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
phone: +81-774-95-1827; fax: +81-774-95-1839
e-mail: {kuwabara, osato}@cslab.kecl.ntt.jp

[‡]Department of Information Science, Kyoto University
Yoshida-honmachi, Sakyo-ku, Kyoto-shi, Kyoto 606-01 Japan
phone: +81-75-753-4821; fax: +81-75-753-5379
e-mail: ishida@kuis.kyoto-u.ac.jp

Abstract

This paper presents basic concepts behind AgenTalk, which is a newly proposed language 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 soon as more software agents start to be built. In this setting, a language for defining and implementing such coordination protocols plays a crucial role in the development of multiagent systems. AgenTalk allows coordination protocols to be defined incrementally and to be easily customized to suit application domains by incorporating an inheritance mechanism. This paper describes the design policies of AgenTalk and demonstrates its capability by describing the contract net protocol and the multistage negotiation protocol in AgenTalk utilizing its inheritance mechanism. A comparison is made with KQML (Knowledge Query and Manipulation Language) and others, and the relationships with related research in distributed artificial intelligence are also discussed.

Conference Track:	Academic
Topic Areas:	Distributed AI systems development, Multiagent coordination
Keywords:	Multiagent, Distributed AI, Coordination, Protocol description, Software agent
Contact Author:	First author (Kazuhiro Kuwabara)

1 Introduction

In multiagent systems, achieving coordination among autonomous agents is a major problem. Generally, agents exchange messages with other agents, and negotiate to reach an agreement. To achieve effective coordination in such a case, these agents have to properly determine the type of messages to exchange and the timing of the exchange. Considering this, a high-level protocol needs to be designed for multiagent coordination.

In the field of distributed artificial intelligence, several protocols for achieving coordination have been proposed: the contract net protocol for task allocation [3, 22], the multistage negotiation protocol for resource allocation under global constraints [2], the unified negotiation protocol for exchanging tasks between agents under various situations [25], 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, there is the DSP (Distributed Scheduling Protocol) for the Enterprise system [14] which applies the contract net protocol to task allocation problems in workstations connected by a local-area network. The contract net protocol has also been applied to a transportation problem (TRACONET [20]). Then, there is the application of the multistage negotiation protocol to a meeting scheduling problem [21]. Furthermore, as research on software agents proliferates [19], demands for customized protocols will increase. Without a proper framework or language, coordination protocols needs to be implemented from scratch, possibly borrowing ideas from existing protocols, in order to develop multiagent-based applications.

This paper describes basic concepts behind a language called AgenTalk which is developed to describe high-level protocols for multiagent coordination. AgenTalk is not meant to be a formal specification language; rather it is meant to be a programming language capable of implementing protocols and agents behaving according to the protocols. One of the main features of AgenTalk is the introduction of an inheritance mechanism into the protocol

description. Using this inheritance mechanism, protocol designers can design new protocols incrementally based on existing protocols, and application designers can implement customized protocols to suit each application based on existing domain independent protocols.

In this paper, problems in describing multiagent coordination protocols are identified first. Then, the design policies of AgenTalk are presented. Next, the contract net protocol and the multistage negotiation protocol, which can be viewed as a generalization of the contract net protocol, are described using AgenTalk. Finally, a comparison is made with KQML (Knowledge Query and Manipulation Language) and others, and the relationships with related research in distributed artificial intelligence are discussed.

2 Problems in Coordination Protocol Description

In this paper, a protocol is assumed to consist of the followings items:

- a format for messages exchanged between agents
- an agent's behavior upon receiving a message (message handling function, which is called a *message handler* hereafter)

The message handler, in a sense, defines the semantics of a message.

Based on this model, we have described the coordination protocol for multistage negotiation [2, 11] and market-based distributed resource allocation [10]. More specifically, a message class is defined for each message type (such as *Task Announcement* in the contract net protocol), and the semantics of a message is defined by the message handler of the message class. However, describing the protocol in this way has the following shortcomings:

- **Protocol sequence description:** We call a sequence of messages exchanged among agents in one negotiation process a *protocol sequence*. When we define the message handler for each message class, that is, define the semantics for each message class separately, it is difficult to define a protocol from the viewpoint of message sequences.

- **Incremental protocol definition:** A protocol is defined by multiple message handlers. Thus, it is difficult to incrementally define a protocol since definitions of multiple message handlers should be changed at the same time.
- **Separation of domain dependent part:** There is no framework for separating the domain dependent part and the domain independent part. Thus, it is difficult to define a domain independent protocol, and customize it to suit the application domain.
- **Conflict resolution between multiple protocol sequences:** A single agent may be involved simultaneously in multiple protocol sequences between which some interactions exist (e.g., the agent's resources are shared). In such a case, it is difficult to describe a conflict resolution mechanism in the protocol description.

3 Design Policies of AgenTalk

In order to overcome the problems outlined above, we designed a language called AgenTalk for describing coordination protocols with the following design policies.

(1) Protocols are described from the viewpoint of message sequences.

(2) Protocols are incrementally defined.

(3) Protocols are easy to customize.

(4) Conflict resolution mechanisms between protocol sequences are easy to describe.

We use an extended finite state machine, which is a finite state machine extended to allow variables, as a basis to describe coordination protocols. The extended finite machine is often used to describe communication protocols [7], and their characteristics are well known. By using this model, coordination protocols can be described from the viewpoint of message sequences (design policy (1)), and techniques developed for the extended finite machine can potentially be applied to the coordination protocols described in this way.

An extended finite state machine represents state transitions of a protocol sequence, not of an agent itself. This keeps a definition of an extended finite state machine small and modular. In addition, by introducing an inheritance mechanism in the definition of the extended finite state machine, protocols can be defined incrementally (design policy (2)).

A state transition rule in a state machine has a programming interface which specifies the portion of a rule that possibly needs to be customized by each application. By using this interface, only the domain specific portion is required to be defined (design policy (3)).

Furthermore, in AgenTalk, multiple state machines are allowed to be executed in an agent, and communication between executing contexts of a script is supported. In this way, conflict resolution mechanisms between protocol sequences can be described (design policy (4)).

The following section describes main functionalities of AgenTalk that implement these design policies in more detail.

4 Functionalities of AgenTalk

4.1 Script

Each protocol sequence corresponds to the extended finite state machine, and its representation is called a *script* in AgenTalk.[1] Script $\mathcal{SC}_i$ consists of a set of states, S_i; an initial state, so_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 defined locally in $\mathcal{SC}_i$. That is, the definition of $\mathcal{SC}_i$ is a combination of I_i and locally defined

[1]The term *script* is used because an agent taking actions according to a *script* is just like an actor acting in a play according to a play script.

components of $\mathcal{SC}_i$. Each state is associated with a name, and 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 not inherited unless it is explicitly specified to inherit rules.

An example showing a use of the inheritance mechanism is described in a later section.

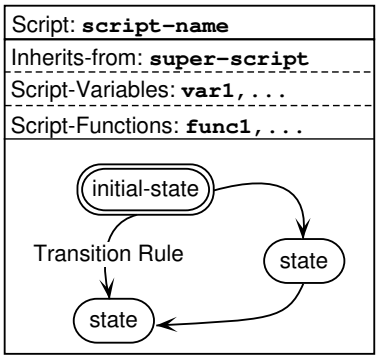

Fig. 1: Script Components

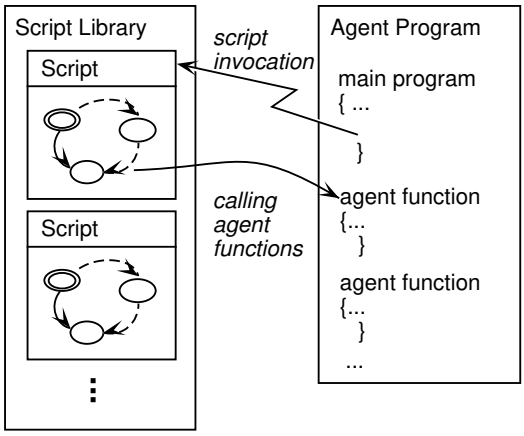

Fig. 2: Script vs. Agent

4.2 Relationship between an Agent and a Script

AgenTalk 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; AgenTalk is mainly concerned with the protocol among agents.

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 functions called from the state transition rules in the script (Fig. 2). We refer to such functions as *agent functions*. Thus, the same script can result in different behaviors of agents by providing different agent functions, which

can be customized for each application domain (design policy (3)).[2]

4.3 State Transition Rule

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 written: (1) message arrival, (2) conditions on script variables, and (3) time-out. The action part describes procedures to be executed when the rule is fired. The following procedure can be written in the action part: (1) transition to another state, (2) invocation of a script, (3) updating script variables, and (4) calling agent functions and script functions.

4.4 Model for Script Execution

When a script is invoked, a *context* is created to hold the values of script variables for each script invocation. Then, the state transition to the initial state 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 defined as the state transition rules in currently invoked scripts. Then, the action part of the selected rule is executed. When another script is invoked from the action part of a state transition rule, a new child context is created, and the script doing the invoking is called the *parent* script and the invoked script is called the *child* script (Fig. 3). Note that when a new script is invoked, active state transition rules are added, and unlike a sub-routine 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.

[2] When an agent function is not defined in the agent which invoked the script, a script function with the same name is searched for. In this way, a script function can provide a default behavior of a script.

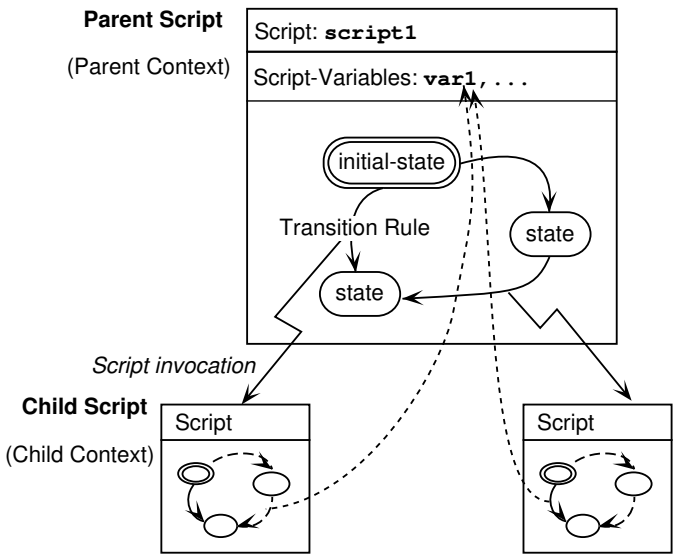

Fig. 3: Script Invocation

A protocol sequence is represented by a script. By invoking multiple scripts from a (top-level) script, multiple protocol sequences 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. That is, child scripts corresponding to protocol sequences can communicate with each other via the script variables in the parent script (context). In this way, conflict resolution among protocol sequences in an agent can be described (design policy (4)). Since a state transition rule is fired one at a time (that is, rules in parent and child scripts are not fired concurrently), we need not consider mutual exclusion in accessing script variables in a parent script (context).

5 Verification of AgenTalk Functionalities

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

5.1 Contract Net Protocol (Script of a Manager)

Task allocation in the contract net protocol is performed as follows: First, an agent (manager) with a task to be executed broadcasts a *Task Announcement* message. An agent that is willing to execute the said 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 the manager which follows the contract net protocol using AgenTalk. First, the format of messages such as *Task Announcement* is defined as a message class. Then, the script for the manager is defined. Though AgenTalk defines the syntax of a script, a figure based on state transitions is used in this paper for illustrative purposes.[3] The manager script `cnet-manager` is shown in Fig. 4. The `cnet-manager` script is supposed to be invoked when the task to be allocated is generated.

As an extension to the basic contract net protocol, a special message class *Direct Award* is used when an agent to which a task is to be allocated is known beforehand [22]. In this case, the agent receiving the *Direct Award* message, responds with an *Acceptance* message or a *Refusal* message. The script of the manager incorporating this extension (`cnet-manager-with-direct-award`) is shown in Fig. 5. This figure shows that the definition of this script inherits a large part of the definition from the `cnet-manager` script.

5.2 Contract Net Protocol (Script of a Contractor)

Let us consider the behavior of a (potential) contractor when a *Task Announcement* arrives. When the contractor can execute a task in the *Task Announcement* by itself, it can simply send a *Bid* message. Otherwise, the agent (contractor) may want to divide the task into subtasks and allocate each subtask to a different agent. If such is the case, a protocol sequence of task allocation is invoked for each subtask generated.

[3] An example script is shown in the appendix.

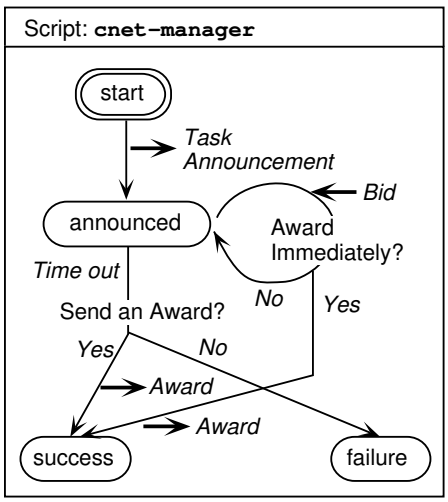

Fig. 4: Script of a Manager in the Contract Net Protocol

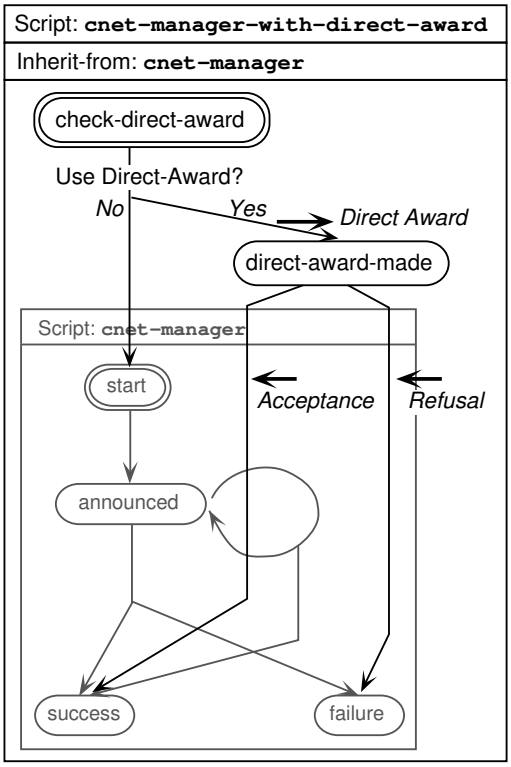

Fig. 5: Script of a Manager with Direct Award

This behavior can be described in AgenTalk as follows. First, a top-level script for a contractor (`cnet-contractor-toplevel` in Fig. 6) is defined, which invokes a `cnet-contractor` script shown in Fig. 7 when an appropriate *Task Announcement* message arrives. The `cnet-contractor` script 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 (`success` state), a *Bid* message is sent in response to the original *Task Announcement* message.

5.3 Multistage Negotiation

The multistage negotiation protocol can be regarded as a generalization of the contract net protocol [2]. In the contract net protocol, *Task Announce-*

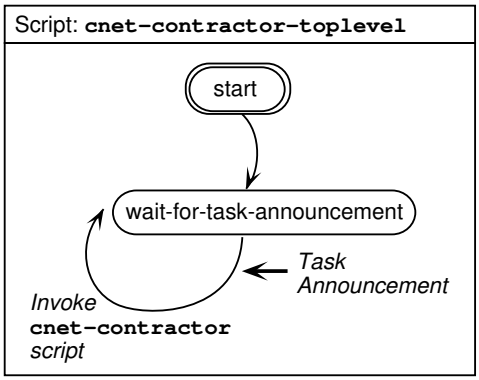

Fig. 6: Script of a Contractor Toplevel

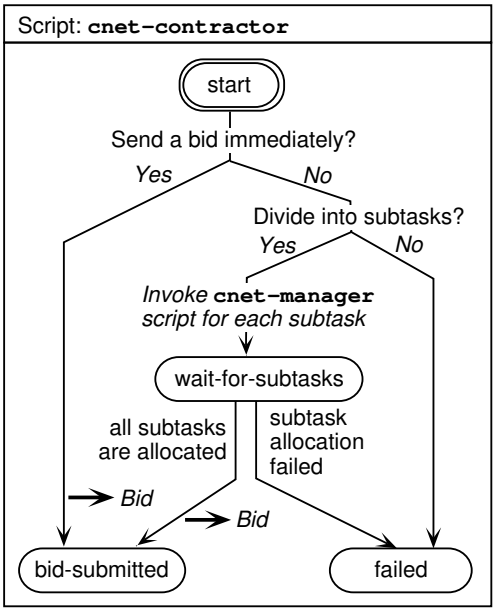

Fig. 7: Script of a Contractor

ment, *Bid*, and *Award* are basically used to allocate tasks. In contrast, the multistage negotiation protocol introduces repetitive information exchanges among agents to satisfy the global constraints. From the viewpoint of protocols, the multistage negotiation protocol has the following characteristics:

- Messages such as *Task Announcement* can be canceled by a *Cancel* message. In addition, messages for finalizing an allocation (*Commit*) and its acknowledgment (*Committed*) are added in the example shown here.
- Multiple negotiation processes exist in one agent and the conflicts among them need to be resolved.

Let us consider how the multistage negotiation protocol can be described in AgenTalk. First, new message classes are defined. Then, the script of the manager in the (extended) contract net protocol (`cnet-manager-with-direct-award`) is further extended for the multistage

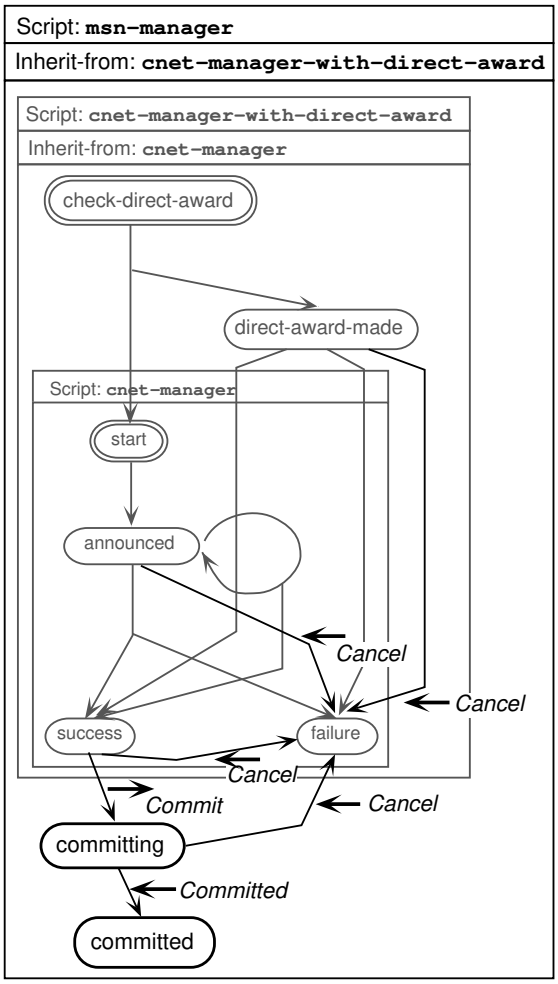

Fig. 8: Script of a Manager Extended for the Multistage Negotiation Protocol

negotiation protocol (`msn-manager` in Fig. 8).[4] 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 done, the manager sends a *Commit* message to the contractor and tries to finalize its task allocation. If a *Committed* message is returned, the allocation is finalized. If a *Cancel* message is received, however, a transition to the failure state occurs. When the failure state is reached, another

[4]The previous implementation assumed that only *Direct Award* is used (that is, *Task Announcement* is not used). The example shown here is a more generalized version.

task allocation can be tried; how this retry is performed is described in the parent script invoking the `msn-manager` script.

The contractor script for the multistage negotiation protocol also needs to handle *Cancel* messages, because a *Task Announcement* message itself can be canceled. This script inherits from the `cnet-contractor` script and adds state transition rules which cause the state transition to the failed state to occur when a *Cancel* message arrives.

There might exist multiple negotiation processes having conflicts over the use of resources in an agent. To counter this, a conflict resolution mechanism can be described using the AgenTalk's functionality that script variables in a parent script (context) can be accessed from a child script, which represents the negotiation process.

6 Related Work

KQML: A coordination mechanism among heterogeneous knowledge bases is receiving increasing attention (e.g., [1, 17]). Within the framework of DARPA's Knowledge Sharing Effort [18], KQML (Knowledge Query and Manipulation Language) is proposed, which is a language and protocol for exchanging information and knowledge among software agents.

In KQML, each agent is associated with a *router* process which interfaces an agent with a network [5]. The programming interface between the router and the agent program is defined as KRIL (KQML Router Interface Language). In addition, a special agent called a *facilitator* is proposed to serve as a mediator between agents. KQML defines performatives as its language primitives. Their semantics are described based on the speech act theory [12].

The relationship between KQML and AgenTalk from the viewpoint of the programming interface hierarchy is depicted in Fig. 9. If the functionalities of KQML are implemented using AgenTalk, a KRIL-like interface can be provided by defining message classes for each performative. In addition, a facilitator can be implemented by defining performatives a facilitator

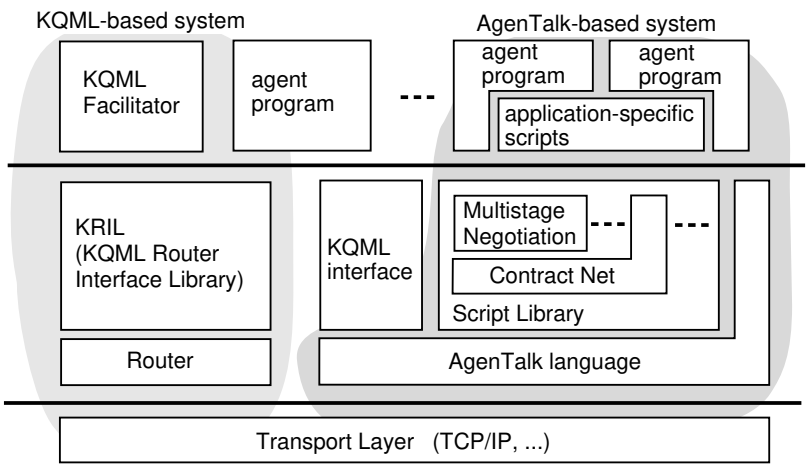

Fig. 9: KQML vs. AgenTalk (a facilitator is treated as a special agent, and shown as one of the agent programs)

needs to understand, such as Subscribe and Broker, by AgenTalk scripts. In KQML, the emphasis is put on each performative's semantics. In contrast, in AgenTalk, coordination protocols can be described from the viewpoint of state transitions and message sequences.

MAI^{2}L: MAI^{2}L language is developed to build multiagent systems in the IMAGINE project [23]. In this language, cooperation primitives, such as *propose*, *refine* are identified. Using these primitives, cooperation methods for constructing and executing domain specific multiagent plans are built. MAI^{2}L is similar to AgenTalk in that it describes domain specific multiagent coordination from domain independent primitives. However, AgenTalk supports incremental protocol definition using the inheritance mechanism.

Procedural Reasoning System: 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 [6]. AgenTalk, however, allows incremental definition of such knowledge.

Distributed Search: Multiagent problem solving can be viewed from a distributed search perspective [13]. Many distributed search algorithms (such as [9]) can be described as message exchanges between agents. For example, an asynchronous backtracking algorithm for distributed constraint satisfaction [24] uses two classes of messages; one is to notify of the local value assignments and the other is to notify of a set of value assignments violating the constraints. Once a script is written based on the asynchronous backtracking algorithm, it is easier to apply the algorithm to different application domains.

7 Conclusion and Future Work

This paper presented basic ideas of AgenTalk for describing multiagent coordination protocols. The design policies of AgenTalk were discussed, and the contract net protocol and the multistage negotiation protocol were described using AgenTalk to demonstrate its description capability.

AgenTalk is being developed, and its first version written in Allegro Common Lisp is currently running on Sun and SGI workstations, where agents communicate over TCP/IP. AgenTalk is used in implementing the agent network called *Socia*, whose goal is to realize a teleorganization [8]. On Socia, a desktop teleconferencing support system was developed. In this application, the programming interface for handling messages is also implemented in the C language, and is used to enable communications between agents and existing programs written in C.

In concurrent object-oriented languages, it is pointed out that reuse of synchronization methods by inheritance is limited. This problem is called *inheritance anomaly* [16], and several mechanisms are proposed to remedy this problem (e.g., [15]). In AgenTalk, a similar problem occurs for the reuse of scripts by inheritance. For example, when a script state (say, 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. Extending the

inheritance mechanism between scripts to allow extensive script reuse is left for future work.

We assumed that scripts are defined beforehand. However, it is an interesting question how to change scripts dynamically and to add new scripts on the fly. This is related to multiagent planning and learning. We also assumed that agents have agreed beforehand on which protocol should be used in a particular situation. However, the (meta-level) protocol to reach an agreement among agents on the (domain-level) protocol to be used is also necessary. How can this meta-level protocol description be integrated with domain-level protocol description is remained for future work.

Furthermore, it is preferable for software agents to be easily customized to a user's preferences. We are planning to investigate a way of evolving AgenTalk into an extension language for software agent customization.

Acknowledgments

The authors would like to thank Takuji Shinohara, Makoto Yokoo, and Futoshi Naya for their contributions in implementing AgenTalk.

References

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

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

[3] Davis, R. and Smith, R. G.: Negotiation as a Metaphor for Distributed Problem Solving, *Artificial Intelligence*, Vol. 20, No. 1, pp. 63–109 (1983).

[4] Durfee, E. H. and Montgomery, T. A.: A Hierarchical Protocol for Coordinating Multiagent Behaviors, *Proc. 8th National Conference on Artificial Intelligence*, pp. 86–93 (1990).

[5] Finin, T., Fritzson, R., McKay, D., and McEntire, R.: KQML as an Agent Communication Language, *Proc. 3rd International Conference on Information and Knowledge Management* (1994).

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

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

[8] Ishida, T.: Bridging Humans via Agent Networks, *Distributed Artificial Intelligence: Papers from the Thirteenth International Workshop, Technical Report WS-94-02*, pp. 142–153, AAAI Press (1994).

[9] Kitamura, Y. and Okumoto, T.: Diffusing Inference: An Inference Method for Distributed Problem Solving, in Deen, S. M. ed., *Cooperating Knowledge Based Systems 1990*, pp. 79–94, Springer-Verlag (1990).

[10] Kuwabara, K. and Ishida, T.: Equilibratory Approach to Distributed Resource Allocation: Toward Coordinated Balancing, in Castelfranchi, C. and Werner, E. eds., *Artificial Social Systems, MAAMAW'92*, Lecture Notes in AI 830, pp. 133–146, Springer-Verlag (1994).

[11] Kuwabara, K. and Lesser, V. R.: Extended Protocol for Multistage Negotiation, *Proc. 9th Workshop on Distributed Artificial Intelligence*, pp. 129–161 (1989).

[12] Labrou, Y. and Finin, T.: A Semantics Approach for KQML – A General Purpose Communication Language for Software Agents, *Proc. 3rd International Conference on Information and Knowledge Management* (1994).

[13] Lesser, V. R.: A Retrospective View of FA/C Distributed Problem Solving, *IEEE Trans. SMC*, Vol. 21, No. 6, pp. 1347–1362 (1991).

[14] 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).

[15] Matsuoka, S., Taura, K., and Yonezawa, A.: Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, *Proc. OOPSLA '93*, pp. 109–126 (1993).

[16] 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).

[17] 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).

[18] Patil, R. S., Fikes, R. E., Patel-Schneider, P. F., Mckay, D., Finin, T., Gruber, T., and Neches, R.: The DARPA Knowledge Sharing Effort: Progress Report, in Nebel, B., Rich, C., and Swartout, W. eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, Morgan Kaufmann (1992).

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

[20] Sandholm, T.: An Implementation of the Contract Net Protocol Based on Marginal Cost Calculations, *Proc. 11th National Conference on Artificial Intelligence*, pp. 256–262 (1993).

[21] Sen, S. and Durfee, E. H.: The Role of Commitment in Cooperative Negotiation, *International Journal of Intelligent and Cooperative Information Systems*, Vol. 3, No. 1, pp. 67–81 (1994).

[22] 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).

[23] Steiner, D., Burt, A., Kolb, M., and Lerin, C.: The Conceptual Framework of MAI^{2}L, Ghedira, K. and Sprumont, F. eds., *Pre-Proc. MAA-MAW '93* (1993).

[24] Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K.: Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving, *Proc. 12th International Conference on Distributed Computing Systems*, pp. 614–621 (1992).

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

Appendix A Script Example

The script of a manager in a contract net protocol (Fig. 4) is shown here. AgenTalk is currently implemented on top of Common Lisp, and the lisp-like syntax is adopted. Common Lisp primitives such as `if`, `progn`, `setf` are readily available. `define-script` macro defines a script, and `define-script-state` macro defines each state in a script. A state transition rule is defined in the definition of the associated state (`define-script-state :rules` option).

```
;;; cnet-manager script
;;; When this script is invoked, the task to be allocated is given.
;;; task (which is treated as a script variable) holds this task information.
(define-script cnet-manager (task)
  :initial-state start
  ;; Script variable bid-queue holds bid messages received.
  ;; Script variable contract-id holds a current contract-id.
  :script-vars (bid-queue contract-id)
  )
```

```
;;; start state
(define-script-state (start cnet-manager)
  ;; Script variables used in this definition of a state
  :script-vars-used (task contract-id)
  ;; :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.
  :on-entry (progn (setf contract-id (! announce-task task))
                   ;; go to the announced state.
                   (goto-state 'announced)))
;;; announced state
(define-script-state (announced cnet-manager)
  :script-vars-used (task bid-queue contract-id)
  :rules
  ;; a bid message with this contract-id arrived?
  ;; !contract-id represents the value of the script variable contract-id.
  ((:when (msg bid :contract-id !contract-id)
    ;; award-immediately? returns T, if an award message is sent.
    :do (if (! award-immediately? msg bid-queue)
            (goto-state 'success)
            ;; store this bid message to the script variable bid-queue.
            (push msg bid-queue)))
   ;; the task's deadline is reached?
   (:when (timeout (task-expiration task))
    ;; award-sent? returns T when an award message is sent.
    :do (if (! award-sent? bid-queue)
            (goto-state 'success)
            (goto-state 'failure)))))
;;; success state
(define-script-state (success cnet-manager))
;;; failure state
(define-script-state (failure cnet-manager))
```