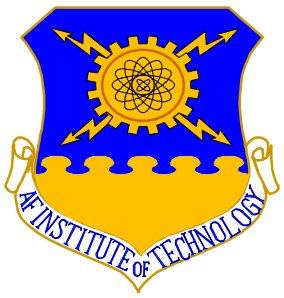
AFIT/EN-TR-00-03
TECHNICAL REPORT
July 2000



Specifying Agent Behavior as Concurrent Tasks: Defining the Behavior of Social Agents

Scott A. DeLoach

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

Abstract

Software agents are currently the subject of much research in many interrelated fields. While much of the agent community has concentrated on building exemplar agent systems, defining theories of agent behavior and inter-agent communications, there has been less emphasis on defining the techniques required to build practical agent systems. While many agent researchers refer to tasks performed by roles within a multiagent system, few really define the what they mean by tasks. We believe that the definition of tasks is critical in order to completely define what an agent within a multiagent system. Tasks not only define the types of internal processing an agent must do, but also how interactions with other agents relate to those internal processes.

In this report, we define *concurrent tasks*, which specify a single thread of control that defines a task that the agent can perform and integrates inter-agent as well as intra-agent interactions. We typically think of concurrent tasks as defining how a role decides what actions to take, not necessarily what the agent does. This is an important distinction when talking about agents since hard-coding specific behavior may not be the ideal case. Often agents incorporate the concept of plans and planning to determine what to do. In these cases, we would develop a concurrent task for determining how the planning and plan implementation occurs, but not to describe the individual plans themselves.

Table of Contents

Table of Figures

1. Introduction

Software agents are currently the subject of much research in many interrelated fields. While much of the agent community has concentrated on building exemplar agent systems, defining theories of agent behavior and inter-agent communications, there has been less emphasis on defining the techniques required to build practical agent systems.

Many agent researchers refer to tasks performed by roles within a multiagent system (Kendall 1998, Wooldridge, Jennings & Kinny 1999). However, few really define the essence of what they mean by tasks. We believe that the definition of tasks is critical in order to completely define what an agent within a multiagent system. Tasks not only define the types of internal processing an agent must do, but also how interactions with other agents relate to those internal processes. Some researchers have focused on coordination (Barber 1999, Wooldridge, Jennings & Kinny 1999) and some on internal agent reasoning (Kinny, Georgeff, & Rao 1996), few have combined the two.

2. Background

In general, our research has focused on developing the methodology, techniques, and tools for building practical agent systems (DeLoach & Wood 2000). To this end, we have developed the Multiagent Systems Engineering methodology (Wood & DeLoach 2000) that defines multiagent systems in terms of agent classes and their organization. We define their organization in terms of which agents can communicate using *conversations*. There are two basic phases in MaSE: analysis and design. The first phase, Analysis, includes three steps: capturing goals, applying use cases, and

The first step *capturing goals* takes user requirements and turns them and top-level system goals. After defining system level goals, we extract system-level use cases and define Sequence Charts for each in the *applying use cases* step. This step defines an initial set of system roles and communications paths. Using the system goals and roles identified in the use cases, we refine and extend the initial set of roles and define tasks to accomplish each goal in the *refining roles* step.

In the next phase, Design, we transform the analysis models into specific constructs that can be used to build a multiagent system. The Design phase has four steps: creating agent classes, constructing conversations, assembling agent classes, and system design. In the first step, *creating agent classes*, we define specific agent classes to fill the roles defined in the Analysis phase. Then, after determining the number and types of agent classes in the system, we can either construct conversations between those agent classes or define the internal components that comprise the agent classes. These two steps may be carried out in parallel in the *constructing conversations* and *assembling agent classes* steps. Once we have completely defined the system structure, we define how the system is to be deployed. During this step the number of individual agents, their locations are defined, and other system specific items are defined.

The most interesting, and most difficult, part of using the MaSE methodology is transforming the roles into agent classes and defining the conversations and internal agent behaviors. To help us accomplish this task, we need to be able to define high-level role tasks that can be transformed into specific agent functionality. This functionality helps us define the internal components of agents as well as the details of the conversations in which the agents participate. In this paper, we proposed use a Concurrent Task model to help define these high-level role and agent class behaviors. We define these concurrent tasks as a finite state automaton that specifies messages between agent classes and internal agent components as well is the activities or functions of the agent. Using concurrent tasks, higher level, complex interaction protocols that require the coordination with multiple agents can be defined. We have also shown that we can actually verify correct operation of such interaction protocols based on Concurrent Tasks (Lacey & DeLoach 2000).

In this paper, we cover the syntax and semantics of concurrent task models and describe different types of tasks that may be developed. We also present a few examples. Although goal of using Concurrent Task models is to help define the internal agent behavior and conversations, we do not define a methodology to help us do that at this time.

3. Concurrent Tasks

We define agent behavior to be defined by a number of *concurrent tasks*. These tasks specify a single thread of control that defines a task that the agent can perform and integrates inter-agent as well as intra-agent interactions. We typically think of concurrent tasks as defining how a role decides what actions to take, not necessarily what the agent does. This is an important distinction when talking about agents since hard-coding specific behavior may not be the ideal case. Often agents incorporate the concept of plans and planning to determine what to do. In these cases, we would develop a concurrent task for determining how the planning and plan implementation occurs, but not to describe the individual plans themselves. Concurrent tasks are specified graphically using a finite state automaton as shown in Figure 1. All tasks are assumed to start execution upon startup of the agent and continue until the agent terminates or an end state is reached.

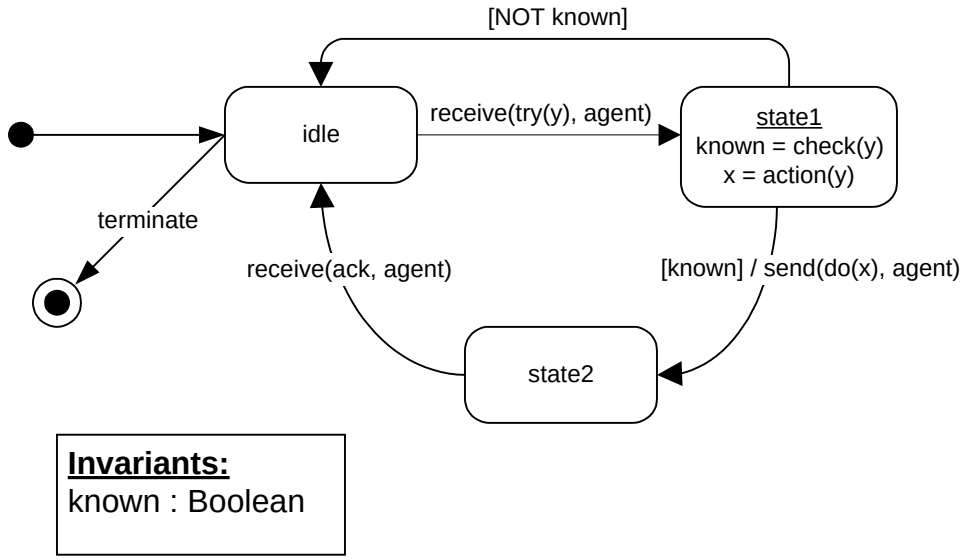


Figure 1. Concurrent Task

We model agent behavior as consisting of *n* concurrent tasks. Each of these tasks executes in parallel to define the behavior of the agent. Activities are used to specify actual functions carried out by the agent and are performed inside the task states. While these tasks execute concurrently and carry out high-level behavior, they can be coordinated using internal events. Internal events are passed from one task to another and are specified on the transitions between states. To communicate with other agents, external

messages can be sent and received. These are specified as internal *send* and *receive* events. These events send and retrieve messages from the message-handling component of the agent, which is assumed to exist. Besides communication with other agents, tasks can interact with the environment via reading percepts or performing operations that affect the environment. This interaction is typically captured in functions defined in the states. By including reasoning within tasks, agents are not "hardwired" or purely reflexive. They can plan, search, or use knowledge-based reasoning to decide on appropriate actions

3.1 Syntax

The syntax of a concurrent task has two components: states and transitions. As defined above, the states and transitions are similar to the states and transitions of most other finite automata models. States encompass the processing that goes on internal to the agent. This processing is denoted by a sequence of activities specified in a functional form. Transitions denote communication between agents or between tasks. Of each of these components are specified completely below.

3.1.1 Transitions

A transition consists of five items. First, each transition has a single source state and a single destination state. A transition also has a *trigger*, which is either a message from an external agent (a receive event) or an internal event from another task. For example, in Figure 1, the transition between the *idle* state and *state1* consists of a trigger, `receive(try(y), agent)`. In this case, an event *receive* is transmitted to the task. The parameters of the trigger event are the message and the agent from which the message was sent.

Each transition may also have a *guard*, which is a Boolean condition that must be true before the transition may occur. In Figure 1 we can see that the transition from *state1* to the *idle* state is simply `[NOT known]`, which has only a single guard condition. The variable `known` is Boolean condition computed in *state1* based on the value of *y*. The other transition leaving *state1*, `[known]/send(do(x),agent)`, also uses `known` for a guard condition.

A transition may also have *transmissions*, which are either a message to an external agent (a send event) or an event sent to another task. The send event in the transition from *state1* to *state2*,

`[known]/send(do(x),agent),` denotes a transmission. In this case, this is a message to *agent* to *do(x)*. Multiple transmissions may be separated with a semicolon (;). The semicolon not only separates transmissions, but also imparts a sequential ordering to their actual transmission. While *null transitions* are not allowed (with the exception of the transition from the *start* state), transitions may be made with just a guard condition and no triggers or transmissions. Transitions that occur with no communications or guard conditions separating two states should be combined into a single state or the state being transitioned to must be the idle state. The syntax for a transition is shown below.

```
Trigger [guard] / transmission(s)
```

Generally, events specified in a trigger or transmissions are assumed to come from another task within the same agent. This allows an agent to coordinate its tasks. For instance the *terminate* event in Figure 1 must come from another task being executed within the same agent. However, two special events are used to indicate that a message is actually sent from the current agent to another agent: send and receive. The *send* event is used to send a message to an external agent and has the following syntax.

```
send(message, agent)
```

The message is defined as a *performative*, which describes the intent of the message, along with a set of parameters that are the *content* of the message. Again, the transition

```
[known]/send(do(x),agent)
```

in Figure 1 involves sending a *do* message to a particular agent. In this case, *do* is the performative while *y* is the content of the message. The format of a message is shown below where p1 … pn denotes *n* possible parameters.

```
performative(p1 … pn)
```

It is possible for a message to contain only a performative with no contents. In this case, only the performative and agent identifier is required. For example, to send a simple acknowledgement message, the message would have the syntax `send(acknowledge, agent)`, where acknowledge is the performative and the second parameter is interpreted as the agent to whom the message is sent.

It is also possible to send a message to a group of agents via *multicasting*. This is a common capability supported by many inter-agent communication frameworks and can be simulated by sending multiple messages, if not supported directly by the communication framework. Instead of specifying a single agent to send a message to, a group name is specified by enclosing the group name with braces (e.g., <group-name>). The syntax for a multicast message is shown below.

```
send(message, <group-name>)
```

The receive event has a similar syntax to a send event as shown below.

```
receive(message, agent)
```

In this case, a receive event is only valid as a trigger and follows the same syntax rules as the send event. In the case of send and receive events, there must always be at least one parameter denoting the agent to or from whom the message was sent or received. An example of a receive event that gets a message `ack` from another agent is shown in Figure 1. In this case, the transition is from *state2* back to the *idle* state. The message `ack` is received with no parameters (i.e., no message contents). Obviously, the `ack` is simply an acknowledgement that the previous `do(x)` message was received by the other agent.

3.1.2 States

States may contain *activities*, which can be used to represent internal reasoning, reading a percept from sensors, or performing actions via effectors. Multiple activities may be included in a single state and are performed in sequence. Once in a state, the task remains in that state until activity processing is complete and a transition out of the state becomes enabled. Activities are defined in the form of functions. Each function may return up to one result and may have a number of input parameters. For example, in *state1* in Figure 1, there are two activities performed in sequence: `check(y)` and `action(y)`. The `check(y)` activity returns a value, which is stored in the variable `known`, while the `action(y)` activity returns a value stored in variable `x`. Once processing starts in *state1*, both activities must complete before either of the transitions out of *state1* are enable. The syntax of an activity statement is shown below.

$$\texttt{result = activity-name(parameter}_1\texttt{, parameter}_2\texttt{, … parameter}_n\texttt{)}$$

Multiple results may be returned from activity using tuple notation such as

```
<x, y> = divide(a, b)
```

Once a multiple value has been returned in a tuple, the individual variables that make up the tuple can be referred to as independent variables.

States with an asterisk (*) following their name denote optional states. *Optional states* are states that do not have to be entered into for the task to be performed in a valid manner. Specification of optional states allow for automated verification of task protocols. They do not affect the semantics of the task.

3.1.3 Task Invariants

Each task may have a set of invariants that must hold during then entire life of the task. These invariants may either specify variable datatypes in the form of

```
variable : datatype
```

or specify a general condition using traditional axiomatic expressions. Invariants are annotated in a box in the lower left of the task model as shown in Figure 1. We do not restrict the actual syntax of the invariant axioms.

3.2 Semantics

Semantics of concurrent tasks are based on standard finite state automata such as Statecharts. However, because a single agent is defined by a number of concurrent task models, the state of an agent is actually defined by the set of current states in each of the agent's active concurrent tasks. Because activities occur in states, agents are typically in a state for a finite amount of time. On the other hand, we assume that transitions between states occur instantaneously. This allows the state of the agent to be precisely determined at any point in time.

The variables used in activity definitions in states and in message and event definitions on transitions are assumed to be globally visible within the task, but not outside of the task. This does not mean that the variables used in a task definition are visible inside the activities defined in the task. The only way to transfer information from a task to an activity is by parameters passed to the activity or the result returned from the activity. The only way to pass information between task models is to explicitly pass information as parameters in an event call. In Figure 1, the variable *y* is set when the task receives the *try* message

from another agent. This value of *y* is now globally available within the task definition. For instance, when the `check(y)` and `action(y)` activities are performed in *state1*, the value of *y* received from the *try* message is passed as a parameter. Since activities are defined as functions, the parameter *y* cannot be changed inside the `check` activity and thus has the same value when passed to `check` and `action`. Likewise the result of `action` gets stored in the variable `x` which is sent out in the `do(x)` message on the transition from *state1* to *state2*.

We also assume that all messages sent between agents and events sent between tasks are queued. This allows us to ensure that all messages are received even if the agent or task is not in the appropriate state to handle the message or event immediately. We also allow the task to search the queue for messages that can be handled in the current state, although messages and events of the same type are handled in the order they are received.

3.2.1 States

Each task is in exactly one state at any point in time. That means that transitions between states are instantaneous while states take time. Generally, states are used for two purposes: waiting for an event or performing internal processing. If there are no activities in a particular state or all activities have been completed and no transitions have been enabled, then the task is idle waiting on a transition to be enabled.

All activities occur in a state and are executed sequentially. When a task enters a state, the first activity is automatically executed. Upon completion of the first activity, subsequent activities are executed one by one. No transitions out of the state are enabled until all activities have been completed. For example, the execution of *state1* in Figure 1 is equivalent to the following code fragment from a traditional imperative programming language such as C or Java (assuming the *send* and change of state happens instantaneously):

```
known = check(y);
x = action(y);
if (known)
    nextState = idle;
if (!known)
    send(do(x), agent) and nextState = state2;
```

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

```
Boolean = move(location)
```

The next pair of predefined activities deals with the ability of an agent to determine the passage of time. To reason about time, tasks provide built in timers. An agent can define a timer using the setTimer activity. The *setTimer* activity takes a time as input and returns a timer that will timeout in exactly the time specified. Thus, the time specified is the offset from the current time. The setTimer activity returns a timer that can be tested by the agent to see if it has timed out using the *timeout* activity. The timeout activity returns a Boolean value that is true if the timer has timed out. Using the setTimer and timeout activities, an agent can use time in carrying out its assigned responsibilities. The syntax for the setTimer and timeout functions is shown below.

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

Actually, the *timeout* activity is not generally used in a state, but on transitions as a guard condition. As shown in Figure 2(a), the *timeout(t)* activity is placed in *state1*. According to concurrent task semantics, the *timeout(t)* activity would be executed only once. If the *t* had timed out, the transition would fire, otherwise, the transition would never fire since the *timeout(t)* activity would never again be executed. In Figure 2(b), the *timeout(t)* activity is actually placed on the transition.

While use of an activity in a guard condition appears to violate the semantics requiring a transition to be instantaneous, it does not. An activity that returns either a Boolean value or an attribute of a specific data structure (such as the size of a queue) may be used in a guard condition of a transition. A Boolean activity on a transition is only tested once all activities in the state have been executed and is continually

tested until a transition is enabled. Multiple Boolean activities on transitions out of the same state are all executed before checking to see if any transitions are enabled. Thus, the semantics of Figure 2(b) is as follows. Upon entering *state1*, the *setTimer(time)* activity is executed. Once it completes, the state goes to an idle states where it continually checks to see if *timeout(t)* is true. When *timeout(t)* becomes true, then the transition fires and we move to *state2*.

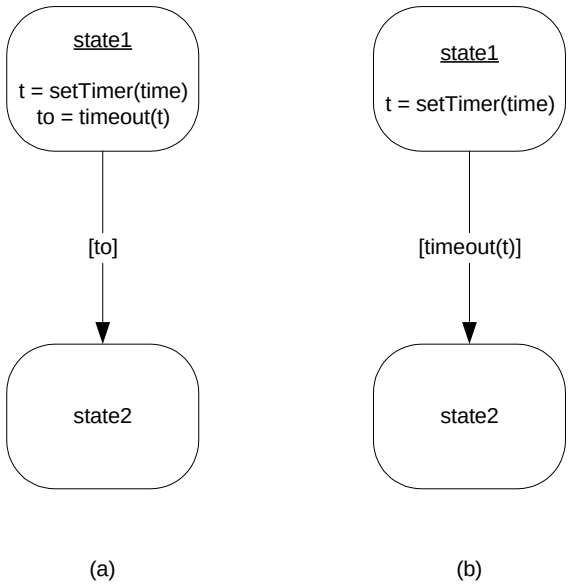


Figure 2. Using timeout(t)

3.2.2 Transitions

As stated above, transitions occur instantaneously and move tasks from one state to another (or possibly the same state). A transition is enabled if all the following conditions hold.

1. The transition's source state is the current state of the task.
2. The transition's trigger event has been generated.
3. The transition's guard condition evaluates to true.
4. All activities in the transition's source state have been completed.

If a transition does not have a trigger or a guard, both conditions are assumed to hold. Once a transition is enabled, it is executed and execution occurs instantaneously. This means that events and messages are sent instantaneously and the current task state becomes the destination state of the transition.

In the case of a transition that sends two events or messages, even though the transition occurs instantaneously, the events and messages are ordered sequentially according to the ordering on the diagram.

If multiple transitions are enabled simultaneously, the following priority scheme is used.

1. Received events. Any transitions whose trigger contains an event received from other tasks are processed first. If multiple transitions with internal events are enabled, then they are processed in the order the events were received. Since events are transmitted instantaneously, there must be a linear ordering to the events.
2. Send events. Any transitions whose transmissions contain an event to be sent to another task are processed next. If multiple transitions with send events are enabled, they must be ordered based on other criteria such as arrival time of trigger, etc. Since guard conditions must be mutually exclusive (see Guard Condition order below) and events and message triggers are ordered based on arrival times, there is always an ordering to the sending of events.
3. Received messages. Any transitions whose trigger contains a message received from another agent are processed next. If multiple transitions with received messages are enabled, then they are processed in the order the messages were received. Since messages, like events, are transmitted instantaneously, there must be a linear ordering to the message receipts.
4. Send messages. Any transitions whose transmissions contain a message to be sent to another agent are processed next. If multiple transitions with send messages are enabled, they must be ordered based on other criteria such as arrival time of trigger, etc. Since guard conditions must be mutually exclusive (see Guard Condition order below) and events and message triggers are ordered based on arrival times, there is always an ordering to the sending of messages.
5. Guard conditions. Transitions that only have guard conditions are the next priority. The rule for guard conditions is that states containing multiple guard condition transitions must be mutually exclusive. Since received events and messages are an implied part of a transition's

guard condition, these may be conjuncted with the guard condition when determining exclusiveness.

6. Null transitions. The only allowable null transition is a transition from the *start* state. If there is a null transition from the start state, there may be no other transitions from the start state.

Thus in Figure 3, if we are in the *wait* state, it is possible to have two transitions enabled at the same time: `[timeout(t)]` and `receive(bid(x),ag)`. In this case, it clear from our priority rules that the *receive* message is handled before the *timeout*. It is also possible that while handling the *receive* message (i.e., we are in *state1* in the *recordBid* activity) we get another *receive* message. Thus when we transition back to the *wait* state and send an *acknowledge* message, we have the same dilemma. Since the *timeout* is in a guard condition and does not have a timestamp associated with it the way a message would, we handle the message first and go through the same process.

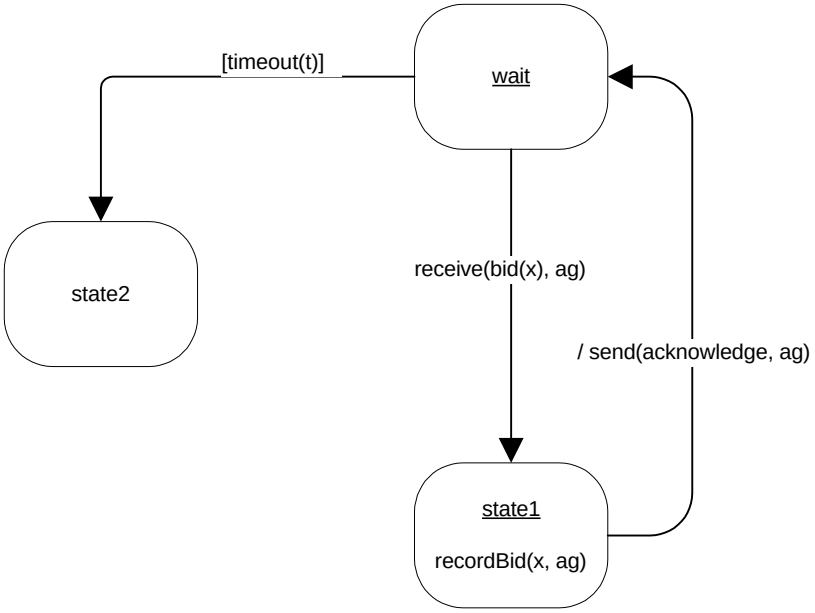


Figure 3. Transition Priority Example

If instead of a guard condition on the transition from the *wait* state to *state2*, we had another receive message, we would get different results. For example, in Figure 4, assume upon entering the *wait* state we have two incoming messages waiting for us: *endBidding* and *bid(x)*. According to our priority rules, we take first message to come in that is enabled. In this case, assume *bid(x)* came in before *endBidding*

(possibly before we even entered the *wait* state). We would read the *bid(x)* message and begin the activity in *state1*. Now if we received a second *bid(x)* message while in *state1*, that message is cued up after the *endBidding* message. Therefore, after the recordBid activity is completed and the task transitions back to the *wait* state, the task processes the *endBidding* message first since it has priority over the second *bid(x)* message.

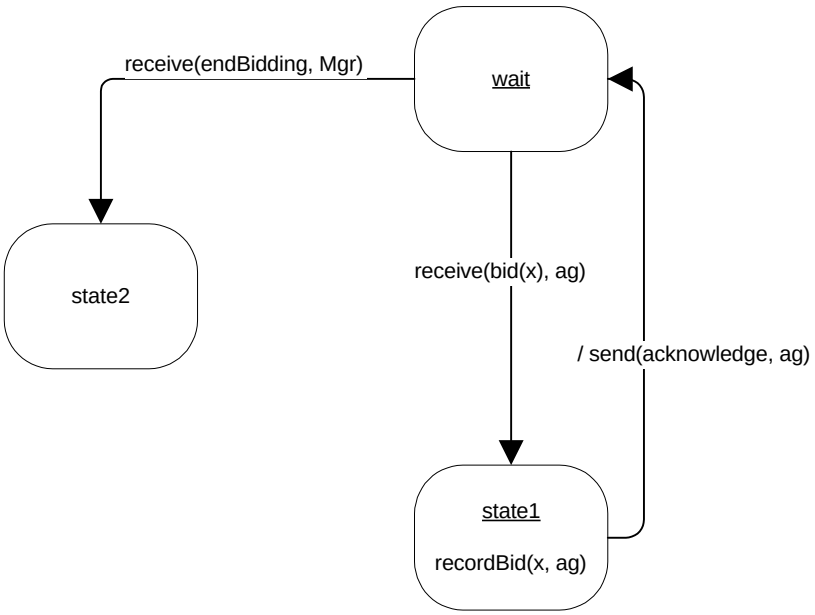


Figure 4. Message Transition Priority

3.2.3 Task Invariants

The semantics of task invariants are straightforward. If the invariant is a variable type definition, the invariant is taken as definitional. If the invariant is axiomatic, then the semantics are that the axiom must be true at all times. In essence, the invariant becomes a post-condition of every activity in the task. For instance, in Figure 5, the invariant `size(list) >= 0` is a post-condition for the `remove` and `add` activities. Using standard convention for add and removing from a list, the invariant could affect the `remove` activity.

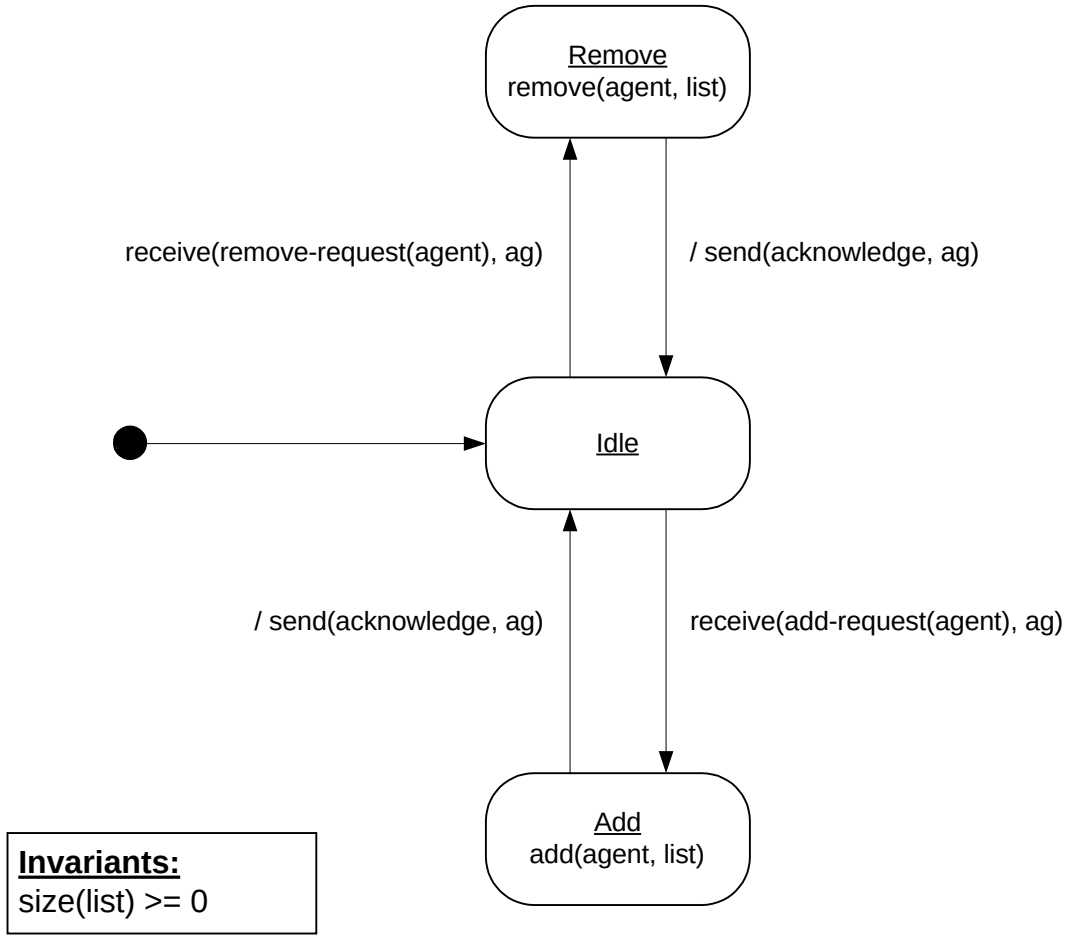


Figure 5. Concurrent Task with Invariants

4. Task Types

As stated initially, the goal of concurrent tasks is to define the behavior of agents, tying the internal reasoning processes of the agent to its interaction with other internal processes as well as externally with other agents. We can categorize these tasks into three types: reactive, proactive, or heterogeneous. A *reactive task* has at least one idle state where it waits for a request – from either another task or agent – before actually starting any processing. A reactive task always starts in one of its idle states. That is, the *start state* has a null transition to an *idle* state. *Proactive tasks* do not have idle states. They are continually generating requests for other agents or tasks.

A heterogeneous task, as the name suggests, is a combination of reactive and proactive tasks. A *heterogeneous task* has idle states, but it does not start in an idle state. It generates at least one request for another agent or task before entering an idle state.

Based on these task definitions, we can categorize agent whose behavior is defined by tasks as either proactive or reactive. A *proactive agent* is an agent with at least one proactive or heterogeneous task while a *reactive agent* is an agent with all reactive tasks.

5. Examples

5.1 Information Registration

Assume we want to define the behavior of an information registration role and have come up with the following goals that this role is be responsible for:

1. Keep list of all registered information sources
    a. Allow information sources to register and de-register
2. Keep a list of all registered requests for information sources
    a. Allow agents to register and de-register on-going requests for information
3. Inform requestors when information is available
    a. One-time requests
    b. Registered requests

The registration role needs to allow information sources to register with it. To register, the information source has to supply information on how it can be reached (address and port number) as well as the type of information it has available. (To simplify this example, we assume that every agent within the system is using the same ontology.) To define the behavior of this role, we develop a concurrent task for each of the three main goals. We call these tasks the Information Source Registration task (goal 1), the Request Registration Task (goal 2), and the Inform Requestors task (Goal 3).

5.1.1 Information Source Registration Task

The Information Source Registration task is shown in Figure 6. This task is relatively simple and made simpler by the fact that we have not added any error checking. Two basic things can happen in this task. First, an external agent can send a `register-source` message to the task with its datatype and source information (name, address, port, etc.). After receiving the message, the role calls its `addSource` activity and returns an `acknowledge` message and sends a `new-info` event to the Inform Requestors task to check if there is anyone who needs to be informed of this new source of information. Having handled this request, the task waits in the idle state.

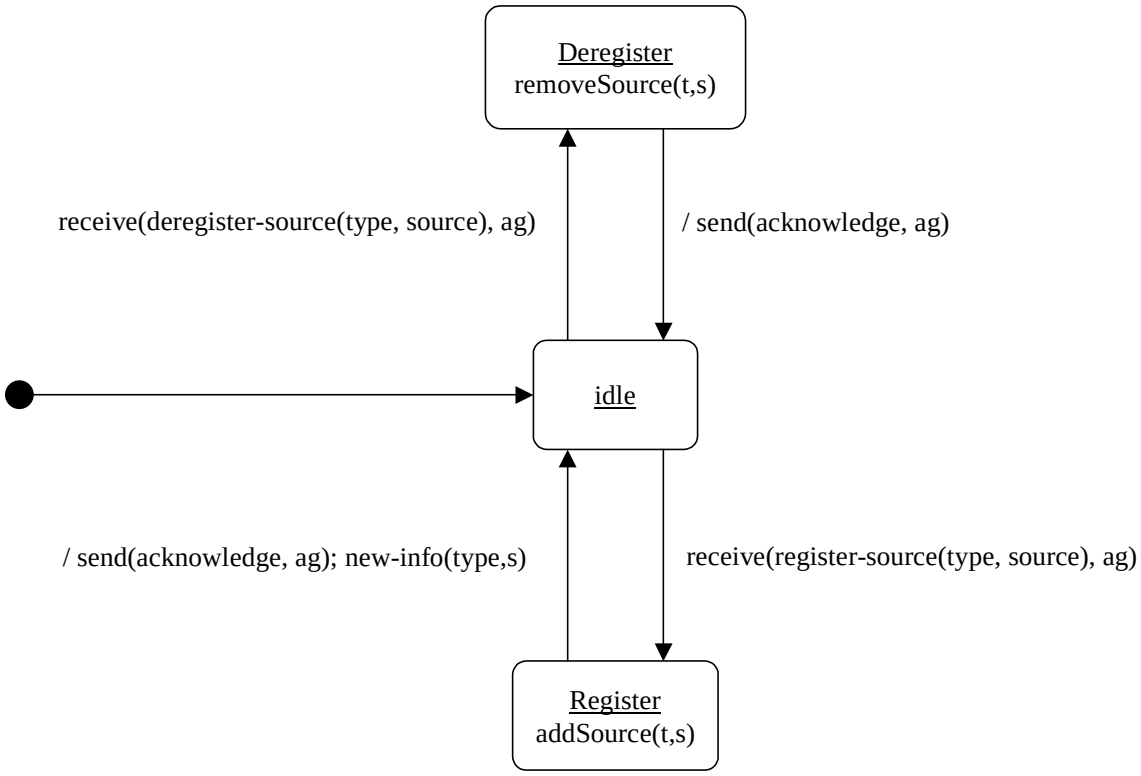


Figure 6. Information Source Registration Task

The second request the task can handle is from an external agent wanting to de-register the fact that it can provide a certain type of information. This happens when a `deregister-source` message is received with appropriate type and source information. After the message is received, the task invokes its `removeSource` activity, which actually removes the information source from the list. After the activity is complete, the task sends an `acknowledge` message to the agent verifying that the request was granted.

5.1.2 Request Registration Task

The Request Registration task is similar to the Information Source Registration task as shown in Figure 7. In this case, the task receives requests to register and de-register standing requests for information sources containing specific types of data. This is accomplished by an external agent sending either a `register-request` or `deregister-request` message. The message cause internal list management activities to be performed with an `acknowledge` message to be returned upon completion. When a new agent registers, an internal `new-request` event is generated and passed to the Inform Requestors task, which searches the list of information sources for any existing information that can be sent immediately.

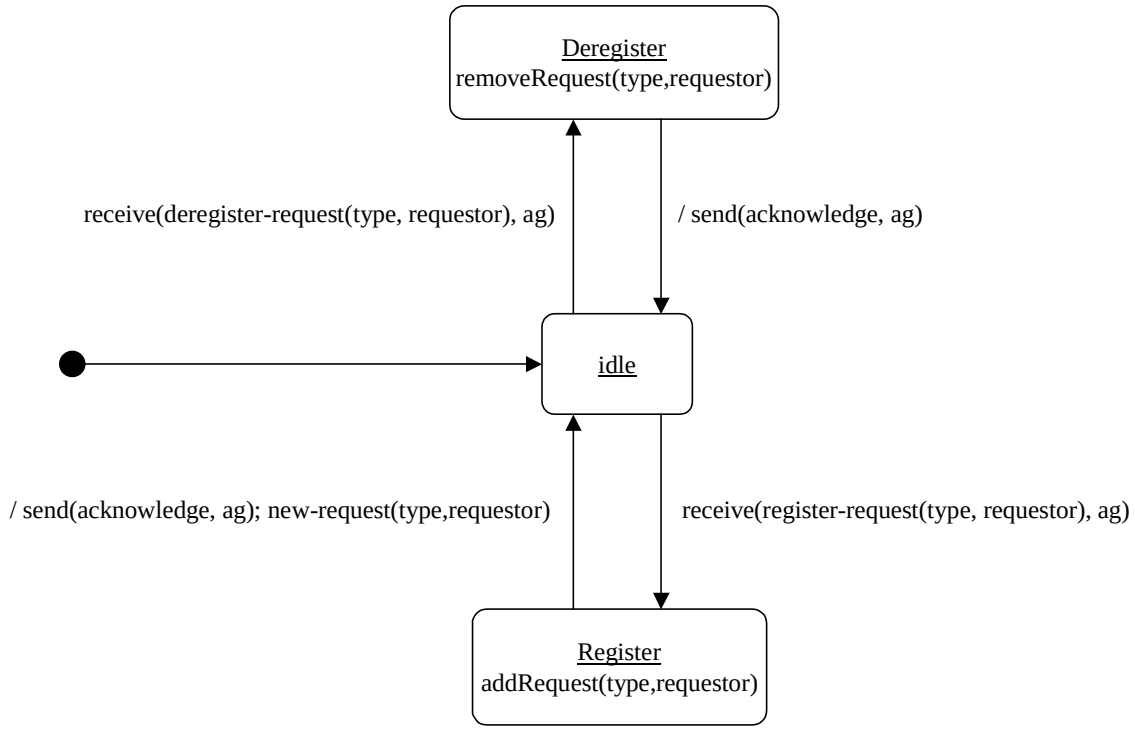


Figure 7. Request Registration Task

5.1.3 Inform Requestors of New Source Task

The Inform Requestors of New Source (Figure 8) task is more complicated than the first two tasks. First, it is set in motion by an event from the Information Source Registration tasks: `new-info`. Once the task is invoked, it searches the list of agents who have requested information of the type provided by this new information source and informs them of the new source.

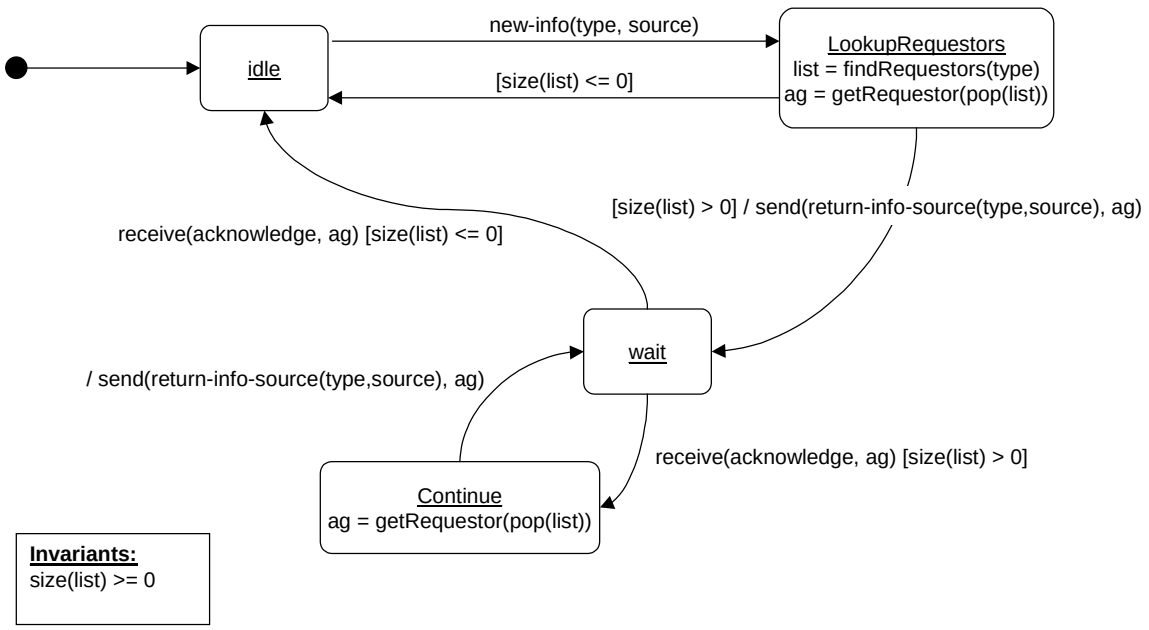


Figure 8. Inform Requestors of New Source Task

There are two inputs to the task: the type of the new information source and the address of the source. Upon invocation, the `LookupRequestors` state is entered where the `findRequestors`, `size`, and `getRequestor` activities are carried out. If no agents have requested the type of information from this new source (`x<=0`), the task is terminated.

Assuming agents interested in the new information source are found (`x>0`), the task goes about the business of sending the new source to each requesting agent, one at a time. In this case, the task transitions out of the `LookupRequestors` state and sends a `return-info-source` message to the requesting agent, passing both the type and address information of the new information source, and enters the `wait` state. Upon entering the `wait` state, the task basically enters a loop where it waits for an `acknowledge` message from the requesting agent and then, if there are more agents on the list, gets the next requesting agent from the list and sends it the new information source. This is shown in the `wait – continue – wait` loop. In the continue state, the list size counter, x, is decremented, the next requestor is popped off the list. If there are no more agents on the list (`x=0`), the last `acknowledge` message sends the task to the `idle` state.

5.1.4 Inform Requestors Task

The Inform Requestors task is very similar to the last task. As shown in Figure 9, there are two differences. The first difference is that the task is initiated by an event from the Request Registration task, `new-request`, or by a one-time request from an agent via a `request` message. The second difference is that it searches the list of registered information sources to find sources that match the request. Once invoked, the task searches the list of registered information sources for those who can provide the type of information requested and informs the requestor.

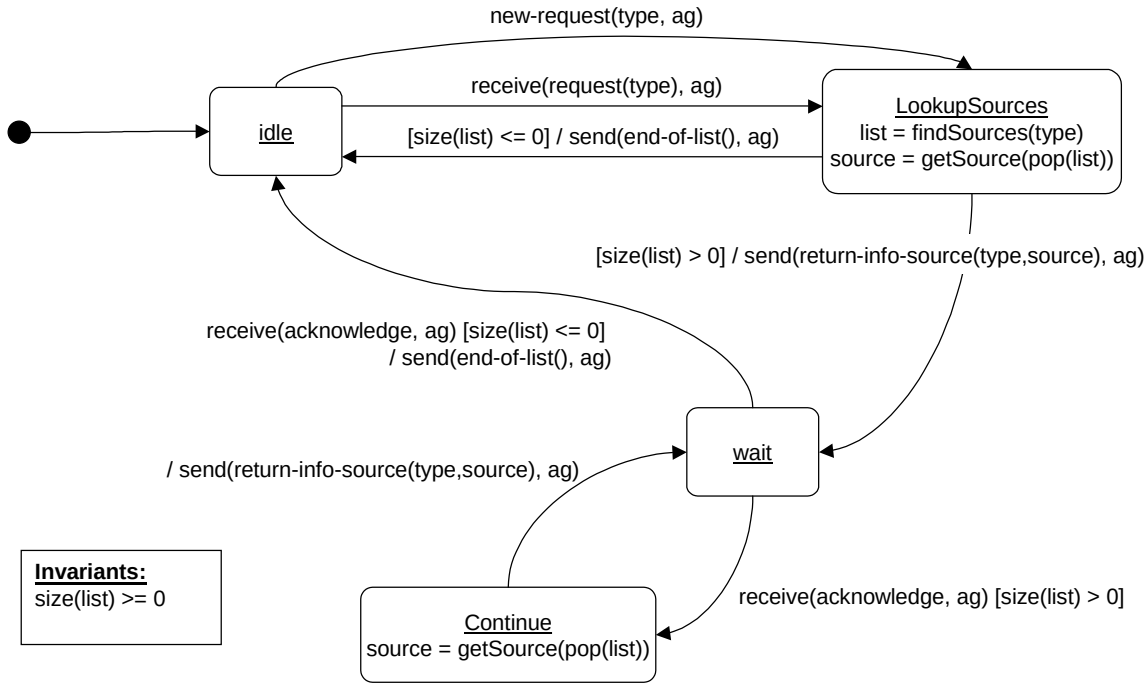


Figure 9. Inform Requestors Task

Regardless of the method of invocation, there are two inputs to the task: the type of information being requested and the agent who needs the information. Upon invocation, the `LookupSources` state is entered where the `findSources`, `size`, and `getSource` activities are carried out. If there are no possible information sources found `(x<=0)`, the task is terminated and the `end-of-list` message is returned to the requesting agent.

Assuming there are information sources of interest found `(x>0)`, the task goes about the business of sending each information source, one at a time. In this case, the task transitions out of the

`LookupSources` state and sends a `return-info-source` message to the requesting agent, passing both the type and address information of the source, and enters the `wait` state. Upon entering the `wait` state, the task basically enters a loop where it waits for an `acknowledge` message from the requesting agent and then gets the next information source from the list and returns it to the agent. This is shown in the `wait – continue – wait` loop. In the continue state, the list size counter, `x`, is decremented, the next information source is removed from the list. When the all information sources on the list have been sent `(x=0)`, the task transitions to the idle state when it receives the last `acknowledge` message.

6. Related Work

Much of our work on Concurrent Task Models stem from work originally done by Harel on *Statecharts* (Harel 1998), which was the basis for *state diagrams* in Rumbaugh's Object Modeling Technique (OMT) (Rumbaugh 1991) and the Unified Modeling Language (UML 1997). The original goal of Statecharts was to model reactive systems, which Harel define as systems that continually interact with their environment using input and output whose timing may be unpredictable. Statecharts are a very large, complex language supporting concurrency, conditional transitions, and event input and output. The basic difference between Concurrent Tasks and Statecharts is the ability to define parameterized events and activities inside states in Concurrent Tasks. Statecharts have no way to define internal processing and how information passed in events relates to those activities. The only significant advantage of Statecharts is the ability to model substates for complex models. We specifically chose to leave substates out of Concurrent Tasks to simplify the language. We assume typical Concurrent Tasks are fairly simple with much of the complex processing performed within activities.

Concurrent Task Models are really much more akin to OMT and UML state diagrams, which are used to model object class or method behavior. However, we have provided an precise semantics that allows for a specific implementation. These semantics differ from OMT or UML in that we assume all tasks run concurrently and that activities within states are non-interruptible. While our simplifications may reduce

the ease of use of Concurrent Tasks, we argue that they have the same general expressiveness and a specific semantics that makes automatic agent synthesis possible.

Concurrent tasks are also similar to Parunak and Singh's use of Dooley graphs for agent coordination (Singh 1998) (Parunak 1996), among others. The strength of Concurrent Tasks in relation to these efforts is its ability to tie agent coordination issues to internal agent reasoning and computation.

7. Conclusions

Concurrent tasks allow us to capture the behavior of agents in a uniform fashion. Whether or not the tasks are internal to a single agent or carried out between multiple agents makes little different other than a minor syntactical change. This allows us to define multiple concurrent tasks and then determine to which agent they should belong.

References

1. K. S. Barber, T. H. Liu, and D. C. Han. Agent-Oriented Design. In: Multi-Agent System Engineering 9th European Workshop on Modelling Autonomous Agent in a Multi-Agent World (MAAMAW'99), Lecture Notes in Artificial Intelligence no. 1647, Springer-Verlag, 1999.
2. Mihai Barbuceanu and Mark S. Fox. COOL: A Language for Describing Coordination in Multi Agent Systems. In Proceedings of the International Conference on Multi-Agent Systems, ICMAS-95, San Francisco, CA, 1995.
3. Scott A. DeLoach and Mark Wood. Developing Multiagent Systems with agentTool. The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL-2000). Boston, MA, July 7-9, 2000.
4. David Harel, and Politi, Michael. Modeling Reactive System with Statecharts : the Statemate Approach. McGraw-Hill, New York, New York. 1998.
5. Elizabeth A. Kendall, Agent Roles and Role Models: New Abstractions for Intelligent Agent System Analysis and Design. International Workshop on Intelligent Agents in Information and Process Management, 1998.
6. David Kinny, Michael Georgeff, and Anand Rao. A Methodology and Modelling Technique for Systems of BDI Agents. Technical Note 58. Australian Artificial Intelligence Institute. January 1996.
7. Timothy H. Lacey and Scott A. DeLoach, Verification of Agent Behavioral Models. The 2000 International Conference on Artificial Intelligence (IC-AI'2000). Las Vegas, Nevada, June 2000.
8. H. V. D. Parunak. Visualizing agent conversations: Using Enhanced Dooley graphs for agent design and analysis. In Proceedings of the 2nd International Conference on Multiagent Systems, pages 275-282. AAAI Press, 1996.

9. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, New Jersey. 1991.

10. Munindar P. Singh, Developing Formal Specification to Coordinate Heterogeneous Autonomous Agents. Proceedings of the *International Conference on Multiagent Systems (ICMAS)*, July 1998, pages 261-268..

11. UML. Unified Modeling Language Notation Guide. Rational. 1997.

12. Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems. Vol 3 (3) 2000.

13. Wood, Mark & Scott A. DeLoach, An Overview of the Multiagent Systems Engineering Methodology. The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), Limerick Ireland, June 2000.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 2000	Technical Report, June 1999 - July 2000

4. TITLE AND SUBTITLE
Specifying Agent Behavior as Concurrent Tasks: Defining the Behavior of Social Agents

5. FUNDING NUMBERS
99NM097 (AFOSR)

HE-WSU-99-09 (DAGSI)

6. AUTHOR(S)
Scott A. DeLoach

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT/ENG
2950 P. Street
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/EN-TR-00-03

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM, 801 N Randolph St. Arlington, VA 22203-1977

Dayton Area Graduate Studies Institute
3171 Research Boulevard, Suite 109 Kettering OH 45420

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Software agents are currently the subject of much research in many interrelated fields. While much of the agent community has concentrated on building exemplar agent systems, defining theories of agent behavior and inter-agent communications, there has been less emphasis on defining the techniques required to build practical agent systems. While many agent researchers refer to tasks performed by roles within a multiagent system, few really define the what they mean by tasks. We believe that the definition of tasks is critical in order to completely define what an agent within a multiagent system. Tasks not only define the types of internal processing an agent must do, but also how interactions with other agents relate to those internal processes. In this report, we define concurrent tasks, which specify a single thread of control that defines a task that the agent can perform and integrates inter-agent as well as intra-agent interactions. We typically think of concurrent tasks as defining how a role decides what actions to take, not necessarily what the agent does. This is an important distinction when talking about agents since hard-coding specific behavior may not be the ideal case. Often agents incorporate the concept of plans and planning to determine what to do. In these cases, we would develop a concurrent task for determining how the planning and plan implementation occurs, but not to describe the individual plans themselves.

14. SUBJECT TERMS
Multiagent systems, agent-oriented software engineering, distributed systems, software engineering

15. NUMBER OF PAGES
27

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL