International Journal of Software Engineering and Knowledge Engineering
Vol. 11, No. 3 (2001) 231-258
© World Scientific Publishing Company

MULTIAGENT SYSTEMS ENGINEERING

SCOTT A. DELOACH, MARK F. WOOD AND CLINT H. SPARKMAN

Air Force Institute of Technology
Graduate School of Engineering and Management
Department of Electrical and Computer Engineering
Wright-Patterson Air Force Base, OH 45433-7765
E-mail: (sdeloach@computer.org, woodm@gateway.net, csparky@flash.net)

This paper describes the Multiagent Systems Engineering (MaSE) methodology. MaSE is a general purpose, methodology for developing heterogeneous multiagent systems. MaSE uses a number of graphically based models to describe system goals, behaviors, agent types, and agent communication interfaces. MaSE also provides a way to specify architecture-independent detailed definition of the internal agent design. An example of applying the MaSE methodology is also presented.

Keywords: Multiagent systems; software engineering; methodologies; analysis; design.

1. Introduction

The advent of multiagent systems has brought together many disciplines in an effort to build distributed, intelligent, and robust applications. They have given us a new way to look at distributed systems and provided a path to more robust intelligent applications. However, many of our traditional ways of thinking about and designing software do not fit the multiagent paradigm. Over the past few years, there have been several attempts at creating tools and methodologies for building such systems. Unfortunately, many of the methods focused on a single agent architecture or have not gone to the necessary level of detail to adequately support complex system development [5]. In our research, we have developed a complete-lifecycle methodology, called Multiagent Systems Engineering (MaSE), for analyzing, designing, and developing heterogeneous multiagent systems.

Much of the current research related to intelligent agents has focused on the capabilities and structure of individual agents. However, to solve complex problems, these agents must work cooperatively with other agents in a heterogeneous environment. This is the domain of multiagent systems. In multiagent systems, we are interested in the coordinated behavior of a system of individual agents to provide a system-level behavior. Sycara [17] describes the challenges facing multiagent systems including the focus of our research, how to engineer practical multiagent systems. MaSE uses the abstraction provided by multiagent systems for developing intelligent, distributed software systems. To accomplish the goal, MaSE uses a number of graphically based models to describe the types of agents in a system and their interfaces to other agents, as well as an architecture-independent detailed definition of the internal agent design.

In our research, we view MaSE as a further abstraction of the object-oriented paradigm where agents are a specialization of objects. Instead of simple objects, with methods that can be invoked by other objects, agents coordinate with each other via conversations and act proactively to accomplish individual and system-wide goals. Interestingly, this viewpoint sidesteps the issues regarding what is or is not an agent. We view agents merely as a convenient abstraction, which may or may not possess intelligence. In this way, we handle intelligent and non-intelligent system components equally within the same framework. In addition, since we view agents as specializations of objects, we build on existing object-oriented techniques and apply them to the specification and design of multiagent systems.

The primary focus of MaSE is to help a designer take an initial set of requirements and analyze, design, and implement a working multiagent system. This methodology is the foundation for the Air Force Institute of Technology's (AFIT) agentTool development system, which also serves as a validation platform and a proof of concept [2]. The agentTool system is a graphically-based, fully interactive software engineering tool for the MaSE methodology. agentTool supports the analysis and design in each of the seven MaSE steps. The agentTool system also supports automatic verification of inter-agent communications and code generation for multiple multiagent system frameworks. The MaSE methodology, as well as agentTool, is independent of any particular agent architecture, programming language, or communication framework. The focus of our work is on building heterogeneous multiagent systems. We can implement a multiagent system designed in MaSE in several different ways from the same design.

The MaSE methodology is a specialization of more traditional software engineering methodologies. The general operation of MaSE follows the phases and steps shown on the right side of Figure 1. The MaSE Analysis phase consists of three steps: Capturing Goals, Applying Use Cases, and Refining Roles. The Design phase has four steps: Creating Agent Classes, Constructing Conversations, Assembling Agent Classes, and System Design. The rounded rectangles denote the MaSE models used to capture the output of each step while the arrows between them show how the models affect each other. While we have drawn it as a single flow from top to bottom, with the models created in one step being the inputs for subsequent steps, in practice the methodology is iterative. The intent is that the analyst or designer be allowed to move between steps and phases freely such that with each successive pass, additional detail is added and, eventually, a complete and consistent system design is produced.

A major strength of MaSE is the ability to track changes throughout the process. Every object created during the analysis and design phases can be traced forward or backward through the different steps to other related objects. For instance, a goal derived in the Capturing Goals step can be traced to a specific role, task, and agent class.

Likewise, an agent class can be traced back through tasks and roles to the system level goal it was designed to satisfy.

The individual steps of the analysis and design phases are discussed in Sections 2 and 3. An overview of where MaSE has been used and future research directions is presented in Section 4 while a comparison with other existing multiagent methodologies is given in Section 5.

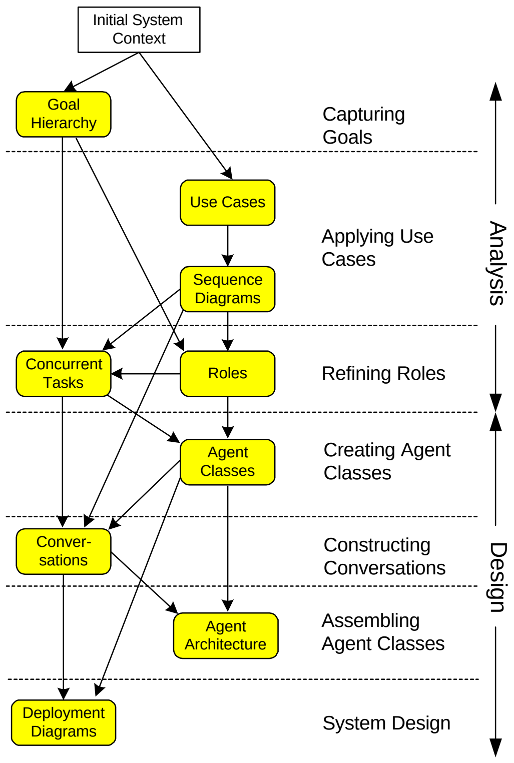

Figure 1. MaSE Phases

2. Analysis Phase

The purpose of the MaSE Analysis phase is to produce a set of roles whose tasks describe what the system has to do to meet its overall requirements. A role describes an entity that performs some function within the system. In MaSE, each role is responsible for achieving, or helping to achieve specific system goals or sub-goals. MaSE roles are analogous to roles played by actors in a play or by members of a typical company structure. The company (which corresponds to system) has roles such as "president", "vice-president", and "mail clerk" that have specific responsibilities, rights and relationships defined in order to meet the overall company goal.

We chose to model multiagent systems using roles since roles are typically goal driven and map conveniently to agents. Because roles are goal-driven, we also chose to abstract the functional requirements into a set of system goals that can be passed on to the individual roles to carry out. A goal is an abstraction of a set of functional requirements. Typically a system has an overall goal and a set of sub-goals that must be achieved to reach the overall system goal.

The overall approach in the MaSE Analysis phase is fairly simple. Define the system goals from a set of functional requirements and then define the roles necessary to meet those goals. While a direct mapping from goals to roles is possible, MaSE suggests the use of Use Cases to help validate the system goals and derive an initial set of roles. The individual steps of the Analysis phase of Capturing Goals, Applying Use Cases, and Refining Roles are presented in Sections 2.1, 2.2, and 2.3 respectively.

2.1 Capturing Goals

The first step in the MaSE Analysis phase is Capturing Goals, which takes an initial system specification and transforms it into a structured set of system goals. In the context of the classic software lifecycle, this phase is concerned with system and software analysis. The initial system context is the collection of anything given to the analyst as a starting point for system analysis. We assume that the initial system context includes a software requirements specification that includes a well-defined set of functional requirements. Functional requirements tell the analyst the services that the system must provide and how the system should or should not behave based on inputs to the system and its current state [14]. The first step in the Analysis phase is to abstract the functional requirements into system goals. Our definition of a goal is similar to that described by Cockburn [1]; however, instead of focusing on the user's goal in using the system, we look at it from the system's point of view. The overall goal of the system is to fulfill the desires of the user. Therefore, if a user has a goal of "keeping track of possible login violations," the system goal would be to "inform user of possible login violations." Stating goals from the system's perspective seems to be more natural when talking about the system itself.

We chose to base the MaSE Analysis phase on goals because system goals are more stable than functions, processes, or information structures that often change with time [7]. Goals embody what the system is trying to achieve and generally remain constant throughout the analysis and design process. This is in contrast to other possible analysis objects, such as functions, that are organized around how something is done. In functional analysis, the details can be overwhelming and rapidly changing [7].

There are two sub-steps in Capturing Goals: identifying goals and structuring goals. First, goals must be identified from the initial system context. Next, the goals are analyzed and structured into a form that can be used later in the Analysis phase. Each sub-step is described in more detail below.

2.1.1 Identifying Goals

The first step in capturing goals is to capture the essence of an initial set of functional requirements. This process begins by extracting scenarios from the initial specification and describing the goal of that scenario. Assume we are given the following function requirements for security violations on a computer system [7].

- The system is responsible for dealing with host violations, in particular login violations and system file intrusions. The system administrator is notified of suspected or attempted intrusions.
- It is necessary to validate the date, time and existence of system files periodically, every few minutes. When a file is not found or a new version appears, the system administrator needs to be notified. When a user tries to modify or delete a system file, the system administrator needs to be notified.
- A user tries to login when he or she does not have a valid account. If this occurs once or twice in a short period of time, it is not a violation. Three or more attempts are a violation that needs to be reported.
- The system administrator may not be available to receive a notification. This can be due to a network failure or the fact that the administrator is performing another task. The report needs to be stored and resent after a delay.

An example of the goals derived from these requirements is shown below. Notice that all the details on how to perform system functions (e.g., "It is necessary to validate the date, time and existence of system files periodically, every few minutes") are not included as goals.

1. Inform administrator of file violations.
2. Inform administrator of login violations.
3. Detect invalid file deletion attempts.
4. Detect invalid file modification attempts.
5. Detect invalid login attempts.
6. Notify administrator of violations.

The purpose of using goals is that identify the critical aspects of the system requirements. Therefore, an analyst should specify goals as abstractly as possible without losing the essence of the requirement. This abstraction can be performed by removing detailed information when specifying goals. For example, to “Detect invalid login attempts” is a goal. How to detect invalid attempts is a requirement that may change with time or between various operating systems and is not a goal.

Once goals have been captured and explicitly stated, they are less likely to change than the detailed steps and activities involved in accomplishing them. These goals provide the foundation for the analysis model; all roles and tasks defined in later steps must support one of the goals identified in this step. If, later in the analysis, the analyst discovers roles or tasks that do not support an existing system goal, either the roles and tasks are superfluous or a new goal can be added to the goal set.

2.1.2 Structuring Goals

The final step in Capturing Goals is structuring the goals into a Goal Hierarchy Diagram, as shown in Figure 2. A Goal Hierarchy Diagram is a directed, acyclic graph where the nodes represent goals and the arcs define a sub-goal relationship. A goal hierarchy is not a tree since a goal may be a sub-goal of more than one parent goal. Each level in the diagram is intended to contain goal "peers" that are at approximately the same level of detail.

To develop the goal hierarchy, the analyst studies the initial set of goals for their importance and inter-relationships. Even though goals have been captured, they are of various importance, size, and level of detail. The Goal Hierarchy Diagram preserves such relationships, and divides goals into levels of detail and importance that are easier to manage and understand.

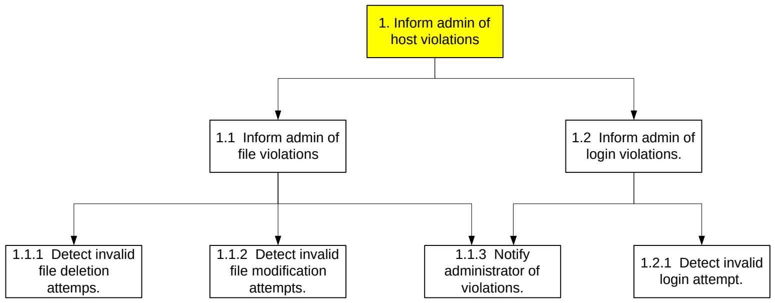
1. Inform admin of host violations
1.1 Inform admin of file violations
1.2 Inform admin of login violations.
1.1.1 Detect invalid file deletion attemps.
1.1.2 Detect invalid file modification attempts.
1.1.3 Notify administrator of violations.
1.2.1 Detect invalid login attempt.

Figure 2. Example Goal Hierarchy Diagram

The first step in building the hierarchy is to identify the overall system goal, which is placed at the top of the Goal Hierarchy Diagram. However, it is often the case, as in our example above, that a single system goal cannot be directly extracted from the functional requirements. In this case, the highest-level goals are summarized to create an overall system goal and the high level goals become sub-goals of the system goal. Once a basic goal hierarchy is in place, goals may be decomposed into new sub-goals. Each sub-goal must support its parent goal in the hierarchy. A goal is a valid sub-goal if it defines what the system must do to support its parent goal.

Goal decomposition is not simply "functional decomposition." Functional decomposition results in a set of steps to achieve a goal. For example, the steps required to implement the goal "Inform administrator of login violations" are to (1) detect all logins, (2) determine if they are valid, and (3) send a message to the administrator for all invalid logins. However, valid sub-goals include "Detect invalid login attempts" and "Notify administrator of violations." The facts that an invalid login is detected and the administrator is notified are the goals, how we detect invalid logins and notify the administrator are not. Goal decomposition continues until any further decomposition would result in a functional requirement instead of a goal (i.e., the analyst starts capturing how the goal should be accomplished). Once a decision of how a goal should be accomplished, the analyst has moved gone too far in the goal decomposition process.

There are four special types of goals in a Goal Hierarchy Diagram: summary, partitioned, combined, and non-functional. Goals can have attributes of more than one special goal type; however, they do not necessarily have to be one of these types at all.

A *summary goal* [1] is derived from a set of existing "peer" goals to provide a common parent goal. This often happens at the highest levels of the hierarchy. For instance, if the analyst decides that goals of "Inform admin of file violations" and "Inform admin of login violations" constitute the highest level of goals for the system, the analyst may abstract them further to create the summary goal "Inform admin of host violations", which is the overall system goal as shown in Figure 2.

Some goals do not directly direct support of the overall system goal, but are critical to the correct functioning of the system. These *non-functional goals* are often derived from non-functional requirements such as reliability or response times. For example, if a system must be able to find resources dynamically, a goal to facilitate locating dynamic resources may be required. While not central to the main goal of the system, this goal allows the system to meet its requirements. In this case, another "branch" of the Goal Hierarchy Diagram can be created and placed under an overall system level goal.

There are often a number of sub-goals in a hierarchy that are identical or very similar that can be grouped into a *combined goal*. For example, the initial goals "Inform administrator of file violations" and "Inform administrator of login violations" are combined in Figure 2 into the single goal of "Notify administrator of violations." In this case, the combined goal becomes a sub-goal of both the "Inform admin of file violations" and the "Inform admin of login violations" goals. By combining goals, the analyst can make the final system more understandable by combining similar functionality into specific roles or agents.

A *partitioned goal* is a goal with a set of sub-goals that, when taken collectively, effectively meet that goal. In essence, the sub-goals must cooperate to achieve their parent goal. While this is always true of summary goals, it may be true of any goals with a set of sub-goals. By defining a goal as "partitioned", it frees the analyst from specifically accounting for it in the rest of the analysis process. Partitioned goals are annotated in a Goal Hierarchy Diagram using a gray goal box instead of a clear box. For example, in Figure 2, Goal 1 is a partitioned goal since it is a summary goal.

At the conclusion of the Capturing Goals step, the system goals have been analyzed, captured, and structured in a Goal Hierarchy Diagram. The analyst can now move to the second step of the Analysis phase, Applying Use Cases, where the initial look at roles and communication paths takes place.

2.2 Applying Use Cases

The objective of the Applying Use Cases step is to capture a set of use cases from the initial system context and create a set of Sequence Diagrams to help the system analyst identify an initial set of roles and communications paths within the system. Use cases define basic scenarios that a system should be able to perform. The Sequence Diagrams capture the use cases as a set of events between the roles that make up the system. These event sequences are used later in the Analysis phase to define tasks that a particular role must accomplish. These tasks eventually find their way into the inter-agent conversations during the Design phase, thus ensuring that the use cases are implemented in the resulting multiagent system.

2.2.1 Creating Use Cases

The first step in Applying Use Cases is to extract use cases from the initial system context. Use cases define a sequence of events that can occur in the system. They are examples of how the user thinks the system should behave. Although part of the Applying Use Cases step, creating use cases may actually elicit more information or clarify existing information about system goals. If this happens, the analyst should immediately go back and add or modify the original Goal Hierarchy Diagram.

Use cases may already exist as part of the initial system context or they may have to be extracted by the analyst. The analyst may extract use cases from requirements specifications, user stories, or any other available source. While having a large number of use cases may be handy in helping to understand the system, it is important not to let the creation of use cases get out of hand. The goal of creating use cases is to identify paths of communication, not to define all possible combinations of events and data in the system. The analyst should attempt to gather enough use cases to cover as many possible event sequences without repeating the same sequence many times with different data or events. In general, the analyst should strive to show how each goal can be accomplished. The analyst should capture both positive and negative use cases. A positive use case describes what should happen during normal system operation. However, a negative use case still describes a desired sequence of events, but is illustrative of a breakdown or error. We are currently investigating the use of *obstacles* (as a dual concept to goals) [18] and their relation to negative use cases.

While use cases cannot be used to capture every possible requirement, they are an aid in deriving communication paths and roles. Cross checking the final analysis against the set of derived goals and use cases provides a redundant method for deriving required system behavior.

2.2.2 Creating Sequence Diagrams

A Sequence Diagram depicts the sequence of events that are transmitted between roles identified from use cases as shown in Figure 3. The boxes at the top of the diagram represent system roles and the arrows between the lines represent events passed between roles. Time is assumed to flow from the top of the diagram to the bottom. Therefore, in Figure 3, the FileViolation event is sent from the FileModifiedDetector to the FileNotifier and must precede the RequestNotification event that is sent to the AdminNotifier.

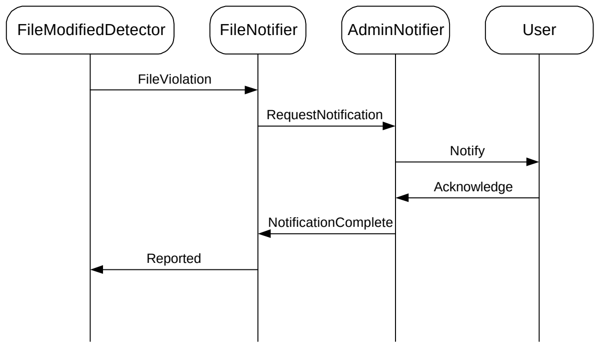

Figure 3: Sequence Diagram

Transformation from use cases to Sequence Diagrams is relatively straightforward. Individual entities named in the use case correspond to roles while any type of communications or information passing between use case entities becomes an event. The sequence of the events is based on the use case description. Every type of participant in a Sequence Diagram becomes a role. The roles identified in Sequence Diagrams form the initial set of roles used in the next step, Refining Roles, where they may be renamed, decomposed into multiple roles, or combined with other roles.

In general, one Sequence Diagram is created for each use case. However, if there are several possible execution sequences, multiple Sequence Diagrams may be created. For instance, if a use case has several alternate resolutions, such as "the diagnosis is sent from the doctor to the medical desk, and from the medical desk to the patient unless the patient is a minor, in which case it is sent to the patient's legal guardian from the medical desk", the analyst should create two similar but distinct Sequence Diagrams to define the use case. One use case could be used to describe what happens when the patient is a minor and the second could describe the more normal case.

After identifying the participating roles, creating the Sequence Diagram consists of reading through the use case and finding all instances of events that occurs between two of the roles. Each event in the use case is drawn as an arrow on the Sequence Diagram in the order that they occur. By applying use cases to create Sequence Diagrams, the main sequences of events from the use cases are explicitly accounted for in the concurrent tasks and conversations designed from these use cases.

2.3 Refining Roles

The objective of the last step of the Analysis phase, Refining Roles, is to transform the structured goals and Sequence Diagrams into roles and their associated tasks, which are forms more suitable for designing multiagent systems. Roles form the foundation for agent class definition and represent system goals during the Design phase. By using roles in this manner, the system goals are carried forward into the system design. It is our contention that system goals will be satisfied if every goal is associated with a role and every role is played by an agent class.

The general case transformation of goals to roles is one-to-one, with each goal mapping to a role. However, there are situations where it is useful to have a single role be responsible for multiple goals. There are many considerations in Refining Roles. Similar or related goals may be combined into single roles for the sake of convenience or efficiency. Commonplace goals often imply roles that can be reused from previous efforts. For example, in the case where a system must find resources dynamically, some type of facilitator role may be required. Facilitator roles are quite common and have been included in many multiagent systems. One mapping of the goals from our previous example to a set of roles is shown below.

FileNotifier	(1.1)
LoginNotifier	(1.2)
FileDeletionDetector	(1.1.1)
FileModifiedDetector	(1.1.2)
AdminNotifier	(1.1.3)
LoginDetector	(1.2.1)

Due to the simplicity of our example, we mapped goals to individual roles with a single exception; goal 1 was not mapped to a role since it was partitioned by Goals 1.1 and 1.2. In general, these decision on mapping goals to roles are based on detailed goal analysis. Possible considerations about when to combine and separate goals are detailed below.

Some goals may go unstated in the requirements and undiscovered until this point in the analysis. For example, interfacing with a user is a requirement that is often overlooked. Since a user interface requires special design techniques, it should be a separate role. If a goal is discovered at this point in system analysis, it should be added to existing goals as if it was part of the original system requirements. The previous steps, such as adding the new goal to the Goal Hierarchy Diagram, are then re-accomplished to keep the system analysis consistent.

Related goals can often be combined into a single role. For example, if we had decomposed our goals into “Notify administrator of file violations” and “Notify administrator of login violations”, we could have combined the roles into a single AdminNotifier role. While making the role more complex, combing goals into a single role simplifies the overall system design. This is a tradeoff that the analyst must make.

Interfacing with external or internal resources generally requires a separate role to act as an interface from a resource to the rest of the system. We generally consider a human user as an external resource. In MaSE we do not explicitly model human – computer interaction. However, we would suggest that a specific role be created to encapsulate the user interface. In this way, we can define the ways in which a user can interface with the system without defining the user interface itself. Other resources such as databases, files or legacy systems may also require their own interface role.

Role definitions are captured in a MaSE Role Model as shown in Figure 4, which includes information on interactions between role tasks and is more complex than traditional role models [8]. Roles are denoted by rectangles, while a role's tasks are denoted by ovals attached to the role. The detailed description of a task's definition is provided via Task Diagrams described in the next section. Lines between tasks denote (possibly named) communications protocols that occur between the tasks. The arrows denote the initiator/responder relationship of the protocol with the arrow pointing from the initiator to the respondent. Solid lines indicate peer-to-peer communications, which are generally implemented as external communications protocols. External protocols involve message passing between roles that may become actual messages if their roles end up being implemented in separate agents. These protocols are derived from the Sequence Diagrams developed in the previous step. Dashed lines denote communication between concurrent tasks within the same role. A lined is dashed if its protocols denote communications occurring only within the same instance of the role.

The tasks are generally derived from the goals for which a task is responsible. For instance, the FileDeletionDetector role is responsible for attaining goal 1.1.1, which is to "Detect invalid file deletion attempts." Therefore, to accomplish this goal, the role must be able to detect file deletion attempts and determine if they are valid. In this case, the designer has decided to separate these into two tasks: Detect File Deletions and Determine Validity.

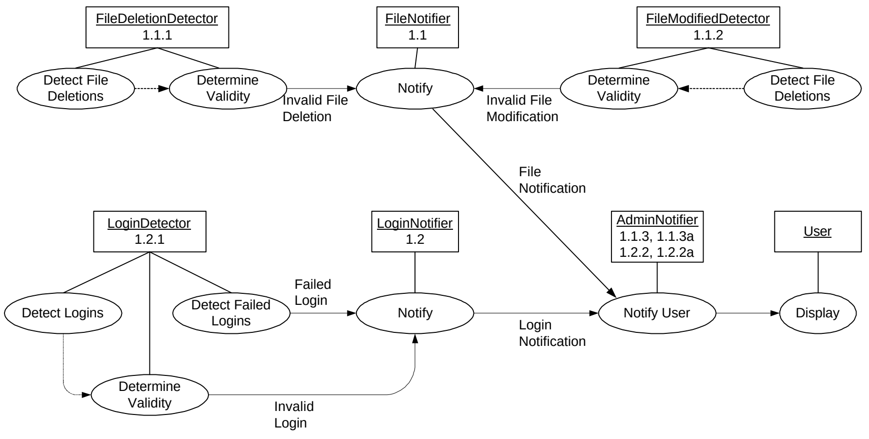

Figure 4: MaSE Role Model

Roles may not share or duplicate tasks. Sharing of tasks is a sign of improper role decomposition. Shared tasks should be placed in a separate role, which can be combined into various agent classes in the Design phase. This does not imply that the more general notion of a task cannot be distributed among various agents in the system. An agent in charge of satisfying a goal may distribute tasks among various agents capable of playing the appropriate role.

2.3.1 Concurrent Task Diagram

After roles are created, tasks are associated with each role that describe the behavior that the role must exhibit to successfully achieve its goals. In general, a single role may have multiple concurrently executing tasks that define the required role behavior. Each task specifies a single thread of control that defines a particular behavior that the role may exhibit and integrates inter- as well as intra-role interactions. Concurrent tasks are specified graphically using a finite state automaton, which we refer to as a Concurrent Task Diagram, as shown in Figure 5. We considered using Petri nets to model the tasks; however, we felt that finite state automata were generally easier to build and understand and provided a more straightforward translation to code.

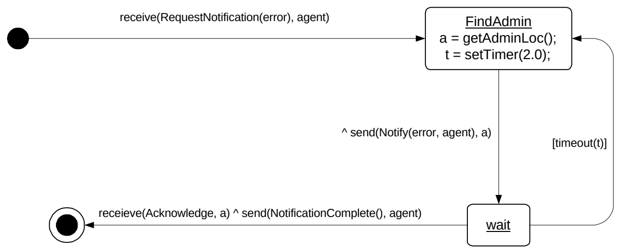
receive(RequestNotification(error), agent)
FindAdmin
a = getAdminLoc();
t = setTimer(2.0);
^ send(Notify(error, agent), a)
[timeout(t)]
receieve(Acknowledge, a) ^ send(NotificationComplete(), agent)
wait

Figure 5: Concurrent Task Diagram

There are two types of tasks: persistent and transient. A *persistent* task is a task that has a null transition from the start state to the first state. In other words, the task does not have an event that initiates its execution. We assume that persistent tasks start when the agent is initiated and continue executing until the agent itself is terminated or until the task reaches an end state. On the other hand, a *transient* task has a specific trigger on the transition from the start state. A transient task is not executed when the agent starts, but waits until its trigger is received by the agent. With transient tasks, it is possible to have multiple, concurrently executing tasks of the same type.

Concurrent tasks consist of states and transitions, which are similar to the states and transitions of most other finite automata models. States encompass the processing that goes on internal to the agent while transitions allow communication between agents or between tasks. A transition consists of a source state, destination state, trigger, guard condition, and transmissions and uses the syntax *trigger [guard] ^ transmission(s).* Multiple transmissions may be separated with a semicolon (;), however, there is no ordering of transmissions implied.

Generally, events specified in a trigger or transmissions are assumed to come from/to another task within the same role, thus allowing internal tasks to coordinate their behavior. However, two special events are used to indicate messages that are sent between agents: send and receive. The *send* (following the syntax *send(message, agent)*) event is used to send a message to another agent while the *receive* event (denoted as *receive(message, agent)*) signifies the receipt of such a message. The *message* is defined as a performative, which describes the intent of the message, along with a set of parameters that are the content of the message. The format of a message is *performative*(p_1 ... p_n) where p_1 ... p_n denotes n possible parameters. It is also possible to send a message to a group of agents via multicasting. Instead of specifying a single agent to send a message to, a group name is specified by enclosing the group name with braces (e.g., <group-name>) .

States may contain activities (or functions) that can be used to represent internal reasoning, reading a percept from sensors, or performing actions via effectors. Multiple activities may be included in a single state and are performed in sequence. Once in a state, the task remains in that state until activity processing is complete and a transition out of the state becomes enabled. Once processing starts in a state, all activities in the state must complete before any transitions out of the state are enabled.

The variables used in activity definitions in states and in message and event definitions on transitions are assumed to be globally visible within the task, but not outside of the task or within activities. All messages sent between roles and events sent between tasks are queued to ensure that all messages are received even if the agent or task is not in the appropriate state to handle the message or event immediately.

We also assume that each task is in exactly one state at any point in time. That means that transitions between states are instantaneous while states take time. If there are no activities in a particular state or all activities have been completed and no transitions have been enabled, then the task is idle, waiting on a transition to be enabled.

Concurrent tasks have predefined activities to deal with mobility and time. The move activity specifies that the agent is to move to a new address. The result of the move activity is a Boolean value that states whether the move actually occurred. It is possible that an agent may want to move to a new location but is unable to for some reason. The agent should be able to reason about this and deal with it accordingly. The syntax for the move activity is *Boolean = move(location).*

To reason about time, the concurrent task model provides a built in timer activity. An agent can define a timer using the setTimer activity. The *setTimer* activity takes a time as input and returns a timer that will timeout in exactly the time specified. The timer that can then be tested by the agent to see if it has timed out using the timeout activity. The *timeout* activity returns a Boolean value that is true if the timer has timed out. Using the setTimer and timeout activities, an agent can use time in carrying out its assigned responsibilities. The syntax for the setTimer and timeout functions is shown below.

t = setTimer(time)
Boolean = timeout(t)

Once a transition is enabled, it is executed and execution occurs instantaneously. This means that events and messages are sent instantaneously and the current task state becomes the destination state of the transition. If multiple transitions are enabled simultaneously, the following priority scheme is used.

1. Transitions whose triggers contain internal events from other tasks.
2. Transitions whose transmissions contain internal events.
3. Transitions whose trigger contains a receive message from other roles.
4. Transitions whose transmissions contain a message to another role.
5. Transitions with valid guard conditions only.

Figure 5 shows the *Notify User* task for the AdminNotifier role. The task is initiated upon receipt of a RequestNotification message from another agent. The error to be sent to the administrator is captured in the parameter, error. After the message is received, the task goes to the FindAdmin state where it locates the administrator and sets a timer. Once these activities are complete, the task sends a Notify message to the administrator, passing along the associated error. The task waits in the wait state until either the timer times out or an acknowledge message is received. If the timer times out, the task returns to the FindAdmin state and the exact same activities are re-accomplished. However, if an acknowledge message is received from the Administrator, the task simply sends a NotificationComplete message to the initiating task and the current task ends. Because the Notify User task is created based on a message receipt and terminates when it has completed, it is a *transient* task.

As discussed above, Sequence Diagrams define the minimum set of messages a role must respond to and send. The analyst can create an initial Concurrent Task Model from a scenario by taking the sequence of messages sent or received by that role and use them to create a sequence of corresponding states and messages. An example of the initial version of the Notify User task, derived directly from the Sequence Diagram in Figure 3, is shown in Figure 6. Obviously, the biggest differences between Figure 5 and Figure 6 are the addition of the parameters, activities, and the timeout capability, which was added for robust operation.

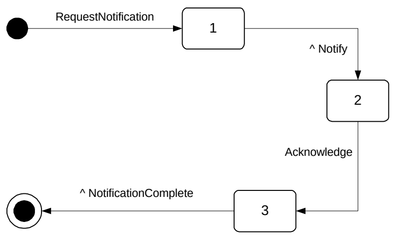

Figure 6. Initial Concurrent Task Diagram

After creating the initial concurrent task diagram, the analyst must determine the internal processing the role must perform to be able to satisfy the use case. This internal processing is captured as activities within the existing states. The analyst also fills in information about the data passed in the messages as well as any additional messages required for robust information exchange.

As tasks are created for each Sequence Diagram, the analyst may notice that several tasks are similar and can be combined. In this case, the analyst may combine multiple tasks into a single, generally more complex, task that can handle all of the use cases.

2.4 Analysis Phase Summary

Once Concurrent Task Models have been defined for each role, the Analysis phase is complete. Although there are three steps in the MaSE Analysis phase, the analyst is able, and even encouraged, to move freely between the steps. The MaSE Analysis phase can be summarized as follows:

1. Identify goals from user requirements and structure into a Goal Hierarchy Diagram.
2. Identify use cases and create sequence diagrams to help identify an initial set of roles and communications paths.
3. Transform goals into a set of roles
    a. Create a Role Model to capture roles and their associated tasks.
    b. Define a Concurrent Task Model for each task to define role behavior.

3. Design Phase

There are four steps to the designing a system with MaSE. The first step is Creating Agent Classes, in which the designer assigns roles to specific agent types. In the second step, Constructing Conversations, the actual conversations between agent classes are defined while in the third step, Assembling Agents Classes, the internal architecture and reasoning processes of the agent classes are designed. Finally, in the last step, System Design, the designer defines the actual number and location of agents in the deployed system. Each of these steps is discussed below.

3.1 Creating Agent Classes

In the Creating Agent Classes step of the Design phase, agent classes are created from the roles defined in the Analysis phase. The end product of this phase is an Agent Class Diagram, which depicts the overall agent system organization consisting of agent classes and the conversations between them. An agent class is a template for a type of agent in the system and is analogous to an object class in object-orientation. An agent is an actual instance of an agent class. During this step, agent classes are defined in terms of the roles they will play and the conversations in which they must participate.

At this point in the methodology, we simply identify the roles and tasks an agent class must play, the internal details of the agent are defined in the Assembling Agent Classes (Section 3.3). To ensure all the system goals are captured in the design, there must be at least one agent class assigned to play each role identified in the Analysis phase. In actuality, agent classes may play many roles, with the roles changing dynamically during execution. Furthermore, agents of the same class may play different roles at the same time.

Roles are the foundation upon which agent classes are designed. Since roles correspond to the set of system goals defined in the Analysis phase, roles form a bridge from what the system is trying to achieve (the Analysis phase and goals) to how it goes about achieving it (the Design phase agent classes). The analyst can easily change the organization and allocation of roles among agent classes during design, since roles can be manipulated modularly. This allows consideration of various design issues. For example, a high communication volume between two roles could imply that those roles should be part of the same agent class. In addition, two roles with large computational requirements are best be played by different agent classes so they can be executed on separate CPUs. Often these decisions are based on standard software engineering concepts such as functional, communicational, procedural, or temporal cohesion.

During this step, we also identify the conversations in which different agent classes must participate. Again, we do not define all the details; these are added during the Constructing Conversations step as described in Section 3.2. The set of conversations an agent class must participate in is derived from the external communications of the roles that the agent plays. For instance, assume roles A and B are defined by concurrent tasks and communicate with each other. Then, if agent 1 plays role A and agent 2 plays role B, the designer must define a conversation between agent 1 and 2 to implement the communication described between roles A and B.

The agent classes and conversations are documented via an Agent Class Diagram, which is similar to object-oriented class diagrams. There are two main differences. First, agent classes are not defined by attributes and methods; they are defined by the roles they play. The second difference is the semantics of the relationships between agent classes. In Agent Class Diagrams, all relationships between classes are conversations that may take place between two agent classes. A sample Agent Class Diagram is shown in Figure 7. The boxes in Figure 7 denote agent classes and contain the class name and the set of roles each agent plays. The lines with arrows identify conversations and point from the initiator of the conversation to the responder. The name of the conversation is written either over or next to the arrow.

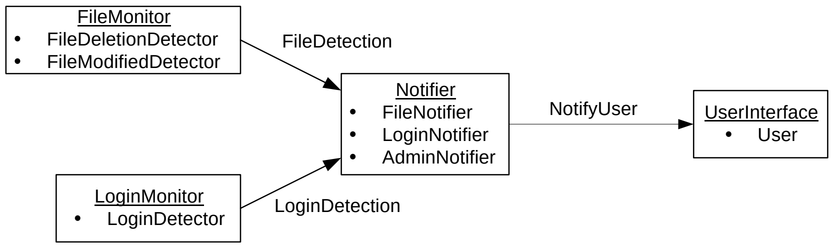

Figure 7: Agent Class Diagram

The Agent Class Diagram is the first design object in MaSE that depicts the entire multiagent system as it will eventually be implemented. If we have carefully followed MaSE to this point, the system represented by the Agent Class Diagram will support the goals and use cases identified in the Analysis phase. Of particular importance at this point is the system organization - the way that the agent classes are connected with conversations.

In Section 2, we stated that roles are the "foundation" for multiagent system design. If that is true, then agent classes are the "bricks" from which the system is actually built. The reason for these two different abstractions is that they provide the ability to manipulate two different system dimensions. Roles provide a way to allocate system goals apart from lower-level considerations. On the other hand, agent classes allow us to consider communications and other system resources, such as databases and external interfaces, without explicitly worrying about the system goals.

3.2 Constructing Conversations

Constructing Conversations is the next step in the MaSE Design phase. Up to this point, the designer has not defined communications between agents beyond stating that they exist. The fact that a conversation must happen between two agents is known; the goal of this step is to actually define the details of those conversations. The internal details of concurrent tasks are indispensable in this pursuit.

A MaSE conversation defines a coordination protocol between two agents. Specifically, a conversation consists of two Communication Class Diagrams, one each for the initiator and responder. A Communication Class Diagram is a finite state automaton that defines the conversation states of the two participant agent classes, as shown in Figure 8. The initiator always begins the conversation by sending the first message. When an agent receives a message, it compares it to its active conversations. If it finds a match, the agent transitions the appropriate conversation to a new state and performs any required actions or activities from either the transition or the new state. Otherwise, the agent assumes the message is a request to start a new conversation and compares it to all the possible conversations the agent can participate in with the agent that sent the message. If the agent finds a match, it begins a new conversation. The syntax of a transition follows conventional UML notation as shown below.

rec-mess(args1) [cond] / action ^ trans-mess(args2)

The above syntax states that if the message *rec-mess* is received with the arguments *args1* and the condition *cond* holds, then the method *action* is called and the message *trans-mess* is sent with arguments *args2*. All elements of the transition are optional. By analyzing the transition from the start state in Figure 8, it is obvious that it corresponds to an initiator half of a conversation since the transition from its start state is triggered by a message transmitted by the agent.

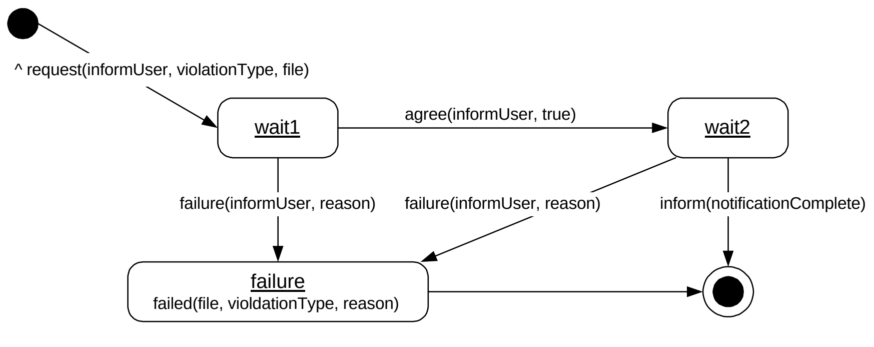
^ request(informUser, violationType, file)
wait1
agree(informUser, true)
wait2
failure(informUser, reason)
failure(informUser, reason)
inform(notificationComplete)
failure
failed(file, violdationType, reason)

Figure 8: Communication Class Diagram for FileDetection Conversation

Therefore, in Figure 8, the FileMonitor agent (the initiator of the FileDetection conversation as defined in Figure 7) sends a message to the Notifier agent requesting that it inform the user of a file violation. At this point, the FileMonitor agent enters a wait state waiting for a response. If the Notifier can notify the user, it sends an agree message and informs the FileMonitor when the notification has been completed. If the Notifier cannot inform the user for the FileMonitor, it returns a failure message with the appropriate reason (e.g., the user may have logged out of the system, or the network may be down, etc.). After receiving a failure message, the FileMonitor performs an internal call to the failed method.

The complimentary side of the conversation, from the point of view of the Notifier agent, is shown in Figure 9. In a well designed conversation, each possible sequence of messages sent/responded to by one side of the conversation must correspond to the messages sent/responded to by the opposite side. Conversations must be deadlock free. Besides deadlock, there are other ways to improperly design a conversation. For example, every message sent from one side of the conversation must be able to be received on the other half of the conversation. Additionally, the conversation must be able to exit every state, meaning that every state must have a valid transition from it that eventually leads to the end state. The agentTool system provides automatic verification of conversations during the design stage. Once a set of conversations has been created, the designer may choose to automatically verify them. The verification process beings with the automated transformation of the system conversations into the Promela modeling language. Then, the Promela model is automatically analyzed using the Spin verification tool to detect errors such as deadlock, non-progress loops, syntax errors, unused messages, and unused states [4]. Feedback from this process is provided to the designer automatically via text messages and graphical highlighting of error conditions. The topic of deadlock and the methods used in agentTool to avoid and detect it are covered in detail by Lacey [11, 12].

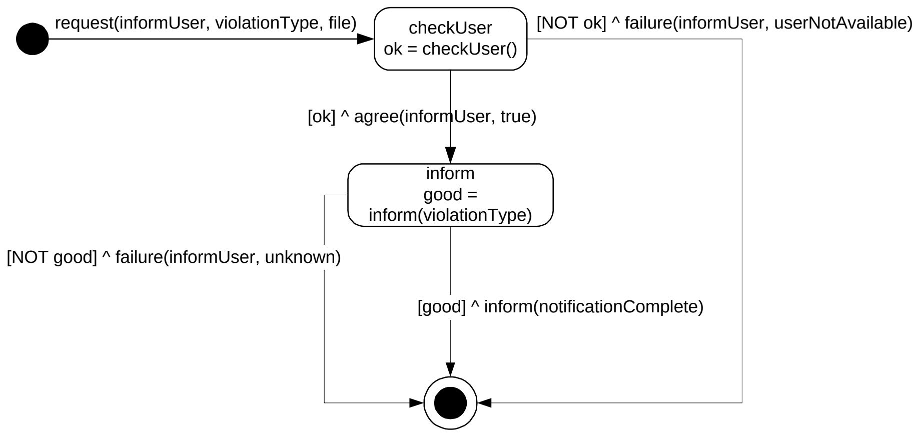
request(informUser, violationType, file)
checkUser
ok = checkUser()
[NOT ok] ^ failure(informUser, userNotAvailable)
[ok] ^ agree(informUser, true)
inform
good =
inform(violationType)
[NOT good] ^ failure(informUser, unknown)
[good] ^ inform(notificationComplete)

Figure 9. Communication Diagram for FileDetection Conversation (Part II)

As discussed above, the designer determines the set of conversations an agent class may participate in by the roles it plays. Likewise, the detailed design of conversations is derived from the concurrent tasks associated with those roles. Since a Concurrent Task Model specifies a single thread of control that integrates inter-role and intra-role interactions, they provide critical information required to define conversations. Basically each task that defines external communication creates one or more conversations. If all the communication within the task is with a single role, or set of roles that have all been mapped to a single agent class, the task can be mapped directly to a single conversation. More generally, however, concurrent tasks are more complex and consist of multiple conversations. The communications between separate roles or agents can be mapped to individual conversations.

Once all the information from Concurrent Task Models has been captured as part of conversations, the designer must ensure that other factors, such as robustness and fault tolerance, are taken into account. For instance, if a particular agent sends a message to another agent requesting some action be done, what happens if the other agent refuses or is unable to complete the request? The conversation should be robust enough to handle these possible problems.

In designing conversations, the designer faces a trade-off between having many simple conversations or a few complex ones. If the system has a large number of simple communications, these should be implemented by a series of smaller conversations. Larger and more complex conversations are only appropriate if an elaborate protocol is required.

3.3 Assembling Agents

During the Assembling Agents step of the Design phase, the internals of agent classes are created. This is accomplished via two sub-steps: defining the agent architecture and defining the components that make up the architecture. Designers have the choice of either designing their own architecture or using predefined architectures such as Belief-Desire-Intention (BDI). Likewise, a designer may use predefined components or develop them from scratch. Components consist of a set of attributes, methods, and, if complex, may have a sub-architecture.

An example of an Agent Architecture Diagram is shown in Figure 10. Architectural components (denoted by boxes) are connected to either inner- or outer-agent connectors. *Inner-agent connectors* (thin arrows) define visibility between components while *outer-agent connectors* (thick dashed arrows) define connections with external resources such as other agents, sensors and effectors, databases, and data stores. Internal component behavior may be represented by formal operation definitions as well as state-diagrams that represent events passed between components. The architecture and internal definition of the components must be consistent with the conversations defined in the previous step. At a minimum, this requires that each action or activity defined in a

Communication Class Diagram be defined as an operation in one of the internal components. The internal component state diagrams and operations can also be used to initiate and coordinate various conversations.

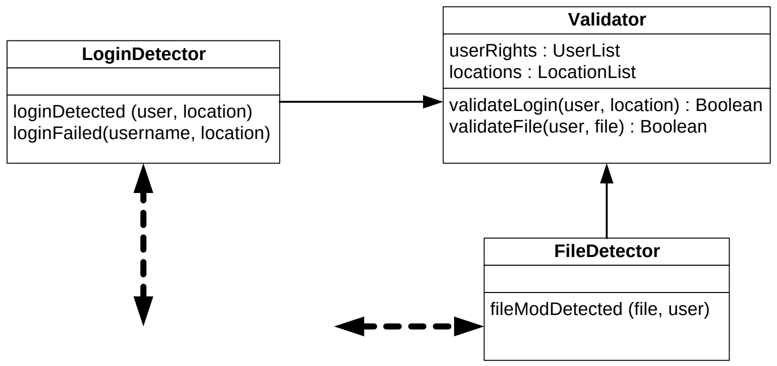
LoginDetector
loginDetected (user, location)
loginFailed(username, location)
Validator
userRights : UserList
locations : LocationList
validateLogin(user, location) : Boolean
validateFile(user, file) : Boolean
FileDetector
fileModDetected (file, user)

Figure 10. FileMonitor Agent Architecture

The FileMonitor agent architecture is shown in Figure 10. The FileMonitor agent has three components. The LoginDetector and FileDetector components work basically the same, interacting with the operating system to detect logins and file modification attempts. Both of these components *call* the Validator component to determine whether the login or file accesses were valid and need to be reported. The *outer-agent connectors* on the detector components denote both the fact that the components interact with the operating system as well as communicate with the Notifier agent via conversations.

While the designer may use existing architectures or design one from scratch, we are currently investigating deriving the agent architecture directly from the roles and tasks defined in the analysis phase [15]. This approach has the advantage of more directly mapping the analysis to the design while possibly losing some flexibility and reuse potential. Basically, each task from each role played by an agent defines a component in the agent class. The concurrent task itself is transformed into a combination of the component's internal state diagram and a set of conversations. Activities identified in the concurrent task become methods of the component.

3.4 System Design

The final step of the MaSE methodology takes the agent classes defined previously and instantiates actual agents. We use a Deployment Diagram to show the numbers, types, and locations of agents within a system. System design is actually the simplest step of MaSE, as most of the work was done in previous steps. The concept of instantiating agents from agent classes is similar to instantiating objects from object classes in object-oriented programming.

Deployment Diagrams describe a system based on agent classes defined in the previous steps of MaSE. Figure 11 shows a Deployment Diagram for our example system. The three-dimensional boxes represent agents while the connecting lines represent actual conversations between agents. The agents are identified by their class name or in the form of *designator:class* if there are multiple instances of a class. Any conversation between agent classes appears between agents of those classes. Furthermore, a dashed-line box indicates agents executing on the same physical platform.

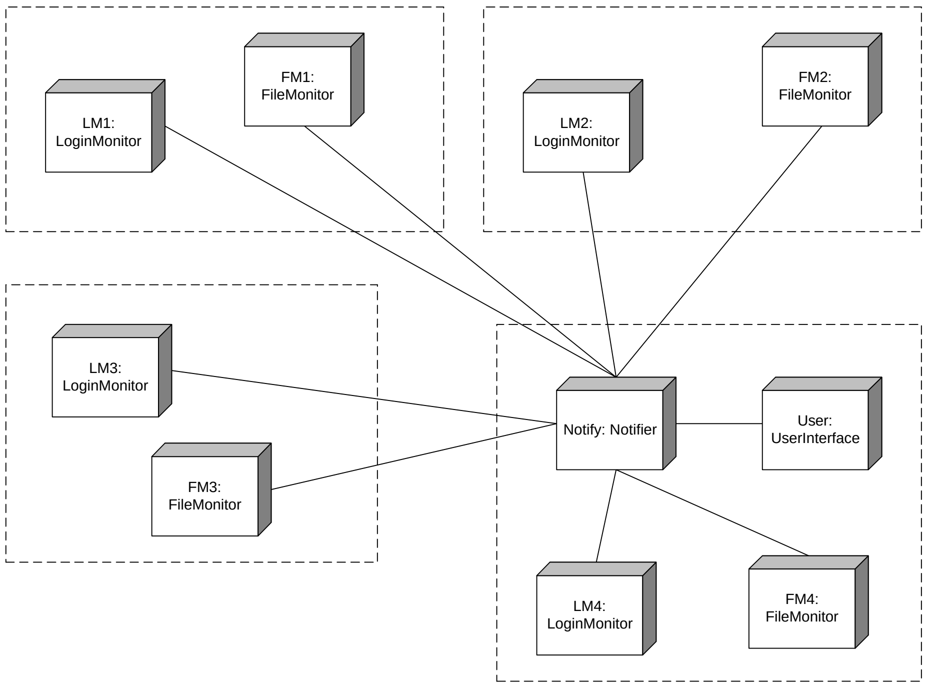

Figure 11: Deployment Diagram

A designer must define the system using a Deployment Diagram before implementing it since agents typically require the information contained in the Deployment Diagram, such as a hostname or address, in order to participate in external communications. Deployment Diagrams also offer an opportunity for the designer to tune the system to its environment. The designer can use Deployment Diagrams to define various configurations of agents and computers to maximize available processing power and network bandwidth. In some cases, the designer may specify a particular number of agents in the system or the specific computers on which certain agents must reside. The designer should also consider the communication and processing requirements when assigning agents to computers. To reduce communications overhead, a designer may choose to deploy agents on the same machine. However, putting too many agents on a single machine destroys the advantages of distribution gained by using the multiagent paradigm. Likewise, if an agent has high processing requirements, the designer can place it on a machine by itself. A strength of MaSE is that a designer can make these modifications after designing the system organization, thus generating a variety of system configurations.

3.5 Design Phase Summary

Once the Deployment Diagrams are finished, the Design phase is complete. The MaSE Design Phase can be summarized as follows:

1. Assign roles to specific agent classes, and identify conversations by examining Concurrent Task Models based on the roles played by each agent class.
2. Construct conversations by extracting the messages and states defined for each communication path in Concurrent Task Models, adding additional messages and states for added robustness.
3. Define the internals of agent classes by defining the architecture of each agent class using components and connectors. Ensure that each action defined in a conversation is implemented as a method within the agent architecture.
4. Define the final system structure using Deployment Diagrams.

4. Summary & Future Work

MaSE is a seven-step process that transforms a set of abstract models into a series of more concrete representations. It begins in the Analysis phase by capturing the essence of an initial system context in a structured set of goals and use cases. Next, the use cases are transformed into Sequence Diagrams so desired event sequences will be designed into the system. Finally, roles are identified from goals and use cases and include tasks, which describe how their associated goals are satisfied. The goal of the Design phase is to define the overall system organization by transforming the roles and tasks defined during analysis into agent types and conversations. Once the system organizational structure is defined, the internal structure of each agent class is defined. The final system configuration is defined in the system design step. Once again, we cannot overstress the importance of moving fluidly between steps and phases in MaSE to achieve the final goal - a robust, flexible, and efficient multiagent system.

Both MaSE and agentTool are works in progress. MaSE, along with our current version of agentTool, has been used to develop over a dozen multiagent systems ranging a few agents to over a hundred. The application areas range from information systems [9, 13] to biologically based immune systems [3] to recent work on teams of uninhabited air vehicles. The results have been promising. Users tell us that MaSE is relatively simple to use, yet is flexible enough to allow for a variety of solutions. We are currently using MaSE and agentTool to develop larger scale multiagent systems.

We are also currently extending MaSE to handle mobility and dynamic systems (in terms of agents being able to enter and leave the system during execution) [16]. We are also looking more closely at the relationship between tasks, conversations, and the internal design of agents [15]. Two additional areas that require further research are the use of obstacles to goals as a way to increase robustness and exception handling abilities and modeling of the information domain. As for agentTool, we are extending it to handle all phases and steps of MaSE including code generation. We are also looking at visualization techniques to make existing MaSE diagrams, or modified versions of them, easier to view and use in agentTool. The agentTool system already includes a code generator that generates complete conversations for multiple communication frameworks.

5. Related Work

There have been several proposed methodologies for analyzing, designing, and building multiagent systems [5], most of which are based on existing object-oriented or knowledge-based methodologies. In the following subsections, we compare MaSE against three of the better-known methodologies. Section 5.1 looks at the Gaia methodology offered by Wooldridge, Jennings, and Kinney, Section 5.2 evaluates the approach of Kinny, Georgeff, and Rao, and Section 5.3 compares MaSE against the MAS-Common KADS approach.

5.1 Gaia Methodology

One of the most recent attempts at defining a full methodology for the analysis and design of multiagent systems is Gaia [19]. The Gaia methodology views the system as a society or organization, with the elements of that society defined by roles. In Gaia, roles are initially captured in a prototypical role model, which are incrementally expanded and fully elaborated by the end of the analysis phase. These roles have direct correspondence to roles and role model defined in MaSE. A key difference between Gaia and MaSE is that Gaia provides no concrete way of determining the organization of the system or the type of roles that should exist in the organization. MaSE, on the other hand, develops the roles in a systematic manner using use cases and sequence diagrams. In MaSE it is often straightforward to recognize the required roles, relationships between roles, and goals for which the roles are responsible.

In Gaia, a role is defined by four attributes: responsibilities, permissions, activities, and protocols. *Responsibilities* determine the functionality of a role and are analogous to goals as defined in MaSE. *Permissions* are the "rights" (generally information resources) associated with a role and identify the resources that are available to a role in order to achieve its responsibilities. MaSE has no counterpart for permissions; however, we generally assume that each resource is encapsulated by a unique role that provides an interface to the rest of the system. Gaia *Activities* define a role’s computations that are carried out without interacting with other roles. Gaia activities correspond directly concurrent task activities. Protocols define how Gaia roles interact with each other and are defined by six attributes: purpose, initiator, responder, inputs, outputs, and processing. In MaSE, protocols are defined in more detail by modeling them as concurrent tasks.

In the design phase, Gaia only provides an abstract, high-level design. It consists of an Agent Model that identifies the agent types, the roles they implement, and their multiplicity within the system. This information is also captured in MaSE using Agent Class Diagrams and Deployment Diagrams. The Gaia Services Model identifies the main services of the agent type in terms of the inputs, outputs, pre- and post-conditions (as derived from safety properties). This does not have a direct parallel in MaSE although services and the details of the interactions (inputs and outputs) are defined in much more detail in MaSE tasks and conversations. Finally, the Gaia acquaintance model simply identifies lines of communications between agent types. In MaSE, this information is captured in the Agent Diagram, which also identifies the types of interactions as individual conversations. MaSE conversations go on to describe the structure of these communications, a level of detail not addressed by Gaia.

5.2 Belief-Desire-Intention (BDI)

An early attempt to define a multiagent systems methodology was developed by Kinney, Georgeff, and Rao [10]. They proposed a set of specialized Object-Oriented models for developing a system of Belief-Desire-Intention (BDI) agents. In this methodology, there are two sets of models: external and internal.

From the external viewpoint, the system is decomposed into agents, their responsibilities, the services they perform, the information they require, and their external interactions. These characteristics are captured in two models: the Agent Model and the Interactions Model. The *Agent Model* describes the hierarchical relationship between different abstract and concrete agent classes, and identifies the agent instances that may exist within the system, their multiplicity, and when they come into existence. The *Interaction Model* describes the responsibilities of an agent class, the services it provides, associated interactions, and control relationships between agent classes. The external viewpoint and associated models are captured in MaSE Agent Class Diagrams using agent classes and conversations.

From the internal viewpoint, the elements required by particular agent architectures are modeled for each agent using three models that describe its informational and motivational state and its potential behavior: the Belief Model, the Goal Model, and the Plan Model. The *Belief Model* describes the information about the environment and internal state that an agent of that class may hold, and the action is may perform. The *Goal Model* describes the goals that an agent may possibly adopt, and the events to which it can respond. Finally, the *Plan Model* describes the plans that an agent may possibly employ to achieve its goals or respond to events it perceives. It consists of a plan set which describes the properties and control structure of individual plans.

The biggest difference between MaSE and this approach lie in the focus on a particular agent type. MaSE is designed to be used with heterogeneous agents as opposed to a single agent architecture (i.e., BDI). In MaSE, attributes such as beliefs, desires, and intentions are initially captured in the analysis phase with goals and tasks. Agents cooperate with each other by passing messages, effectively changing each other's beliefs (which corresponds to a state transition in a task). These same concepts are carried forward into the design phase of MaSE as agent classes assume roles (and their associated goals) and the role tasks are translated into conversations and component state diagrams in the internal agent architecture. More complicated belief computations are easily added to the way MaSE handles states and their transitions.

5.3 MAS-CommonKADS

Another proposed multiagent system methodology is the MAS-CommonKADS methodology [6]. This methodology extends CommonKADS for multiagent systems by adding techniques from object oriented methodologies and protocol engineering. The general software engineering process combines a risk-driven approach with a component-based approach. The first phase of analysis is Conceptualization, where the analyst determines use cases from the initial user requirements and then formalizes them with Message Sequence Charts. The purpose of this phase is to capture roles and to develop an initial understanding of the interactions that must take place between those roles. This Conceptualization phase is analogous to the Applying Use Cases phase in MaSE.

After the Conceptualization phase, a system requirements specification is generated using six models, each consisting of constituents (entities to be modeled) and the relationships between them. The Agent model specifies the agent characteristics, which include reasoning capabilities, sensors, effectors, services, agent groups and hierarchies (both modeled in the organization model). The Task model describes the tasks that the agents can carry out and include goals, decompositions, ingredients and problem-solving methods. The Expertise model defines the information sources needed by the agents to achieve their goals. The Organization model specifies the organization and the social organization of the agent society. The Coordination model describes the conversations between agents: their interactions, protocols and required capabilities. Finally, the Communication model details the human-software agent interactions and the human factors required for developing these interfaces.

Many of the steps and models seem to be similar to those produced in MaSE. Unfortunately, there were no examples of most of the models and some of the modeling steps were not well defined, as some of the models described above were never mentioned in any particular step. One concept that MAS-CommonKADS fails to capture is the concurrency of the activities being performed by the agent roles. However, MAS-CommonKADS does appear to provide a more detailed definition of the methods or functions (here called tasks) and information sources associated with each role.

5.4 Summary

Based on the comparisons above, we believe that MaSE is a comprehensive methodology for the analysis of multiagent systems and provides solid foundation for the design and development of multiagent systems. MaSE not only takes advantage of goal-driven development, but also uses the power of multiagent systems by defining roles, protocols and tasks in the analysis phase. Another less obvious advantage that MaSE has is that its steps are defined at a fine level of granularity making the transition between models simpler and more straightforward than many of the techniques discussed above. MaSE also provides more guidance on how models relate to each other.

6. Acknowledgements

This research was supported by grants from the Air Force Office of Scientific Research and the Dayton Area Graduate Studies Institute. 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. A. Cockburn, "Structuring Use Cases with Goals," Journal of Object-Oriented Programming, Sep-Oct, 1997 and Nov-Dec, 1997.
2. S. A. DeLoach and M. Wood, "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.
3. P. K. Harmer, G. B. Lamont, G.B, "An Agent Architecture for a Computer Virus Immune System," in Workshop on Artificial Immune Systems at Genetic and Evolutionary Computation Conference, Las Vegas, Nevada, July 2000.
4. G. J. Holzmann, "The Model Checker Spin," IEEE Transactions On Software Engineering, vol. 23(5), pp. 279-295, 1997.
5. C. Iglesias, M. Garijo, and J. González, "A Survey of Agent-Oriented Methodologies," in Intelligent Agents V. Agents Theories, Architectures, and Languages, Lecture Notes in Computer Science, vol. 1555, J. P. Müller, M. P. Singh, and A. S. Rao (Eds.), Springer-Verlag, 1998.
6. C. Iglesias, M. Garijo, J. González, and J. Velasco, "Analysis and Design of Multiagent Systems using MAS-CommonKADS," in INTELLIGENT AGENTS IV: Agent Theories, Architectures, and Languages, Springer Verlag, Berlin Heidelberg, 1998.
7. E. A. Kendall, U. Palanivelan, and S. Kalikivayi, "Capturing and Structuring Goals: Analysis Patterns," Proceedings of the Third European Conference on Pattern Languages of Programming and Computing, Bad Irsee, Germany, July 1998.
8. E. A. Kendall, "Agent Roles and Role Models: New Abstractions for Multiagent System Analysis and Design," Proceedings of the International Workshop on Intelligent Agents in Information and Process Management, Bremen, Germany, September 1998.

9. S. C. Kern, M. T. Cox, and M. L. Talbert, "A Problem Representation Approach for Decision Support Systems," Proceedings of the Eleventh Annual Midwest Artificial Intelligence and Cognitive Science Conference, AAAI Press, Fayetteville, Arkansas, April 2000.

10. D. Kinny, M. Georgeff, and A. Rao, "A Methodology and Modelling Technique for Systems of BDI Agents," in Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '96. Lecture Notes in Artificial Intelligence, vol. 1038. Springer-Verlag, Berlin Heidelberg, 1996.

11. T. H. Lacey, and S. A. DeLoach, "Verification of Agent Behavioral Models," in Proceedings of the International Conference on Artificial Intelligence, CSREA Press, Las Vegas, Nevada, July 2000.

12. T. H. Lacey, and S. A. DeLoach, "Automatic Verification of Multiagent Conversations," in Proceedings of the Eleventh Annual Midwest Artificial Intelligence and Cognitive Science Conference, pp. 93-100, AAAI Press, Fayetteville, Arkansas, April 2000.

13. J. T. McDonald, M. L. Talbert, and S. A. DeLoach, "Heterogeneous Database Integration Using Agent Oriented Information Systems," in Proceedings of the International Conference on Artificial Intelligence, CSREA Press, Las Vegas, Nevada, July 2000.

14. I. Sommerville, Software Engineering, Pearson Education Limited, Essex England, 2001.

15. Self, "Design & Specification of Dynamic, Mobile, and Reconfigurable Multiagent Systems," MS thesis, AFIT/ENG/01M-11. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base Ohio, USA, March 2001.

16. C. H. Sparkman, "Transforming Analysis Models into Design Models for the Multiagent Systems Engineering Methodology," MS thesis, AFIT/ENG/01M-21. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base Ohio, USA, March 2001.

17. K. P. Sycara, "Multiagent Systems," AI Magazine vol. 19(2), pp. 79-92, 1998.

18. A. van Lamsweerde, and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering," IEEE Transactions on Software Engineering vol. 26(10), pp. 978-1005, 2000.

19. M. Wooldridge, N. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," Journal of Autonomous Agents and Multi-Agent Systems. vol. 3(3), 2000.