Meta-Level Control of Coordination Protocols

Kazuhiro Kuwabara*
NTT Communication Science Laboratories
2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 JAPAN

Abstract

This paper proposes a meta-level control mechanism of coordination protocols in a multi-agent system. In an open environment, an agent needs to respond to unexpected situations (exceptions). By introducing a meta-level control mechanism in the coordination protocol execution, an agent can have flexible control over the coordination with other agents. As a meta-level control mechanism, methods to detect and handle exceptions are presented. For example, the proposed mechanism allows an agent to dynamically switch a coordination protocol for use when an exception occurs. In this paper, AgenTalk, a coordination protocol description language, is extended to include primitives for the meta-level control. In this extension, the meta-level control is described using the same mechanism for describing an agent's behavior in a coordination protocol.

Introduction

In a multi-agent system which consists of multiple autonomous agents, how to coordinate actions of agents is one of the major issues. Generally, coordination among agents is achieved by exchanging messages between them. We call the protocol in these message exchanges a *coordination protocol.*

Multi-agent coordination needs to adapt to changes in the environment. Many unexpected situations may occur, such as a message failing to arrive at its destination, or the arrival of an unexpected message. An agent needs to respond to these exceptions, possibly changing its behavior in the coordination with other agents.

In this paper, we propose a mechanism to introduce a meta-level control of coordination protocols in order to achieve more flexible control over the execution of coordination protocols and the handling of exceptions. There has been a lot of research done on meta-level control, especially in the cooperative distributed problem solving area (e.g., (Corkill & Lesser 1983)). The research mainly focuses on the sophisticated control of the reasoning processes of a problem solver. In contrast, this paper focuses on the issue of controlling the execution of coordination protocols which are already described in a coordination protocol description language.

There are several languages proposed for describing coordination protocols among agents. For example, the agent communication language KQML (Finin *et al.* 1994), which was developed as part of DARPA's knowledge sharing effort (Patil *et al.* 1992), defines message types based on speech act theory. A language called COOL was developed to describe coordination protocols using KQML messages. In COOL, protocol description is based on a finite state machine.

There is also another language, called CooL (Kolb 1995), which was evolved from the language called MAI^{2}L (Steiner *et al.* 1995) for describing a multi-agent system in the IMAGINE project (Haugeneder, Steiner, & McCabe 1994). In CooL, protocols are described using cooperation methods. Another language which allows for the easy customization of protocols was developed in the COSY project (Burmeister, Haddadi, & Sundermeyer 1995).

It is not easy, however, to explicitly describe a meta-level control of coordination protocols in these languages. In this paper, we extend a coordination protocol description language called AgenTalk (Kuwabara, Ishida, & Osato 1995) and introduce primitives of a meta-level control into this language.

This paper is structured as follows. First, an outline of the original AgenTalk is presented. Next, a meta-level control of coordination protocols is described, and AgenTalk is extended to include primitives for the meta-level control. Then, an example of the meta-level control is shown.

AgenTalk

AgenTalk is a language for describing coordination protocols. It is intended to be used to design a real world multi-agent system, rather than a formal description language of multi-agent coordination protocols. AgenTalk facilitates the development of application-

*Currently with NTT Research and Development Headquarters, 3-19-2 Nishi-shinjuku, Shinjuku-ku, Tokyo 163-19 Japan (e-mail: kuwabara@yamato.ntt.jp).

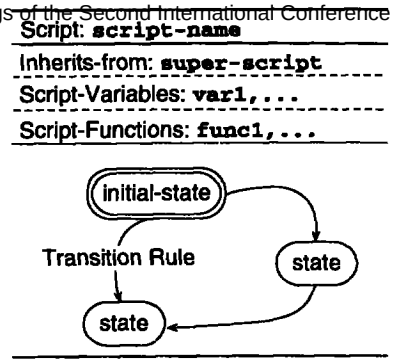

Figure 1: Script Component

specific protocols by providing the following features:

- an inheritance mechanism as seen in object-oriented languages to allow the incremental definition of protocols, and
- the interface for each agent to customize coordination protocols.

Description Model

In AgenTalk, a coordination protocol is described based on an extended finite state machine, and its representation is called a *script*. More specifically, the behavior of an agent following a protocol is described as a script. By introducing an inheritance mechanism in the definition of a script, a script can be defined incrementally, inheriting the definition of an existing script.

Script: A script is defined as a tuple of states, an initial state, state transition rules, script variables, script functions, and an inherited script (Figure 1). Script variables can potentially have different values for each invocation of a script. A local function in a script can be defined as a script function.

State transition rule: A state transition rule in a script definition consists of a condition part and an action part. In the condition part, the following conditions can be written: (1) the condition on a message arrival (*msg-condition*), (2) the timeout condition (*timeout-condition*), and (3) the condition on script variables (*script-var-condition*).

In AgenTalk, a message is represented as an instance of a message class, which declares slots of its instance (message). Thus, a message is represented as pairs of a slot and its value. There is also an inheritance of slot declarations among message classes.

The condition on a message arrival (*msg-condition*) is represented as a *message pattern*, which contains conditions regarding a message class, a sender of the message, and slot values. Using an inheritance relationship among message classes, it is defined that a message can match the message pattern which specifies the super class of the message.

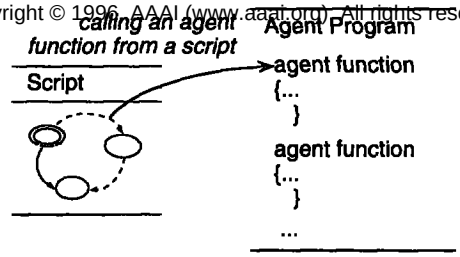

Figure 2: Agent Function

In the action part, various procedures can be written including (1) the state transition to another state, (2) the update of script variables, and (3) the control of the script execution, which is described later.

Agent function: A script has an interface with which an agent can customize the protocol. This interface is called an *agent function*, and it is a kind of callback function, to be called from the state transition rules during the script execution. A protocol can be customized for each agent by defining its own agent function.

The current version of AgenTalk is implemented in Common Lisp and uses S-expression-like syntax. The syntax of the script definition is described in Figure 3. The `define-script` macro declares a script, and the `define-script-state` macro defines a state and state transition rules which are to be active in the state. In addition, the `define-script-state` macro can define a procedure which is to be invoked when a state transition to the state being defined occurs (`:on-entry` option).

Execution Model

When an agent decides to follow a certain protocol, it invokes a corresponding script. Once invoked, a script is basically executed as follows: first, a state transition rule whose condition part holds is selected from currently active rules; next, its action part is executed; then the script execution goes back to the first step.

Meta-Level Control

In the proposed meta-level control mechanism, a script is utilized to control the execution of another script. In other words, a script describes a (meta-level) control of a protocol (script). Describing the script control mechanism by a script itself makes it possible to utilize the inheritance and customization mechanisms of script definitions.

```
(define-script script-name lambda-list {script-options}*)
script-options ::= :initial-state initial-state-name
                 | :script-vars ({script-var | (script-var initial-value)}*)
                 | :inherits-from script-name

(define-script-state (state-name script-name) [ :on-entry lisp-form ] [ :rules ({rule}*) ])
rule ::= (:when condition :do action) | :inherited
condition ::= msg-condition | timeout-condition
            | [ msg-condition | timeout-condition ] script-var-condition
msg-condition ::= (msg message-class [ sender ] {slot-name value-pattern}*)
timeout-condition ::= (timeout lisp-form)
script-var-condition ::= (test lisp-form)
```

Figure 3: Syntax of Script Definition (in part)

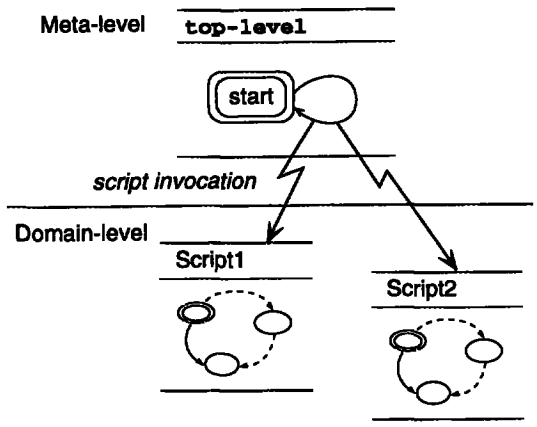

Figure 4: Top-level Script

Script Invocation

Another script can be invoked using the `run-script` function at the action part of a state transition rule. The script which invokes another script is called a *parent* script, and the invoked script is called a *child* script. Basically, multiple scripts can be executed concurrently.[1]

When an agent receives a message, it searches the state transition rules in the currently invoked scripts for a rule whose condition part holds. This search starts at the most recently invoked script and continues in the reverse order of the script invocations. That is, a child script is searched before its parent script, and the initially invoked script is searched last. Thus, a parent script can be used to handle background jobs (with lower priority), while child scripts handle each individual jobs.

[1]There is also another mode in the invocation of a script, which suspends the execution of a parent script temporarily until the execution of a child script terminates. This mode is provided mainly for a programmer's convenience.

The top-level script, which is invoked when an agent is started, can be viewed to provide a meta-level control of protocols. That is, this top-level script will invoke scripts of domain-level protocols as necessary (Figure 4). By using the agent function interface, the top-level script itself can be customized for each agent. In addition, the top-level script can be extended inheriting the definition of a basic top-level script.

Script Execution Control Primitives

The `run-script` function returns a *script execution context*. The context holds the information regarding the execution of a script. When multiple scripts are invoked, a context is created for each script invocation. From a context, the current state, parent context, child context(s), and descendent context(s) can be obtained (Table 1). A script variable can have a context as its value. By describing a condition on the script variables in a state transition rule, a script can control the execution of another script based on the status of the script execution.

In addition to `run-script`, there are other primitives provided for controlling the execution of a script. They can terminate, suspend, and resume the execution of a script taking a context as an argument (Table 2).

Detecting Exceptions

The main goal of the meta-level control described in this paper is to handle unexpected situations. Therefore, it is necessary to detect an exception and notify the (meta-level) script execution (program) of the exception occurred. Basically, exceptions are detected either by an AgenTalk system or by a meta-level script. The former case includes a message delivery problem, and the latter case includes a message which no currently activated scripts can handle.

In order that an exception can be notified, a message class `exception` is defined. For each specific exception, its derived class is defined. The examples are as follows.

Table 1: Primitives for Accessing a Script Execution Context

primitive	description
`current-state` *context*	returns a current state name
`parent-context` *context*	returns a context of an immediate parent
`child-contexts` *context*	returns a list of child contexts
`descendant-contexts` *context*	returns a list of descendant contexts

Table 2: Primitives for Script Execution Control

primitive	description
`run-script` *script-name-and-options* {*argument*}*	invokes a script
`exit-script` [*return-value*] [*context*]	terminates the script execution
`suspend-script` [*context*]	suspends the script execution
`resume-script` [*context*]	resumes the script execution

Message delivery error: When a message fails to be delivered, the AgenTalk system sends a message of the `message-not-delivered` class, which is a subclass of `exception`, to the sender agent. A state transition rule which handles this type of message can be defined to describe an action to be taken when a message delivery problem occurs.[2]

Message unmatched: At least one state transition rule in currently activated scripts should match an incoming message. When a message which does not have a matching state transition rule arrives, it should be considered an exception, and an exception notifying message (`message-unmatched`, which is a subclass of `exception`) is sent to the agent (a receiver of the original message).

More specifically, the top-level script has a state transition rule whose condition part matches any message. Since the top-level script will be searched last, this rule will be fired only when no other scripts have a matching rule. The action part of this rule is to send a message of the `message-unmatched` class to the agent. A user needs to define a script which handles this `message-unmatched` message.

Coping with Exceptions

An agent needs to respond to the exceptions. In this section, a mechanism to handle a `message-unmatched` exception is described. The basic idea is to search for a script which can handle the unmatched message and change the script of an executing context to the script found. Using this mechanism, it is possible that a basic protocol (script) is invoked first, and an agent changes the executing script to an extended and more specific protocol (script) as the need arises (i.e., the arrival of a message which cannot be handled).

This default behavior is defined in the top-level script. Since a matching state transition rule is searched for in the reverse order of the script invocations, this default behavior can be overridden by invoking a script which contains a rule that can handle a `message-unmatched` message.

AgenTalk supports an inheritance in the script definitions. This inheritance relationship is utilized to change the executing script dynamically. We call a script whose definition is inherited a *superclass* script, and a script which inherits the definition of another script a *subclass* script.

A primitive `change-script` is introduced which changes the executing script of a context to another script.[3] In order to change the executing script, the following conditions must be met.

- Both original and new scripts must have the same superclass script (including itself).
- The state with the same name of a current state of the context must exist in a new script.

The former condition can reasonably limit the range of scripts when a new script is searched for. The latter condition ensures that the current state can be determined in a new script. If the script of a context changes from a superclass script (more general one) to a subclass script (more specific one), the above conditions will hold, since a new script will have a state with the same name as the current state, inheriting from the superclass script.

When the script of a context changes, script variables may also change. When a script variable is no longer necessary in a new script, the current value of the variable is temporarily stored. When the executing script is changed later and a script variable with

[2]This kind of state transition rule is often in effect in all states. Thus, there is a way to define this kind of exception handling rules separately from typical state transition rules.

[3]When we view a script definition as a class definition, and its executing context as an instance of a class as in object-oriented languages, changing the executing script of a context corresponds to changing the class of an instance (e.g., `change-class` in Common Lisp).

```
;;;script declaration
(define-script top-level ()
  :initial-state 'start)
;;;state transition rules
(define-script-state (start top-level)
  :rules
   ;;this rule matches a message-unmatched message.
  ((:when (msg message-unmatched :message ?msg)
    ;;find a script which can handle the message (?msg).
    ;; context refers to the current execution context.
    ;; note that Common Lisp functions/macros can be used here.
    :do (let ((context-list (descendant-contexts context)))
          ;;try each context in the descendant contexts.
          (dolist (this-context context-list)
            ;;is an appropriate script found?
            (let ((scripts (find-script this-context ?msg)))
              (when scripts
                ;;if found, change the executing script of this context to the top of the script list.
                (change-script this-context (car scripts) ?msg)
                ;;exit the dolist.
                (return nil))))))
   ;;other default rules such as handling a message-not-delivered message should come here...
   ...
   ;;the last rule matches any message.
   (:when (msg message)
    ;;send an exception notifying message to itself (self).
    :do (send-message self 'message-unmatched :message msg))
  ))
```

Figure 5: Example Code of the Top-level Script (in part)

the same name is needed again, the previously stored value is restored.

The primitive `change-script` has an optional argument *unmatched-message* so that the message which has triggered the script change can be processed when `change-script` is called. In addition, a primitive (`find-scripts`) is provided to find a list of candidate scripts from (a library of) existing scripts. This primitive takes a context and an unmatched message as its arguments and returns a list of possible scripts to which the executing script of a given context can be changed. Using these primitives, an example top-level script can be written as seen in Figure 5.

Example

Let us consider an example taken from the contract net protocol (Smith 1980). The basic contract net protocol uses a task announcement, a bid, and an award to allocate a task among agents. The state transition of a *manager* agent, which allocates a task to another agent, can be described as seen in Figure 6 (the state transition diagram) and Figure 7 (the sample code) (Kuwabara, Ishida, & Osato 1995). After a manager agent broadcasts a task announcement message, it waits for a bid message, and the manager agent sends an award message if a proper bid is received.

Many extensions to the basic contract net protocol are possible. For example, when an agent receives a task announcement but cannot send back a bid, the agent may send an immediate response bid (Smith 1980) to notify the manager agent of the reason why a bid cannot be sent. Similarly, we can consider a *counter proposal* to the task announcement instead of a bid. This counter proposal message would suggest a task specification the agent can handle. If we incorporate this extension, a counter proposal message needs to be handled in the *announced* state in addition to bid messages. When a manager agent recognizes that an award cannot be sent, it checks the counter proposal messages received so far and decides whether it goes back to the start state and sends another (possibly modified) task announcement (Figure 8 and Figure 9).

Let us suppose that agents cannot decide which specific extension of the contract net protocol to use at the start. Using the script change mechanism described above, an agent can start with the basic pro-

```
(define-script cnet-manager (task)
  :initial-state 'start
  :script-vars (bid-queue contract-id))
;;;start state
(define-script-state (start cnet-manager)
  ;;broadcast a task-announcement message by invoking the agent function announce-task.
  :on-entry (progn (setf ($ contract-id) (! announce-task))
                   (goto-state 'announced)))
;;;announced state
(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)
            (goto-state 'success)
            (push msg ($ bid-queue))))
    ;;check a timeout condition.
   (:when (timeout (task-expiration task))
    :do (if (! send-award-if-possible)
            (goto-state 'success)
            (goto-state 'failure)))))
;;;success state
(define-script-state (success cnet-manager)
  :on-entry (exit-script t))
;;;failure state
(define-script-state (failure cnet-manager)
  :on-entry (exit-script nil))
```

Figure 7: Example Script of a Manager in the Contract Net Protocol

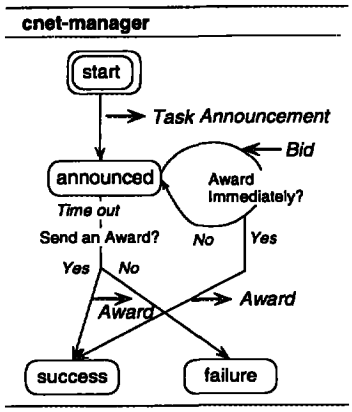

Figure 6: State Transition of a Manager in the Contract Net Protocol

tocol (`cnet-manager` script) and switch the executing script to the extended script when a message which cannot be handled by the current script arrives (in this case, `counter-proposal`). When such a message arrives, the default rule in the top-level script sends a `message-unmatched` exception message to the agent. This message results in switching the executing script to the `cnet-manager-with-counter-proposal` script (Figure 10).

Of course, it is possible to define a script which includes all the possible extensions and invoke that script from the start. However, as a protocol becomes complicated, it may not be possible to define a script which can handle all the extensions, because some of the extensions may be incompatible with each other. In such a case, switching the executing script is useful.

Conclusion

This paper described the meta-level control of coordination protocols and presented a way to implement it as an extension to AgenTalk. The primitives for controlling the script execution are introduced and a script itself is used to control the execution of other scripts.

Currently the meta-level control mechanism described in this paper is being implemented in AgenTalk. In addition, AgenTalk is planned to be extended to represent the procedural knowledge (Georgeff & Lansky 1986) of an agent. The original version of AgenTalk which runs on Allegro Com-

```
(define-script cnet-manager-with-counter-proposal (task)
  :script-vars (counter-proposal-list)
  :inherits-from cnet-manager)
;;;redefine the announced state.
(define-script-state (announced cnet-manager-with-counter-proposal)
  :rules
  (;;state transition rules are inherited.
   :inherited
   ;;a state transition rule which handles a counter-proposal message is added.
   (:when (msg counter-proposal :contract-id !($ contract-id))
    ;;record counter proposals.
    :do (push msg ($ counter-proposal-list)))))
;;;redefine the failure state.
(define-script-state (failure cnet-manager-with-counter-proposal)
  ;;check counter proposals received so far.
  :on-entry (if (! retry-using-counter-proposal-p)
                ;;go back to the start state.
                (goto-state 'start)
                ;;exit the execution of this script.
                (exit-script nil)))
```

Figure 9: Example Script of a Manager in the Extended Contract Net Protocol with a Counter Proposal

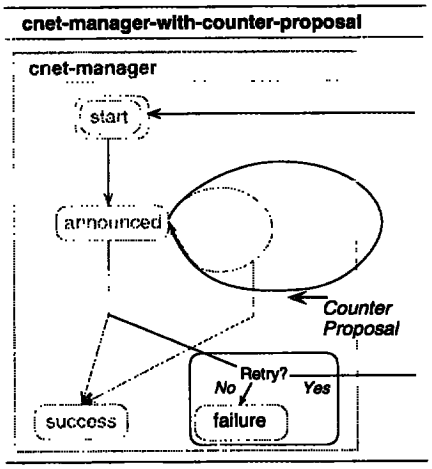

Figure 8: Extended Contract Net Protocol with a Counter Proposal

mon Lisp and Macintosh Common Lisp is available as free software.[4]

The meta-level control mechanism proposed in this paper is rather simple one. For example, it is not easy to make certain that an unmatched message is a response to the previously sent message. Moreover, since multiple script execution contexts may exist in one agent simultaneously, the meta-level control mechanism needs to find an appropriate context when switching a script. Currently, selecting the correct context is up to a user; in a user program, a message class may need to have a slot which holds an ID of a message exchange sequence (conversation) in order to distinguish between script execution contexts. Extending the AgenTalk language to handle these matters more easily is left for the future work.

Finally, we plan to evaluate the mechanism proposed in this paper by building multi-agent applications such as a conference registration support system.

[4]URL: `http://www.cslab.tas.ntt.jp/at/`

Acknowledgments

The author would like to thank Toru Ishida and Nobuyasu Osato, the co-designers of the original AgenTalk. He would also like to thank Kazunori Asayama, Motohide Otsubo, Takuji Shinohara, Isoo Ueno, and Sen Yoshida for their contributions to the AgenTalk language and its implementation.

References

Burmeister, B.; Haddadi, A.; and Sundermeyer, K. 1995. Generic, configurable, cooperation protocols for multi-agent systems. In Castelfranchi, C., and Müller, J.-P., eds., *From Reaction to Cognition, MAAMAW '93*, Lecture Notes in AI 957. Springer-Verlag. 157–171.

Corkill, D. D., and Lesser, V. R. 1983. The use of meta-level control for coordination in a distributed problem solving network. In *Proc. 8th International Joint Conference on Artificial Intelligence (IJCAI '83)*, 748–756.

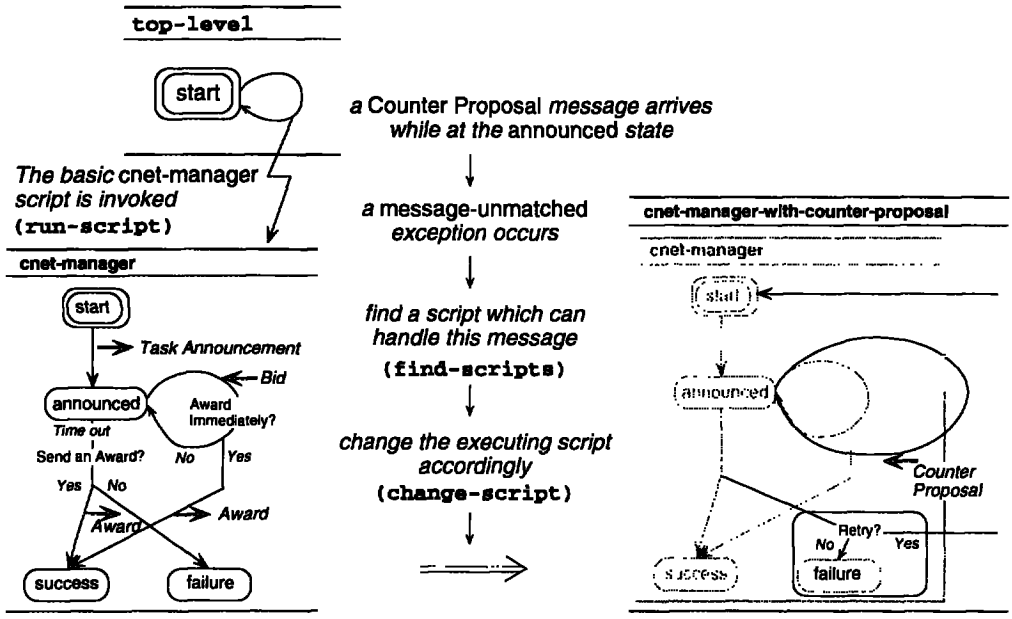

Figure 10: Example of a Script Change

Finin, T.; Fritzson, R.; McKay, D.; and McEntire, R. 1994. KQML as an agent communication language. In *Proc. Third International Conference on Information and Knowledge Management (CIKM '94)*.

Georgeff, M. P., and Lansky, A. L. 1986. Procedural knowledge. *Proceedings of the IEEE* 74(10):1383–1398.

Haugeneder, H.; Steiner, D.; and McCabe, F. G. 1994. Imagine: A framework for building multi-agent systems. In *Proc. Second International Working Conference on Cooperating Knowledge Based Systems (CKBS '94)*, 31–64.

Kolb, M. 1995. A cooperation language. In *Proc. First International Conference on Multi-Agent Systems (ICMAS '95)*, 233–238.

Kuwabara, K.; Ishida, T.; and Osato, N. 1995. AgenTalk: Describing multiagent coordination protocols with inheritance. In *Proc. 7th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '95)*, 460–465.

Patil, R. S.; Fikes, R. E.; Patel-Schneider, P. F.; Mckay, D.; Finin, T.; Gruber, T.; and Neches, R. 1992. 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.

Smith, R. G. 1980. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers* 29(12):1104–1113.

Steiner, D.; Burt, A.; Kolb, M.; and Lerin, C. 1995. The conceptual framework of MAI^{2}L. In Castelfranchi, C., and Müller, J.-P., eds., *From Reaction to Cognition, MAAMAW '93*, Lecture Notes in AI 957. Springer-Verlag. 217–230.