role uses to accomplish its goals, and the communication paths between the roles necessary to complete their tasks. Roles are the abstract entities that exist in the system, and are defined much like an actor in a play, or a position in an organization

(President, Vice President, Manager, etc.). Each role is responsible for accomplishing one or more system level goal, and there must be at least one role responsible for each goal. In this way, the analyst is able to ensure that all of the initial system requirements have been captured.

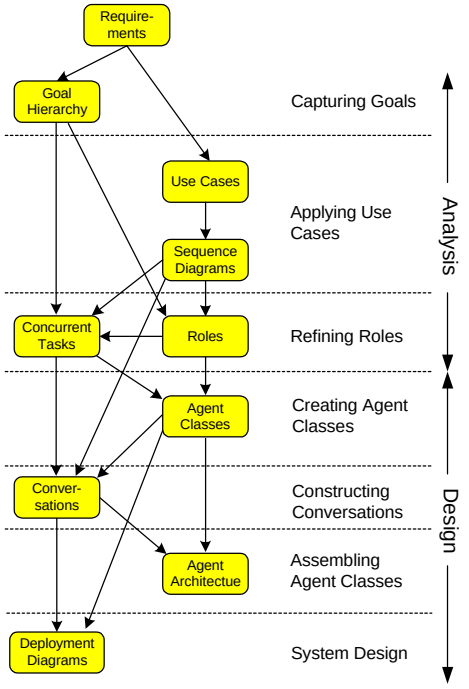

Fig. 1. MaSE Methodology

One example of a Role Model is shown in Fig. 2. The roles in the system are depicted as rectangles, and the goals that a role is responsible for are listed under the role. Each task that a role has is denoted by an oval attached to the role. The lines between the tasks denote communication protocols that occur between the tasks. The arrows indicate the initiator and responder tasks in the protocol, with the arrow pointing from the initiator to responder. Solid lines indicate peer-to-peer communication, which is external communication either between two tasks of different roles, or between two different instances of the same role. Conversely, dashed lines denote communication between two tasks of the same role instance.

As part of defining the Role Model, the analyst must define the tasks that each role has. Tasks describe the behavior that a role must exhibit in order to accomplish its goals, and are specified graphically using a finite state automaton, as shown in Fig. 3. A single role may have multiple concurrent tasks that define the complete behavior of the role. Each task is assumed to operate under its own thread of control, thus each task has its own state diagram that executes independently of the other tasks. An

important aspect of multiagent systems is the ability of agents to interact to accomplish their goals. Concurrent tasks capture this interaction and can be used to specify complex communication protocols such as Contract Net, Dutch Auction, etc. [6]. Concurrent tasks also lay the foundation for conversations between the agent classes in the design phase of MaSE.

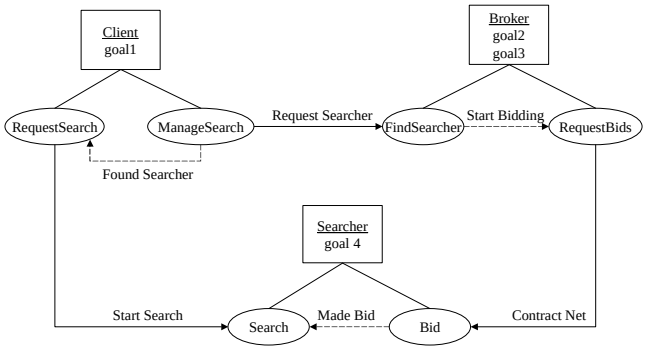

Fig. 2. Role Model

An important property of concurrent tasks is that they are able to capture communication with multiple tasks in order to accomplish their goals. In other words, Concurrent Task Diagrams naturally intertwine events belonging to different protocols. The other tasks being communicated with can belong to the same role, or they may belong to a different role. Tasks that belong to the same role can coordinate with each other through internal events. In Fig. 3, the *^backup(file)* event on the transition from the Wait_For_File state to the Wait_For_Backup state is an example of an internal send event, and the *done* event on the transition from the Wait_For_Backup state to the Process_File state is an example of an internal receive event. In order for a task to communicate with a task of another role, events that represent external communication are specified using *SendEvents* and *ReceiveEvents*. These events are defined to send and retrieve messages from an implied massage-handling component of the agent. The *^send(request(filename), server)* event on the transition from the Get_File_Name state to the Wait_For_File state is an example of an external *SendEvent*, and the *receive(inform(file), server)* event on the transition from the Wait_For_File state to the Wait_For_Backup state is an example of an external *ReceiveEvent.*

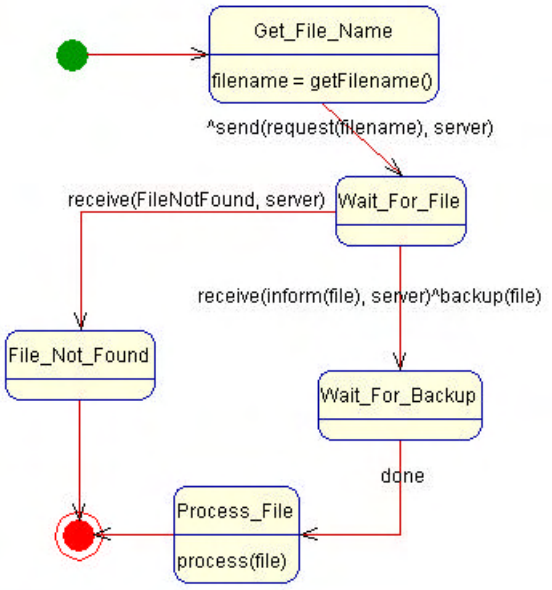

Fig. 3. Concurrent Task Diagram

2.2 Design Models

In the design phase of MaSE, the designer takes the Role Model in the analysis phase, and produces an Agent Class Diagram, as shown in Fig. 4. Each rectangle represents an agent class, with the roles played by each agent listed under the agent's name. The directed lines between the agents represent conversations between the agents, with the arrow pointing from the initiator to the responder. In order to ensure that all system goals are being met, each role must be played by at least one agent class, providing a traceable link from the goals in the analysis phase to the agents in the design phase.

Conversations define detailed coordination protocols between exactly two agents, and consist of a pair of Communication Class Diagrams, one each for the initiator and responder. Conversations are at the heart of any multiagent system as they detail how the agents communicate with each other. Like tasks, conversations are described using finite state automata that define each half of the conversation. Since conversations are point-to-point communication between two agents, every event within a Communication Class Diagram is represents a message to or from the other agent in the conversation. Conversations do not allow for communication with multiple agents simultaneously or for internal events to be exchanged with components internal to the agent. An example of a Communication Class Diagram is shown in Fig. 5.

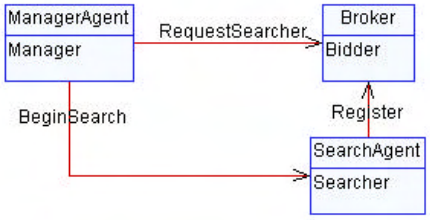

Fig. 4. Agent Class Diagram

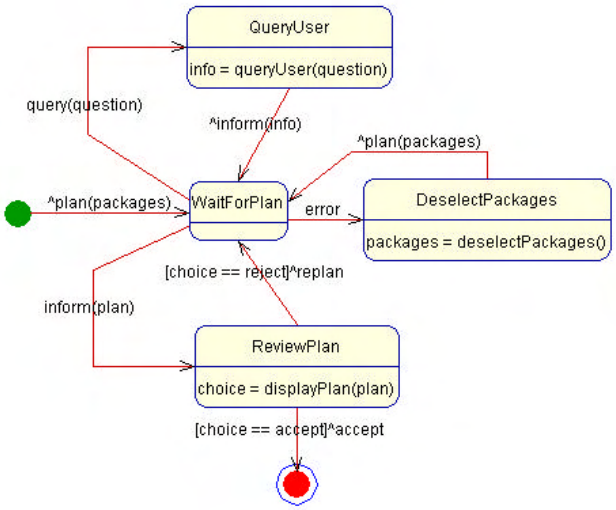

Fig. 5. Communication Class Diagram

In addition to the conversations that agents participate in, agents have internal components defined using an architectural modeling language combined with the Object Constraint Language (see Fig. 6). Components allow users to logically decompose the agents and define attributes and functions that are needed for the agent to carry out its tasks. The dynamic characteristics of the components are defined using a state diagram. The events passed within a component's state diagram are limited to internal events with other components that belong to that agent.

2.3 agentTool

In addition to the MaSE methodology, AFIT has developed a CASE tool named agentTool that serves as a validation platform and a proof of concept for MaSE. agentTool has a graphical user interface that allows a user to develop a multiagent

system using the MaSE analysis and design models. agentTool is also able to generate Java code for a system based on the design models. Currently, the code generator is able to generate code for two different frameworks, agentMom [7] and Carolina [14], but work is being done to integrate agentTool with the AFIT Wide Spectrum Object Modeling Environment that is looking at the more general code generation problem [9].

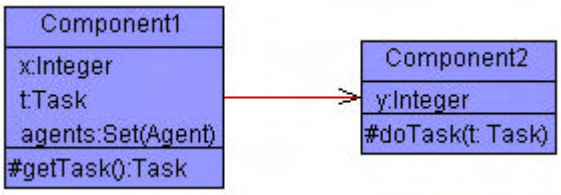

Fig. 6. Internal Agent Components

3 Analysis to Design Transformations

Before defining the specific transformations, this section first describes how the analysis models map to the design models. The MaSE methodology makes it clear that an agent class' roles, in conjunction with the protocols between the tasks, determine the conversations each agent class will have. However, if the external events are simply removed from the tasks to create the conversations, the problem we are faced with is that there will be nothing left in the design to capture how to coordinate the conversations and there will be no guarantee that the agent will behave consistently with the initial concurrent tasks. We must also capture the internal events in the design as well.

To solve this problem, we create a separate component for every task in each role that an agent is assigned to play. We then copy the concurrent task definition to the associated component state diagram. Next, we extract the states and transitions belonging to conversations and replace them with actions that represent the execution of the conversation. Using this approach, the component's state diagram retains the coordination and internal events necessary to ensure the behavior of the component matches the task from which it was derived.

Prior to this segment of our research, we had defined conversations as belonging directly to agents. However, based on the approach discussed above, we have redefined the generic architecture to have conversations belong to components. Fig. 7 illustrates how the models in the analysis phase translate to the models in the design phase as well as the relationship between the design models. Ultimately, the roles that the designer chooses for an agent to play determine that agent's components, as well as the set conversations in which the agent participates. To accurately capture the

behavior as defined in concurrent tasks, we assume each component also executes as an independent thread.

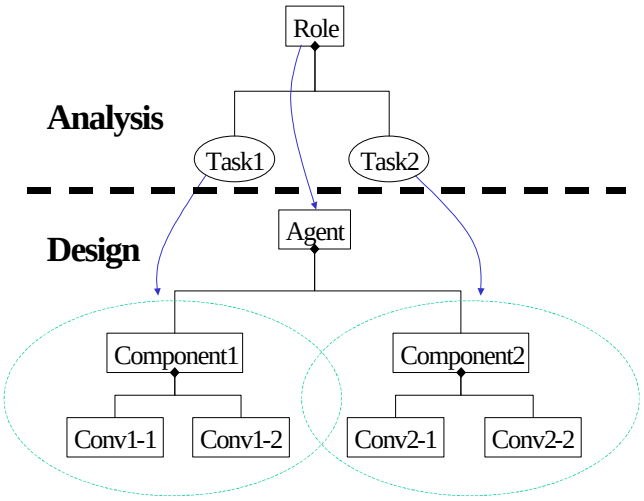

Fig. 7. Model Influences

Besides components derived from concurrent tasks, our transformations also create a special *Agent Component* for each agent [15]. This *Agent Component* captures how the agent coordinates its different components. Fig. 8 shows the basic state diagram for the *Agent Component*, which is designed to handle both transient[1] and persistent[2] components. The *Agent Component* can also be transformed to account for special agent characteristics like mobility, where the agent must halt all of its active components, move to a new location, and then resume the components where they were interrupted.

The transformation system created in this research is actually a series of small steps that incrementally change the roles and tasks in the analysis phase into agent classes, components, and conversations in the design phase. The process logically decomposes into three stages. Before the transformations can take place, the developer must analyze the system and develop a Role Model, which defines the roles that are present in the system, and a set of concurrent tasks, which the roles perform to accomplish their goals. The developer must also decide which agent classes will be in the system and the roles that each agent class will play.

[1] A transient component is started in response to the receipt of a specific event. There may be multiple transient components of the same type executing at any one time.

[2] A persistent component is started when the agent is instantiated and runs until its completion or the agent is terminated. There is only one instance of a persistent component running.

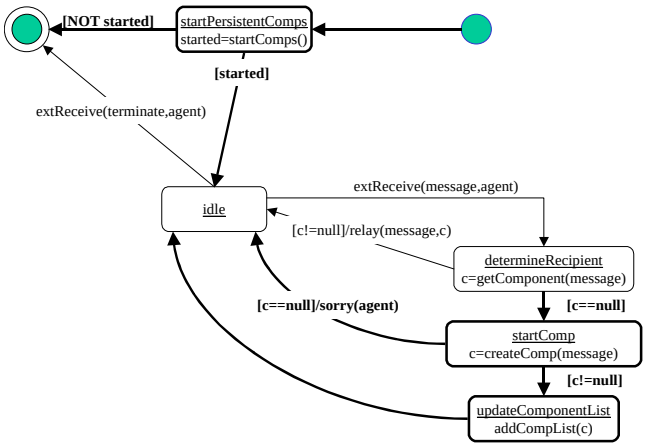

Fig. 8. Basic Agent Component State Diagram

During the first stage of the transformation process, the components for the agent classes are created based on the roles assigned by the developer. The set of protocols to which each external event belongs is also determined. The second stage centers on annotating the component state diagrams and matching external events in the different components that become the initial messages of a conversation. During the last stage the component state diagrams are prepared for removal of the states and transitions belonging to conversations. They are then removed and added to the state diagrams of the corresponding conversation halves. As they are removed from the components they are replaced with a single transition that has an action that starts the conversation.

Each transformation is defined by a predicate logic equation of the form: `condition` $\Rightarrow$ `result`, where the condition is the set of requirements that must be true for the transformation to take place, and the result describes what is guaranteed to be true after the transformation is performed. This notation is similar to defining functions with pre-conditions and post-conditions. These transformations describe *what* must take place, not *how* it must be done.

3.1 Creating Agent Components

Once the designer has developed the Role Model, defined the concurrent tasks, and assigned roles to agent classes, the transformation process can begin. The first transformations in stage one of the transformation process determine the protocols to which each external event belongs. This is important because the specific protocols that events belong to are used to determine where conversations begin and end in the component state diagrams. While the protocols for most events can be automatically determined, there are ambiguous cases where the designer must be asked to decide to which protocol specific events belongs.

Next, for every task of every role that an agent plays, a component is created for that task. Once again, the component's state diagram is initially identical to that of the

task it was derived from. The rest of the transformation process is focused on moving the external events from these component state diagrams into conversations. The following predicate logic equation formally defines this transformation:

$\forall$ a : **Agent**, r : **Role**, t : **Task** • (r $\in$ a.roles $\wedge$ t $\in$ r.tasks)
$\Rightarrow$ ($\exists$ c : **Component** • c $\in$ a'.components $\wedge$ c.stateTable = t.stateTable $\wedge$ c.name = t.name)

As an example of this transformation, consider the example Role Model shown in Fig. 9. If the developer decides in the design phase to create the agent classes with the roles shown in Fig. 10, then the transformation system creates the components shown for the agents. Since both agents play Role 2, there is a component created for each agent for Role 2's Task 2. Fig. 10 is not a MaSE diagram, but is presented to illustrate the internal agent components based on the initial Agent Class Diagram.

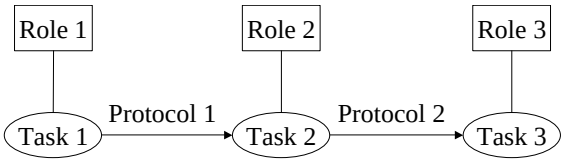

Fig. 9. Initial Role Model

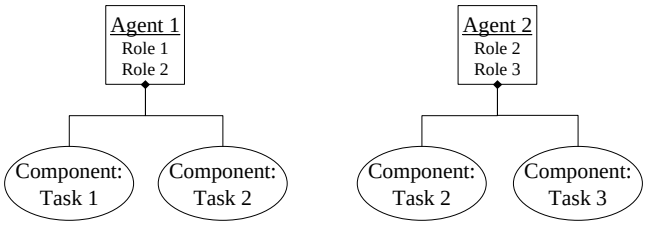

Fig. 10. Agent Components Created from Roles' Tasks

Once the agent components are created, for each pair of roles that are combined into an agent class, the designer must determine whether each protocol that exists between components of that agent is either internal or external. If a protocol is defined as internal, all events belonging to that protocol become internal events between components and not messages in a conversation.

3.2 Annotating Component State Diagrams

After the agent components are created, the next stage of the transformation process involves annotating the component state diagrams to prepare for conversation

extraction. There are many different cases in which tasks are defined in the analysis phase that make removing conversations problematic. One such case occurs when multiple events not belonging to the same protocol reside on the same transition. To solve this (and other similar) problem, we defined a transformation that converts the component's state diagrams into a canonical form, which splits transitions having events belonging to different protocols. This canonical form simplifies conversation extraction while remaining consistent with the initial task specification.

Next, each transition is given a set of protocols that is based on the protocols for the external events on the transition. Then the state diagrams are annotated to indicate where each conversation begins and ends. Conversations are defined as point-to-point communication between two agent instances. Therefore, any time a component's state diagram has a transition with external communication to a different agent than one of the preceding transitions, a new conversation must begin, and that transition is labeled as the start of a conversation. The following six conditions indicate the start of a conversation by a change in who the agent is communicating with, which in most cases is due to a change in the protocols.

1. A transition has a protocol not found in at least one transition into its *from* state.
2. A transition has a non-empty set of protocols that is different than another transition leaving the same state.
3. A transition has a non-empty set of protocols, but lacks a protocol of another transition into its *from* state.
4. A transition has a non-empty set of protocols, and there is another transition into or out of its *from* state with an empty set of protocols.
5. A transition has an empty set of protocols and at least one SendEvent. In these cases, there is either a multicast event, or there are SendEvents that belong to different protocols.
6. A transition has a SendEvent whose recipient was previously determine by an action.

Similarly, when a component state diagram has a transition with external communication not guaranteed to continue on transitions leaving its *to* state, that transition is labeled as the end of a conversation. The following four conditions indicate that a transition is the end of a conversation

1. A transition has a protocol not found in a transition leaving its *to* state.
2. A transition has an empty set of protocols and at least one SendEvent.
3. A transition has a non-empty set of protocols and there is a start transition leaving its *to* state.
4. A transition to the end state has a non-empty set of protocols.

Once the start and end labels have been added to the component state diagrams, the initial messages of the conversations must be "matched up" (i.e., both sides of a conversation must start and end with the same message types). In most cases, this can be done automatically, but in some ambiguous cases the designer is required to decide how to match conversation halves.

3.3 Extracting Conversations

The last stage of the transformation process removes the conversations from the component state diagrams and places them in their appropriate conversation halves. To extract a conversation from a state diagram, each of its end transitions must exit to the same state. If different transitions of a conversation exit to different states, a transformation is applied to create a new "dummy" end state for the conversation. Then, the states and transitions that belong to the conversation are replaced with a single transition from the state where the transition originates to the state where the conversation ends. An action is added to the transition that represents the execution of the conversation.

Other transformations in this stage prepare variables in the states and transitions before they are removed from the components and placed in the conversations. If a variable is not exclusive to a single conversation, that variable must be stored in parent component to ensure any other conversations extracted from the component references the *same* variable. To annotate this, these variables are pre-pended with "parent.".

As the transitions are moved from the components into the conversations, the special "send" and "receive" parts of the events are removed from the events. They are used in the component state diagrams to distinguish between internal and external events, but are not needed in the conversations since conversations, by definition, define binary communication between exactly two agents.

4 Example

This section presents an example to demonstrate the results of the transformation system. The transformations were implemented in agentTool [5], and most of the figures are screen shots from the tool. Fig. 11 shows the initial Role Model for a simple multiagent system. There are three roles, each with a single task. The Manager role uses the ContractNet protocol to solicit bids for search tasks. The Bidder role bids on the tasks, and if awarded the contract requests a search from the Searcher role. The Bid task, shown in Fig. 12, demonstrates how the transformation system derives the agent components and conversations in the design phase from the roles and tasks in the analysis phase.

For the purposes of this example, we assume that the designer initially defines a SearchManager agent class, which plays the Manager role, and a MobileSearcher agent class, which plays both the Bidder and Searcher roles. During the first stage of the transformation process, the designer determines that the SearchRequest protocol is internal communication within the MobileSearcher agent. Therefore, every event in the MobileSearcher's Bid and Search components that belongs to the SearchRequest protocol is transformed into an internal event. The resulting architecture for the multiagent system is shown in Figure 13. Once again, this is not a MaSE model, but simply demonstrates the architecture created for the agents based on the roles they play. The external protocol, ContractNet, generates several conversations to carry out that communication.

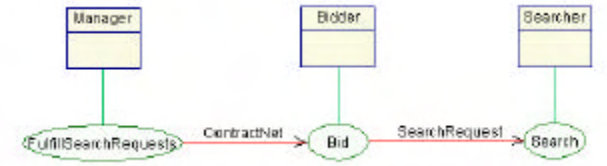

Fig. 11. Role Model

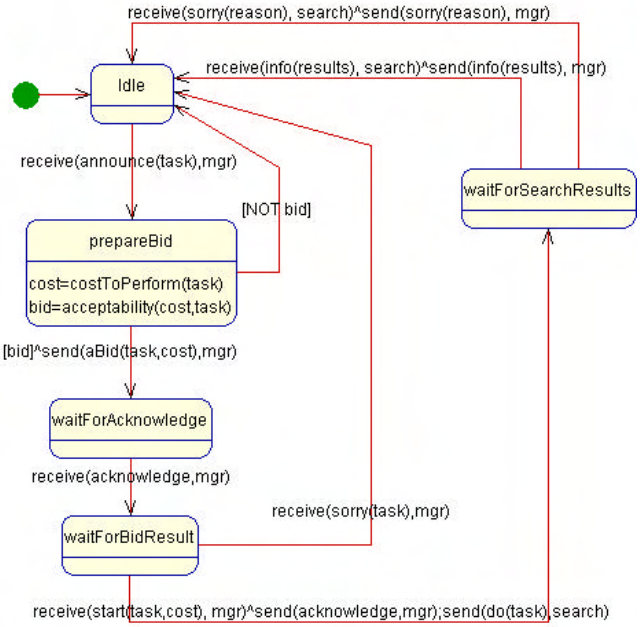

Fig. 12. Bid Task

Fig. 14 shows the Bid Component after being annotated in the second stage of the transformation process. The three events that belong to the SearchRequest protocol are now internal events, and three new null states have been added to split transitions that had both internal and external events. The letters “S” and “E” on the transitions denote where the conversations begin and end.

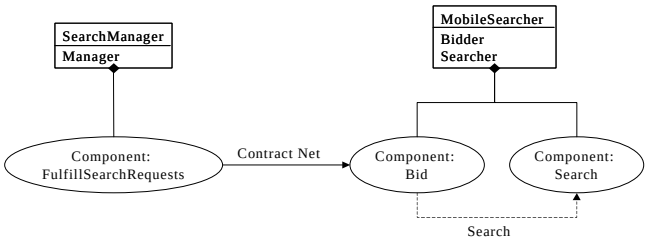

Fig. 13. Agent Architectures

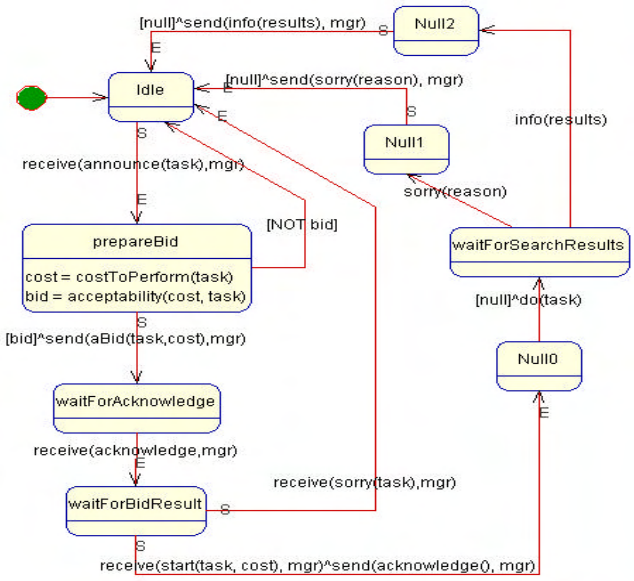

Fig. 14. Annotated Bid Component

A total of six different conversations were extracted from the events belonging to the ContractNet protocol. Some were due to the internal events passed with the Search component, while others were due to the way the SearchManager's FulfillSearchRequest component (not shown) was annotated. For example, the reason the transition from the Idle state to the *prepareBid* state is both the start and end of a conversation is because the corresponding send event for the *receive(announce(task), mgr)* event in the FulfillSearchRequest component is a multicast. Similarly, the transitions leaving the waitForBidResults state are the start of different conversations because the corresponding send event for the *receive(sorry(task), mgr)* event in the FulfillSearchRequest is a multicast to all of the

losers, and the corresponding send event for the receive*(start(task, cost), mgr)* event is only sent the winner agent, so they must be different conversations.

Fig. 15 shows the bid component after the third stage of the transformation process. The states and transitions that belong to the conversations were removed, and each conversation was replaced with a transition that has an action that instantiates it. When the conversation completes, the action is finished and the component enters the next state, thus preserving the original semantics of the state diagram.

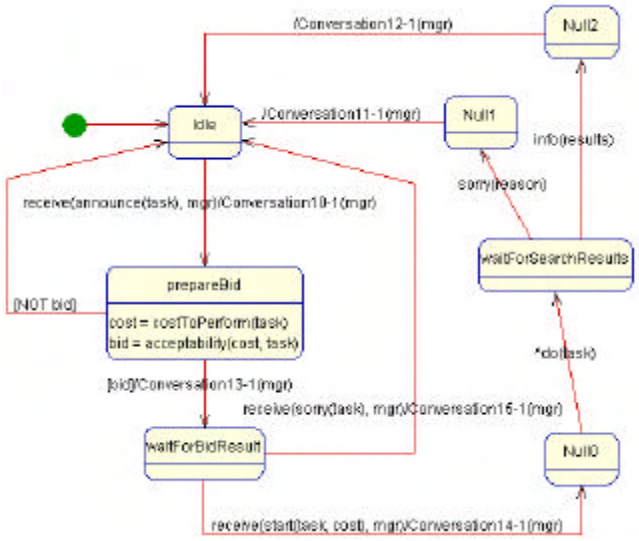

Fig. 15. Bid Component After Extracting the Conversations

Fig. 16 shows the initiator half of Conversation13-1, which was one conversation extracted from the MobileSearcher's Bid component. It is easy to see that the waitForAcknowledge state and the transitions to and from that state were taken directly from the Bid component. The *task* and *cost* variables were prepended with "parent." because they are both used either in the Bid component or in another conversation. Fig. 17 shows the Agent Class Diagram derived by our transformation system. Note that all external communication defined by the Contract Net protocol is captured in six conversations.

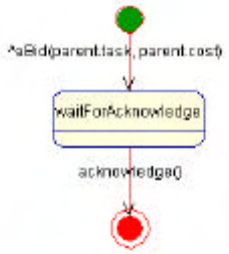

Fig. 16. Initiator Half of Conversation13-1

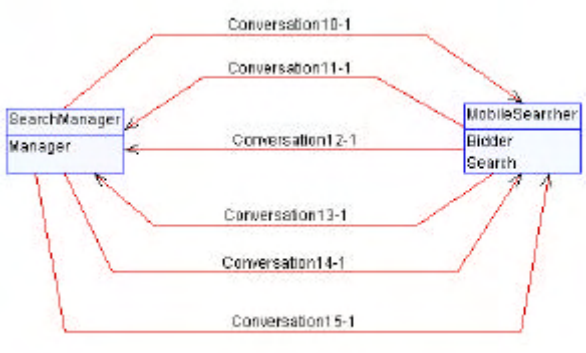

Fig. 17. Agent Class Diagram

4.1 Evaluation

The above example was analyzed and designed using the agentTool environment. The semi-automatic transformation system built into agentTool dramatically improved the time required to take the analysis specifications and create complete design specifications. The initial design was created in less than an hour, a task that would normally take several hours, or days, just to enter into agentTool. The CPU time actually taken during this transformation was less than 10 minutes with the remainder of the time being spent by the designer looking at the "ambiguous cases" described in Section 3.2. These cases required the designer to look at the analysis models to determine the correct action for the transformation to perform. Even more dramatic was the time required to re-design the system after analysis specification changes. As we discussed in the introduction, high-level specifications are rarely correct the first time. After making minor changes to the analysis specifications, the time required to re-design the system was just a few minutes since the ambiguous cases were already known.

Although the time saved in the analysis to design transformation was impressive, the fact that the design was correct with respect to the analysis specification was the key factor. We did not perform the analysis to design transformations manually and thus were not able to measure the number of errors or inconsistencies that might have been introduced. The only place that manual design could improve on the current transformation process is in efficiency. We will discuss this in detail in the next section.

5 Future Work

This research has opened the door to many areas of future work. Although transformation system produces a design that corresponds to the analysis specification, many times the result is not an optimal solution. One example is the way that conversations are created. After applying the transformations, there may be two conversations between two agents that do *exactly* the same thing. Although this is not necessarily wrong, an additional set of optimizing transformations could remedy these problems.

Another area of future work deals with what we refer to as embedded conversations. In many cases, the current transformations halt one conversation to carry out a dialog with another agent only to resume communication with the initial agent. This results in a single protocol being decomposed into several simple conversations that, by themselves, have little semantic meaning. Alternate approaches would be to allow one conversation to instantiate another conversation, or allow conversations to halt while a component carries out other communication, which would result in more robust and semantically intact conversations.

A final area of future research is in the area of transforming concurrent tasks to run in a single threaded execution environment. As described earlier, we assume that each component runs as its own thread; however, any many situations, we would rather have a single thread running. The challenge would be to capture a single threaded design that would behave consistently with a concurrent specification.

6 Conclusions

The multiagent paradigm provides a framework for developing increasingly complex and distributed software systems. However, better methods are needed to develop multiagent systems that can guarantee correctness, reliability, and robustness. Using formal transformation systems for multiagent system synthesis is one way to meet this growing need.

This paper presented a transformation system that generates design models from the analysis models, including the internal agent architectures and the specific conversations for the components. It is predominantly an automatic process, requiring only a few key design decisions from the system developer. Since each transformation preserves correctness from one model to the next, the developer has

much more confidence that no inconsistencies or errors occurred during the design process. The transformation process also provides clear traceability between the analysis and design, simplifying the verification process.

Furthermore, when implemented in a development environment, such as agentTool, the transformations allow the developer to maintain the system in the more abstract analysis models and regenerate the design when any changes are made. How many times during a software development project are the models in the analysis phase forgotten once the project enters the design phase? In many cases, there is simply not enough time or manpower to maintain the consistency between the models in the two phases. The transformation system presented here can eliminate that problem for system engineers using the MaSE methodology.

7 Acknowledgements

This work was performed while the authors were at the Air Force Institute of Technology and was supported by the Air Force Office of Scientific Research. The views expressed in this article 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 US Government.

References

[1] Brauer, W., Nickles, M., Robatsos, M., Weiss, G., Lorentzen, K.: Expectation-Oriented Analysis and Design. In this volume (2001)

[2] Clarke, E. Wing, J.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys. **28** (4) (1996)

[3] Dastani, M., Jonker, C., Truer, J.: A Requirement Specification Language for Configuration Dynamics of Multi-Agent System. In this volume (2001)

[4] DeLoach, S. A., Wood, M., Sparkman, C.: Multiagent Systems Engineering. To appear in the Intl. J. on Software Engineering and Knowledge Engineering (2001)

[5] DeLoach, S. A., Wood, M.: Developing Multiagent Systems with agentTool. In: Castelfranchi, C., Lesperance, Y. (eds.): Intelligent Agents VII: Agent Theories Architectures and Languages, Proceedings of the 7th International Workshop, ATAL 2000. Lecture Notes in Artificial Intelligence, Vol. 1986. Springer-Verlag, Berlin Heidelberg New York (2001)

[6] DeLoach, S. A.: Specifying Agent Behavior as Concurrent Tasks. Proceedings of the Fifth International Conference on Autonomous Agents. ACM Press, New York (2001) 102-103

[7] DeLoach, S. A.: Using agentMom. Air Force Institute of Technology, (2000)

[8] Green, C., Luckham, D., Balzer, R., et al.: Report on a Knowledge-Based Software Assistant. In Rich, C., Waters, R. C. (eds.): Readings in Artificial Intelligence and Software Engineering. Morgan Kaufmann, San Mateo, California (1986) 377-428

[9] Hartrum, T. C., Graham, R.: The AFIT Wide Spectrum Object Modeling Environment: An AWESOME Beginning. Proceedings of the National Aerospace and Electronics Conference. IEEE (2000) 35-42

[10] Jennings, N.: On Agent-based Software Engineering, Artificial Intelligence: **117** (2000) 277-296
[11] Lind, J.: Issues in Agent-Oriented Software Engineering. In Ciancarini, P., Wooldridge, M. (eds.): Agent-Oriented Software Engineering: First International Workshop, AOSE 2000. Lecture Notes in Artificial Intelligence, Vol. 1957. Springer-Verlag, Berlin Heidelberg (2001) 45-58
[12] Omicini, A.: SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems. In Ciancarini, P., Wooldridge, M. (eds.): Agent-Oriented Software Engineering: First International Workshop, AOSE 2000. Lecture Notes in Artificial Intelligence, Vol. 1957. Springer-Verlag, Berlin Heidelberg (2001) 185-194
[13] Rana, O.: A Modelling Approach for Agent Based Systems Design. In Ciancarini, P., Wooldridge, M. (eds.): Agent-Oriented Software Engineering: First International Workshop, AOSE 2000. Lecture Notes in Artificial Intelligence, Vol. 1957. Springer-Verlag, Berlin Heidelberg (2001) 195-206
[14] Saba, G. M., Santos, E.: The Multi-Agent Distributed Goal Satisfaction System. Proceedings of the International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA '2000) (2000) 389-394
[15] Self, A.: Design & Specification of Dynamic, Mobile, and Reconfigurable Multiagent Systems. MS thesis, AFIT/GCS/ENG/01M-11. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, (2001)
[16] Sparkman, C.: 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 (AU), Wright-Patterson AFB, OH (2001)
[17] Wood, M.: Multiagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems. MS thesis, AFIT/GCS/ENG/00M-26. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, 2000
[18] Wooldridge, M., Jennings, N., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Intl. J. of Autonomous Agents and Multi-Agent Systems. **3** (3) (2000) 285-312
[19] Wooldridge, M., Ciancarini, P.: Agent-Oriented Software Engineering: the State of the Art In Ciancarini, P., Wooldridge, M. (eds.): Agent-Oriented Software Engineering: First International Workshop, AOSE 2000. Lecture Notes in Artificial Intelligence, Vol. 1957. Springer-Verlag, Berlin Heidelberg (2001) 1-28
[20] Zambonelli, F.: Abstractions and Infrastructures for the Design and Development of Mobile Agent Organizations. In this volume (2001)
[21] Zhu, H.: A Formal Specification Language for MAS Engineering. In this volume (2001)