Extendable and Composable Units for Multi-Agent Coordination (ECUMAC)

Trong Khiem Tran

A Thesis

in

The Department of Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 05

Your file Votre référence
ISBN: 0-494-04453-5
Our file Notre référence
ISBN: 0-494-04453-5

Abstract

Extendable and Composable Units for Multi-Agent Coordination (ECUMAC)

Trong Khiem Tran

As multi-agent systems evolve, coordination among agents becomes crucial in the execution of tasks to ensure the correctness of such system. Much effort is spent in ensuring the correctness in the interaction of the agents. Such efforts usually result in very complex designs which increase the maintenance cost or the cost for further development. In this thesis we propose a model to leverage the effort spent in maintenance and extension by promoting modularity of the system for improved understandability and reusability to save effort. The model, called ECUMAC, is based on the concept that a set of coordination requirements can be realized by a set of "small coordination achievement" called the coordination units. A coordination unit is a skeleton description of a coordination pattern which the application developer can further refine to suit the specific nature of the application. ECUMAC defines a model of these coordination units so that they are extendable and composable to support modularity and code reuse. Two types of coordination units have been identified: static coordination unit and dynamic coordination unit. The static coordination unit allows the definition of coordination structures which require a fixed pattern of interaction. On the other hand, the dynamic coordination unit relies on the spontaneous reaction of the agents to occurrence of certain state of the system.

TABLE OF CONTENTS

List of Figures

1. Introduction

As multi-agent systems evolve, coordination among agents becomes crucial in the execution of a task to ensure the correctness of the system or to improve efficiency of the system by avoiding duplicated work. Research has been conducted to define coordination models which allow agent-system developers to specify the coordination behavior of the agents within the system. However, these models provide only low-level primitives which do not provide enough separation of concern between the computational parts from the coordination parts. As a result, the application's computation is mingled with coordination primitives which reduces the reusability of the components.

In this thesis, ECUMAC (Extendable composable unit model) is presented as a model to accomplish the coordination requirements of the multi-agent systems while promoting the modularity of the design of such systems thus enhancing software understandability and supporting code reusability. ECUMAC is a coordination framework providing generic coordination constructs from which developers can extend the design to implement coordination among agents. To achieve this goal, the model proposes the use of the concept of extendable and composable units.

The thesis is organized as follows. chapter 2 defines the agent coordination problem. Chapter 3 discusses the limitations of the current approaches to coordination. Chapter 4 presents ECUMAC. Chapter 5 shows how the coordinations units in the model can be composed. A design is proposed for the implementation of ECUMAC in chapter 6. In chapter 7, an example of how to model an automated e-commerce application using ECUMAC is given. Finally, chapter 8 provides the highlights of the model and draws a roadmap for future research on the topic.

2. Agent Coordination Problem

The meaning of the word coordination has been overloaded in many contexts. In Jini's framework -[25] , coordination means the ability for agents to communicate in spite of differences in terms of their interfaces. In the tuple space model, as presented in -[14] -[23] -[36] -[37] -[38] -[39] -[40] -[41] and -[42] , the focus of coordination has shifted from compatibility issue to synchronization problem. In this thesis, only the synchronization issue is addressed. This section briefly discusses the context of coordination and gives a working definition of a coordinated multi-agent system, first to limit the scope of the presentation and second to set up a vocabulary set for the later discussion.

An agent is assigned many tasks to be performed. The agent has the ability to plan the tasks to be performed. However, one agent's task is related to another agent's task. Therefore one agent's execution is constrained by another agent's execution. There are three types of relationships among the tasks:

- Dependency
- Follow up

- Mutual exclusion

The next three subsections describe these three types of relaitionships and the final subsection gives a definition of a coordinated system based on these three properties.

2.1 Dependency

Tasks can be dependent on one another. For example, in a data mining application, the data analysis step is dependent on the data collection step. Hence the requirement for dependent tasks is that the order of execution of the tasks must be maintained to respect the dependency, i.e. a task cannot be started until all its dependencies have been fulfilled. In this example, data collection must be completed before the analysis step.

A task is a unit of work performed by an agent. A task can be in one of three states: pending, executing, or completed. When a task has not yet acquired all its dependencies, it cannot proceed to the execution. If the dependent task is executing then all the dependencies have been accomplished. The completion of a task will make a transition of the state of the entire system from executing to completed, thus possibly triggering the execution of another task. In more complex task structures, a finer-grained analysis of the tasks may reveal that to start execution of a task, not all dependencies are required. The first few steps of the task can be executed with only a preliminary set of dependencies satisfied. In such a case, the task should be decomposed into subtasks which are then considered as tasks themselves and the ordering requirement can be applied to the subtasks. By exposing the subtasks, the complexity of the overall task structure increases due to the additional dependencies within the group of subtasks.

In a multi-agent system, due to the dependencies among various tasks, agent executions are themselves dependent on each other. Therefore agents need to coordinate their actions so that the relative ordering of their tasks are respected.

2.2 Follow up

Another relationship among the task is "follow up", meaning that if one task has been performed then the system must ensure that the companion tasks will also be performed. Consider the example of the data mining application. The analysis of each batch of data collected must be analyzed. Thus there is a follow up relationship between the data collection task and the data analysis task. The distinction between the dependency relationship and the follow up is that the dependency relationship describes what must have happened before the current task is executed whereas the follow up describes what must happen after the current task has executed.

There are two variants of this relationship: atomic and non-atomic. In the atomic form of the relationship, when one task is performed, then the companion task must be performed before any other tasks within the agents. In other words, at no point in time should the agent be able to see a state where task one has completed and task two is incomplete. In contrast, the non-atomic form of the relationship only ensures that the corresponding task will eventually be executed but offers no guarantees that there is no interleaving of other tasks between the completion of the first task and the completion of its companion task. The follow up property does not necessarily relates tasks from the same agents and thus

the agents involved must coordinate their actions so that they are aware of each other's task engagement.

2.3 Mutual Exclusion

Multi-agent systems are often faced with the problem of mutually exclusive tasks. For example in an automated restaurant reservation application, the agent acting on behalf of the client is assigned a task to make a reservation at one restaurant. In order to accomplish its tasks, it must negotiate with customer services agents from each restaurant to find out the best deal. Hence the client agent is now engaged into multiple negotiation tasks. However, exactly one reservation is to be made. Thus the reservation tasks at the various restaurants are mutually exclusives. This example illustrates mutual exclusion between a set of tasks within the same agent.

There are cases where the mutually exclusive tasks could be from different agents. Consider for example, a team of agents performing a collective search on the Internet. In order to avoid redundant work let us assume that each site is to be processed by exactly one agent. The searching tasks at each site are mutually exclusive yet they are to be executed by different agents. In both cases, agents in the system must coordinate their actions to ensure that no two mutually exclusive tasks are executed.

2.4 A definition for coordinated agent system

Given the description of the three forms of coordination, a definition can be given for a coordinated system. A coordinated agent system is one in which all agent tasks can be

modeled using the three forms of coordination of dependency, follow up and mutual exclusion as specified in the requirements of the application. The next chapter addresses the limitations of the current approach to agent coordination and the remaining chapters is dedicated to present ECUMAC, the proposed model, as a solution to support the modeling of the coordination requirements as well as the design of a system based on the model.

3. Existing coordination models

This section reviews three representative models of coordination from the literature and discusses their respective limitations. The three models are message passing, associative broadcasts and tuple space. The reader should note that some of these models provide mechanism to address both synchronization and compatibility issues. However our discussion will be focused on the former aspect only.

3.1 Message passing

Message passing provides the basic *send* and *receive* functions implemented with or without blocking. Coordination via message passing is achieved through causality: the event of a receipt of a message is caused by the sending of that message. Hence message pattern can be developed to ensure that the dependency among the tasks by having a blocking *receive* call to suspend the execution of a task until a particular message is received. The follow up can be accomplished by using *handshake protocols*, with or

without blocking depending on atomicity requirements. The mutual exclusion property can be ensured by a series of message exchanges among agents so that proper serialization is accomplished whenever choices arise.

The difficulty of expressing agent coordination using message passing is due to the lack of *synchrony-guarantee*. Two messages can be sent at the same time yet arrive at destinations at different times. Hence in order to ensure correctness of the system, complex protocols need to be implemented to meet the synchrony requirements. The Foundation for Intelligent Physical Agents (FIPA) [47] has developed an agent communication language (ACL) and a set of specifications for the most commonly used protocols such as voting, auction, etc. However, FIPA does not suggest means to reuse these protocols once they are implemented. In FIPA specification, the possibility of interference among the protocols are not defined and such interference are implementation dependent.

3.2 Associative broadcast

Associative broadcast, as described in [7] and -[8] , is an interesting extension of message passing where messages are not sent directly to individual agents but are broadcast to all agents and each agent must set filters to catch the right messages to be processed. Coordination is then achieved via setting up the filters. Dependency between agents can be implemented as static or dynamic creation of a filter that will catch the message from the dependencies. The follow up construct can be implemented as a pair of broadcast and receive filter. To implement a mutual exclusion relation among a group of tasks, a

protocol must be followed by the agents to determine which task will be performed exclusively and ensure other agent tasks will not overlap. Figure 1 shows a protocol to ensure mutual exclusion between agents A1, A2 and A3. At system startup, there is an agent holding the lock. Suppose the agent A1 attempts to enter a critical section and thus must broadcast a message to acquire the lock of the section and wait for a response. The agent currently holding the critical section will have set a filter to receive the lock request messages. When the lock holder exits the critical section, it will send the lock to the next agent in the requests queue. When agent A1 receives the lock, it can safely enter the critical section. The advantage of using this model over message passing is the indirectness of communication. The sender does not need to know the receiver.

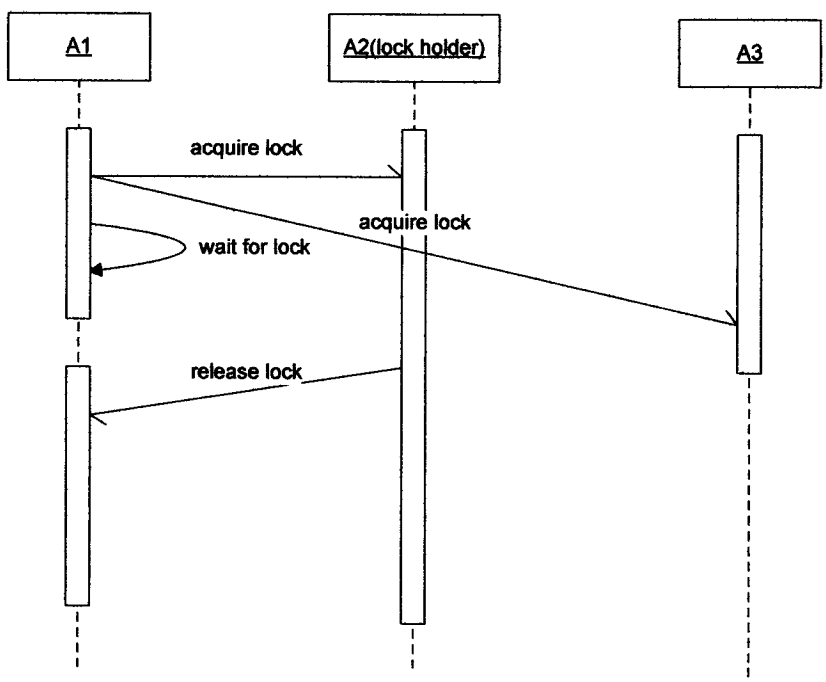


Figure 1 Mutual exclusion protocol with Associative Broadcast

Although this model relieves the sender from knowing to which agents messages are to be sent, the burden is now shifted to the receiver which must decide which messages are to be received. Design correctness depends on proper insertion of filters and could be prone to error. Since associative broadcast is an extension of message passing, it still suffers from the asynchrony in message delivery and thus results in an unpredictability of sequence of behavior. Thus, to achieve coordination, the application developer is still constrained to design complex protocols to ensure the right sequence of actions. These protocols cannot be easily reused or extended due to the potential interference with other messages in the application.

3.3 Tuple space

Message passing and associative broadcast both suffer from *high coupling* between the computation and coordination parts thus lead to complex and hard to develop reusable protocols. The tuple space model has the solution to this problem. The model is based on a shared memory model with basic access primitives *in*, *out* and *read*. The *out* primitive puts a tuple in the tuple space. The *in* primitives destructively retrieves a tuple from the shared space. The *read* primitives also retrieve data contained in a tuple but leave the tuple in the tuple space. The model ensures atomicity of the access functions. Details on tuple space based framework can be found in -[23] -[34] and -[42]

Dependency among agent tasks can be implemented by having one agent producing a tuple in the space using the "out" primitive and the other agents consuming the tuple using the "read" or "in" primitive depending on whether multiple destination of

dependency exists. Follow up can be achieved by having one agent constantly performing the "read" or "in" operation to wait for the specific tuple and the producing agent "out"ing the corresponding tuple. The mutual exclusion among tasks can be implemented as a "out"/"in" pair. When one agent produces an "out", only one agent will get the tuple and only that agent will execute its task. Hence tuples can serve as locking mechanism in the tuple space to ensure mutal exclusion.

Although this communication model is simpler, the tuple space is inherently a centralized shared memory and thus it is a bottleneck for information exchange at runtime. In addition, even though the tuple space primitives are simple, the message exchange still requires implantation of some protocols.

Considering the effort involved in designing these protocols in either of these models, it would be interesting to explore if we can create certain *patterns* or *units* and be able to reuse these patterns with only small modifications. In this thesis, ECUMAC is presented as a model to design coordination patterns in the form of *units* which can be refined and composed to accomplish application level coordination. Using ECUMAC as building block, one can design the coordination unit for a distributed application. Then, if we were to consider the distributed application consisting of two separate parts, computational part and coordination part, the designers of a distributed system will have much less work to do for the coordination part.

4. ECUMAC

The goal of ECUMAC is to decouple the computational part from the coordination parts so that both can be reused and can serve as building block for future development. In order to fulfill this goal, coordination is broken into coordination units. The coordination requirements of each of these units are accomplished by groups of software agents interacting with each other. This section presents the different types of coordination units and the role of the group formation among the coordination units.

4.1 Coordination Units

A coordination unit is a pattern that describes the coordination structure information. An example of coordination unit is a leader election. The voting activity involved requires coordination among agents. The system must ensure that the voting procedure follows the specified rules. Hence the coordination unit specifies the constraints that the agents must respect in order to reach coordination. Given a coordination unit, an application developer only has to provide the computational part to complement it.

A coordination unit can be described by what the coordination unit ensures in the functioning of the system. A set of coordination units can be seen as a guard that keeps the system within the boundary of valid states.

From the three coordination properties of dependency, follow up and mutual exclusion, two types of coordination can be identified: static coordination and dynamic coordination. Static coordination handles cases where the sequence of agent actions is decided at the design time. The existence of a coordination unit has a clear beginning and end. For example, during election of a leader, all voters must follow the rules of proposing a candidate then cast their votes which are then counted. In such a protocol, all the participants have a prescribed sequence of action to follow. This type of protocol is a special case of static coordination. Dynamic coordination handles the case where agent actions need not be executed in a predetermined sequence but some reaction has to take place when the system reaches a certain special state that we call a *triggering state*. The reactions of the agents are not mutually related. The only ordering imposed is that the reaction happens after the occurrence of the triggering event. In contrast to the static coordination, the dynamic coordination does not have a definite beginning and end. The reactions are triggered as many times as the triggering state occurs. For example, when the price of a stock increases beyond a certain threshold, it will trigger reactions in the agents that monitor the stocks and their reactions may be completely unrelated to each other. Coordination is reached by adjusting their actions according to the current state of the system to maintain a desirable state. Corresponding to the two types of coordination, there are two types of coordination units: static coordination unit and dynamic coordination unit.

4.2 Static Coordination Unit

A static coordination unit describes a coordination in which the sequence of agent's action can be statically defined. Hence, any agents participating in a static coordination must comply with the prescribed sequence of actions. A static coordination unit can be specified by the following parameters:

- Termination condition
- Agent local inputs
- Group computation

The termination condition determines what the coordination unit ensures about the system's state upon completion of the static coordination unit and during the execution of the coordination unit for dynamic coordination. The termination condition can be specified as a Boolean expression on the state of the system. The agent's local input specifies the interface for agents to provide input to the coordination and when such inputs are expected in the lifetime of the coordination unit. The group computation describes what is going to be done once knowledge has been gathered from all individual agents in the group and what is going to be produced at each round of an iterative computation. It can be noted that each local input is associated with a group computation. Consider the example of a leader election protocol. This can be modeled as a static coordination unit. The termination condition that the unit must provide is that upon completion, all agents participating in the coordination must agree as to who is the newly elected leader. The input would consist of a list of candidates and a vote for the preferred

candidate. The global computation consists of first combining the lists of candidates into one common list and second to compute the votes and returning the elected candidate who gets the highest number of votes.

4.2.1 Group Formation

The notion of a static coordination unit is based on the concept of a group. The termination condition of a coordination unit results from the accomplishment of members of a group of agents. Hence, in order to specify the behavior of a static coordination unit, a developer must specify how the group is formed in the first place. In this section, a formal description of how groups can be formed is provided for a static coordination unit. A coordination unit is characterized by the corresponding group of coordinating agents being in a certain state of coordination. Any coordination unit can be abstracted to be in one of the following states: waiting, ready to start, started, completed. A coordination unit is in the waiting state if one agent has created a group but not all required participants have joined the group yet. The ready state is a state in which all required participants have joined the group and the starting conditions have been met. In the started state, actions from all participating agents have been triggered. When the execution of the coordination unit has completed, then the state transits to *completed.*

Let A denote the set of all agents in the system and S_s be the set of all coordination unit states. A coordination unit c can now formally be defined as a set of agents A_s and a state s in S_s. Formally,

$c = (A_s, s)$, where $A_s \subseteq A$, $s \in S_s$ and S_s={waiting, ready, started, completed}.

The state of an agent system can be abstracted by the collection of the individual state of the agents. Since agents are forming groups to execute coordination units, the system can then be represented by the state of coordination untis. Let CS denotes the set of all possible static coordination units in the system and C_s denote the set of coordination units currently in the system. Then the state of a multi-agent system consisting solely of static coordination units is simply described by C_s.

The model provides the following primitives to support the group formation process:

- createCoordinationGroup(coordinationUnit)
- joinCoordinationGroup(agent, coordinationUnit)
- initiateCoordination(coordinationUnit)

The *createCoordinationGroup* primitive creates a coordination group to execute the coordination unit *coordinationUnit*. This coordination unit will have no agents and will be in the waiting state. The *joinCoordinationGroup* primitive adds *agent* to the set of participants in the coordination unit *coordinationUnit*. The *initiateCoordination* primitive sets the state of the *coordinationUnit* to ready. This will trigger the participants' reaction.

In what follows, the state machine model is used to describe the state transitions of the system using guarded command notation as described in [46] . A guarded command has the following form:

Precondition on the system's state → Resulting state

If the precondition is true, then the state of system will transit to a state where the resulting state is true. The system is modeled using these guarded commands only. At each iteration, one of the commands whose precondition is true is selected and executed

atomically meaning either the entire command is executed or the command is not executed at all.

At system startup, the set of coordination units in the system is empty as no agents has yet created a coordination group.

$C_s = \varnothing$

Upon call to createCoordinationGroup, a new group is created with an empty coordination unit. The precondition for creating a coordination group is that no other group with the same coordination unit already exists in the system.

createCoordinationGroup(coordinationUnit)

$\forall c \in C_s$: $c \neq$ *coordinationUnit* → coordinationUnit.s = waiting;

$C_s = C_s \cup \{coordinationUnit\}$

The next primitive, joinCoordinationGroup with its two parameters i.e., the agent and the coordination unit, adds the joining agent to the group. The precondition for the execution of this primitive is that the group must already exist in the current coordination unit sets C_t and the group is still in the waiting state.

joinCoordinationGroup(*agent*, *coordinationUnit*)

$\exists c \in C_s$ c=*coordinationUnit* and c.s = waiting → $c.A_c = c.A_c \cup \{agent\}$

The third primitive, initiateCoordination, changes the state of the coordination from waiting to ready. The precondition is that the coordination unit must exist and the unit must be in the waiting state.

$\exists c \in C_s$ c = *coordinationUnit* and c.s = waiting → c.s = ready

It can be noted that when the initiateCoordination primitive is executed, the group state changed from the *waiting* state to the *ready* state rather than the *started* state. The reason is that the group is made up of multiple agents and thus not all agents will receive the initiation signal at the same time. Therefore there is a delay between the execution of the initiateCoordination primitive and the actual start of the execution of the coordination unit. Hence the transition from *ready* to *started* takes place when all participants have acknowledged receipt of the initiation signal.

4.3 Dynamic Coordination Unit

A dynamic coordination unit responds to unscheduled occurrence of system state. This may arise in an open system or due to runtime race. Such response need not follow any particular sequence of agent actions but must take place after the system has reached a particular state. Hence the dynamic coordination unit can be described by the following parameters:

- System state constraint
- Triggering condition
- Response

The triggering condition is a Boolean expression to be evaluated based on the global state of the system. When the predicate becomes true, it will trigger the response from the monitoring agent. The response is the reaction of the agents in the system to the occurrence of the predicate. Consider again the example of the buyer agent in a synthetic market. The buyer agent needs to monitor the prices of the products which are controlled

by seller agents. In this context, the triggering predicate would be when any of the prices posted by the sellers is below the client's set threshold price. The response of the buyer agent would be to conduct the purchase of the product on behalf of the client.

Unlike the static coordination unit which has a definite beginning and end, the dynamic coordination unit is a long term relation between agents. The association of the triggering predicate with the corresponding response will remain until an agent explicitly cancels the relation. The system state constraint describes the type of relationship that the coordination unit will enforce. The dimensions describing the relationship are multiplicity and synchrony. The multiplicity is the number of times the response will be triggered and the synchrony states whether the response is executed synchronously with the occurrence of the predicate or asynchronously. With the synchronous execution, the response is triggered and completed before any action that could cause changes to the triggering condition. For instance, when a bank system that a user account is accessed simultaneously via two different ATM machines, it should immediately halt one of the transactions before the other transaction completes. On the other hand, the asynchronous reaction does not provide guarantee that at the time the response is executed the triggering predicate is still true. For example, in a synthetic robot soccer game, when a player detects that its team has lost control of the ball, that player needs to switch to defensive mode. However, in the mean time, other players are not suspended from taking action to regain possession of ball until that player has completed its plan update.

As mentioned earlier, the concept of coordination unit allows the separation of concern between the computational part from the coordination part. With the two types of

coordination units, an application developer can customize the behavior of each coordination units and compose them in various ways to define a coordination structure that fits the application's requirements.

4.3.1 Predicate group formation

Similar to the case of static coordination, a dynamic coordination unit is also based on the concept of a group. The dynamic coordination unit relates the event triggered by the change of state of a group of agent to a corresponding agent's action. There are two roles to be played by different agents in a dynamic coordination unit. The first role is that of a monitored agent. A monitored agent allows its state to be monitored by other agents. The collection of concurrent agent's states form a global state of the system as described in [44] . The second role is that of a monitoring agent whose responsibility is to observe a subset of the global state variables and react correspondingly to the occurrence of certain conditions of interest. These conditions are called predicate. In this section, an approach for the formation of monitored agents group and monitoring agents group will be presented.

Given the dual role of the relationship, the dynamic coordination unit can result in one of the following states:

- Created
- Reporting
- Responding

In the *created* state, the group of monitored agents has been created but no monitor has yet registered in observing the group. When a monitoring agent registers with the group,

the coordination unit state transits to reporting. In the *reporting* state, the monitored agents within the group starts reporting their state to the monitor. Finally when the predicate has occurred then the coordination unit enters the *responding* state. It should be noted however that the responding state overlaps with the reporting state because while the monitoring agents are responding, the monitored state continues to report. Thus, it is possible that the predicate occurs more than once. Each occurrence of the predicate will fire a distinct instance of the response which will be queued and executed sequentially. A predicate is said to have occurred when the system transits from a state where the predicate is false to a state where the predicate is true.

A dynamic coordination unit can be characterized by a set of monitored agents, a set of monitoring agent and a dynamic coordination unit state. Let A denotes the set of all agents, and S_d denotes the set possible states of a dynamic coordination unit. Then a dynamic coordination unit c can formally be defined as follows:

$c = (A_{md}, A_{mg}, s)$

where $A_{md} \subseteq A$, $A_{mg} \subseteq A$, $s \in S_d$ and S_d = {Created, Reporting, Responding}

(A_{md} is the set of monitored agents and A_{mg} is the set of monitoring agent)

Recall from the description of the group formation of the static coordination unit, a multi-agent system consisting only of static coordination unit is defined by the set C_s that contains all the static coordination units. Similarly the state of a system consisting of only dynamic coordination units can be described by a set of dynamic coordination coordination units C_d. Let CD denote the set of all possible dynamic coordination units then

$C_d \subseteq CD$

Our model provides the following primitives to support the group formation process:

- createCoordinationGroup(coordinationUnit)
- joinCoordinationGroupAsMonitoredAgent(agent, coordinationUnit)
- joinCoordinationGroupAsMonitor(agent, coordinationUnit)

The *createCoordinationGroup* primitive creates a dynamic coordination unit. The *joinCoordinationGroupAsMonitoredAgent* primitive allows an agent to join a created dynamic coordination unit by playing the role of a monitored agent, meaning that it will allow its state for monitoring. The *joinCoordinationGroupAsMonitor* allows an agent to join a created dynamic coordination unit by playing the role of a monitoring agent. The monitoring agent will be responsible for responding to occurrences of the triggering condition.

At system startup, the set of coordination units in the system is empty as no agents has yet created a coordination group:

$C_d = \varnothing$

Upon call to *createCoordinationGroup*, a new *coordinationUnit* is created and added to C_d. The original state of the newly created coordination unit is set to *created* and the two agent sets A_{md} and A_{mg} are both empty. The precondition of this primitive is that no such coordination unit already exists in C_d.

createCoordinationGroup(*coordinationUnit*)

$\forall c \in C_d$: $c \neq$ coordinationUnit → *coordinationUnit*.$A_{md} = \varnothing$;

coordinationUnit.$A_{mg} = \varnothing$;

coordinationUnit.s = *created*;

$C_d \cup \{$*coordinationUnit*$\}$

The next primitive, *joinCoordinationGroupAsMonitoredAgent,* with its two parameters the *agent* and the *coordinationUnit*, adds the joining agent to the group in the monitored agent set (A_{md}). The precondition for the execution of this primitive is that the group must already exist in C_d.

joinCoordinationGroupAsMonitor(*agent*, *coordinationUnit*)

$\exists c \in C_d$ c = *coordinationUnit* → $c.A_{md} = c.A_{md} \cup \{agent\}$;

The third primitive, joinCoordinationGroupAsMonitor with parameters the *agent* and the *coordinationUnit*, adds the joining agent to the group in the monitoring agent set (A_{mg}). If the joining agent is the first monitor to join then the group state will transit from *created but not yet observed* to *reporting*. The precondition for the execution of this primitive is that the coordination unit must already exist in C_d.

joinCoordinationGroupAsMonitor(*agent*, *coordinationUnit*)

$\exists c \in C_d$ c = *coordinationUnit* → if($c.A_{mg} = \varnothing$) then c.s = *reporting*;

$c.A_{mg} = c.A_{mg} \cup \{agent\}$;

Since a multi-agent system (MAS) consists of a collection of several groups executing static coordination units and a collection of groups executing dynamic coordination units, it can be expressed formally as follows:

$MAS = (C_s, C_d)$

where C_s represents the list of static coordination group and C_d represents the list of dynamic coordination group.

5. Composing a system with coordination units

In the previous chapter, a description of the two types of coordination was provided with an indication of how each one can be used to fulfill the coordination requirements in terms of the three properties of dependencies, follow up, and mutual exclusion. Since the goal of these coordination units is to promote reuse of the coordination code, these units must have a model for composing them in order to generate new system properties. In this chapter, the reusability and extensibility issues of the coordination units is discussed. One of the challenges posed by a coordinated system is due to the extensiveness of communication among the agents in order to reach coordination thus yielding a very high coupling among the agents in the system. Thus, a simple modification in the code might cause a ripple effect on the entire system and lead to mysterious errors which are very difficult to debug or even to detect. In order to overcome this challenge, the coordination units provide a way to localized the effect of the communications while providing a mechanism to retain the necessary coupling among the agents.

This chapter is divided into two sections. Section 5.1 discusses how the coordination units are executed and how a developer can define the behavior of a coordination unit. The discussion gives details for both the static and dynamic version of the coordination units. Section 5.2 will turn the focus to the interaction and interference among the coordination units.

5.1 Behavioral definition of a coordination unit

As mentioned earlier, a coordination unit is a skeleton containing coordination pattern information where application developer can fill in the computational part to accomplish the required task. Details on how to define these coordination units is specific to the type of coordination units. The following sub-sections give a treatment for each of the coordination unit.

5.1.1 Static coordination unit

In many coordination problems involving predetermined sequence of action, two perspectives are observed: the perspective of the agents as individuals and the perspective of all agents as a group. In each perspective, there is a recurring sequence of actions that can be observed: gathering information, processing the information, and publishing the information. Hence defining the static coordination unit consists of defining the interface between the agents and the group. This section presents a model to define the static coordination unit and provides an analysis of the application of the model to define

coordination units that solves common coordination problems such as consensus, voting, etc.

5.1.1.1 Static coordination model

From the observation made earlier, the coordination unit can be modeled as a sequence of information gathering, computation, and state update. Figure 2 illustrates the sequence of actions from a group perspective. The group first gathers information from the agents, then performs the computation that needed data from all agents in the group and finally broadcast the result to the group.

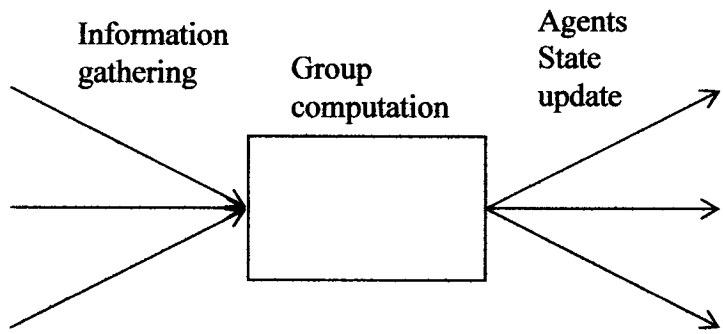


Figure 2 Sequence of actions from group perspective

Figure 3 illustrates the sequence of actions from an individual agent's perspective. The individual agent first retrieves information from the group, then performs computation that needed synchronized data from the group and finally submits the next batch of data to the group for the next round, if any.

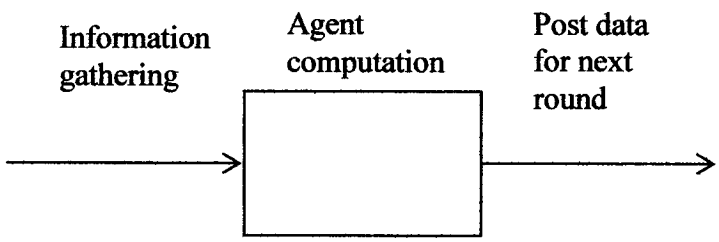


Figure 3 Sequence of actions from a individual agent's perspective

A general static coordination unit looks like Figure 4. Individual agents gather information then perform local computations, then post data to the group from which the group carry out the group computation and finally update the agents about the collective result and the cycle repeats again if necessary.

In order to define a coordination unit, a developer needs to specify the data interface so that the data produced by the agent computation matches the data gathered by the group and vice versa.

In the following section, examples of coordination problems illustrate how a developer can map the coordination requirements to this model of a coordination unit.

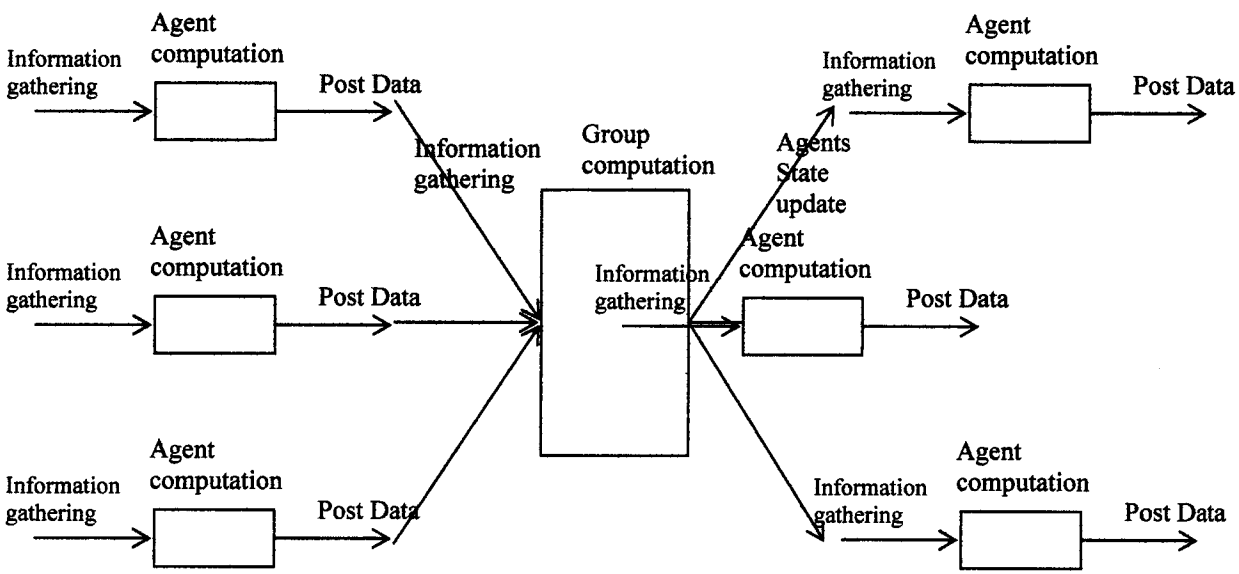


Figure 4 Generic coordination unit

5.1.1.2 Coordination pattern mapping to static coordination unit

In order to overcome the problem of agent coordination, patterns of recurring coordination structure have been developed to solve specific problem. Among these patterns, two are most commonly used: consensus and voting. Below, we demonstrate how an application developer can map these patterns to a coordination unit.

5.1.1.2.1 Consensus

Very often, agents are faced with the problem of collectively picking a value for a shared state variable. The difficulty is how to make all agents in the group agree on the same value chosen and doing so distributively. One solution of achieving this is to have one agent dedicated for making the final decision. The problem with such a solution is that if the decision maker agent fails an election protocol would be required to replace the lost agent. Another problem is the asymmetry imposed by the solution, one agent has to be the decision maker but no criteria are relevant to select such an agent.

The coordination unit model has the solution to this problem. Figure 5 shows a mapping to accomplish the consensus requirement. The agents that want to reach a collective agreement on the value to be selected join a group that executes the consensus coordination unit. Once all agents have joined the group, determined by the starting condition of the coordination unit, execution starts and each agent performs their local computation and post a value to the group. The group then gathers the data and performs a selection, in this example a random selection was used for illustration purposes. Further refinement can be provided to suit the application. When the group selection is made, all

agents in the group get updated about the selected value. The individual agents then receive collectively the selected value. This coordination unit ensures that at the end of its execution, all agents will have agreed on the value selected.

Although the decision making process is centralized, the behavior of the agents remains distributed because the central synchronization is only localized to a group. The rest of the system still executes concurrently. In addition, the group promotes the symmetry in terms of the roles of the agents. No agent needs to handle the central synchronization as it is done implicitly by the group coordinator which is an external entity.

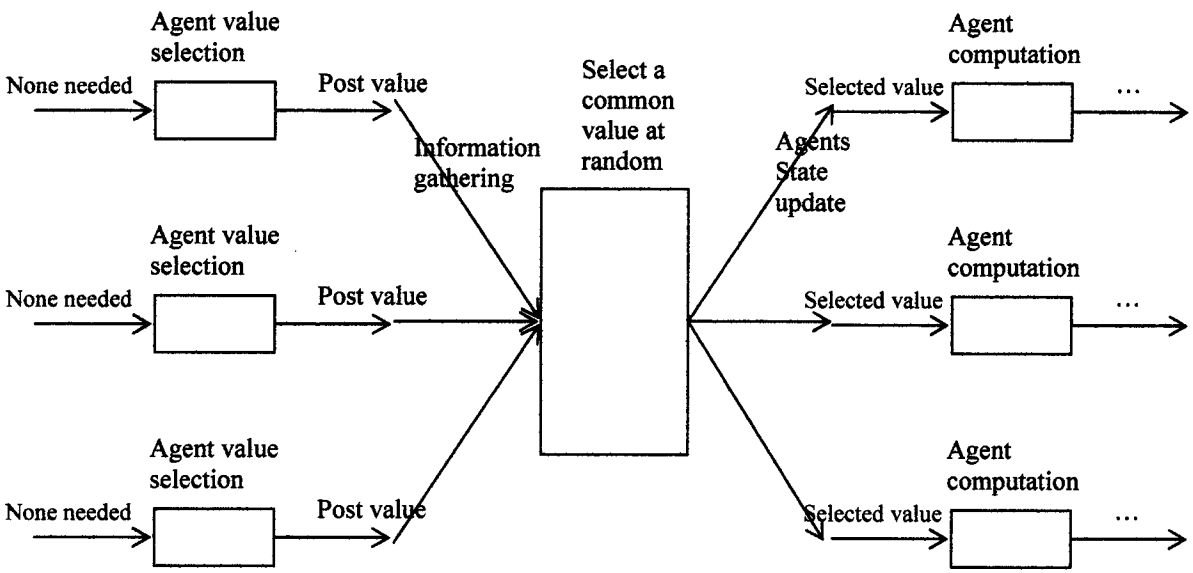


Figure 5 Consensus mapping to coordination unit

5.1.1.2.2 Voting

Another common pattern of coordination in agent-systems is voting. Agents are confronted with problems like electing a leader, selecting an alternative that will maximize the group's utility function. Hence each agents must have an input to say how much they value each leader or what their utility function is with respect to a set of alternatives to be chosen. Thus in addition to the agreement on the final selection, there is a maximizing function that needs to be computed.

Figure 6 illustrates a coordination unit handling a voting protocol using combined list computation and the maximum number of vote as group computations. In the first phase, the agents post a list of candidates. The group then combines all the list of candidates and builds a complete list of candidates based on all the proposals. The combined list is then fed back to the agents who will then compute a vote to express their utility function on the set of candidates. They cast a vote. Then the group computes the candidate with the maximum number of votes and return it to the agents. By the end of the execution, the voting coordination unit ensures that all agents agree on the same candidate being elected and the candidate is representative of all the individual agent's utility function.

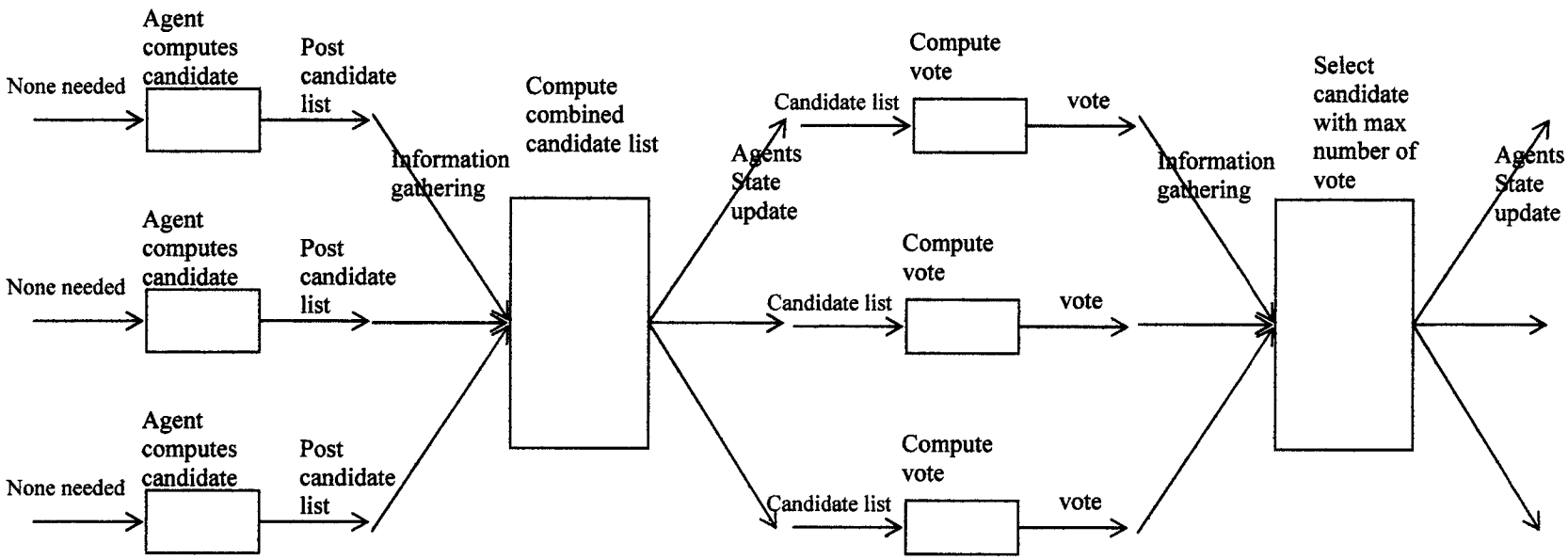


Figure 6 Voting mapping to coordination unit

From the two examples presented, it can be observed that the patterns of message exchange are similar yet depending on the coordination requirements, the basic patterns can be repeated. The consensus protocol was modeled using one phase of message exchange where the agents are involved only once. Whereas the voting protocol involves two phases of the information exchange where the group computation produces intermediary which is fed back to the agents before they produce the next round of input.

5.1.2 Dynamic coordination unit

The pattern of execution of a dynamic coordination unit is unique: when the triggering predicate of the unit occurs, the response must take place. However, the predicate being monitored and the type of response may vary. In this section, we present a description of the structure of the dynamic coordination unit and an example of a mapping.

5.1.2.1 Dynamic coordination unit model

As mentioned earlier, a coordination unit is accomplished by two roles: the monitored agents and the monitoring agents. A monitored agent is responsible for reporting any changes in its own state and a monitoring agent is responsible for reacting when the monitored predicate has occurred. Hence, in this process there is an intermediary component called the predicate detector which triggers an event when the predicate occurs. Figure 7 illustrates the general dynamic coordination unit model. Unlike in the case of a static coordination unit, not all monitored agents are required to report state changes at the same time. That is the predicate evaluator does not wait for all reports to

start processing but process them as each of them arrives. Hence in this figure, the solid lines represent actual communication in the current predicate evaluation and the dashed lines represent communication in the former or potential predicate evaluation. In this example, monitored agent 1 reports its relevant state change. The predicate evaluator evaluates the predicate and if the predicate is found to be true then the evaluator notifies the monitoring agents registered with this predicate so that they can take corresponding action.

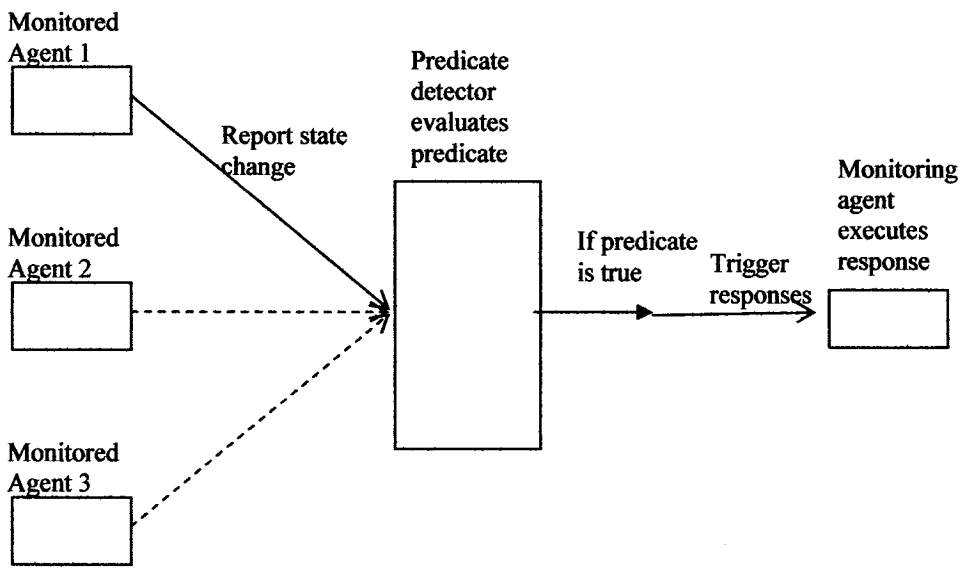


Figure 7 Dynamic coordination model

In order to define a dynamic coordination unit, the developer needs to define what state from the monitored agent to be reported and when to report, how to evaluate the predicate and what response is to be taken when the predicate has occurred. Once these entities are

specified, the relationship will hold until all registered monitoring agents have cancelled their registration.

5.1.2.2 Coordination pattern mapping to dynamic coordination unit

A coordination unit establishes a relationship among agents in the system. In current systems, many such relationships exist. One such relationship is the *race condition prevention*. For example, a distributed database must prevent two agents from accessing the same data simultaneously where one of them is a write, since it might cause inconsistency in the data. This example can be generalized as mutual exclusion. Another type of relationship is called *safe progression*. This relationship ensures that the safe condition happens before taking an action. This relationship is commonly used in military strategies to safely occupy territory. In what follows, a mapping of the mutual exclusion and safe progression relationship are presented.

5.1.2.2.1 Mutual exclusion

In an agent system, agents are not always in communication with each other. There are activities which cannot be executed concurrently. Such activities are mutually exclusives. Hence without proper coordination, an agent might not be aware that another agent is currently executing the mutually exclusive activity and consequently start the activity itself and causes corruption in the system. The property of mutual exclusion is to prevent such consequences from happening. In this section, a formalization of the mutual exclusion requirement will be given and an illustration of how the mapping to a dynamic coordination unit can solve the problem.

Let us define the following notations to be used in the specification of the requirements for the mutual exclusion property.

A: a finite set of agents on which the mutual exclusion property is to be enforced

ai: is an agent with identity i

ai.isInState(exclusiveTask) denotes a state of ai where ai is executing *exclusiveTask*

The mutual exclusion property can be formally defined as follows:

$\forall ai \forall aj \in A: ai \neq aj \rightarrow \neg$ (ai.isInState(*exclusiveTask*) $\wedge$ ai.isInState(*exclusiveTask*))

Hence in order to be notified for a mutual exclusion violation, the above predicate needs to be negated to yield:

$\exists ai \exists aj \in A: ai \neq aj \wedge$ ai.isInState(*exclusiveTask*) $\wedge$ ai.isInState(*exclusiveTask*)

Hence the mutual exclusion is simply a disjunction of conjunctions of local agent state on a finite set. Consider the case where the group has 3 agents. The predicate would be:

(a1.isInState(*exclusiveTask*) $\wedge$ a2.isInState(*exclusiveTask*)) $\vee$

(a1.isInState(*exclusiveTask*) $\wedge$ a3.isInState(*exclusiveTask*)) $\vee$

(a1.isInState(*exclusiveTask*) $\wedge$ a3.isInState(*exclusiveTask*))

The mapping to a dynamic coordination unit can be done as shown in Figure 8.

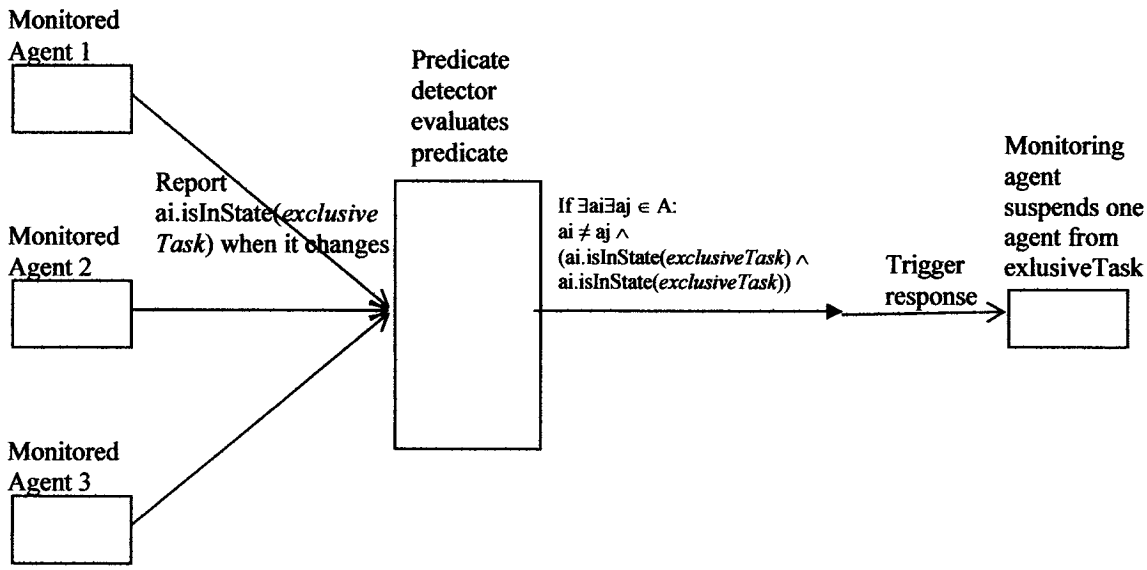


Figure 8 Mutual exclusion mapping to dynamic coordination unit

Each agent must report to the predicate evaluator whenever the state ai.isInState(exclusiveTask) changes to become true. The predicate evaluator then evaluates the truth value of the global predicate ∃ai∃aj ∈ A: ai ≠ aj ∧ (ai.isInState(*exclusiveTask*) ∧ ai.isInState(*exclusiveTask*)). If the predicate is true then it will trigger the response from the monitor. Once triggered, the monitor then suspends execution of one of the agent to save the mutual exclusion property. It should be noted that the response is executed synchronously with the triggering event. This means that the cancellation of the second access will occur before the its transaction is completed, hence the transaction will never be committed.

5.1.2.2.2 Safe progession

Safe progression, as the name implies, is based on the concepts of safety and progression. Agents in a safe progession are to achieve a goal yet they must bound to safety criteria. Consider the military scenario depicted in Figure 9. The goal of the friend troops is to move toward the target by passing in between the enemy bases. Yet, the safety criteria requires that the troop does not move forward until the enemy bases are both under the allies control. The scenario shows two scout planes S1 and S2 that will be reporting to the friend troops the states of the enemy bases.

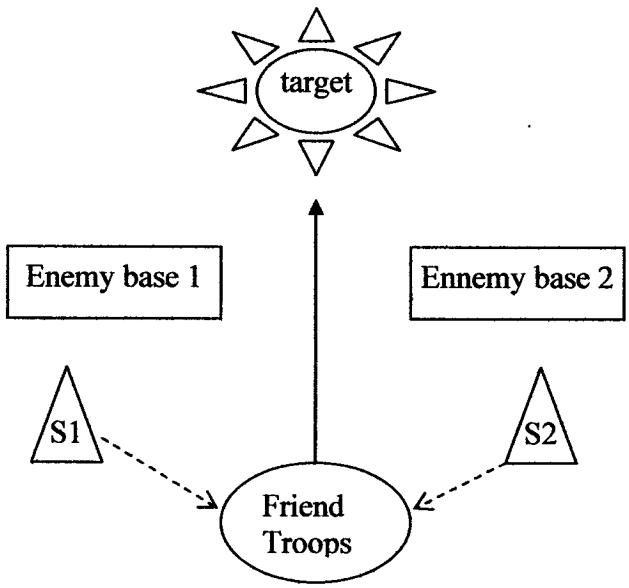


Figure 9 Military Safe Progression

Using the dynamic coordination unit model, we can model this relationship. The safety criteria can be broken up into small portions to be monitored by individual agents independently. These agents will report the change of their local state to the monitoring agent. When the predicates are concurrently true then the monitoring agent takes action to progess toward the goal.

In the particular example of the military scenario, the predicate to be monitored is:

EnemyBase1.state = alliedControlled and EnemyBase2.state = alliedControlled.

The response is to advance toward the target. Figure 10 shows the mapping of this progression to the dynamic coordination unit. It can be noted that unlike the mutual exclusion requirement, the response in a safe progession need not be synchronous with the triggering event.

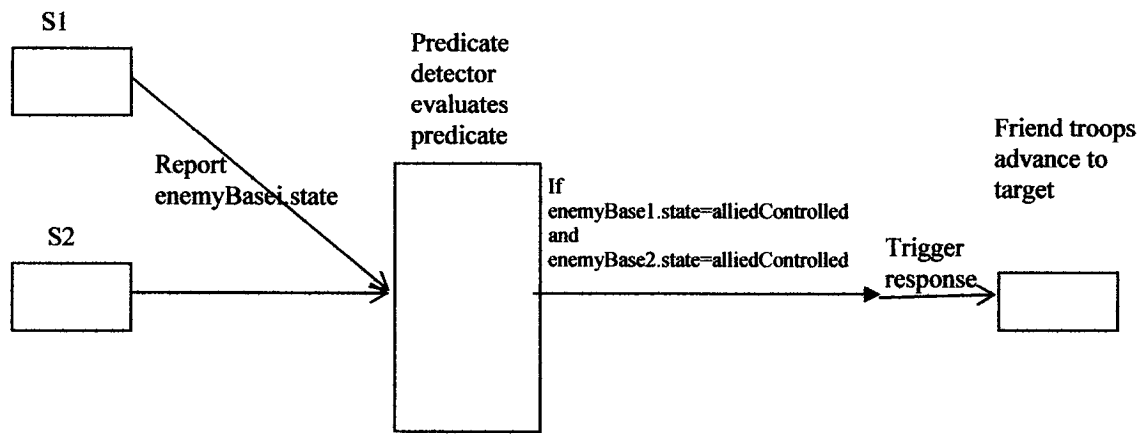


Figure 10 Safe progession mapping to dynamic coordination unit

5.2 Composition of coordination units

In section 5.1 we described how individual coordination units can be instantiated. However, an agent system requires multiple coordination units to work together to fulfill all the coordination requirements of the underlying application domain. The coordination units can be composed in 3 ways:

- sequentially
- in parallel

- by containment

Section 5.2 discusses each of these three types of composition.

5.2.1 Sequential composition

The most basic type of composition of the coordination units is *sequentially*. This results in ordering one construct after another with respect to one agent. Let us first consider the case of a static coordination. As described in section 4.2 the static coordination unit can be in one of the four states: waiting, ready, started, and completed. The static coordination unit is divided into two phases: group preparation and execution. The waiting and ready states of the coordination unit correspond to the group preparation phase. The started and completed states correspond to execution phase. The execution of a static coordination unit is performed as if it is atomic. There is no interference with other coordination units. Hence, when the same agent is involved in two static coordination units, the execution of the two units is serialized by sequencing one after the other. Figure 11 illustrates a sequence of static coordination unit. The agent is involved in two static coordination units SCU1 and SCU2. The sequencing has restricted the execution phase of SCU2 to be followed after the execution phase of SCU1. If multiple agents participate in at least two common coordination units then conflict might arise due to potential *cyclic dependency*. Hence the requirement to the application developer is to make sure that if a sequence of coordination unit is to be enforced then there must not be a cyclic dependency.

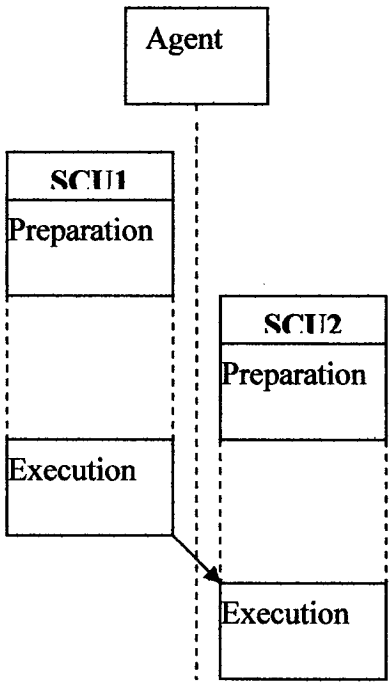


Figure 11 Sequential static coordination unit

Consider now the case of the dynamic coordination units. The dynamic coordination unit can also be divided into the same two phases: preparation and execution. The preparation phase consists of *created* state, whereas the execution phase consists of *reporting* and *responding* states. Since the execution of a dynamic coordination unit depends on unpredictable occurrence of predicates, it is not possible to statically order two dynamic coordination units. However, it is possible to partially order a dynamic coordination unit with respect to a static coordination unit. Figure 12 shows an ordering of a dynamic coordination unit DCU2 after the execution of a static coordination unit SCU1 hence the execution of DCU2 will happen after SCU1. However, there is no guarantee that when DCU2 starts execution, the state will be as if SCU1 has just completed because of the potential delay between monitoring phase and the execution phase. This delay is attributed to the message transit time.

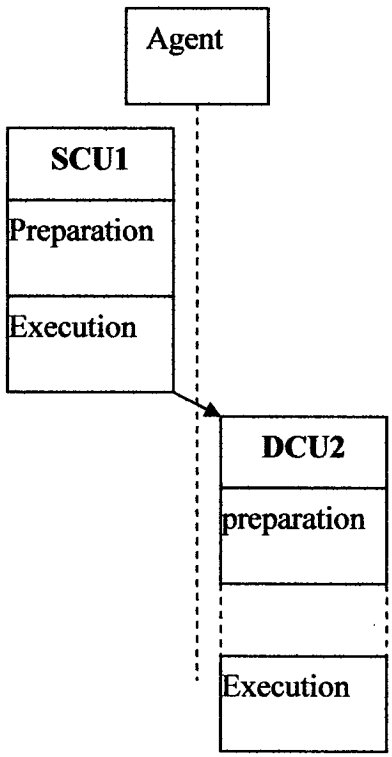


Figure 12 Sequencing a dynamic coordination unit with respect to a static coordination unit

5.2.2 Parallel composition

So far, the description of the coordination units have ensured *consistency* among agents, in the sense that agents in the system satisfy the properties of dependencies, follow up, and mutual exclusion. However, in most cases, an agent is called upon to play multiple roles or to service multiple requests concurrently, each of which requires coordination with other agents. Such agent may be involved in more than one coordination unit. Composition in parallel allows the agent to participate in multiple coordination units simultaneously. A composition in parallel consists of preparing the coordination units involved without specifying the dependencies among them. The result would be to let the runtime system serialize the executions of the coordination

units. Figure 13 shows two possible serializations of the parallel execution of two static coordination units SCU1 and SCU2.

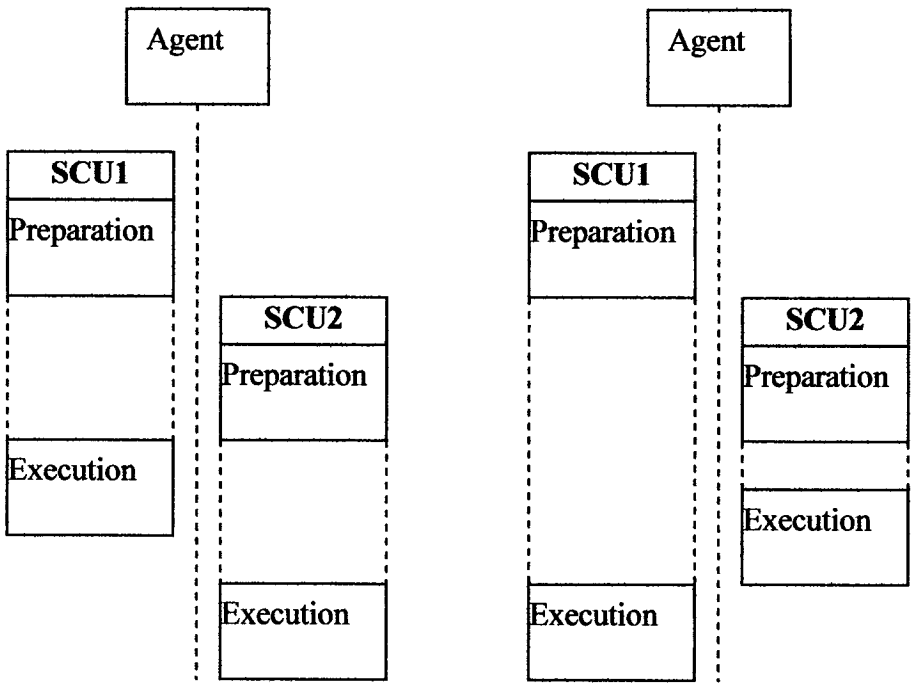


Figure 13 Composition in parallel of two static coordination unit

Figure 14 illustrates a possible serialization of a dynamic coordination unit DCU2 and a static coordination unit SCU1 composed in parallel. In both cases, the serialization ensures that the executions of the coordination unit are performed atomically or as if atomic.

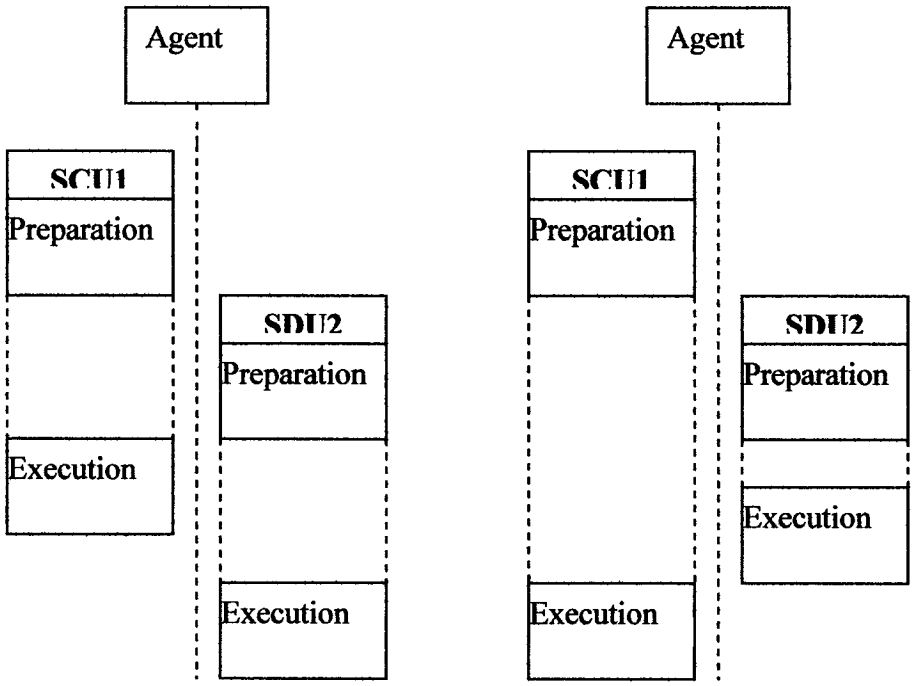


Figure 14 Parallel composition of a static and a dynamic coordination unit

5.2.3 Composition by containment

The third type of composition is containment. This concept stems from the field of functional programming: A coordination unit may call for help from other coordination units to fulfill its task. Hence a coordination unit is started within another. There are 4 possible cases:

- case 1: a dynamic coordination unit starting another dynamic coordination unit
- case 2: a dynamic coordination unit starting a static coordination unit
- case 3: a static coordination unit starting a dynamic coordination unit
- case 4: a static coordination unit starting a static coordination unit

Case 1 is illustrated by Figure 15. During the execution phase of the dynamic coordination unit DCU1, it creates an internal dynamic coordination unit DCU2. It can be

noted that the execution of DCU2 can be within the execution of DCU1 or occur after the execution of DCU1 depending on the occurrence of the associated triggering event. Only the preparation phase of DCU2 is guaranteed to be completed inside the execution of DCU1.

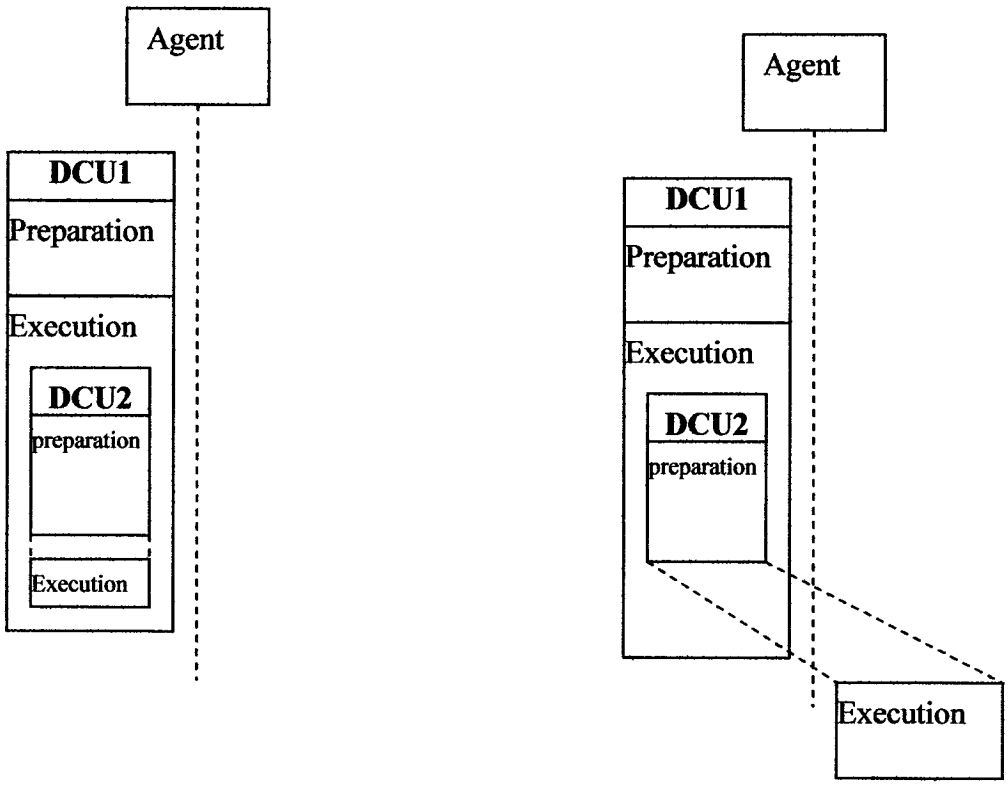


Figure 15 Containment of a dynamic coordination unit by a dynamic coordination unit

Figure 16 shows a dynamic coordination unit DCU1 containing a static coordination unit SCU2. SCU2 is initiated during the execution of DCU1 and must complete its execution with the execution phase of DCU1.

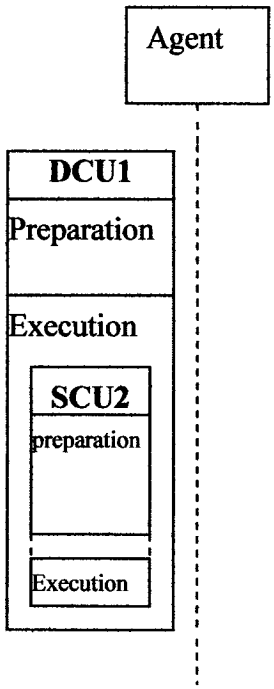


Figure 16 Containment of a static coordination unit by a dynamic coordination unit

Figure 17 illustrates a static coordination unit SCU1 containing a dynamic coordination unit DCU2. DCU2 is initiated during the execution of SCU1. DCU2 may or may not complete its execution within the execution phase of SCU1.

Figure 18 illustrates a static coordination unit SCU1 containing a static coordination unit SCU2. SCU2 is initiated during the execution of SCU1. SCU2 must complete its execution within the execution phase of SCU1.

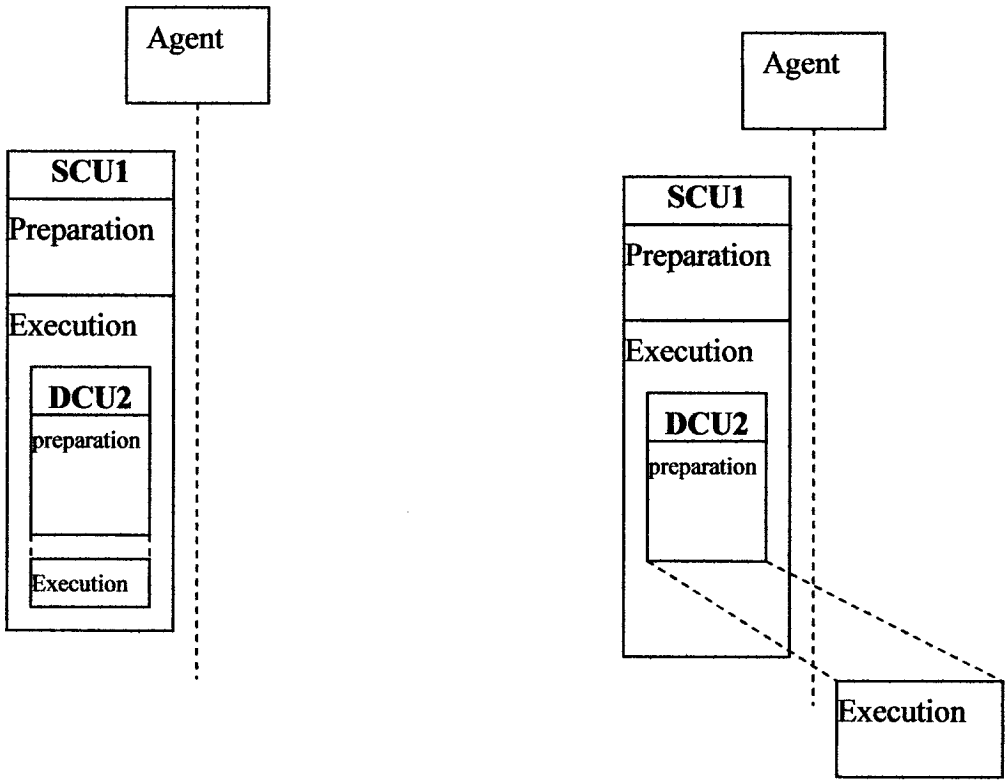


Figure 17 Containment of a dynamic coordination unit by a static coordination unit

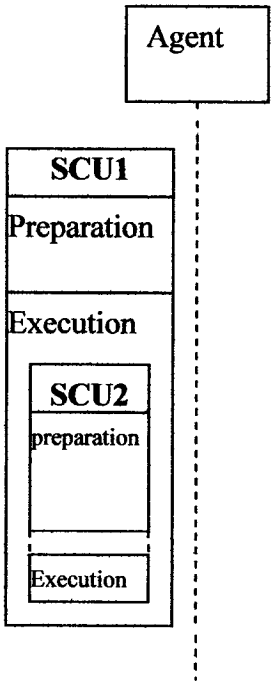


Figure 18 Containment of a static coordination unit by a static coordination unit

Composition by containment is useful for on demand monitoring and on demand coordination. Consider a hierarchy of agents performing election. A set of agents is divided into small groups of agents. Each group elects a leader to represent them at the higher level of the hierarchy. Hence, when each leader is to cast its vote, it will first consult its group members. Containment of a static coordination unit by a static coordination unit pattern can be applied to resolve this problem. This problem would not have been possible to resolve with any other composition techniques.

6. ECUMAC Design

The previous chapters have laid out the various constructs of the Extendable and Composable Unit for Multi-agents Coordination (ECUMAC) model. This chapter proposes a design to implement the model. The design is based on the JADE [45] agent platform and the Java programming language. JADE is based on the behavioral model of an agent and the underlying method of communication is message passing. This chapter is divided into the following sections: use cases, class diagram design, and scenarios description.

6.1 Use cases

6.1.1 Static coordination

This section describes the use cases pertaining to the usage of the static coordination units. The discussion starts with an introduction to the roles of the various agents within a

static coordination unit and proceeds with describing the primitives that a developer can use to define a coordinated agent system.

6.1.1.1 Roles of agents in static coordination

The model does not mention any roles in the static coordination unit. However, in order to make the unit fit with the rest of the system, we identify three roles that an agent may have:

Initiator

Participant

Observer

The initiator is the one that senses the need to coordinate through its own local state and initiates a coordination protocol. For each coordination protocol, there is only one initiator. A participant is the one that responds to the coordination call and provides input to the group actions. An observer is only interested in the result of the coordination and does not have any data to feed in the coordination. However some of its future states would be dependent on the result of the coordination. Note that an agent can play more than one of these roles. For instance an agent can be the initiator, a participant, and an observer of the coordination. It can also be noted also that the role only exists as long as the coordination is in progress.

6.1.1.2 List of Primitives

The following is a list of primitives that the application developer can use to create and manipulate a static coordination unit:

1- CoordinationObserver joinCoordinationGroup(GroupName, CoordinationResponse)

This function allows an agent to join a coordination group. If the requested group does not exist then a new group is created with the requested group name. The coordination response represents the computational part of the agent during the execution of the coordination unit. The primitive returnins a *CoordinationObserve* which contains a coordination ID to allow the calling agent to follow up on the status of the coordination unit. The agent can join the group as an observer or as a participant. By joining the group with a null *CoordinationResponse*, the agent joins the group as a mere observer, meaning that the agent can only wait and get the result of the coordination but cannot provide input to the coordination unit. It can be noted that the addition of the notion of an observer member does affect the atomicity assumption of the model as the result is only made available to the observers when all the coordination unit execution has completed, i.e. no partial result is available. If the agent joins with an instantiated *CoordinationResponse,* then the agent joins as a participant.

2- CoordinationObserver initiateCoordination(GroupName, startingCondition, GroupAction)

This function notifies all the group members that the coordination has been instantiated. The rationale for this primitive is that when agents join a group, they are only registered with a lookup service that it is in the group. The coordination unit is not yet allocated resources to start execution. Only after the initiator, who detains the definition of the *groupAction* (or group computation phase), has called the initiateCoordination primitive that the system knows what group action to fill in for the coordination unit and only then the system allocates resources to the coordination unit. At this point, the state of the

coordination unit has transited from waiting to initiated, meaning that the group action of the coordination unit has been installed. What is left is to wait for sufficient agents to join the group or for certain condition to be met for the coordination to start. Thus the starting condition argument of this primitive is evaluated to tell when the coordination unit is ready to start. When the starting condition has been met then the coordination unit is closed to any further registration and a start message to sent to all participants.

3- cancelCoordination(CoordinationID)

This primitive abort all operation done so far on the coordination unit. The agents restore their original state as they were at start of the coordination.

4- waitForCoordinationResult(CoordinationObserver)

This primitive allows the agent to block waiting for the static coordination unit to complete before resuming the current work. This primitive is used to order the execution of the various coordination unit as described in the sequential composition section.

5- pollForCoordinationResult(CoordinationObserver)

This primitive allows the agent to check up on the result of the coordination unit. If the coordination unit has not completed then the primitive does not block and returns null.

6.1.2 Dynamic Coordination

This section describes the use cases pertaining to dynamic coordination units. The discussion starts with the introduction to the roles of the various agents within a dynamic

coordination unit and follows with a description of the different primitives that a developer can use to define a coordinated agent system.

6.1.2.1 Role of agents in dynamic coordination

There are two roles in a dynamic coordination units:

monitored agent

monitoring agent

The monitored agent exposes its state so that the monitors can observe and react to occurrences of certain joined states. In this class of coordination, there is no clear event indicating the end of a role. The roles are maintained to ensure the continuous coordination of all the agents.

6.1.2.2 Primitives list

The following is a list of primitives that the user can use to create and manipulate a dynamic coordination unit:

1- setPredicateTrigger(GroupName, PredicateEventListener, is_sync_flag)

This primitives allows the monitoring agent to register a response to the occurrence of a predicate. The registration will be made with the monitored agents associated with the *GroupName*. The *PredicateEventListener* contains the predicate detections mechanism as well as the reaction of the monitor. The *is_sync_flag* is used to determine whether the reaction is to be executed synchronously with the occurrence of the predicate. If the flag is set to true, then the agent whose report has triggered the reaction will be not allowed to proceed until the reaction has completed.

2- setupMonitoredPredicate(GroupName, MonitoringObject, MonitoredObject[ ])

This primitive is to be used by the monitored agent to specify which part of its state to be monitored. The result is that the set of local variables is registered with the lookup service as being monitored under the *GroupName*. The *MonitoringObject* is responsible for monitoring the local *MonitoredObject* and reporting when necessary.

6.2 Class diagram Design

This section provides a class view of the design. A description of each of the main classes is given along with a description of the interrelations among them. The section is divided into two parts: static coordination and dynamic coordination.

6.2.1 Static Coordination

Figure 19 illustrates a class diagram for the implementation of the static coordination. There are three main components to manage the coordination units within the agent and in the whole system: (a) C*oordinatedAgent*, (b) C*oordinationManager* and (c) *LookupService*. The *CoordinatedAgent* is a placeholder for application agents to extend. It encapsulates primitives for specifying coordination units and implements the managements of the coordination units. The *CoordinationManager* is the component within the *CoordinatedAgent* that manages the message passing related to the

coordination units. The *LookupService* provides a mean for agent to explore existence of other agent in the system.

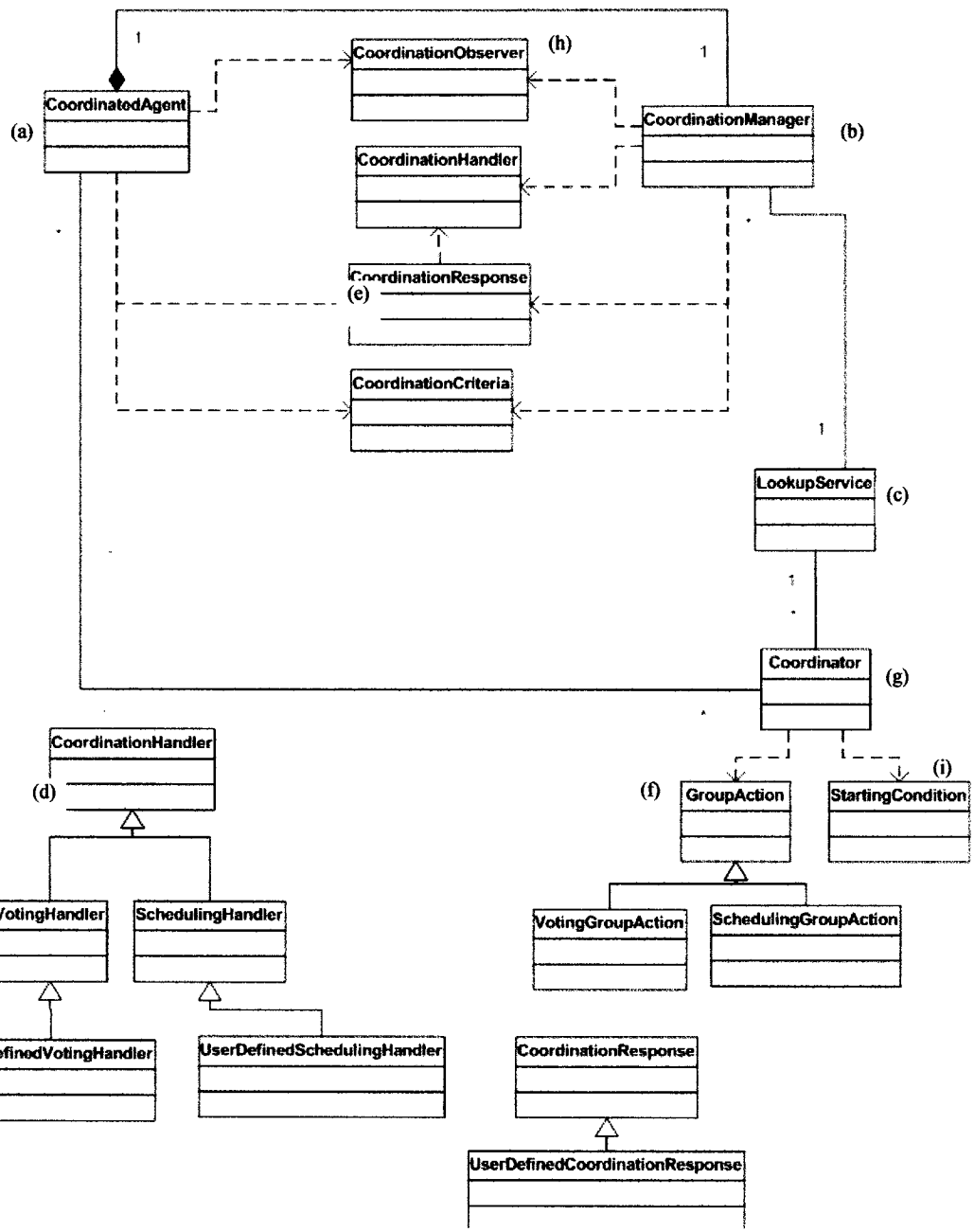


Figure 19 Static coordination class diagram

6.2.1.1 Static coordination class description

The following provides a detailed description of each of the classes.

6.2.1.1.1 CoordinationHandler

The role of the *CoordinationHandler*(d) is to perform the local computation which will contribute to the group coordination. This class is an abstract class which the application developer must extend to implement the application specific action. The developer does so by overwriting the abstract method *doLocalAction*:

abstract Object doLocalAction(Object argumentList)
throws DataTypeMismatchException

This method takes as parameter a list of arguments which is of type *Object*. The list contains abstract objects received from the coordinator. This is the place where the developer needs specify the interface between the group and the agent. The data passed to the function must be cast into the proper format to be used in the computation. Once the local computation is completed, the function returns an object which will be sent back to the coordinator. The *CoordinationHandler* hierarchy in Figure 19 (see d) shows how the developer can refine the requirement to move from the abstract concept toward the application specific handler. For instance, the *VotingHandler* can be a generalization of the voting coordination units and thus can perform the data conversion. However, each agent has its own policy for casting a vote, thus the *VotingHandler* is further extended to specify these specific policies.

6.2.1.1.2 CoordinationResponse

The role of the *CoordinationResponse*(see Figure 19 e) class is to decide whether or not to join in the coordination based on the current state of the agent and create a

CoordinationHandler if the agent decides to participate. This is an abstract class that represents an agent response to a coordination invitation. The developer must extend this class to implement the decision making about when to join the coordination and create a corresponding CoordinationHandler. Since the decision is based on the state of the agent, the CoordinationResponse must have some reference to state objects of the agent. The class contains an abstract method *doParticipate* which the developer must override to give specific details as to whether or not to join the coordination group when asked.

abstract CoordinationHandler doParticipate(CoordinatedAgent initiator)

The method takes as parameter a CoordinatedAgent who has initiated the coordination. This parameter may influence the decision of the agent to participate or not in the coordination. The function must return a CoordinationHandler if the agent is going to participate in the coordination and NULL otherwise. Note that for the case where the agent is only interested in the result of the coordination and not in participating in the coordination then this function will always return NULL.

6.2.1.1.3 GroupAction

A GroupAction has the role of performing the actions of the coordination that requires input from all participants. This class (f in Figure 19) is an abstract class which the developer must extend to implement the specific group actions. In the extension, the developer must override the abstract methods *doGroupAction* and *isComplete*..

abstract Object doGroupAction(Object ArgumentList)
throws DataTypeMismatchException

This method takes as parameter a list of arguments as a generic Object. The list contains abstract objects received from the coordinated agents. The function returns an object

which will be broadcast to the participants. Since a coordination may involve more than one phase, the doGroupAction must set a flag telling the coordinator when the coordination is complete.

boolean isComplete()

This method is meant to be used by the coordinator to check whether the coordination action has been completed. The application developer must override this function with the specifics of the actual instance of the coordination unit.

It can be noted that the number the doLocalAction from the CoordinationHandler and *doGroupAction* must agree on the number of computation phases and the data being passed at each phase. This is the way the developer specifies the specific behaviors of the various static coordination untis.

6.2.1.1.4 Coordinator

The *Coordinator* gathers the input from the participants, execute the group action and scatter the result to registered agents. This is an actual agent. The coordinator has two behaviors: initialization and execution. During the initialization, the coordinator sends invitation to all registered agent to join the group and then waits to receive join request from the participants. When the starting conditions are met, the initialization behavior terminates and the execution starts. The execution behavior is a repeated behavior with the following steps:

- gather input from coordinated agents
- invoke the *doGroupAction* in the *GroupAction*
- scatter the result returned by *doGroupAction* to all participants

The above three steps are repeated until the *GroupAction* completes in which case it will scatter a terminate message

It can be noted that the C*oordinator,* as shown in Figure 19g, is hidden from the application developer. This is an internal construct to manage the messages exchange between the agents and the group.

6.2.1.1.5 CoordinationObserver

A *CoordinationObserver* (Figure 19 h) has the role to provide the *CoordinatedAgent* with status of a coordination. When a *CoordinatedAgent* has registered for a coordination or has initiated a coordination, it is returned a CoordiationObserver to follow up on the progress of the coordination. A coordination can be in one of 4 states: not intiated, initiated, started, completed. Once a coordination has completed, the registered agents can get the results via the CoordinationObserver. A coordination result is stored until another coordination unit with the same group name is initiated.

The *CoordinationObserver* provides the following methods:

- CoordinationState getCoordinationState()

This method returns the state in which the coordination unit is. The return state is one of the coordination state mentioned above.

- Object syncGetCoordinationResult()

This method is provided to synchronously retrieve the result of the coordination. If the state of the coordination is "completed" then the last result is returned. Otherwise, if the coordination is in any of the other state, the *CoordinatedAgent* is blocked until the coordination completes and then the result is returned.

- Object asyncGetCoordinationResult()

This method is provided to asynchronously poll the result of the coordination. If the state of the coordination is "completed" then the last result is returned. Otherwise, if the coordination is in any of the other state a NULL pointer is returned. This function does not block the calling agent.

- void subscribeReaction(Observer, isRepeat)

This method allows a user to subscribe a reaction for each time a coordination unit complete its execution. If the user wants a reaction without having to poll or block for the coordination then it can use the subsribeReaction to set a reaction for the incoming result. The reaction can be set to execute once or to be repeated each time the coordination completes.

6.2.1.1.6 StartingCondition

The role of the *StartingCondition* (Figure 19i) is to determine when the first execution of the GroupAction of a coordination can be started. The rationale for this class is due to the fact that there is a delay between the initiation of the coordination unit and the response from the agents. Thus the coordination can only start when all agents joining satisfy a certain condition called the *StartingCondition.* That condition is application specific; hence this class is an abstract class that the application developer must extend to specify when the GroupAction within a coordination can be started. The following abstract method is to be overridden by the developer.

boolean isReady(CoordinatedAgentList, ElapsedTime)

This method takes as parameter a list of CoordinatedAgent that have registered to participate in the coordination and returns a Boolean value telling whether the group is ready to start. The decision can be a function of elapsed time since the initiation, a function of the number of participants or a combination of both. Hence the *ElapsedTime* is also included in the list of parameters.

6.2.1.1.7 CoordinationManager

The role of the *CoordinationManager* is to do the book keeping of the communication between the *CoordinationHandler* and the corresponding *Coordinator*. This class provides the application programmer interface (API) to the user to register with as well as to initiate coordination units. The following methods are provided:

- CoordinationObserver initiateCoordination(GroupName, startingCondition, GroupAction)

This method is to be called when an agent senses a need for coordination. This method will create a coordinator for the coordination and invite all agents registered to join the group.

- CoordinationObserver joinCoordinationGroup(GroupName, CoordinationResponse)

This method registers the calling agent with the group so that the agent is notified the next time a coordination is initiated on the group.

6.2.2 Dynamic Coordination

Figure 20 illustrates a class diagram for the implementation of the dynamic coordination. There are three main components to manage the coordination units within the agent and in the system: (a) C*oordinatedAgent*, (b) *MonitoredStateManager* and (c)*MonitoringManager*. The *CoordinatedAgent* can play both the roles of monitored agent and a monitoring agent. Hence each *CoordinatedAgent* contains a *MonitoredStateManager* to manage the internal objects being monitored and the *MonitoringManager* to manage the predicate being monitored and their corresponding reactions.

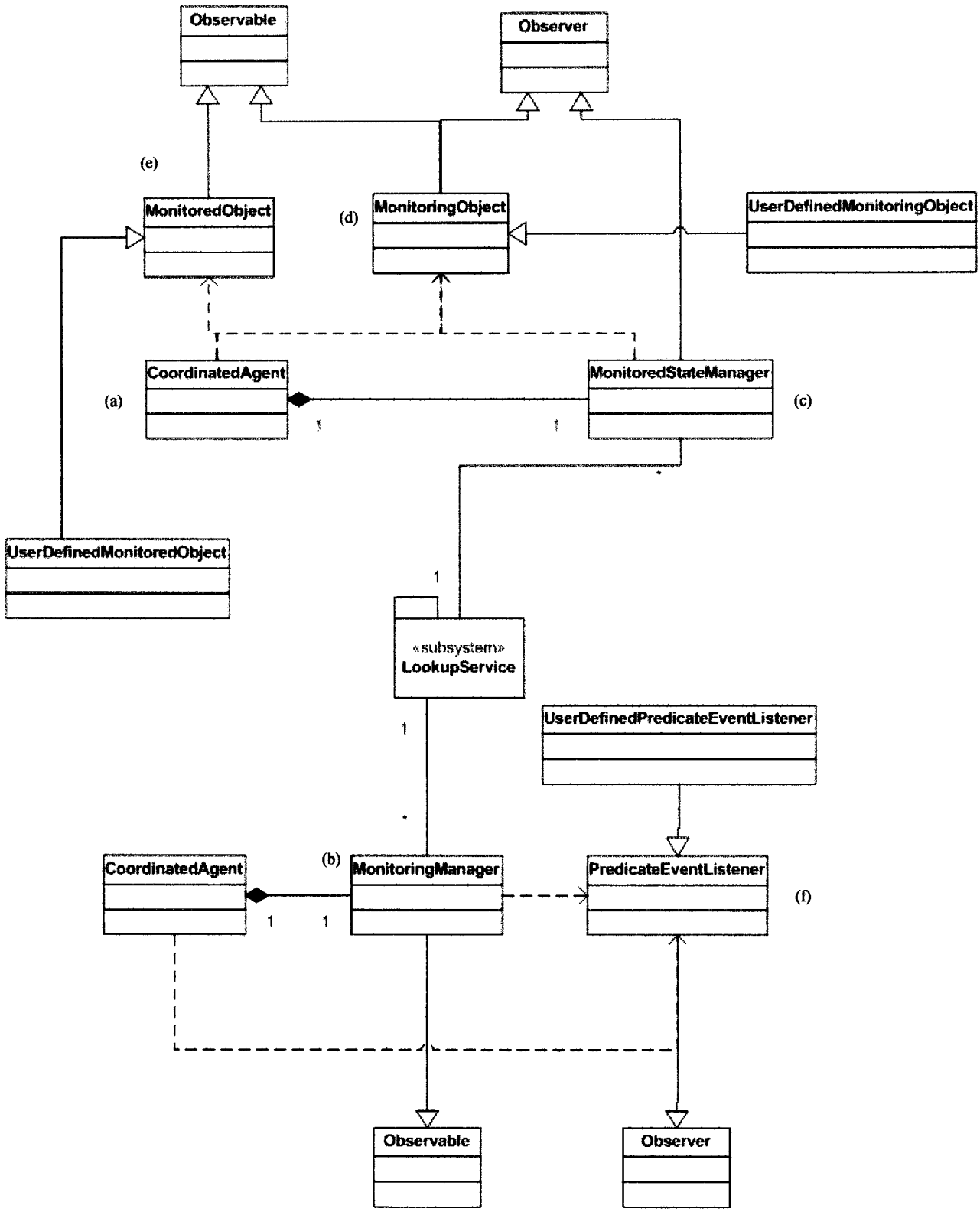


Figure 20 Dynamic coordination class diagram

6.2.2.1 Class description

This section gives a detail description of the classes composing a dynamic coordination unit.

6.2.2.1.1 MonitoringObject

The role of a *MonitoringObject* (Figure 20d) is to receive notification of changes in the *MonitoredObject* and to evaluate whether the change needs to be reported and report to the monitor if necessary. This class is placeholder for the developer to extend to implement the specific reporting policies. The MonitoringObject contains a list of MonitoredObject.

The following is an abstract method that the developer must override.

boolean mustReport()

This method uses the list of *MonitoredObject* to evaluate whether the current local state should be reported to the monitor. This method is called when there is a change with any of the *MonitoredObject* in the list.

6.2.2.1.2 MonitoredObject

The role of *MonitoredObject* (Figure 20e) is to report changes in the state of the object to the *MonitoringObject.* This is an abstract class that the application developer must extend to expose part of the state of the agent to be monitored. This class is an extension of the Observable class thus it has a method to subscribe MonitoringObject as observers. A rule to define the MonitoredObject is to notify about all changes to the state of the object but

this is open to the user. A guideline is to have "set" methods for each of the state variables and the notification can be appended to those "set" methods.

6.2.2.1.3 MonitoredStateManager

The role of *MonitoredStateManager* (Figure 20c) is to keep track of the different local predicates being monitored and perform registration of the monitor with the corresponding *MonitoringObject*

This class acts like a glue layer between the agent and the local Monitoring system. It provides an interface to set up a local monitoring. It provides the following method.

setupMonitoredPredicate(GroupName, MonitoringObject, MonitoredObject[ ])

This method is called when the user wishes to expose part of the agent's state to be monitored in the form of a local predicate. The method registers the *MonitoringObject* to the *MonitoredObject* and creates an entry with the *MonitoringObject* indexed by *GroupName*.

6.2.2.1.4 MonitoringManager

The *MonitoringManager's r*ole (Figure 20b) is to keep track of the different global predicates being monitored and their corresponding *PredicateEventListener.* This class acts like a glue layer between the agent and the external Monitoring system. It provides an interface to set up a monitor for a global predicate with the following method.

setPredicateTrigger(GroupName, PredicateEventListener, is_sync_flag)

This method is called when the user wants to monitor for a global predicate involving multiple agents. This method will register the monitor with the corresponding monitored

agents and create an entry with the *PredicateEventListener* and the *is_sync_flag* indexed by *GroupName* identifying the group being monitored. The *is_sync_flag* indicates whether the reaction is to be taken atomically with the occurrence of the predicate or not.

6.2.2.1.5 PredicateEventListener

The role of *PredicateEventListener* (Figure 20b) is to react to occurrence of the global predicate being monitored. This class is an abstract class that the developer must extend to define what action should be taken when the global predicate becomes true.

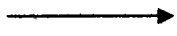
The following abstract method is the mean to specify the reaction by the application.

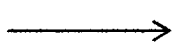
abstract void reactToTrue( )

This method does not take any parameter and returns void. The developer must implement the application specific reaction to take place when the global predicate becomes true.

6.3 Scenarios

This section illustrates the roles of the various classes by means of several sequence diagrams to show the interaction among the classes. The following notation is being used for distinguishing the type of communications:


Function call


Message passing

Return value

The section is divided into two parts: static coordination description and dynamic coordination description.

6.3.1 Static coordination

This section describes the interaction of the various classes to realize the functionality specified for each of the supported primitives of the static coordination unit:

- CoordinationObserver joinCoordinationGroup(GroupName, CoordinationResponse)
- CoordinationObserver initiateCoordination(GroupName, startingCondition, GroupAction)
- cancelCoordination(CoordinationID)
- waitForCoordinationResult(CoordinationObserver)
- pollForCoordinationResult(CoordinationObserver)

6.3.1.1 JoinCoordinationGroup

Figure 21 shows a sequence diagram for the joinCoordinationGroup. A *CoordinatedAgent* joins a coordination group by calling the joinGroup method from the

CoordinationManager with parameters a *GroupName* and a *CoordinationResponse.* The *CoordinationManager* creates a CoordinationObserver for the corresponding GroupName and then forward the GroupName to the LookupService for publication. Upon receipt of the request to join the group, the *LookupService* checks whether the group already exists. If the group already exists then the agent ID is added to the group. Otherwise, a new group is created with the given name and the agent is added to the newly created group. Finally when the *CoordinationManager* has received acknowledgement from the *LookupService*, the *CoordinationObserver* for the static coordination unit is returned to the *CoordinatedAgent.*

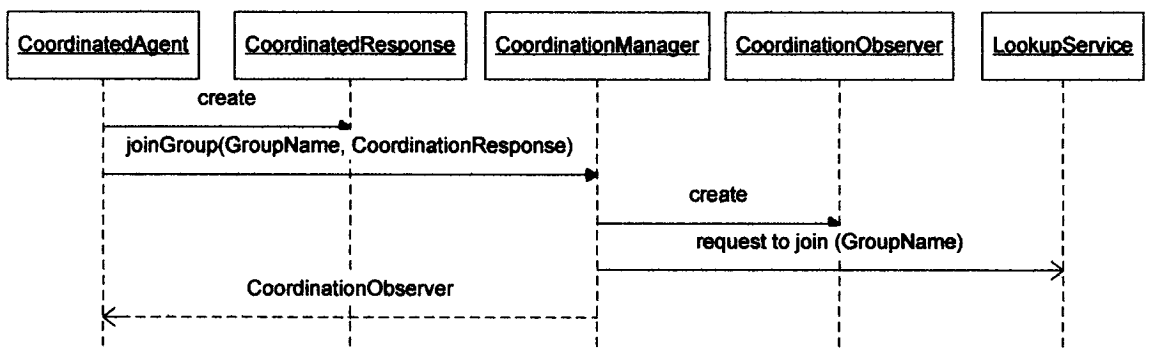


Figure 21 JoinCoordinationGroup sequence diagram

6.3.1.2 InitiateCoordination

Figure 22 shows a sequence diagram of the *InitiateCoordination* primitive. In this scenario, it is assumed that a group has already been created, and the coordinated agents

A and B are currently members of the group. Let B play the role of initiator and A play the role of a participant. Then the sequence is as follows.

1- CoordinatedAgent B initiates coordination by calling the *initiateCoordination* method in *CoordinationManager* with parameters a *GroupName*, *StartingCondition* and *GroupAction.*

2- The *CoordinationManager* sends a lookup request given a *GoupName* to the *LookupService* to get the list of agents currently registered in the group.

3- The *LookupService* does the match making between the *GroupName* and if a match is found then the list of all *CoordinatedAgent* belonging to the group is sent back to the initiator

4- Upon receipt of the list of *CoordinatedAgent*, agent B then creates a *Coordinator* with the list of *CoordinatedAgent* and add the newly created *Coordinator* to the *CoordinationManager* for later book keeping.

5- The *Coordinator* then notifies all registered agents about the initiation of the coordination

6- CoordinatedAgent A who is a participant in the coordination unit receives the notification of the coordination initiation and passes the message to the *CoordinationManager*

7- The *CoordinationManager* updates the status of the *CoordinationObserver* to *initiated*, and call upon the *CoordinationResponse* to decide whether or not to participate in the coordination.

8- If the *CoordinationResponse* decides to participate in the coordination then a *CoordinationHandler* is created and returned to the *CoordinationManager* and the manager can then send a join coordination message to the *Coordinator*

9- Otherwise, if the *CoordinationResponse* decides not to participate in the coordination but wishes to know about the result, then nothing is returned to the *CoordinationManager*, the manager then sends a register for result message to the *Coordinator* for receipt of the coordination result when the later completes.

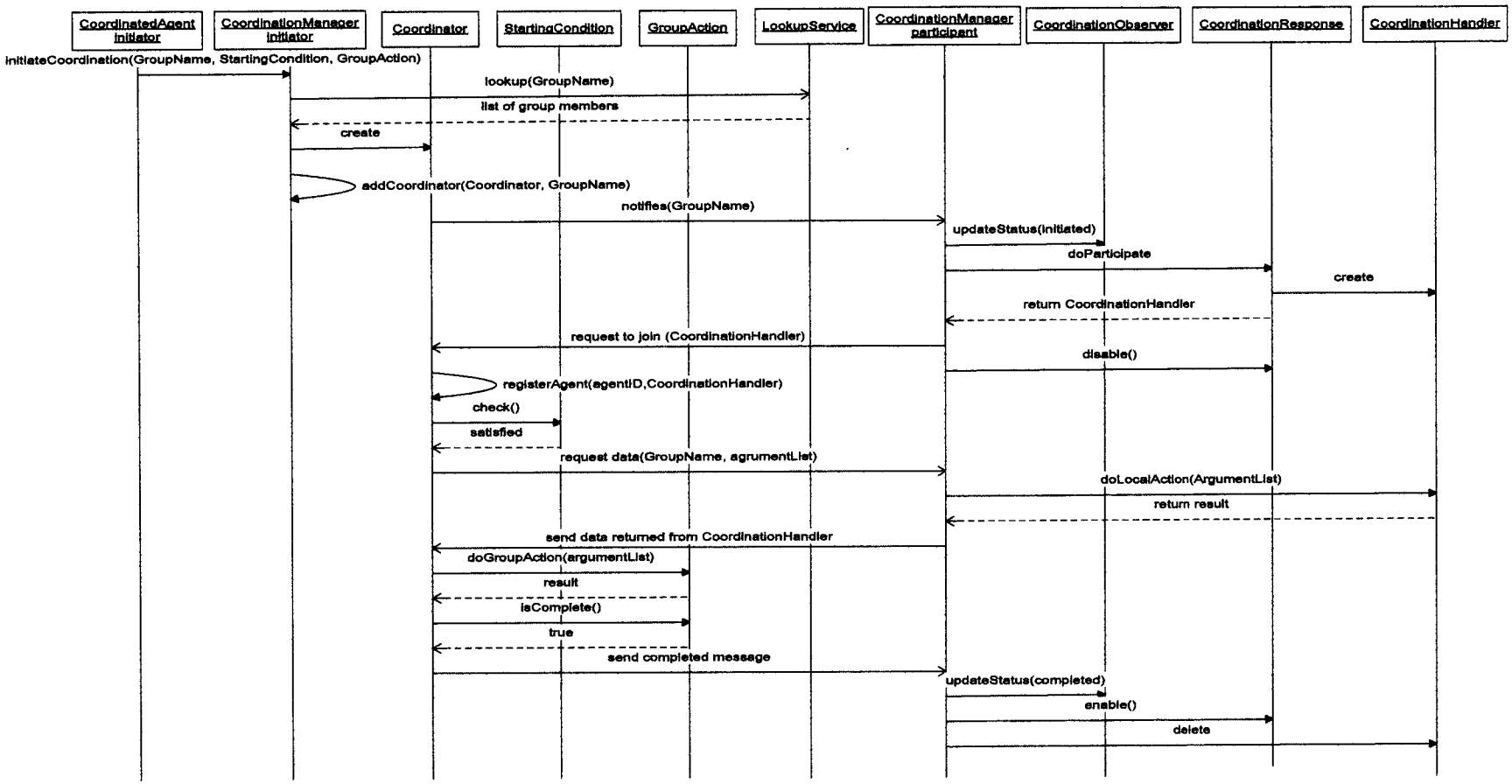


Figure 22 InitiateCoordination sequence diagram

10- In either case, the *CoordinationManager* after sending the reply will disable the *CoordinationResponse*. The rationale for this is that a group is formed to execute one instance of coordination. Hence if at some point another agent, who has not yet received the invitation, happens to initiate the same coordination unit once again, then by disabling the coordination response, the system guarantees that only one instance of the coordination unit is executed at a time.

11- The *Coordinator* receives the response from the agent and then registers the agent according to whether they are a participant or an observer

12- When the *StartingCondition* for the coordination is met, the coordinator starts the coordination action and ignore any further incoming registration messages except for observers.

13- If *StartingCondition* is can no longer become true then the coordination will be aborted in which case an *abort* message is sent to all registered agents so their *CoordinationManager* can reset the status of the coordination *waiting*.

14- Otherwise the *Coordinator* sends "request for information" to all participants

15- Participant's *CoordinationManager* receives request and forward to the corresponding *CoordinationHandler*

16- The CoordinationHandler computes the required information and sends it back to the *Coordinator*

17- The *Coordinator* waits for information from all agents to come and then execute the *GroupAction*

18- The Coordinator then broadcast the result of the group action to all participants

repeat 14-18 until all *GroupAction* have completed.

19- Upon completion of all *GroupAction*, the *Coordinator* then broadcasts a *complete* message to all registered agents

20- The receiving *CoordinationManager* will then notify all behaviors waiting on this coordination, set the status of the *CoordinationObserver* to *complete* with the corresponding timestamp and re-enable the *CoordinationResponse*

6.3.1.3 CancelCoordination

After the call to *initiateCoordination*, the initiator can call for a cancellation of the coordination using the coordination ID to terminate the coordination unit execution. Upon such call, a *cancel* message is going to be sent to the *Coordinator.* The *Coordinator* then broadcasts an *abort* message to all registered agents. Upon receipt of the message the *CoordinationManager* reset the corresponding *CoordinationHandler* and reset the status of the coordination in the *CoordinationObserver* to *waiting.*

6.3.1.4 WaitForCoordinationResult

At any point in time after the *CoordinatedAgent* has joined the coordination group, the *CoordinatedAgent* can use the *CoordinationObserver* to call *waitForCoordinationResult* to wait for receipt of the result of the coordination. This primitive registers the behavior making the call to be blocked on the CoordinationObserver with the CoordinationManager. That behavior will remain blocked until the coordination has completed, in which case the result is returned.

6.3.1.5 PollForCoordinationResult

This primitive is similar to the WaitForCoordinationResult except that the calling behavior is not blocked.

6.3.2 Dynamic coordination

This section describes the interaction of the various classes to realize the functionality specified for each of the supported primitives of the dynamic coordination unit:

- setupMonitoredPredicate(GroupName, MonitoringObject, MonitoredObject[ ])
- setPredicateTrigger(GroupName, PredicateEventListener, is_sync_flag)

In addition, two scenarios show the mechanism that supports the synchronous and asynchronous trigger of the reaction.

6.3.2.1 SetupMonitoredPredicate

Figure 23 shows the sequence diagram to realize the *SetupMonitoredPredicate* primitive.

A *CoordinatedAgent* calls *setupMonitoredPredicate* from the *MonitoredStateManager* by passing a *GroupName*, a *MonitoringObject* and a set of *MonitoredObject.*

The *CoordinatedAgent* then subsribes the *MonitoringObject* to each of the *MonitoredObject's.*

An entry is then created in the *MonitoredStateManager* to keep track of the local data being monitored within the agent.

The *MonitoredStateManager* then sends a group registration to the *LookupService* given a *GroupName* so that other agents can start monitoring.

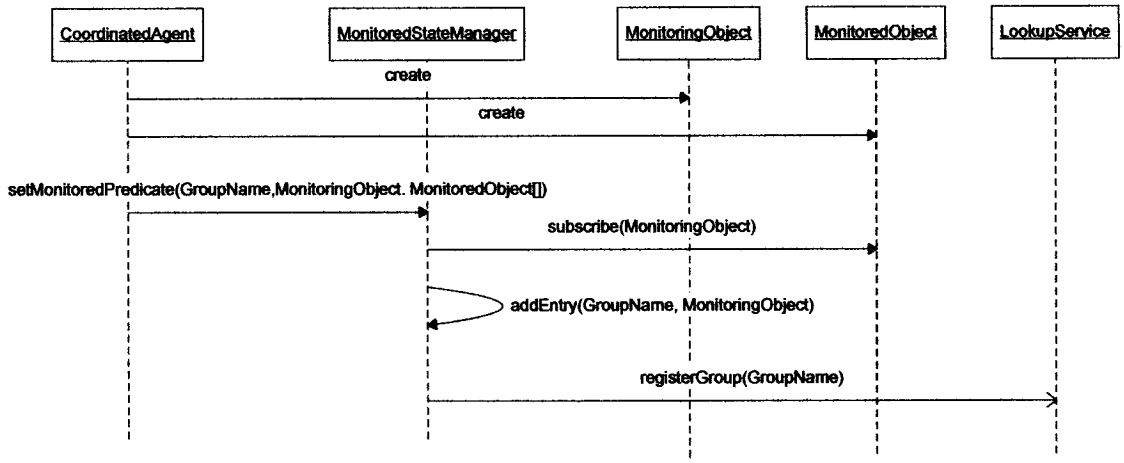


Figure 23 SetupMonitoredPredicate sequence diagram

6.3.2.2 SetPredicateTrigger

Figure 24 shows the sequence diagram to realize the SetPredicateTrigger primitive.

A *CoordinatedAgent* playing the role of the monitor call setGlobalPredicateMonitoring from the *MonitoringManager* by specifying a *GroupName*, *PredicateEventListener* to react when the predicate becomes true and a flag indicating whether the reaction is synchronous or asynchronous.

The *MonitoringManager* then sends the global predicate to the *LookupService*

The *LookupService* searches for all agents belonging to the monitored group then sends the list of such agents to the *MonitoringManager*

The *MonitoringManager* then sends a monitoring request to each agents in the group

The *MonitoredStateManager* receiving the monitoring request will match with the corresponding *MonitoringObject* and registers the monitor with the *MonitoringObject* and set the corresponding synchrony flag.

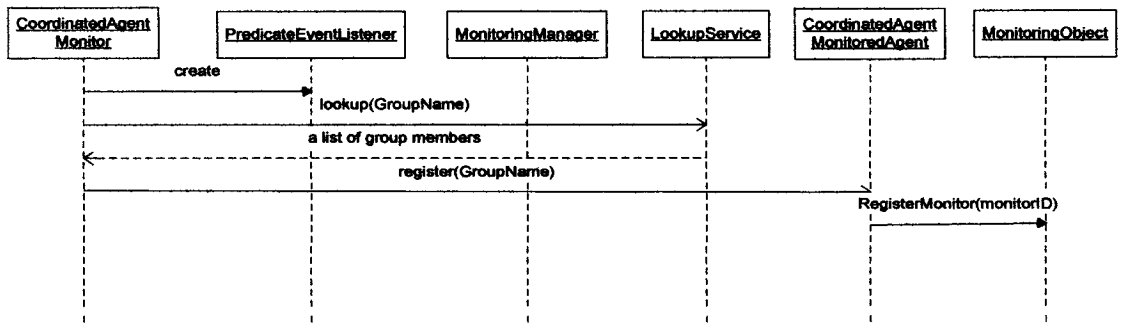


Figure 24 SetPredicateTrigger sequence diagram

6.3.2.3 Synchronous Predicate event triggering

Figure 25 shows the sequence diagram to realize the Synchronous Predicate event triggering.

A *CoordinatedAgent* updates the state of one of the *MonitoredObject*

The *MonitoredObject* reports changes to the *MonitoringObject*

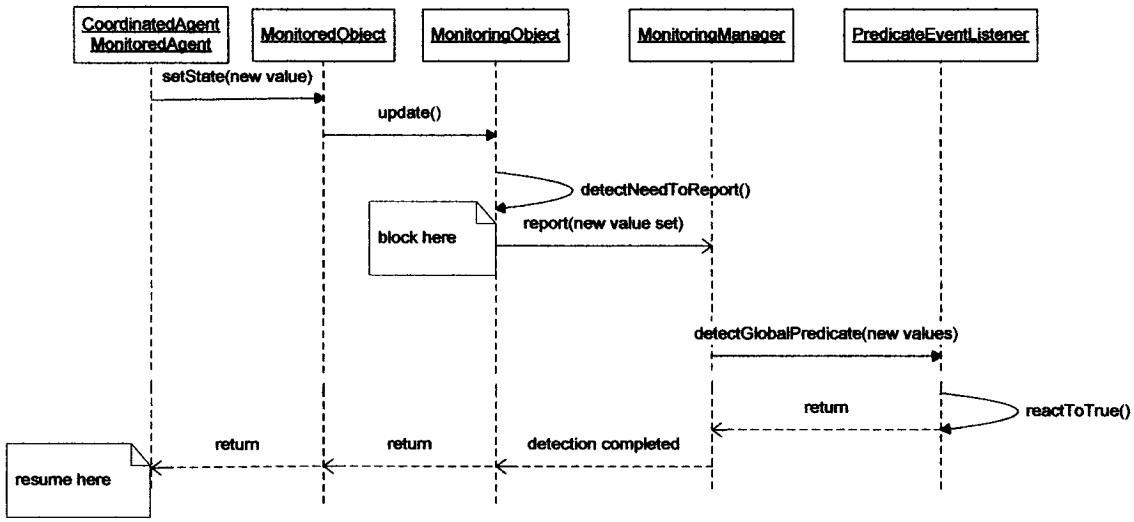


Figure 25 Synchronous Predicate event triggering sequence diagram

The *MonitoringObject* detects necessity to report to monitor and report to the monitor and block waiting for the response from the monitor

The *MonitoringManager* receives the report and dispatch to the corresponding *PredicateEventListener*

The *PredicateEventListener* detects the global predicate.

If the global predicate has occurred then the reaction is triggered

Upon completion, the *MonitoringManager* send a completion message of to the monitored agent

When the completion messages from all the monitors have been received then the *CoordinatedAgent* can resume execution

6.3.2.4 Asynchronous Predicate event triggering

Figure 26 shows the sequence diagram to realize the Synchronous Predicate event triggering.

A *CoordinatedAgent* updates the state of one of the *MonitoredObject*

The *MonitoredObject* reports changes to the *MonitoringObject*

The *MonitoringObject* detects necessity to report to monitor and report to the monitor and resume execution

The *MonitoringManager* receives the report and dispatches it to the corresponding *PredicateEventListener*

The *PredicateEventListener* detects the global predicate.

If the global predicate has occurred then the reaction is triggered

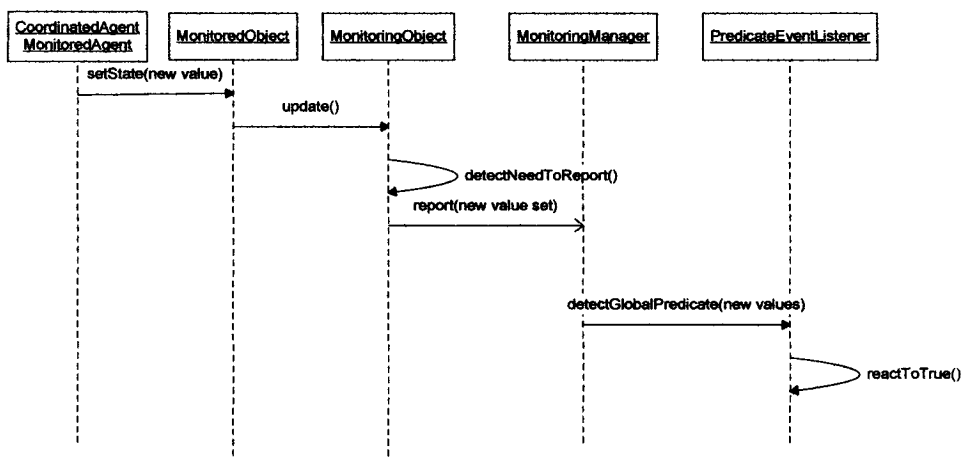


Figure 26 Asynchronous Predicate event triggering sequence diagram

7. Automated E-Commerce Modeling

This chapter shows how a developer can use the proposed framework to design applications involving coordination. The example of an automated e-commerce application is used to illustrates the concept. The chapter starts with an analysis of the coordination requirements in the example e-commerce application. The second part provides a mapping of the requirement to the constructs in our framework. Finally, an extension of the designed application is shown to illustrate the reusability and the flexibility of the proposed ECUMAC framework.

7.1 Coordination requirement for ecommerce application

Agents in an ecommerce application plays two main role: buyer and seller. Agents represent their corresponding users. Then depending on the tasks assigned by the user,

the agent can be playing either one of the two roles. In order to appreciate the need for coordination requirement, consider the following cases of coordination failure:

(1) A buyer does not take an action when there is a seller who is selling the product satisfying the buyer's product specification. (right opportunity wrong action).

(2) A buyer commits a buying action when there is no product item satisfying the buyer's product specification being sold. This happens when more than one buyer attempts to buy the same product from the same seller who has only one item to sell. Both of them will commit the transaction but only one of them will get the product. (right action wrong opportunity).

These two coordination failures translate into the requirement of awareness of the buyers about the market's current situation and the right action being taken in the right time interval. By symmetry the same can be said about the sellers.

This section shows how the proposed ECUMAC platform can be used to model a synthetic market activity. A human buyer can specify a set of tasks to be fulfilled by the corresponding buyer agent. A task consists of buying a set of products. The product in the shopping list must satisfy the conditions, as specified by the buyer user, simultaneously. That is either all products in the list are bought or none of them is bought. These products can be tickets for a flight trip. A user, based in Nunavut, may request to book flights for a trip starting from Nunavut going to Montreal then to Toronto and finally back to Nunavut. Knowing that flights to Nunavut are very rare and comes every six months, the user cannot afford to wait for the next available flight. Either the whole trip is booked or no trip is booked at all. The final system should address this issue with user involvement only at the specification time.

7.2 Mapping to the coordination framework

In order to map the coordination requirements to the proposed model, the following steps should be followed.

- Identify the coordination units
- Define the coordination units
- Connect the coordination units

7.2.1 Identify the coordination units

The reason for the problem of *right opportunity wrong action* is that the buyer does not know about the sellers and in particular their joint states. The remedy is to make the buyer monitor all sellers who sell the products of interest. In this way, the buyer is informed of any changes the seller might make to its product's availability and can consequently react correspondingly. This pattern of interaction resembles the dynamic coordination unit model. Hence, this requirement for product monitoring necessitates a dynamic coordination unit.

The second problem has to do with timing. The seller and buyers can negotiate the price and terms, but when the seller has committed to one buyer then all other negotiations with the seller about that specific product item must be stopped. Hence to correct the *right action wrong opportunity problem*, one can use the static coordination model to develop an atomic transaction where only one buyer and one seller will commit to each

transaction and there is no agent which is half-committed to a transaction. Such half-commitment must be aborted to bring the agent back to a state where the commitment process has never started.

7.2.2 Define the coordination units

The previous section has identified two coordination units that will satisfy the coordination requirements of this example application named: product monitoring and auction. The product monitoring is a dynamic coordination unit and auction is a static coordination unit. In this section, the specific each of these units will be mapped to the corresponding coordination unit model.

7.2.2.1 Product monitoring as a dynamic coordination unit

To accomplish the product monitoring task, there are two layers of subtasks the buyer agent needs to do: monitor each product from different vendors and monitor the joint status of the different products. For example, suppose that the buyer agent B needs to buy product X and Y. X is sold by seller S1 and S2, and Y is sold by S3 and S4. The first layer for B consists of two subtasks. Subtask 1 is to monitor the product advertisement of X from sellers S1 and S2 and subtask 2 is to monitor the product advertisement of Y from sellers S3 and S4. Let say that subtask 1 or 2 succeeds when one of the sellers advertise the product that matches the buyer user specification. Hence the second layer consists of monitoring the status of subtasks 1 and 2 to detect when the two subtasks succeed jointly and then perform the transaction on both products. Consequently, to accomplish its

market monitoring tasks, a buyer agent needs helper agents to perform the subtasks in the first layer.

Figure 27 shows the mapping to the first layer. The buyer helper agent monitors the product advertisement from the different sellers. When one ad matches the buyer user specification then the buyer helper agent reports to the main buyer agent. Figure 28 shows the mapping to the second layer. The buyer helper reports to the main buyer. The buyer agent then detects whether the joint state where all the user specification have been matched has been reached. If so then the buyer agent goes forward in making the transactions.

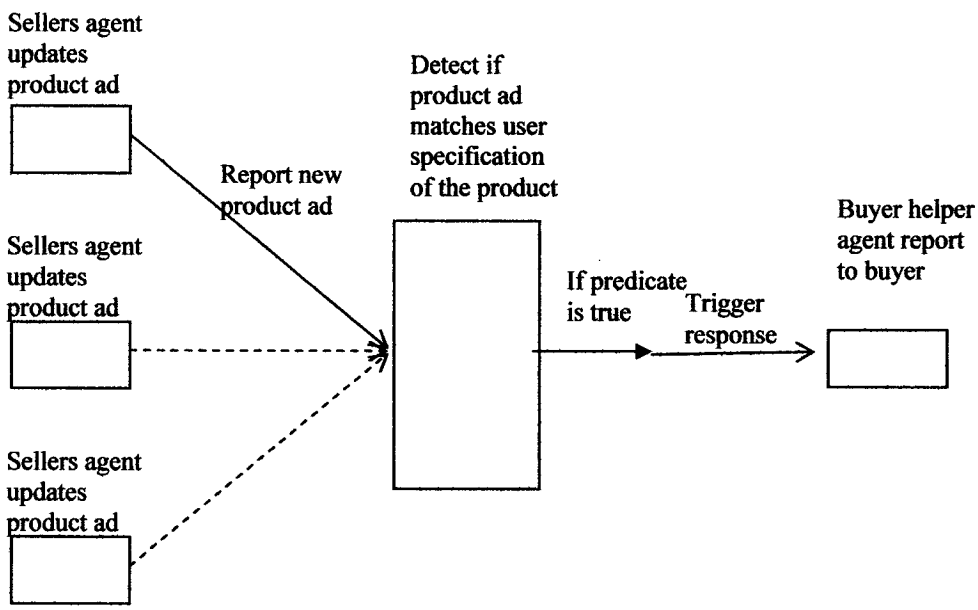


Figure 27 Product monitoring mapping to a dynamic coordination unit (first layer)

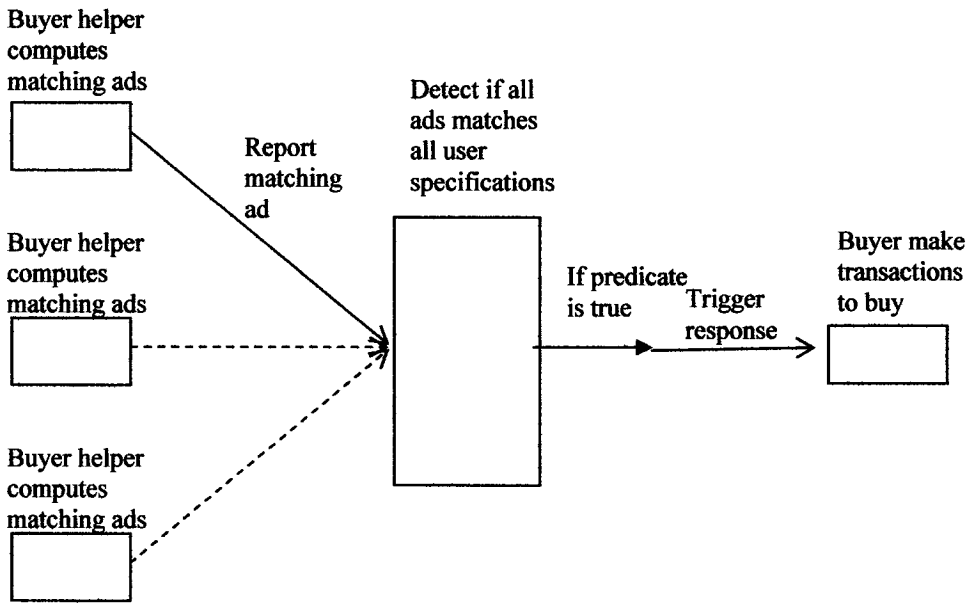


Figure 28 Product monitoring mapping to a dynamic coordination unit (second layer)

7.2.2.2 Atomic transaction as a static coordination unit

Joint state monitoring alone is not enough to ensure that the buyer agent buys all the products in the shopping list at once. Consider the case where the buyer agent has just been notified that all products requested matches the advertisement of 3 vendors, the buyer agent then sends to request to buy these products. However, due to the concurrency of the processing of these requests, one of the product might be bought by another agent. Thus only two out of three products can be bought. As in the description of the scenario of user planning the trip, the consequence might be very severe. Hence there is a need to make the transaction atomic. This can be achieved via a static coordination unit.

Figure 29 shows one round of the mapping of the atomic transactions to a static coordination unit.

Upon receipt of the buyer request, the seller agent processes the request to see if the quantities are available. If so, then the seller posts an offer to the group, otherwise it posts a reject message. The group then checks to see if all the input received are offers. If they all are, then the group commit to the sellers. Otherwise, the group sends an abort message. Once the result from the group is received, the sellers commit or abort the transaction correspondingly. It can be noted that the coordination unit is initiated by the buyer who provides the group action yet the buyer is not a participant of the coordination unit, it is only an observer.

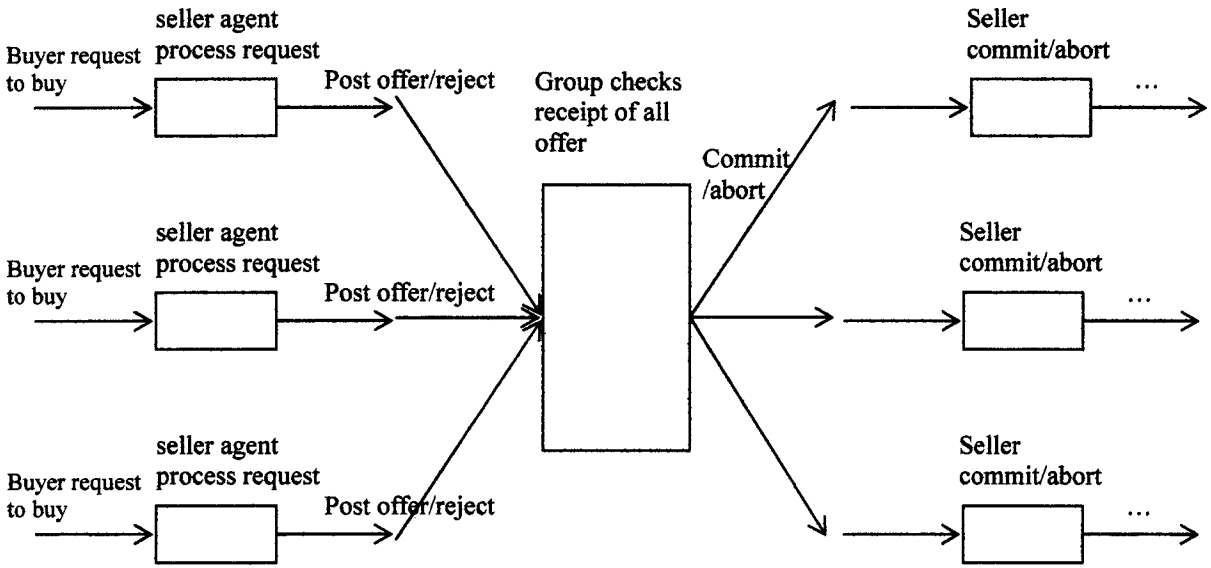


Figure 29 Atomic transaction mapping to a static coordination unit

7.2.3 Connect the coordination units

Now that the individual coordination units have been defined, this section illustrates how to connect them together. In this example, it can be observed that the atomic transaction must be initiated and completed within the reaction of the product monitoring unit. Hence the two units can be combined via the containment composition. Figure 30 shows the containment of the atomic transaction unit by the product monitoring unit in the reaction section.

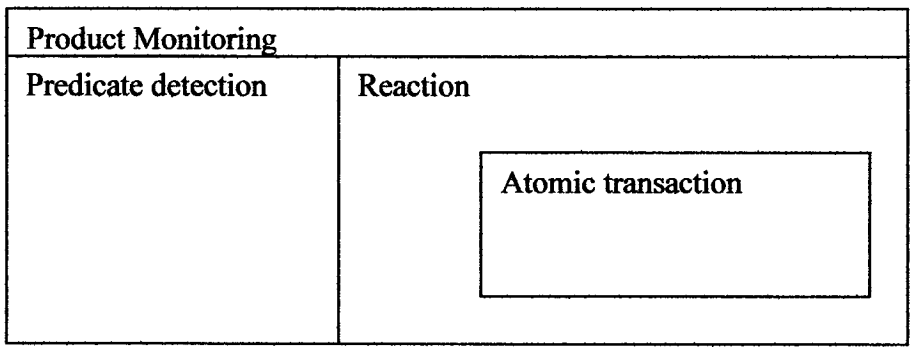


Figure 30 Containment of the atomic transaction unit by the product monitoring unit

7.3 Extension to the application

This section illustrates the easiness to extend the current system to accommodate new features. There are two levels at which an application can be extended: computational level and coordination level. This section discusses examples of extending a current application by adding new features requiring changes at each these two levels.

Consider the following extensions to the above synthetic market application. A new feature is needed to support the buyer user in setting up a backup plan in case the shopping list cannot be matched for a certain period of time. A backup plan could be to relax the constraints to tolerate only a subset of the shopping list yet only in some specified combination. This new feature can be achieved via a computation extension. In Figure 29, the group action consists of checking whether all offers have been received. It will send an abort message if any of the messages received is a reject message. To accommodate the new feature, the only need is to replace the current group computation by a new version that is more tolerant. The new version will have additional checks to see if any of the acceptable combinations of offers have been received. Only if all the combinations fail then the abort message is sent. Effort is minimized in making such a change because that computation is totally independent on the state of the coordination.

Consider further that the system is now to support group purchase. That is a group of agents gather together to purchase some good so that the quantity transacted will be greater and the price will be lower. To support this feature, a coordination structure change is required. An addition of static coordination unit is needed to first of all elect a representative for the group during the purchase. The scenario is as follows. The first buyer agent who wants to perform a group purchase creates a group for that purpose and joins that group. Later on other agents will also join that group. When one of the agents in the group decides to start the purchase, that agent initiates the coordination unit by setting the voting handler as the group action. At the end of the coordination, all members know who the representative is for the group and that representative can start the monitoring and proceeds as before. However, prior to do any commitment, the members

of the group must be consulted again. To do that, the representative sends to the members of the group purchase the sellers offers. Then it creates another static coordination unit the consultation. That coordination unit takes as input from the participant an acceptance or refusal to commit to the purchase. If any one of the agent refuses then the transaction is aborted. Figure 31 shows the re-composition of the coordination units to support the group purchase feature. The atomic transaction is replaced by a similar coordination unit which contains the additional coordination unit for the members' consultation.

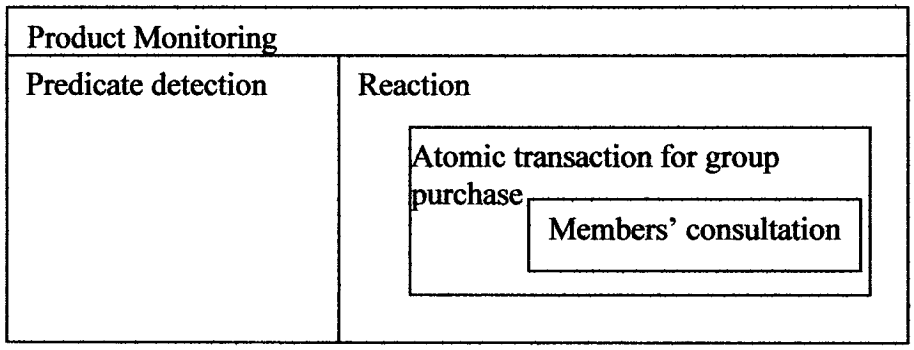


Figure 31 Group purchase composition

8. Conclusion and Future Work

Developers of multi agent system have always been faced with the problem of coordinating agents. The problem stems from the fact that the agents are running concurrently yet they are related to each other. Much effort is spent to ensure that such relations hold at all time during the execution of the system. Such effort usually results in very complex systems and the complexity increases the maintenance cost or the cost for further development. The coordination support through ECUMAC, the model proposed in this thesis, has the solution to leverage the effort spent in maintenance and extension. The model is based on the concept that a coordination requirement set can be realized by a set of small coordination primitives called the *coordination units*. These units can be classified into two categories: the static coordination units and the dynamic coordination units. With these two abstractions, applications can refine the details and compose the units to define multi-agent systems. As the needs change, the computation or structure of the coordination units can be replaced to satisfy the new sets of requirements. A sample case study is used to illustrate the ease with which the application developer can use the proposed model.

Considering the support provided by ECUMAC through the modularity of the components and through the composition model, a conclusion can be drawn that the proposed solution has promises to support the large scale development of multi-agent systems. However, ECUMAC is only a first step to support such large scale developments. Further research in the following areas will represent major complementary to the proposed framework:

- Coordination units' library. Since the coordination units can be extended and refined, it can also be abstracted from a collection of specific units to represents classes of coordination thus forming a hierarchy of units. Later if an application needs coordination constructs for a particular class of coordination, a lookup in the library will gives the possible option to reuse the existing code and then refine it to fit the application needs.
- Visual integrated development environment. The fact that the coordination units are composable suggests that there is a possibility to build a graphical user interface where the user can build a system with simple drag-and-drops of the coordination units. With such tools, development can significantly be reduced.

9. References

[1] Sonia Bergamaschi, Gionata Gelati, Francesco Guerra and Maurizio Vincini, "Experiencing AUML for the WINK Multi-Agent System", Proceeding AIIA and TABOO Workshop: From Object to Agents, 2003.

[2] H. Van Dyke Parunak, James Odell, "Representing Social Structures in UML", Autonomous Agents, 2001.

[3] O. Kornienko, S. Kornienko and P. Levi, "Collective decision making using natural self-organization in distributed systems", CIMCA, 2001: 461-471.

[4] Panzarasa, P. and Jennings, N. R. and Norman, T. J. "Formalising collaborative decision making and practical reasoning in multi-agent systems". Journal of Logic and Computation, (2001): 55-117.

[5] Onn Shehory, Sarit Kraus, "Methods for task allocation via agent coalition formation". ACM. Artificial Intelligence. Volume 101, 1998: 165-200.

[6] K. Schelfthout, T. Coninx, A. Helleboogh, T. Holvoet, E. Steegmans, and D. Weyns, "Agent Implementation Patterns", Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies. 2002: 119-130

[7] B.Bayerdorffer, "Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems", PhD. dissertation, Dept.of Computer Sciences, Univ. of Texas at Austin. Dec. 1993

[8] James C. Browne, Kevin Kane and Hongxia Tian, "An Associative Broadcast Based Coordination Model for Distributed Processes", ACM, Lecture Notes In Computer Science. Vol. 2315. 2002: 96-110

[9] Sandra C. Hayden, Christina Carrick, Qiang Yang, "A Catalog of Agnet Coordination Patterns", ACM. International Conference on Autonomous Agents, 1999: 412-413

[10] C. Castelpietra, L. Iocchi, D. Nardi, M. Piaggio, A. Scalzo, A. Sgorbissa, "Coordination among Heterogeneous, Robotic Soccer Players", In Proc. of International Conference on Intelligent Robots and Systems, 2000

[11] Nicholas Carriero, "Coordination Languages and their Significance", ACM Communication. Vol 35. February 1992: 97-107

[12] Henry Muccini, Fabio Mancinelli, "Eliciting Coordination Policies from Requirements", ACM. Symposium on Applied Computing. 2003: 387-393
[13] P. Ciancarini, F. Franze and C. Mascolo, "A Coordination Model to Specify Systems Including Mobile Agents", IEEE. International Workshop on Software Specifications & Design, 1998: 96-106
[14] David Gelernter, "Generative Communication in Linda", ACM. ACM Transactions on Programming Languages and Systems. Vol 7. 1985: 80-112
[15] Peiyi Tang, Yoichi Muraoka, "On-Demand Coordination of First-Order Multiparty Interactions", Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems. November 1999
[16] Andrea Omicini, "On the Semantics of Tuple-based Coordination Models", ACM. Symposium on Applied Computing. 1999: 175-182
[17] Victor R. Lesser, "Reflections on the Nature of Multi-Agent Coordination and Its Implication for an Agent Architecture", Autonomous Agents and Multi-Agent Systems. Volume . 1998: 89-111
[18] Dwight Deugo, Michael Weiss and Elizabeth Kendall, "Reusable Patterns for Agent Coordination", Chapter 14 in Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R. (eds.), "Coordination of Internet Agents: Models, Technologies, and Applications", Springer 2001
[19] Edmund H. Durfee, "Scaling Up Agent Coordination Strategies", IEEE. Computer. Volume 34. 2001: 39-46
[20] Nicholas V. Findler and Raphael M. Malyankar, "Social Structures and the Problem of Coordination in Intelligent Agent Societies", IMACS. Agent-Based Simulation, Planning and Control in IMACS. 2000: 122-127
[21] J.A. Giampapa and K. Sycara, "Team-Oriented Agent Coordination in the RETSINA Multi-Agent System", tech. report CMU-RI-TR-02-34, Robotics Institute, Carnegie Mellon University, December, 2002.
[22] Paul Valckenaers et al., "The Design of Multi-Agent Coordination And Control Systems Using Stigmergy", Proceedings of the IWES'01 Conference, March 2001
[23] Andrea Omicini and Franco Zambonelli, "TuCSoN: a Coordination Model for Mobile Information Agents", 1st International Workshop on Innovative Internet Information Systems, June 1998
[24] Mirko Viroli, "Comparing Semantic Frameworks for Coordination: On the Conformance Issue for Coordination Media", ACM. Symposium on Applied Computing. 2003: 394-401
[25] M. Pinto, L. Fuentes, M. E. Fayad and J. M. Troya, "Separation of Coordination in a Dynamic Aspect Oriented Framework", ACM. Aspect-oriented software development. 2002: 134 -140
[26] Jason I. Hong, "An Overview of the Jini Coordination Framework", University of California 2000
[27] Martin Beer et al., "Negotiation in Multi-Agent Systems", Knowledge Engineering Review. volume 14. 1999: 285-289

[28] Gruia-Catalin Roman and Jamie Payton, "Agent Coordination Paradigms in Mobile UNITY", Technical Report, Washington University, Department of Computer Science and Engineering, St. Louis, Missouri. 2003
[29] Robert Tolksdorf, "Models of Coordination", Springer-Verlag . Proceedings of the First International Workshop on Engineering Societies in the Agent World: Revised Papers. 2000: 78-92
[30] Gal A. Kaminka et al., "Gamebots: A Flexible Test Bed for Multiagent Team Research", Communications of the ACM. 2002: 43-45
[31] Thuc Vu et al., "MONAD: A Flexible Architecture for Multi-Agent Control", ACM. International Conference on Autonomous Agents. 2003: 449-456
[32] Gal A. Kaminka et al., "Monitoring Teams by Overhearing: a Multi-Agent Plan Recognition Approach", Journal of Artificial Intelligence Research. volume 17. 2002: 83-135
[33] Kyungkoo Jun et al., "A subscription-based Monitoring Model for Distributed Object Systems", Department of Computer Science, Purdue University 1998
[34] Brett Browning, Gal A. Kaminka and M. Veloso, "Principled Monitoring of Distributed Agents for Detection of Coordination Failure", the Seventh International Symposium on Distributed Autonomous Robotic Systems. June 2002
[35] Amy L. Murphy, Gian Pietro Picco and Gruia-Catalin Roman, "LIME: A Coordination Middleware Supporting Mobility of Hosts and Agents", Technical report. University of Rochester and Washington University in St. Louis, 2003
[36] Philipp Obreiter and Guntram Graf, "Towards Scalability in Tuple Space", ACM. Symposium on Applied Computing. 2002: 344-350
[37] A. I. T. Rowstron, A. M. Wood, "BONITA: A Set of tuple space primitives for distributed coordination", IEEE. Proceedings of the 30th Annual Hawaii International Conference on System Sciences. 1997:379-388
[38] Antony Ian Taylor Rowstron, "Bulk Primitives in Linda Run-Time Systems", PhD. Thesis. Department of Computer Science, University of York. 1996
[39] Antony Rowstron, Andrew Douglas and Alan Wood, "COPY-COLECT: A new primitive for the Linda model" Technical Report. Department of Computer Science, University of York. 1996
[40] Antony Rowstron and Stuart Wray, "A Run Time System for WCL", Springer-Verlag. Workshop on Internet Programming Languages. 1998: 78-96
[41] A. I. T Rowstron and A.M. Wood, "Solving the Linda multiple rd problem using the copy-collect primitive", Elsevier North-Holland. Science of Computer Programming. 1998: 335-358
[42] Antony Rowstron and Alan Wood, "Solving the Linda multiple rd problem", Springer-Verlag. Proceedings of the First International Conference on Coordination Languages and Models. 1996: 357-367
[43] P. Wyckoff, "T Spaces", IBM Systems Journal. 1998
[44] Vijay K. Garg, "Elements of Distributed Computing", Wiley-Interscience, 2002
[45] Java Agent Development Framework, http://jade.tilab.com

[46] Edsger W. Dikjstra, "Guarded commands, nondeterminacy and formal derivation of programs", ACM Press, 1975: 453-457
[47] Foundation for Intelligent Physical Agens, http://www.fipa.org