AgenTalk: Describing Multiagent Coordination Protocols with Inheritance

Kazuhiro Kuwabara† Toru Ishida‡ Nobuyasu Osato†

†NTT Communication Science Laboratories
2-2 Hikaridai, Seika-cho, Soraku-gun
Kyoto 619-02 Japan
e-mail: {kuwabara, osato}@cslab.kecl.ntt.jp

‡Department of Information Science
Kyoto University
Sakyo-ku, Kyoto-shi, Kyoto 606-01 Japan
e-mail: ishida@kuis.kyoto-u.ac.jp

Abstract

This paper presents the basic concepts behind AgenTalk, a language used for describing coordination protocols in multiagent systems. Many coordination protocols such as the contract net protocol have been proposed, and many application specific coordination protocols are expected to be required as soon as building of more software agents begins. Thus, a language for defining and implementing such coordination protocols plays a crucial role in the development of multiagent systems. By incorporating an inheritance mechanism, AgenTalk allows coordination protocols to be incrementally defined and easily customized to suit application domains. Its capability is demonstrated by describing the contract net and multistage negotiation protocols in AgenTalk using its inheritance mechanism.

1 Introduction

In multiagent systems, achieving coordination among autonomous agents is a major problem. In general, agents coordinate their actions by exchanging messages. We view a sequence of these interactions among agents as a *coordination process*, and we call a high-level protocol used in a coordination process a *coordination protocol*.

In the field of distributed artificial intelligence, several protocols for achieving coordination have been proposed: the contract net protocol for task allocation [1], the multistage negotiation protocol for resource allocation under global constraints [2], the unified negotiation protocol for exchanging tasks between agents under various situations [3], and the hierarchical protocol based on the concept of a behavior hierarchy [4], just to name a few. These protocols have also been further customized to suit various application domains. For example, the DSP (Distributed Scheduling Protocol) for the Enterprise system [5] is an application of the contract net protocol for task allocation among workstations connected by a local area network. Furthermore, as research on software agents proliferates [6], demands for customized protocols will increase.

On the other hand, a coordination mechanism among heterogeneous knowledge bases is receiving increasing attention (e.g., [7, 8]). KQML is a language and protocol for exchanging information and knowledge among software agents [9]. It defines message classes as performatives, and their semantics are described based on the speech act theory [10, 11].

However, only defining a message class is not enough to describe coordination protocols. Based on KQML messages, a language called COOL has been proposed for describing a coordination protocol [12]. COOL describes the protocol based on state transition rules.

In addition, another language called CooL has been proposed [13]. CooL describes a coordination protocol based on a multi-agent plan, which consists of procedures agents to follow and communication primitives.

However, these languages do not support defining new protocols based on the existing protocols or customizing existing protocols to suit various application domains. In contrast, AgenTalk introduces an inheritance mechanism into the protocol description. Using this inheritance mechanism, protocol designers can incrementally design new protocols based on existing protocols, and application designers can implement customized protocols to suit each application based on existing domain independent protocols.

In this paper, following the presentation of AgenTalk design policies, its functions are described. The contract net protocol and the multistage negotiation protocol, which can be viewed as a generalization of the contract net protocol, are then described to demonstrate the capabilities of AgenTalk.

1082-3409/95 $04.00

2 Design Policies of AgenTalk

AgenTalk is not meant to be a formal specification language, but a programming language capable of implementing protocols and agents that behave according to the protocols. In addition, AgenTalk is mainly concerned with the interactions among agents; it is not concerned with the detailed implementation of an agent. Therefore, various agent models such as a blackboard or a production system can be used to implement an agent. The design policies of AgenTalk are as follows [14].

Explicit state representation of a protocol: In our previous work, the coordination protocols for multistage negotiation [2] and market-based distributed resource allocation [15] were described by defining a message class (such as *Task Announcement* in the contract net protocol) and its message handler. However, it was difficult to define a message handler without explicitly defining states in a coordination protocol. In order to counter this, an extended finite state machine is used as a basis to describe coordination protocols in AgenTalk. The extended finite machine is a finite state machine extended to allow variables and is often used to describe communication protocols [16]. Its representation is called a *script*[1] in AgenTalk. Using this model, states of a coordination protocol can explicitly be defined, and actions of an agent can easily be defined for each state.

Incremental protocol definition: By introducing an inheritance mechanism in the definition of a script as in an object-oriented language, a protocol can be incrementally defined by inheriting a definition of an existing script. An example showing a use of the inheritance mechanism is described in a later section.

Easy protocol customization: A programming interface that specifies the portion of a state transition rule possibly needing to be customized for each application is defined. In AgenTalk, this interface is implemented as an *agent function*, which is a kind of callback function invoked from a script. Protocols can easily be customized by using this interface.

Conflict resolution between coordination processes: A single agent may be simultaneously involved in multiple coordination processes (e.g., different task allocation processes) having some interactions between them (e.g., the agent's resources are shared). By allowing multiple scripts to be executed in an agent and supporting communication between these scripts, a conflict resolution mechanism between coordination processes can be described.

[1]The term *script* is used because an agent's actions are based on a *script*, just like an actor in a play acting according to a script.

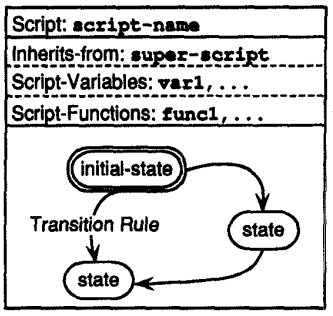

Figure 1: Script Components

3 Functions of AgenTalk

3.1 Script Definition

A script represents the state transition of an agent in a coordination process. Script $\mathcal{SC}_i$ consists of a set of states, S_i, an initial state, $so_i \in S_i$, a set of state transition rules, R_i, and a set of variables called *script variables*, V_i. It also has a set of functions called *script functions*, which is local to a script, F_i, and a script from which the definitions are inherited, I_i (Fig. 1). That is, $\mathcal{SC}_i =< S_i, so_i, R_i, V_i, F_i, I_i >$.

The components of script I_i are treated as if they are to be locally defined in $\mathcal{SC}_i$. That is, the definition of $\mathcal{SC}_i$ is a combination of I_i and locally defined components of $\mathcal{SC}_i$. Each state is associated with a name; if $\mathcal{SC}_i$ has the locally defined state with the same name, a state defined in I_i is overridden in $\mathcal{SC}_i$. In addition, each state transition rule is associated with a state. When a state is overridden in $\mathcal{SC}_i$, rules associated with the overridden state are only inherited when this is explicitly specified.

3.2 Protocol Customization

An agent can view a script as a set of special procedures. When an agent decides to follow a certain protocol, this agent invokes a corresponding script. On the other hand, from the viewpoint of a script, the agent should provide an *agent function* which is to be called from state transition rules contained in a script (Fig. 2). Thus, by providing different agent functions, the same script can cause different behaviors of agents. When an agent function is not defined in the agent that invoked the script, a script function with the same name is called instead. This allows a script function to provide the default behavior of a script. In other words, a script can be customized by redefining only its script functions, while inheriting all the other components of a script.

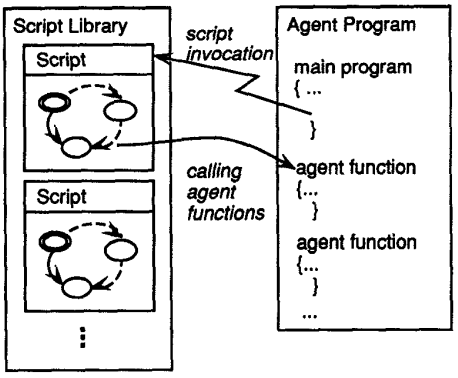

Figure 2: Script vs. Agent

3.3 Execution Model

A state transition rule consists of a condition part and an action part. The condition part describes conditions that must be satisfied for the rule to be fired. The following conditions can be stated: (1) message arrival, (2) conditions on script variables, and (3) timeout. The action part describes procedures to be executed when the rule is fired. The following procedure can be stated: (1) transition to a state, (2) invocation of a script, (3) updating script variables, and (4) calling agent functions.

When a script is invoked, a *context* is created to hold the values of the script variables for each script invocation. The state transition to the initial state then occurs, and the following loop is entered. First, a rule whose condition part is satisfied is selected from the *active* state transition rules, which are the state transition rules defined in a current state of invoked scripts. The action part of the selected rule is then executed. When another script is invoked from the action part of a state transition rule, a child context is created. The script doing the invoking is called the *parent* script, and the invoked script is called the *child* script. Note that when a new script is invoked, active state transition rules are added, not replaced. Unlike a subroutine call, the execution of the parent script is not suspended.

When there exist multiple state transition rules whose condition parts are satisfied, the rule in the most recently invoked (child) script is selected. The motivation behind this mechanism is that when there are no state transition rules to be fired in child scripts, a parent script can do some background chores.

By invoking multiple scripts from a (toplevel) script, multiple coordination processes in an agent can be represented. In addition, script variables in a parent script (context) can be accessed from state transition rules in child scripts. In other words, child scripts corresponding to coordination processes can communicate with each other via the script variables in the parent script (context). This allows conflict resolution among coordination processes in an agent to be described. Since state transition rules are fired one at a time (that is, rules in parent and child scripts are not fired concurrently), we need not consider mutual exclusion when accessing script variables in a parent script (context).

A script can be viewed as procedural knowledge for an agent to attain a certain goal. In this sense, AgenTalk has much in common with Procedural Reasoning System (PRS) [17]. AgenTalk describes the protocol among agents, whereas PRS describes the interaction between an agent and an environment.

4 Verification of AgenTalk Functions

In order to verify that AgenTalk has enough functions, the contract net protocol and the multistage negotiation protocol are described in AgenTalk.

4.1 Contract Net Protocol (Manager)

Task allocation in the contract net protocol is performed as follows. An agent (manager) with a task to be executed broadcasts a *Task Announcement* message. An agent that is willing to execute the task sends a *Bid* message. The manager agent selects an agent (contractor) to which the task is to be allocated and sends an *Award* message to it.

Let us describe the behavior of a manager that follows the contract net protocol using AgenTalk. New message classes such as *Task Announcement* are defined, and then, the script for the manager is defined. The state transition of the manager is shown in Fig. 3.

The current version of AgenTalk is implemented on top of Common Lisp; functions and macros defined in Common Lisp are readily available. In AgenTalk, `define-script` macro declares a script with its script variables, its initial state, and a script to inherit.[2] `define-script-state` macro defines each state in a script. This macro also defines a state transition rule in the state. Using these macros, the `cnet-manager` script can be written as shown in Fig. 4. This script is invoked when the task to be allocated is generated.

4.2 Extending the Contract Net Protocol

As an extension to the basic contract net protocol, a special message class called *Direct Award* is used when an agent to which a task is to be allocated is known beforehand [1]. The agent receiving

[2]Please refer to the AgenTalk reference manual [18] for details.

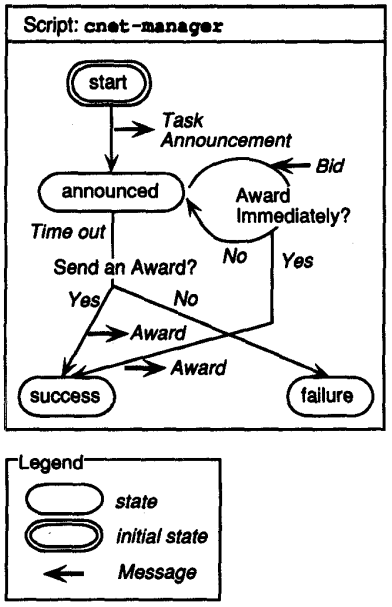

Figure 3: State Transition of a Manager in the Contract Net Protocol

the *Direct Award* message responds with either an *Acceptance* message or a *Refusal* message. The state transition of the manager incorporating this extension (`cnet-manager-with-direct-award`) is shown in Fig. 5. The figure shows that the definition of this script inherits a large part of the definition from the `cnet-manager` script.

4.3 Contract Net Protocol (Contractor)

Let us consider the behavior of a (potential) contractor when a *Task Announcement* arrives. When the contractor can execute a task contained in the *Task Announcement*, it need only send a *Bid* message. However, the contractor may want to divide the task into subtasks and allocate each subtask to a different agent; a task allocation process will then be started for each subtask generated.

This behavior can be described in AgenTalk as follows. First, a top-level script for a contractor is defined; it invokes a `cnet-contractor` script when an appropriate *Task Announcement* message arrives. The `cnet-contractor` script then checks if decomposition to subtasks is necessary. If so, the `cnet-manager` script is invoked for each subtask. When all the task allocations for subtasks are completed, a *Bid* message in response to the original *Task Announcement* message is sent.

```
(define-script cnet-manager (task)
  :initial-state start
  ;; bid-queue holds Bid messages received.
  ;; contract-id holds a current contract-id.
  :script-vars (bid-queue contract-id))

(define-script-state (start cnet-manager)
  ;; :on-entry is executed each time this state is entered.
  ;; ! denotes the agent-function call. In this example,
  ;; announce-task sends a Task Announcement
  ;;  message and returns the contract-id.
  ;; $ denotes script-variable access.
  :on-entry
  (progn (setf ($ contract-id)
               (! announce-task ($ task)))
         (goto-state 'announced)))

(define-script-state (announced cnet-manager)
  :rules
  ;; wait for a Bid message with the same contract-id.
  ((:when (msg bid :contract-id !($ contract-id))
    :do
    (if (! send-award-immediately-if-possible
           msg ($ bid-queue))
        (goto-state 'success)
        (push msg ($ bid-queue))))
   ;; check a timeout condition.
   (:when (timeout (task-expiration task))
    :do (if (! send-award-if-possible
               ($ bid-queue))
            (goto-state 'success)
            (goto-state 'failure)))))

(define-script-state (success cnet-manager)
  :on-entry (exit-script))

(define-script-state (failure cnet-manager)
  :on-entry (exit-script))
```

Figure 4: Example Definition of a Manager's Script

4.4 Multistage Negotiation

The multistage negotiation protocol can be regarded as a generalization of the contract net protocol [2]. In the contract net protocol, *Task Announcement*, *Bid*, and *Award* are basically used to allocate tasks. In contrast, the multistage negotiation protocol introduces repetitive information exchanges among agents to satisfy global constraints. The multistage negotiation protocol has the following characteristics.

- Messages such as *Task Announcement* can be canceled by a *Cancel* message. Messages for finalizing an allocation (*Commit*) and acknowledging it (*Committed*) are added in the example shown here.

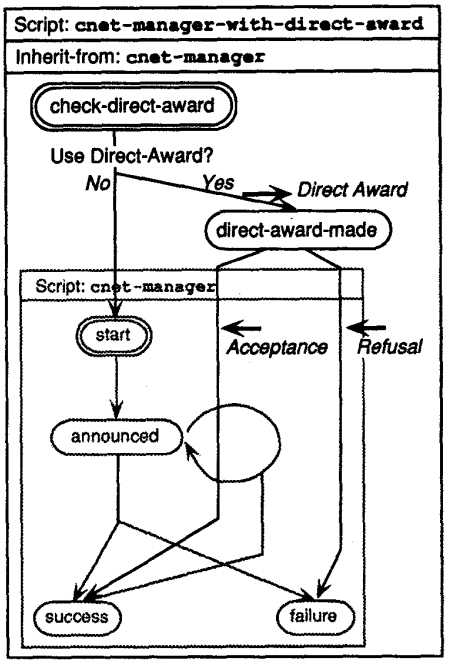

Figure 5: State Transition of a Manager with Direct Award

- Multiple negotiation processes exist in one agent and conflicts among them need to be resolved.

Let us consider how the multistage negotiation protocol can be described in AgenTalk. After new message classes are defined, the script of the manager in the extended contract net protocol (`cnet-manager-with-direct-award`) is further extended for the multistage negotiation protocol (`msn-manager` in Fig. 6).[3] The state where the allocation succeeds (success state) in the contract net protocol corresponds to tentative allocation in the multistage negotiation protocol. After the tentative allocation is completed, the manager sends a *Commit* message to the contractor and tries to finalize task allocation. If a *Committed* message is returned, the allocation is finalized. However, if a *Cancel* message is received, a transition to the failure state occurs. When the failure state is reached, another task allocation can be tried; how this retry is performed is described in the parent script invoking the `msn-manager` script.

[3]The original protocol assumed that only *Direct Award* is used (i.e., *Task Announcement* is not used). The example shown here is a more generalized version.

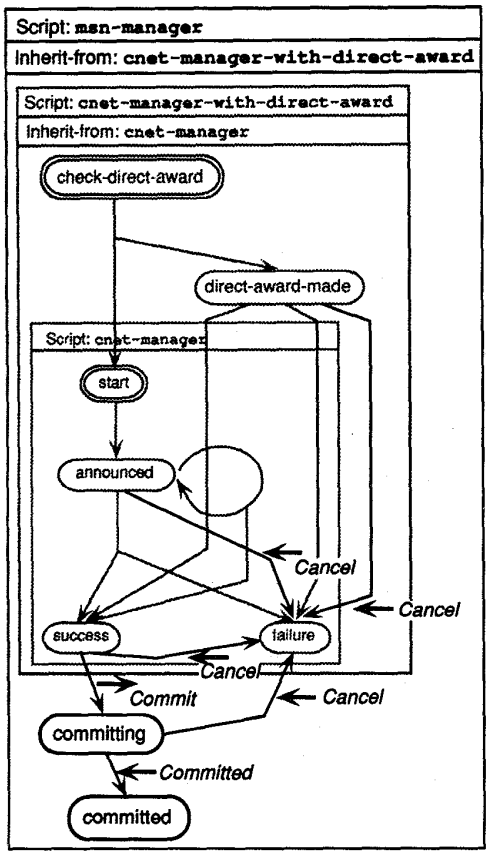

Figure 6: State Transition of a Manager Extended for the Multistage Negotiation Protocol

5 Conclusions and Future Work

This paper presented a basic outline of AgenTalk, which is designed for describing multiagent coordination protocols. Its design policies were discussed, and its capability was demonstrated through a description of the contract net protocol and the multistage negotiation protocol.

AgenTalk is being developed, and its preliminary version written in Allegro Common Lisp is currently running on Sun and SGI workstations, where agents communicate over TCP/IP.[4] AgenTalk is used to implement the agent network called *Socia*, whose goal is to build a teleorganization [19]. On Socia, a desktop teleconferencing support system was developed. In this application, the programming interface for handling messages was also developed in the C language,

[4]The software is available at the followings:
`http://www.cslab.tas.ntt.jp/at/`
`ftp://ftp.cslab.tas.ntt.jp/pub/at/at.tgz`

which enables communications between agents and programs written in C.

In this paper, we showed how a new state can be added as an incremental script definition. However, dividing an existing state into multiple states in order to make a script more detailed one might be useful for incremental definition. In this case, the redefinition of state transition rules sometimes becomes necessary. For example, when a script state (e.g., State A) is partitioned into several states in an inheriting script, a state transition rule whose destination is State A also needs to be redefined.

A similar problem was already pointed out in the concurrent object-oriented language research. Namely, the reuse of synchronization methods by using inheritance is limited when a method specifies a set of methods that can be handled next. This problem is called *inheritance anomaly*, and several mechanisms are proposed to remedy this problem [20]. Extending the inheritance mechanism of scripts in AgenTalk to allow extensive script reuse remains as future work.

Acknowledgments

The authors would like to thank Takuji Shinohara for his contributions in implementing the current version of AgenTalk.

References

[1] Smith, R. G.: The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, *IEEE Trans. Comput.*, Vol. 29, No. 12, pp. 1104–1113 (1980).

[2] Conry, S. E., Kuwabara, K., Lesser, V. R., and Meyer, R. A.: Multistage Negotiation for Distributed Constraint Satisfaction, *IEEE Trans. Systems, Man, and Cybernetics*, Vol. 21, No. 6, pp. 1462–1477 (1991).

[3] Zlotkin, G. and Rosenschein, J. S.: Cooperation and Conflict Resolution via Negotiation Among Autonomous Agents in Noncooperative Domains, *IEEE Trans. Systems, Man, and Cybernetics*, Vol. 21, No. 6, pp. 1317–1324 (1991).

[4] Durfee, E. H. and Montgomery, T. A.: A Hierarchical Protocol for Coordinating Multiagent Behaviors, *Proc. 8th National Conference on Artificial Intelligence (AAAI '90)*, pp. 86–93 (1990).

[5] Malone, T. W., Fikes, R. E., Grant, K. R., and Howard, M. T.: Enterprise: A Market-like Task Scheduler for Distributed Computing Environments, in Huberman, B. A. ed., *The Ecology of Computation*, pp. 177–205, Elsevier Science Publishers (1988).

[6] Riecken, D. ed.: *Special Issue on Intelligent Agents*, Communications of ACM, Vol. 37, No. 7 (1994).

[7] Aiba, A., Yokota, K., and Tsuda, H.: Heterogeneous Distributed Cooperative Problem Solving System Helios and Its Cooperation Mechanisms, *Proc. FGCS'94 Workshop on Heterogeneous Cooperative Knowledge-Bases*, pp. 77–84 (1994).

[8] Nishida, T. and Takeda, H.: Towards the Knowledgeable Community, in Fuchi, K. and Yokoi, T. eds., *Knowledge Building and Knowledge Sharing*, pp. 155–164, Ohmsha, Ltd. and IOS Press (1994).

[9] Finin, T., Fritzson, R., McKay, D., and McEntire, R.: KQML as an Agent Communication Language, *Proc. Third International Conference on Information and Knowledge Management (CIKM '94)* (1994).

[10] Cohen, P. R.: Communicative Actions for Artificial Agents, *Proc. First International Conference on Multi-Agent Systems (ICMAS '95)*, pp. 65–72 (1995).

[11] Labrou, Y. and Finin, T.: A Semantics Approach for KQML – A General Purpose Communication Language for Software Agents, *Proc. Third International Conference on Information and Knowledge Management (CIKM '94)* (1994).

[12] Barbuceanu, M. and Fox, M. S.: COOL: A Language for Describing Coordination in Multi Agent Systems, *Proc. First International Conference on Multi-Agent Systems (ICMAS '95)*, pp. 17–24 (1995).

[13] Kolb, M.: A Cooperation Language, *Proc. First International Conference on Multi-Agent Systems (ICMAS '95)*, pp. 233–238 (1995).

[14] Kuwabara, K., Ishida, T., and Osato, N.: AgenTalk: Coordination Protocol Description for Multiagent Systems, *Proc. First International Conference on Multi-Agent Systems (ICMAS '95)*, pp. 455 (1995).

[15] Kuwabara, K., Ishida, T., Nishibe, Y., and Suda, T.: An Equilibratory Market-Based Approach for Distributed Resource Allocation and Its Applications to Communication Network Control, in Clearwater, S. H. ed., *Market-Based Control: A Paradigm for Distributed Resource Allocation*, World Scientific Publishing (1995).

[16] Holzmann, G. J.: *Design and Validation of Computer Protocols*, Prentice-Hall (1991).

[17] Georgeff, M. P. and Lansky, A. L.: Procedural Knowledge, *Proceedings of the IEEE*, Vol. 74, No. 10, pp. 1383–1398 (1986).

[18] Kuwabara, K.: *AgenTalk Reference Manual (preliminary draft)* (1995).

[19] Ishida, T.: Bridging Humans via Agent Networks, *Proc. 13th International Workshop on Distributed Artificial Intelligence* (1994).

[20] Matsuoka, S. and Yonezawa, A.: Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Agha, G., Wegner, P., and Yonezawa, A. eds., *Research Directions in Concurrent Object-Oriented Programming*, pp. 107–150, MIT Press (1993).