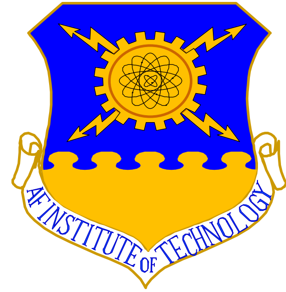
AFIT/EN-TR-00-02
TECHNICAL REPORT
June 2000



MULTIAGENT SYSTEMS ENGINEERING: THE ANALYSIS PHASE

Scott A. DeLoach and Mark F. Wood

GRADUATE SCHOOL OF ENGINEERING AND MANAGEMENT
AIR FORCE INSTITUTE OF TECHNOLOGY
WRIGHT-PATTERSON AIR FORCE BASE, OHIO

Approved for public release; distribution unlimited

Abstract.

This report describes the Analysis phase of the Multiagent Systems Engineering (MaSE) methodology. MaSE is a general purpose, methodology for developing heterogeneous multiagent systems. The goal of MaSE is to guide a system developer from an initial system specification to a multiagent system implementation. This is done by directing the designer through this set of inter-related system models. Although the majority of the MaSE models are graphical, the underlying semantics clearly and unambiguously defines specific relationships between the graphical models.

MaSE uses a number of graphically based models to describe system goals, behaviors, agent types, and agent communication interfaces. MaSE is designed to be applied iteratively. Under normal circumstances, we would expect a designer to move through each step multiple times, moving back and forth between models to ensure each model is complete and consistent. While this is common practice using most design methodologies, MaSE was specifically designed to support this process by formally capturing the relationships between the models.

MaSE is independent of a particular multiagent system architecture, agent architecture, programming language, or communication framework. Systems designed using MaSE can be implemented in a variety ways. For example, a system could be designed and implemented that included a heterogeneous mix of agent architectures using any one of a number of existing underlying communication frameworks. The ultimate goal of MaSE and agentTool is the automatic generation of code that is correct with respect to the original system specification.

.

Table of Contents

1 INTRODUCTION **1**

2 MULTIAGENT SYSTEMS ENGINEERING **2**

2.1 CAPTURING GOALS 3

2.1.1 Identifying Goals *4*

2.1.2 Structuring Goals *5*

2.2 APPLYING USE CASES 7

2.2.1 Creating Use Cases *7*

2.2.2 Creating Sequence Diagrams *8*

2.3 REFINING ROLES 9

2.3.1 Concurrent Task Model *12*

2.4 ANALYSIS PHASE SUMMARY 15

3 EXAMPLE **15**

3.1 REQUIREMENTS 15

3.2 CAPTURING GOALS 16

3.3 APPLYING USE CASES 17

3.4 REFINING ROLES 19

4 SUMMARY **21**

5 RELATED WORK **22**

6 FUTURE RESEARCH **23**

7 ACKNOWLEDGEMENTS **23**

REFERENCES **23**

Table of Figures

FIGURE 1. MASE PHASES 3

FIGURE 2: GOAL HIERARCHY DIAGRAM 4

FIGURE 3: STRUCTURED GOALS 6

FIGURE 4: GOAL HIERARCHY DIAGRAM 6

FIGURE 5: SEQUENCE DIAGRAM 8

FIGURE 6. ROLES 10

FIGURE 7. TRADITIONAL ROLE MODEL 12

FIGURE 8: MASE ROLE MODEL 13

FIGURE 9: REGISTRATION TASK 13

FIGURE 10. INITIAL CONCURRENT TASK DIAGRAM 14

FIGURE 11: GOAL HIERARCHY DIAGRAM 17

FIGURE 12: EGADS SEQUENCE DIAGRAM 18

FIGURE 13: EGADS ROLE MODEL 20

FIGURE 14. INITIAL COMPLEX ROLE MODEL 20

FIGURE 15. HANDLETASKINGS TASK MODEL 21

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 tools focused on specific agent architectures (Drogoul & Collinot 1998) (Kinny, Georgeff, & Rao 1996) (Nwana, Ndumu, Leel & Collis 1999) or have not gone to the necessary level of detail to adequately support complex system development (Iglesias, Garijo, & Gonzalez 1998) (Wooldridge, Jennings, & Kinny 2000). In our research, we have been developing both a complete-lifecycle methodology and a complimentary environment for analyzing, designing, and developing heterogeneous multiagent systems. The methodology we are developing is Multiagent Systems Engineering (MaSE) while the tool we are building to support that methodology is agentTool (DeLoach & Wood 2000).

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 (Sycara 1998) lists six challenges of multiagent systems:

1. How to decompose problems and allocate tasks to individual agents.
2. How to coordinate agent control and communications.
3. How to make multiple agents act in a coherent manner.
4. How to make individual agents reason about other agents and the state of coordination.
5. How to reconcile conflicting goals between coordinating agents.
6. How to engineer practical multiagent systems.

Our research in Multiagent Systems Engineering (MaSE) is an attempt to answer the sixth challenge, how to engineer practical multiagent systems, and to provide a framework for solving the first five challenges. It uses the abstraction provided by multiagent systems for developing intelligent, distributed software systems. A second goal of this research is to define a methodology specifically for formal agent system synthesis. To accomplish the first 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.

2 Multiagent Systems Engineering

The Multiagent System Engineering (MaSE) methodology takes an initial system specification, and produces a set of models in a graphically based style. 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. The MaSE methodology 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 in Figure 1 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 may move between steps and phases freely and that each successive pass will produce additional detail providing a complete, yet consistent system design.

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.

While MaSE covers analysis and design, we concentrate only on the Analysis phase in this report. We are currently preparing a complimentary report on the Design phase. The individual steps of the Analysis phases are discussed in Sections 2.1, 2.2, and 2.3. A more complete example of using MaSE for system analysis is presented in Section 3.

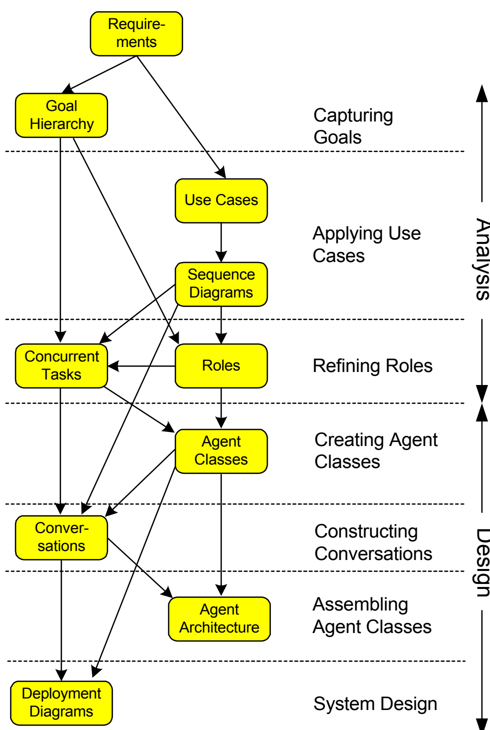


Figure 1. MaSE Phases

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, such as that described by Pressman (Pressman 1992), this phase is concerned with system and software analysis. The MaSE Analysis phase is goal-based because goals are generally a more stable structure than functions or processes, which may change over time. Goals embody *what* the system is trying to achieve and generally remain constant throughout the rest of 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 (Kendall, Palanivelan & Kalikivayi 1998).

In MaSE, a *goal* is always defined as a system-level objective. Lower-level constructs may inherit or be responsible for goals, but goals always have a system-level context. In consequence, every action

within a system must support a particular goal. A goal is a statement that is usually phrased as a declaration of system intent, as if it began with the words: "The system shall…"

The *initial system context* is the collection of anything given to the analyst as a starting point for system analysis. It may come from a variety of sources and may take many forms, such as a formal requirements document or a collection of user stories. It is the conceptualization of the system from the user's point of view. The initial system context is the input to the Capturing Goals step. Pieces of the initial context may alternately be referred to in a more descriptive manner such as "the system requirements" or "the user specification".

The product of the Capturing Goals step is a structured hierarchy of goals called a Goal Hierarchy Diagram, as shown in Figure 2. The *Goal Hierarchy Diagram* is modular to allow for modifications, additions, and deletions. The arrows denote sub-goals of a higher-level goal. Each level in the diagram is intended to contain goal "peers" that are at approximately the same level of detail. Though a Goal Hierarchy Diagram may initially appear to be a tree, it can be more complex. As discussed later, identical sub-goals may be combined to avoid redundancy, thus combining nodes of a branch and creating a directed, acyclic graph.

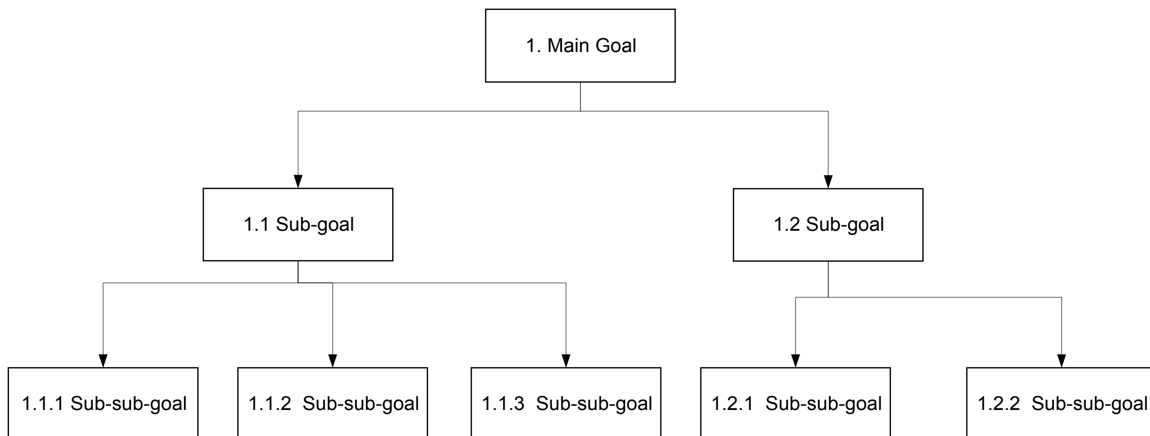


Figure 2: Goal Hierarchy Diagram

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 distill the essence of a set of requirements from the initial system context. As discussed above, this context may be drawn from detailed technical documents, user stories, or formal specifications. This process begins by extracting scenarios from the initial specification

and describing the goal of that scenario. An example of a list of goals for a computer intrusion detection system is shown below.

- Detect and notify administrator of host violations.
- Detect and notify administrator of system file violations.
- Determine if files have been deleted or modified.
- Detect user attempts to modify files.
- Notify administrator of violations.
- Ensure the administrator receives notification.
- Detect and notify administrator of login violations.
- Determine if an invalid user tries to login.
- Notify administrator of login violations.
- Ensure the administrator receives notification.

Once these goals have been captured and explicitly stated, they are less likely to change than the detailed steps and activities involved in accomplishing them. These system 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, a new goal can be created and added to the goal set.

Goals should be specified as abstractly as possible without losing the essence of the requirement. An analyst can perform this abstraction by removing detailed information when specifying goals. For instance, the overall goal of a system might be to "Determine if an invalid user tries to login." Thus, one of the sub-goals is to detect login violations. *How* to detect violations is a requirement that may change with time or between various operating systems and should not be included as a goal or sub-goal.

2.1.2 Structuring Goals

The final step in Capturing Goals is structuring the goals into a Goal Hierarchy Diagram. To accomplish this structuring, the analyst studies the 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.

The first step is to identify the overall system goal, which is placed at the top of the Goal Hierarchy Diagram. If there are two or more main goals without a single system goal, an overarching system goal should be created to encompass all the other goals. Each sub-goal must pertain to its parent goal in the hierarchy. In other words, they must relate to the same functions or mode of operation as their parent, but at a lower level of granularity.

Goals should be decomposed into smaller sub-goals to the point where it appears plausible that each goal might be handled by a simple agent. While there is no hard and fast rule for how far to decompose goals, it is generally better to go too far in decomposing the goals than not far enough. As discussed in

Section 2.2, roles are derived from the goals in the Goal Hierarchy Diagram. If a goal has not been decomposed enough, there is no way to break it down in later steps. If the goals have been decomposed too far, they can be easily grouped back together when forming roles from goals.

Finally, some goals are clearly not in direct support of the primary system goal, but are important parts of the system. For example, if a system must be able to find resources dynamically, a goal to help facilitate finding dynamic resources may be required. While not central to the main functional goal of the system, this facilitator goal allows the system to meet its other requirements. In that case, another "branch" of the Goal Hierarchy Diagram is created and placed under an overall system level goal. The goals in Figure 3 can be easily re-written as a Goal Hierarchy Diagram since they are based on a numerical hierarchy. The primary goal, goal 1, has sub-goals 1.1, 1.2, etc. The analyst need only remove duplicate goals from the tree to form a Goal Hierarchy Diagram. A typical Goal Hierarchy Diagram is shown in Figure 4. Notice that goals 1.1.3.a and 1.2.2.a from Figure 3 have been combined in Figure 4.

1. Detect and notify administrator of host violations.
    - 1.1 Detect and notify administrator of system file violations.
        - 1.1.1 determine if files have been deleted or modified.
        - 1.1.2 Detect user attempts to modify files.
        - 1.1.3 Notify administrator of violations.
            - 1.1.3a Ensure the administrator receives notification.
    - 1.2 Detect and notify administrator of login violations.
        - 1.2.1 Determine if an invalid user tries to login
        - 1.2.2 Notify administrator of login violations
            - 1.2.2.a Ensure the administrator receives notification

Figure 3: Structured Goals

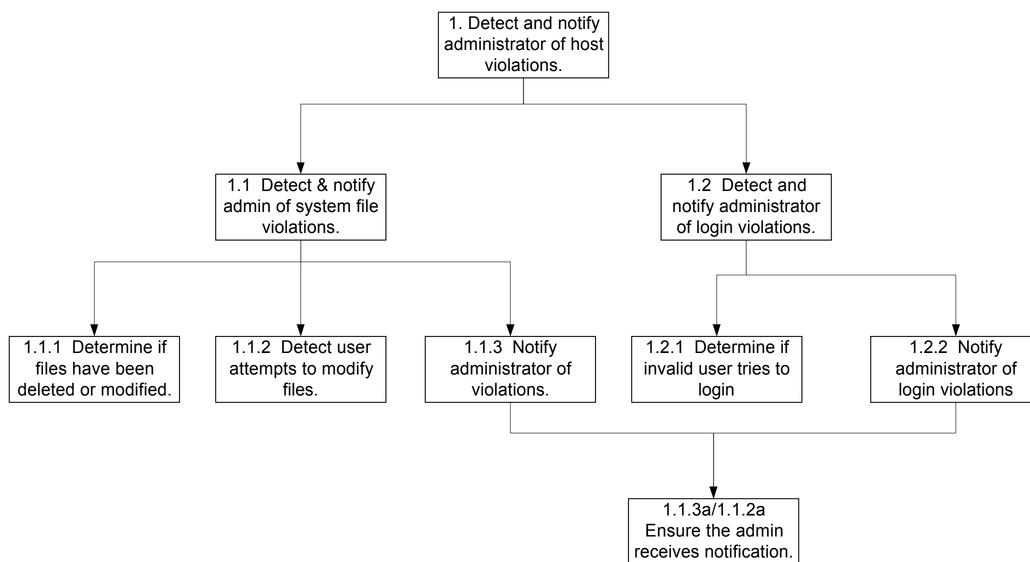


Figure 4: Goal Hierarchy Diagram

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 in identifying 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 should be able to occur in the system. They are examples of how the user, or requirements author, thinks the system should behave. Although part of the Applying Use Cases step, creating use cases may 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. Use cases can be extracted from the requirements specification, user stories, or another available source. If the user is available, this is the time to ask, "What should the system do if *this* happens?" 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 potential paths of communication, not to define all possible combinations of events and data in the system. A good guideline is that every sequence of system events that differs significantly from others should be formed into a use case. A significant difference could be the involvement of different participants in an event sequence, a data stream that goes in a reverse direction, or a different ordering of events in the sequence. The analyst should attempt to gather enough use cases to cover as many possible sequences of events as possible without repeating the same sequence many times with different types of data or events.

The analyst should be sure to capture both positive and negative use cases. The most common form of a use case is the *positive use case*, which describes what should happen during normal system

operation. However, a *failure use case* still describes a desired sequence of events, but is illustrative of a breakdown or error. Failure use cases are similar to exception handling in some programming languages. If the analyst does not capture both positive and failure use cases, the result will be an incomplete system description.

2.2.2 Creating Sequence Diagrams

A Sequence Diagram depicts the sequence of events that are transmitted between roles identified from use cases. A *role* is an abstract description of an entity's expected function and contains the system goals that it is responsible for fulfilling. Roles are created to *do* something (Kendall 1998) and are analogous to roles played by actors in a play or by members of a typical company structure. The company has roles such as "president", "vice-president", and "mail clerk" arranged in a hierarchy. Roles have more than just titles; they have certain responsibilities associated with them as well. For instance, the "mail clerk" is charged with delivering the mail within the company. These responsibilities are associated with particular goals, so a role can be thought of as an abstraction that encompasses a particular goal or set of goals. Eventually, roles will be mapped to agents who are responsible for satisfying those goals.

An example of a Sequence Diagram is shown in Figure 5. 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.

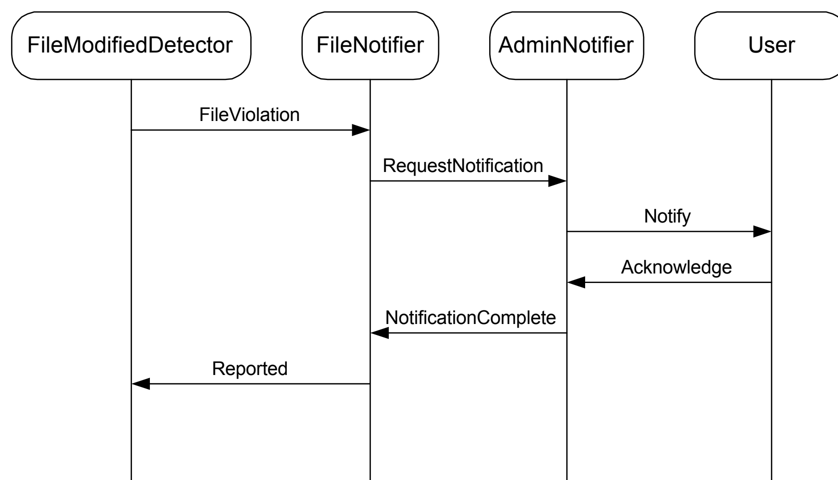


Figure 5: Sequence Diagram

A Sequence Diagram is used to determine the minimum set of events that must be passed between roles. If an event is passed between two roles, then there must be a corresponding communication path between them. A high communication volume between two roles may suggest combining the roles into a single agent class, since agents that play these roles inherit the communications between them. In

addition, a communication path between roles played by separate agent classes means that a conversation must exist between the two agent classes to pass the event. The agent class playing the role that initiated the communication becomes the *initiator* of that conversation, and the receiving agent class becomes the *responder*.

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 participant in a Sequence Diagram becomes a role. The roles identified in Sequence Diagrams form the initial set of roles used by the next step, Refining Roles. In Refining Roles, roles identified in this step may be renamed, decomposed into multiple roles, or composed with other roles.

In general, one Sequence Diagram is created for each use case. Typically, it is only possible to create one sequence from a use case. However, if there are several possibilities, then 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.

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 conversations designed from these use cases. Furthermore, since Sequence Diagrams operate between the system roles, their creation defines required communication paths between the roles and eventually agent classes.

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.

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 in a multiagent

system, 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.

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. Common goals imply particular roles, and may be reused. 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 have been included in many multiagent systems (Finin, Labrou, & Mayfield 1997).

An example of a mapping a set of goals to roles is shown in Figure 6. In general, we mapped goals from Figure 4 to individual roles with a couple of exceptions. Goals 1.1.2, 1.1.2.a, 1.1.3, and 1.1.3.a were combined into a single role, while Goal 1 was not mapped to a role since Goals 1.1 and 1.2 completely partitioned it, thus completely satisfying Goal 1. These types of decisions are based on detailed goal analysis. Possible considerations about when to combine and separate goals are detailed below.

Some implied goals may go unstated in the requirements and undiscovered until this point in the analysis. For example, interfacing with a user is likely a requirement that is often overlooked. A user interface implies a role by itself (and thus a goal), and should be a separate role. If an implied 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.

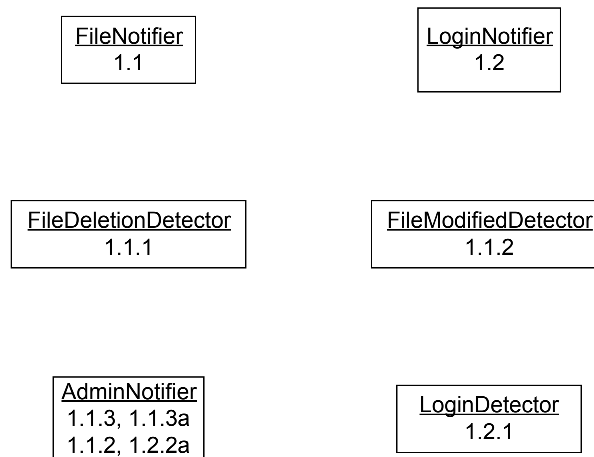


Figure 6. Roles

Related goals can often be combined into a single role. For example, given the following set of constraints,

- no student or instructor may have a class during both the first and last meeting hours of the day
- no course may have less than 3 students
- all students must carry at least eight credit hours

the analyst might choose to combine them all into a single role of “Constraint Enforcer”. Of course, the analyst could have originally combined the three goals into a single goal of “the schedule must meet all applicable constraints”. This is an example where creating too many goals in the Capturing Goals step is compensated for in the Transforming Goals to Roles step. Either solution is acceptable.

In the case that sub-goals actually partition their parent goals (the logical conjunct of the sub-goals is equivalent to the parent goal), the parent goal need not be mapped to its own role. For instance, none of the roles in Figure 6 play the part of the main goal “Detect and notify administrator of host violations” since goals 1.1 and 1.2 (see Figure 5) taken together completely satisfy the overall system goal.

Some goals imply distributed roles. Any mention of separate machines or other type of distribution generally requires a role for each “side” of the distributed relationship. Interfacing with an external or internal resource is similar. One role is required as an interface to the resource while a second is necessary to interface with the rest of the system. In addition, data persistence may imply an information server role with an associated client role as well.

Broadcasting a single message to multiple recipients implies the need for a broadcast manager of some sort to register the members of the broadcast group and to distribute the messages. While this role could be combined with a facilitator role during the Design phase or even handled by the underlying communication framework, it should be a separate role in the Analysis phase.

There should be a role defined for each use case participant that supports the goal the use case is trying to achieve. These roles often exist but have different names. In this case, the names should be changed to a common name to reduce ambiguity.

Any interface with a system user requires a role. In MaSE we do not explicitly model human – computer interaction, we leave than to user interface designer. However, we do encapsulate the user interface with a role. In this way, we can define the ways in which a user can interface with the system without defining all the nuances of the user interface.

Role definitions are captured in a traditional Role Model (Kendall 1998) as shown in Figure 7. MaSE also allows a more complete version of a Role Model, as shown in Figure 8, which includes information on interactions between role tasks. However, the traditional version of the Role Model is more useful at the outset of the role definition process before tasks have been defined, as well as later in the analysis to provide a high-level view of the system. In the traditional Role Model, lines between roles

denote possible communications paths between roles. These paths are derived from the Sequence Diagrams developed in the previous step. Notice we have added a user role to interface with the system administrator.

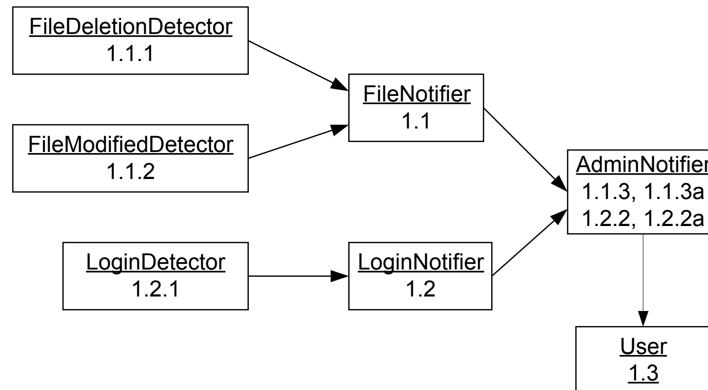


Figure 7. Traditional Role Model

In MaSE, roles are typically documented in a more detailed version of a Role Model as shown in Figure 8. First, the goals associated with each role are listed under the role name. It also shows the set of tasks associated with each role, which are used to define the role's behavior. Roles are denoted by rectangles, while the role tasks are denoted by ovals attached to the role. Tasks are simply identified in the MaSE Role Model. The detailed description of a task's definition is provided 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. Dashed lines denote communication between concurrent tasks within the same role. A lined is dashed if it will only occur within the same instance of the role in the final system. 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.

2.3.1 Concurrent Task Model

After roles are created, tasks are associated with each role that describe the behavior that a role must exhibit to successfully achieve its goal. In general, a single role may have multiple concurrent tasks that define the required role behavior. Task definition is usually performed after role creation since, to accomplish their goals, tasks must communicate with other tasks, both in other roles as well as within the same role.

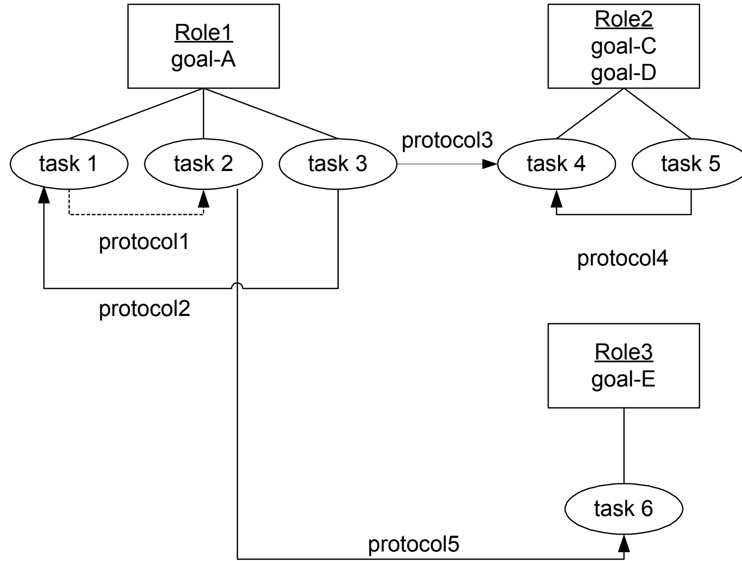


Figure 8: MaSE Role Model

We specify role behavior as a set of *n* concurrent tasks. Each task specifies a single thread of control that defines a particular behavior that the role may exhibit. Tasks also integrate inter-role 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 9. All tasks are assumed to start execution upon startup of the role and continue until the role terminates or an end state is reached.

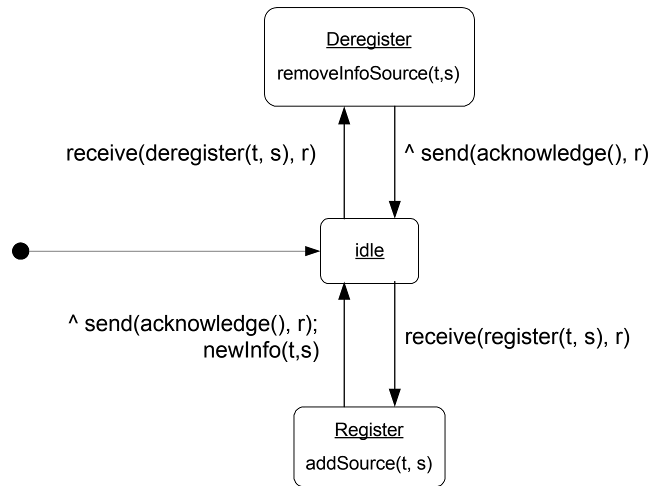


Figure 9: Registration Task

Activities are used to specify actual functions carried out by the role and are performed inside the task states. While these tasks execute concurrently and carry out high-level behavior, they can be coordinated using internal events. Internal events are passed from one task to another and are specified on the transitions between states. To communicate with other roles, external messages can be sent and received. These external messages are specified using *send* and *receive* events. These events send and

retrieve messages from the message-handling component of the role. Besides communication with other roles, tasks can interact with the environment via reading percepts or performing operations that affect the environment. This interaction is typically captured in functions defined in the states. By including reasoning within tasks, roles are not “hardwired” or purely reflexive. They can plan, search, or use knowledge-based reasoning to decide on appropriate actions

Figure 9 shows a sample task for a registrar role. The task begins in the start state and automatically transitions to the *idle* state. Once in the *idle* state, the role can perform two activities – register or deregister an external resource. If the registrar receives a *register* message from role r (where t is the type of information source and s is the source address), the task adds the resource to the registration list via the *addSource* activity and sends an *acknowledge* message to role r. The *newInfo* event on the transition from the *Register* state informs a second task in the registrar role that a new source has been added. The de-registration process takes place in much the same way.

As stated above, tasks define what a role must do to accomplish its goals. The Sequence Diagrams defined in the Applying Use Cases phase are used as the starting point to help the analyst in defining tasks. 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 for a scenario by taking the sequence of messages (sent or received) and creating a sequence of corresponding states and messages. An example of the initial concurrent task derived from the Sequence Diagram in Figure 5 for the *AdminNotifier* role is shown in Figure 10.

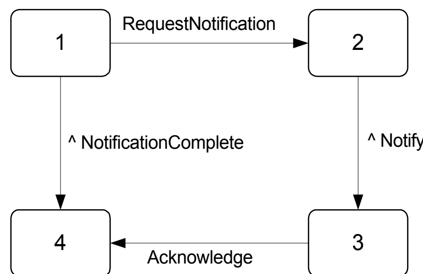


Figure 10. Initial Concurrent Task Diagram

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. The analyst also fills in information about the data passed between roles via messages. Additional messages may also be added 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. If multiple tasks are created it is possible that the tasks

will also have to coordinate their activities. Intra-role coordination must be added at this point in terms of events between tasks.

2.4 Analysis Phase Summary

Once Concurrent Task Models have been defined for each role, the Analysis phase is complete. The MaSE Analysis phase can be summarized as follows:

1. Capture system level goals by
    a. Identifying goals from user requirements,
    b. Structuring goals into a Goal Hierarchy Diagram.
2. Identify use cases and create sequence diagrams to help identify an initial set of roles and paths of communications between them.
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.

As discussed earlier, although there are three steps in the MaSE Analysis phase, the analyst is able, and even encouraged, to move freely between the steps.

3 Example

To further illustrate the MaSE methodology, a single example system, the Electronic Intelligence Gathering and Decision System (EGADS), is presented. The example goes through all three steps in the Analysis phase to provide further clarification on how the methodology works. The requirements of this example are presented in Section 3.1. The requirements form the initial system context and are the inputs to the Analysis phase. The initial set of goals are derived in Section 3.2, use cases and Sequence Diagrams are detailed in Section 3.3, and roles and tasks are defined in Section 3.4.

3.1 Requirements

EGADS links a Commander to intelligence gathering assets in the field. A group of dissimilar data collectors must communicate intelligence data to the commander via an analyst in an intelligence processing unit. The commander can issue *taskings* asking for specific data or status reports. The results of these taskings are received as processed intelligence reports from the intelligence processing unit.

Details:

- The system must link to many different kinds of intelligence assets including airborne sensors, imaging sensors, land and sea-based radar, satellites, and preprocessed intelligence data.

- The commander's taskings may include requests about a particular area, resource (air, land, and sea units), or combination. They may be one-time requests, or the commander may define a time window within which all movements must be reported.
- A commander may request a status report of any open intelligence tasking. The report will return completed portions of the tasking, and status of uncompleted portions including an estimated completion time, which may be unknown.
- A commander may set preferences to determine how all reports (status and intelligence) will be presented.
- Before presentation to the commander, all intelligence reports must be sent through the intelligence processing unit.
- Data sources must be able to be added or removed from the system as they become available or obsolete.

Example Scenario:

A commander desires to know what sort of air defenses will exist in a target area of an air strike that must be executed within 72 hours. At stake is when will the strike be launched and what additional sorties must be deployed to suppress enemy air defenses.

3.2 Capturing Goals

The EGADS system requirements define the overall goal and several general requirements and constraints. Determining the purpose for each scenario listed in the requirements identifies certain system goals. By analyzing the requirements and the initial scenario, the EGADS system analyst has extracted the goals shown below.

- Allow the commander to issue intelligence taskings and receive intelligence reports
- Link to non-homogeneous intelligence assets
- Accept tasking for areas, resources, and combinations
- Accept one-time and time-window taskings
- Allow the commander to request and receive status reports
- Allow the commander to set presentation preferences
- Ensure that intelligence reports go to the intelligence unit before commander
- Allow addition and deletion of data sources

In EGADS, the main goal is the first one, "Allow the commander to issue intelligence taskings and receive intelligence reports." The analyst determines this from the initial statement of the system requirements: "EGADS links a Commander to intelligence gathering assets in the field."

The analyst also notices that many of the other system goals support the primary goal. For example, the ability of a commander to request and receive status reports about intelligence taskings is subordinate to the goal of initiating intelligence taskings and receiving intelligence reports. Furthermore, there are many goals in EGADS that are easily decomposed into sub-goals. The "request and receive status

reports" goal identified above breaks readily into "request" and "receive". The analyst also decides that the example scenario implies that an intelligence analyst must be able to access raw intelligence data for processing and enter results into the system.

In EGADS, the ability to connect with a changing set of many different data sources is an important system goal, and although not directly specified as part of the initial system context, it is required for EGADS to perform properly. Therefore, the analyst creates a new sub-branch in the Goal Hierarchy Diagram (1.4) to handle this derived goal. In the example, the process of structuring goals described above continues and the system analyst comes up with the Goal Hierarchy Diagram shown in Figure 11.

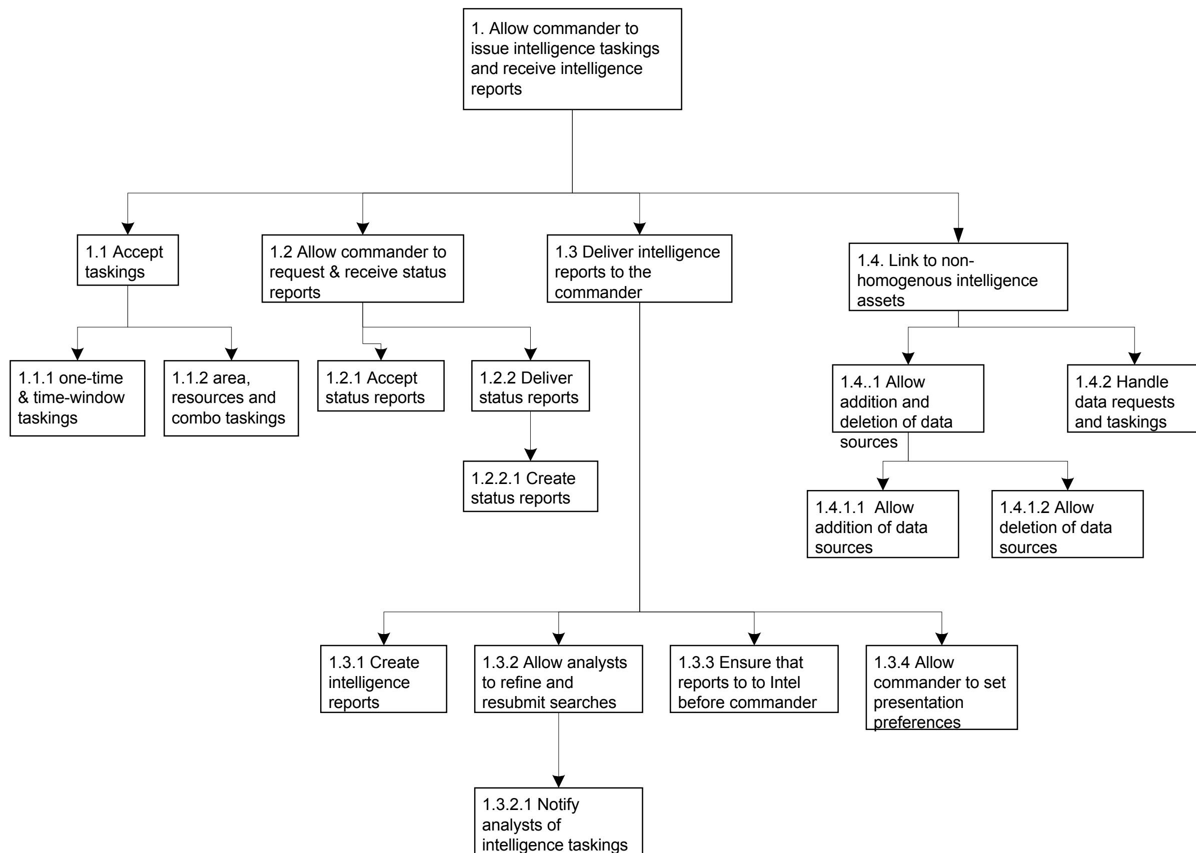


Figure 11: Goal Hierarchy Diagram

3.3 Applying Use Cases

As described in Section 2.2, transforming a use case into a Sequence Diagram is straightforward. The example EGADS scenario from Section 3.1 translates nicely into a use case. Since the example suggests a sequence of events, but does not specify them, the analyst must ask the user to provide detail on the intended sequence. The results are documented in the following use case.

A commander desires to know what sort of air defenses will exist in a target area of an air strike that must be executed within 72 hours. At stake is when will the strike be launched and what additional sorties must be deployed to suppress enemy air defenses.

The commander prepares and executes an intelligence tasking specifying air defense units, the target zone, and a 72-hour time window with a due date of 6 hours. The tasking is distributed to mission controllers who have the ability to detect such units. In this case, the tasking is sent to a satellite controller and a joint intelligence data controller.

Both controllers have data on the position available. The joint intelligence data is immediately available and the satellite information takes a few hours to receive. A status request by the commander during this period would indicate this. The data indicates a medium anti-aircraft presence and a light infantry-based surface-to-air missile presence.

The data, once obtained, is then collated and sent to the intelligence processing unit. An intelligence analyst works the request and knows from experience that historical data may be useful in such situations. She expands the search to backtrack several days and widens the search area to cover more of the opposition's forces, but focuses the search to only check the faster-retrieving joint intelligence data in order to make the 6-hour due date. The new data indicates that two units of mobile surface-to-air missiles are heading toward the target area from rear positions, and are expected to be operational in 48 hours.

The entire report is sent to the commander, who realizes an attack within the next 36 hours would not require usage of extensive Wild-Weasel (anti surface-to-air missile) assets. He orders the attack.

The Sequence Diagram created by the EGADS analyst from the use case above is shown in Figure 12. The use case has clearly identified participants, with the exception of the Task Controller role. The analyst created this role for the express purpose of accepting tasks from the commander and parceling out requests to appropriate Mission Controllers.

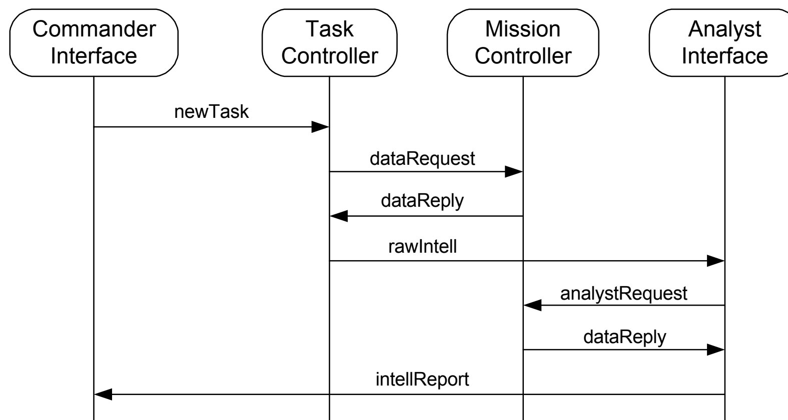


Figure 12: EGADS Sequence Diagram

After identifying the participating roles, creating the Sequence Diagram consists of reading through the use case and finding all instances of an event that occurs between two of the roles. Events identified in the use case are drawn as arrows on the Sequence Diagram in the order they occur. This analysis is straightforward until the EGADS analyst runs into the part of the use case concerning the intelligence

analyst. In this case, the intelligence analyst has the choice of whether or not to resubmit a search for more data. The question is whether or not the EGADS analyst requires a second Sequence Diagram to capture the scenario when the analyst decides to resubmit for a new search. However, since the two-event resubmission sequence would not break the order of the other events, the analyst decides that a new Sequence Diagram is not needed since it is an "alternate resolution". In other words, the addition or deletion of the resubmission (*analystRequest*) events does not affect the sequence of the other events. Therefore, the sequence that includes a resubmission subsumes one that does not. The Sequence Diagram is also consistent with the system goal that the Commander can never receive raw intelligence data. An intelligence report can only be sent to the Commander by the intelligence unit.

The analyst continues developing use cases and creating Sequence Diagrams until all the major scenarios are covered. Sometimes these are given by the user, but more often the details have to be extracted from other requirements and discussions with the user. Once a sufficient number of use cases have been developed, the analyst can move on the next step in the methodology, refining roles and tasks.

By applying the use cases to create Sequence Diagrams, all potential sequences of events are captured and made available for developing the tasks and conversations. Furthermore, since Sequence Diagrams operate between the system roles, their creation helps define the roles and communication paths in the next step.

3.4 Refining Roles

The EGADS analyst takes the Goal Hierarchy Diagram and Sequence Diagrams defined in the last two steps and creates the roles shown below (the parenthesis indicate goals associated with each role).

- Commander Interface (1, 1.2, 1.3, 1.3.4)
- Analyst Interface (1.3.1, 1.3.2, 1.3.2.1)
- Mission Controller (1.4)
- Data Source Interface (1.4.2)
- Status Reporter (1.2.1, 1.2.2, 1.2.2.1)
- Registrar (1.4.1, 1.4.1.1, 1.4.1.2)
- Task Controller (1.1, 1.1.1, 1.1.2, 1.3.3)

The Commander Interface and Analyst Interface were created because separate roles are needed to provide and interface to the various system users. The Mission Controller and Data Source Interface roles form two sides of a distributed relationship, since it is clear that the data sources must reside across a network. The Status Reporter and Registrar roles support particular goals and were participants in use cases identified earlier. Finally, the Task Controller role is created as a representative of a particular task. The Task Controller is the "system" with which the other roles talk. Each task controller ensures that its intelligence task is formatted, handled, and routed correctly.

The initial version of the Role Model was defined from the roles described above (Figure 13). The communications paths were derived from the Sequence Diagrams defined in the previous step.

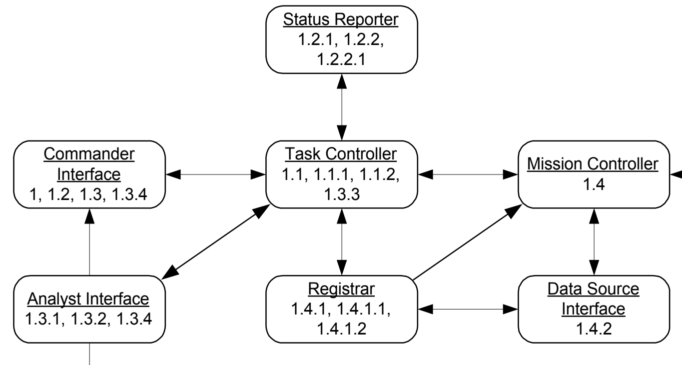


Figure 13: EGADS Role Model

Next, the analyst sets out to define the tasks required to implement the use cases. Using the Sequence Diagram in Figure 12 as an example, the analyst created a set of tasks, which are added to the Role Model as shown in Figure 14. The basic communications paths have been removed and replaced with specific protocol paths between individual tasks. In this instance, as is generally the case, the analyst created one task for each role participating in the use case.

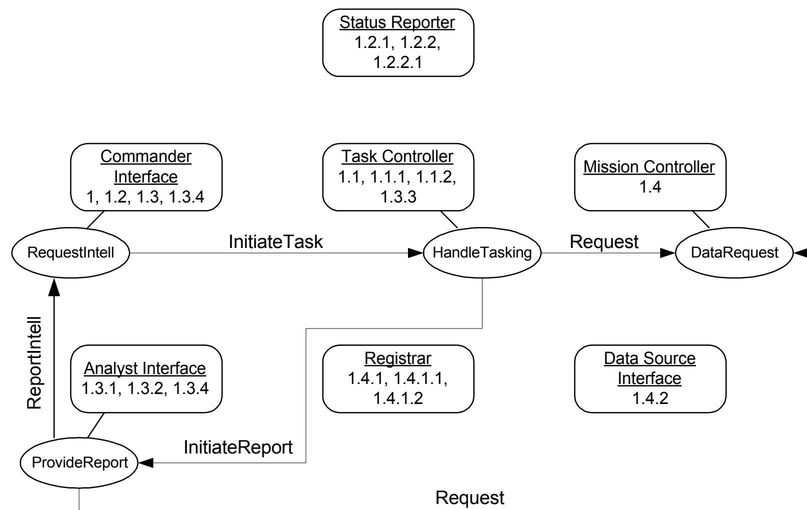


Figure 14. Initial Complex Role Model

Once the tasks have been identified, the analyst can go about defining the internal details of the tasks, which define each role's behavior in the use case. The HandleTaskings task is the most interesting task of the four identified and we use it here to illustrate the definition of a Concurrent Task Model (Figure 15). When creating this task, the analyst started with the associated Sequence Diagram (Figure 12). The Commander Interface role initiates the scenario by sending a *newTask* message to the Task

Controller, which starts the HandleTaskings task. After starting the task, the Sequence Diagram requires the Task Controller to send a *dataRequest* message to a Mission Controller. However, upon further investigation into the use case, the analyst found that it was actually possible to send *dataRequest* messages to multiple Mission Controllers based on whether they had the capability to collect the required data. Therefore, the analyst added the ability to look up existing controllers and send messages to each of them (the <list> notation in the Concurrent Task Model indicates a multicast to multiple recipients indicated by the list). The Concurrent Task Model then waits and collects all the data from the Mission Controllers and sends it, along with the tasking, on to the Analyst Interface role for processing by the intelligence analyst. For robustness, the analyst also added the ability for the task to return a *noSources* message to the Commander Interface if there were no Mission Controllers with the ability to handle the intelligence request. An alternate solution would be to send the tasking on the Analyst Interface role with no additional data.

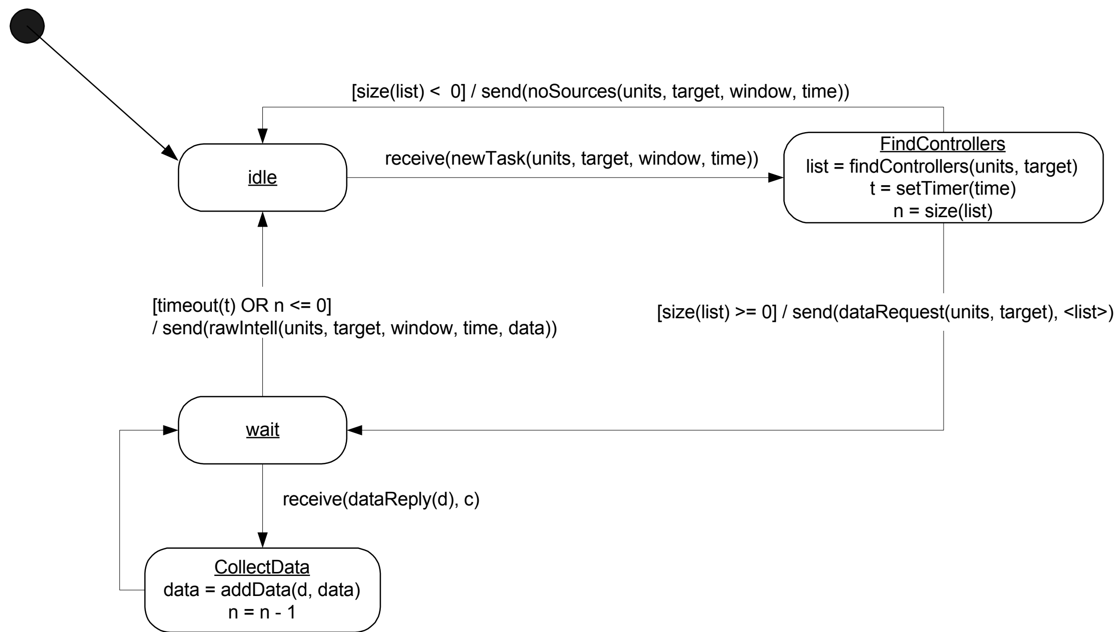


Figure 15. HandleTaskings Task Model

The EGADS analyst would continue to create detailed task definitions, possibly combining them into more complex tasks until all tasks were defined. When task creation is finished, the MaSE Analysis phase is complete. At this point, the analyst has a complete description of the system goals as well as the roles and the tasks required to satisfy those goals.

4 Summary

MaSE is a seven-step process designed around transforming a set of abstract models into a more detailed set of models until, eventually, a concrete system design has been developed. 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. Upon completion of role and task definition, the Analysis phase of MaSE is complete. However, 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.

5 Related Work

There have been several proposed methodologies for analyzing, designing, and building multiagent systems (Iglesias, Garijo, & Gonzalez 1998). The majority of these are based on existing object-oriented or knowledge-based methodologies. Since MaSE is derived from the object-oriented paradigm, we only compare MaSE with those here. In fact, we restrict our comparison to the widely published Gaia methodology (Wooldridge, Jennings, & Kinny 2000).

MaSE is more detailed than Gaia and provides more guidance to the system analyst. The first step in Gaia is to extract roles from the problem specification without real guidance on how this is done. MaSE, on the other hand, develops Use Cases and a Goal Hierarchy to help define what roles should be developed. In MaSE, roles are generated to carry out the goals derived from the initial system specification.

The use of roles by both methodologies is similar. It is used to abstractly define the basic components within the system. In Gaia, roles define functionality in terms responsibilities – functions plus invariants. A Concurrent Task Model defines role functionality in MaSE. A Concurrent Task Model (with added invariants) effectively models the same thing as a Gaia role responsibility. It describes "potential trajectories" in terms of state-based functions and messages. In fact, Concurrent Task Models provide more detail in terms of detailed messages – with the associated information – sent between roles.

Gaia permissions define the use of resources by a particular role (including limitations such as read, change, generate, etc.) as well as the ability to parameterize the resource. This seems to be missing in MaSE although there is the suggestion that an information agent be defined for each information resource to the system. The Gaia approach may be better here. If that is the case, we should consider adding resources to a MaSE role definition.

Gaia uses activities to model computations performed within roles and protocols to define interactions with other roles. These are captured with MaSE tasks. In fact, MaSE tasks provide a lot more detail on when activities are done, the information input and output from the activities, and how the activities relate to the interaction protocols.

As stated above, Gaia does not go to the detail that MaSE does. Specifically, Gaia does not exhibit the following MaSE concepts:

- Goals
- Use Cases
- Sequence Charts describing system operation
- Detailed protocol information
- Message types
- Data passed
- Sequencing information

6 Future Research

As stated throughout this report, MaSE and agentTool are works in progress. MaSE, along with our current version of agentTool, has been used to develop five to ten small to medium sized multiagent systems ranging from information systems (Kern, Cox, & Talbert 2000) (McDonald, Talbert, & DeLoach 2000) to biologically based immune systems (Harmer & Lamont 2000). 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). We are also looking more closely at the relationship between tasks, conversations, and the internal design of agents. As for agentTool, we are extending it to handle all phases and steps of MaSE including code generation. We currently have a code generator that generates complete conversations for a single communication framework. We are also working on integrating agentTool with the AFIT Wide Spectrum Object Modeling Environment that is looking at the more general code generation problem.

7 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] DeLoach, Scott A. and Mark Wood. "Developing Multiagent Systems with agentTool," The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL-2000). Boston, MA, July 2000.

[2] Drogoul, A., and Collinot A. 1998 "Applying an Agent Oriented Methodology to the Design of Artificial Organizations: A Case Study in Robotic Soccer" *Autonomous Agents and Multi-Agent Systems*, 1(1), 113-129

[3] Finin, T., Labrou, Y. and Mayfield, J. KQML as an Agent Communication Language. In: *Software Agents*, J.M. Bradshaw (Ed.), Menlo Park, Calif., AAAI Press, 1997, pages 291-316.

[4] Harmer, P.K., Lamont, G.B.: An Agent Architecture for a Computer Virus Immune System. Genetic and Evolutionary Computation Conference (2000)

[5] Iglesias, C., Garijo, M., Gonzalez, J.: A Survey of Agent-Oriented Methodologies. Intelligent. In: Müller, J.P., Singh, M.P., Rao, A.S., (Eds.): Intelligent Agents V. Agents Theories, Architectures, and Languages. Lecture Notes in Computer Science, Vol. 1555. Springer-Verlag, Berlin Heidelberg (1998)

[6] Kendall, E.A., U. Palanivelan, S. Kalikivayi: Capturing and Structuring Goals: Analysis Patterns. European Pattern Languages of Programming. (1998)

[7] Kendall, E.A.: Agent Roles and Role Models: New Abstractions for Multiagent System Analysis and Design. International Workshop on Intelligent Agents in Information and Process Management (1998)

[8] Kern, S.C.G., Cox, M.T., Talbert M.L.: A Problem Representation Approach for Decision Support Systems. Proceedings of the Eleventh Annual Midwest Artificial Intelligence and Cognitive Science Conference (2000) 68-73

[9] Kinny, D., Georgeff, M., Rao, A.: A Methodology and Modelling Technique for Systems of BDI Agents. 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) 56-71

[10] McDonald, J.T., Talbert, M.L., DeLoach, S.A.: Heterogeneous Database Integration Using Agent Oriented Information Systems. Proceedings of the International Conference on Artificial Intelligence (2000)

[11] Nwana, H., Ndumu D., Leel, & Collis J. "ZEUS: A Toolkit for Building Distributed Multi-Agent Systems," *Applied Artificial Intelligence Journal*, Vol. 13(1), p 129-185, 1999.

[12] Sycara, K. P. 1998, Multiagent Systems. *AI Magazine* 19(2): 79-92.

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