ELSEVIER

Artificial Intelligence 114 (1999) 57–94

Artificial Intelligence

www.elsevier.com/locate/artint

Smart office robot collaboration based on multi-agent programming

F. Mizoguchi[1], H. Nishiyama[2], H. Ohwada[3], H. Hiraishi[*]

Science University of Tokyo, Noda,Chiba 278, Japan

Received 15 February 1998; received in revised form 16 November 1998

Abstract

As a new Artificial Intelligence (AI) application to our everyday life, we designed and implemented a smart office environment in which various information appliances work collaboratively to support our office activities. In this environment, many cameras and infrared sensors allow handling robots and mobile robots to perform complex tasks such as printing and delivering document. The delivery task is a typical example of an important class of tasks supporting humans in the smart office. In this paper, such robots are modeled as robotic agents, and collaboration between the agents is realized using multi-agent programming. We have developed a multi-agent robot language (MRL) as an evolution of concurrent logic programming. MRL provides synchronous and asynchronous control of agents based on guarded Horn clauses. It also supports describing an advanced negotiation protocol using broadcast and incomplete messages, and making decisions using a set of logical rules. These features are unified within an MRL framework, yielding an intelligent integration of the robotic agents. We view the smart office environment as a human assistant system through agent collaboration, and this view is novel and extendable as AI for everyday functions. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Multi-agent programming; Smart office; Robot collaboration; Agent architecture; Information appliance

1. Introduction

Artificial Intelligence (AI) can be applied in many areas, such as medical diagnosis, molecule design, and spaceship control. New applications of AI are being developed to

* Email: hiraishi@imc.sut.ac.jp.
1 Email: mizo@imc.sut.ac.jp.
2 Email: nisiyama@imc.sut.ac.jp.
3 Email: ohwada@imc.sut.ac.jp.

keep pace with increasing computer and internet use, such as information retrieval from the WWW. These new uses indicate that the targets of AI are shifting from functioning in highly technical systems to working in agent systems that support human activities. AI technology is expanding from knowledge representation and reasoning to agent-oriented language, allowing us to apply concepts such as human belief and negotiation to computers. The original concept of an agent arose from social sciences; however, AI faces new challenges within this concept and there are many proposals based on the agent concept [6,9,12,18,30,38].

Although the definition of an agent varies, the common concept is that an agent can be introduced implicitly into the idea that computers can support human beings. For example, some interface agents are filtering agents, which classify e-mail according to the user's preference [19,24], and perform meeting scheduling [22]. We focus here on the support of everyday activities in our homes and office environments as a new application for AI based on the agent concept. This proposal provides a viewpoint of AI usage for everyday activities, in contrast to psychology for everyday things by Norman [26].

Information appliances such as sensors, monitors, cameras, and robots in future offices or houses must cooperate with each other and provide intelligent support using AI for everyday activities. For example, office activities can be made more efficient by a delivery task, where mobile robots deliver documents and printed papers, monitored by cameras and infrared sensors embedded in the office environment. A TV conference system in a conference room with several cameras located on the ceiling can help a conference proceed smoothly. A speaker accesses the TV conference system with a hand-held computer, and one camera automatically focuses on the speaker. A projector then displays the image from the camera on a screen. These information appliances do not work independently in such an office environment, but they support our activities by cooperating with each other. We refer to this future office as a smart office.

Unlike an engineered solution usually used for such information appliance control, an approach based on AI are necessary for intelligent cooperation among robots and sensors in a smart office. For a delivery task, the decision of which cameras and infrared sensors a mobile robot should cooperate with is based on negotiation, such as a contract net protocol [33]. In the contract net protocol, a task request message is broadcast to all the cameras, and one camera is selected from among the cameras that accepted the message. The requested cameras must determine whether they can perform the task by themselves. Reasoning from an existing situation based on a rule set of the available functions of the cameras is necessary to reply the requested task.

We must regard these cameras as intelligent ones and model each camera as an agent in order to realize these functions. Here, the meaning of an agent includes not only supporting human beings but also cooperating with other information appliances including robots. For example, a camera that accurately guides mobile robots can be regarded as an agent cooperating with the mobile robots. We define such an agent as a robotic agent which collaborates with other robotic agents.

We started by developing a multi-agent programming language as a basis for collaboration among robotic agents. This approach from a language design can objectively provide the details of the agent architecture and the method for constructing an agent, since the concept of the agent is represented as programs. This enables us to clarify the definition and functions of agents in a smart office. The multi-agent robot programming language, MRL, that we developed extends concurrent logic programming. MRL supports logic-style programming that enables a description of the concurrency with synchronization and asynchronization. Basically, multiple robots and sensors work concurrently as they synchronize with each other for a cooperative task. This concurrent control of agents is easily realized by MRL. MRL programs are compiled into KL1[4] programs, which are then compiled, using the KLIC [2] system, into C programs running on UNIX-based systems.

Unlike previous concurrent logic programming languages, MRL supports task broadcasting, like a contract net, and MRL has various message communication functions to realize advanced agent negotiation. Moreover, MRL has a theorem-proving function to process queries regarding whether an agent can achieve the requested task. These functions are not control engineering, but they are based on the technology of AI to make the robotic agents intelligent. Multi-agent programming by MRL not only controls distributed information appliances cooperatively in a smart office, but also provides the means to realize inter-agent collaboration such as commitment, consensus and competition to achieve given tasks.

In this paper, we focus on document delivery and printed out document delivery tasks by mobile robots, as an example of a general class of smart office tasks. Although we could use various smart office examples such as TV conference system, this delivery task is a typical application and requires complex collaboration that makes full use of multi-agent programming. Through this task, we describe the characteristics of a smart office and how such a smart office can be realized using MRL. Furthermore, we clarify the usefulness of multi-agent programming to design a smart office based on our experience.

This paper is organized as follows. Section 2 provides related works to emphasize the unique feature of a smart office. Section 3 presents the system structure and how we achieve collaboration between robotic agents for the delivery task. We also describe an experimental environment for the smart office with robots and sensors, and we demonstrate the necessity of multi-agent modeling for the smart office. Section 4 explains the features of MRL and clarifies how to realize the delivery task using MRL. We describe how to extend a concurrent logic programming framework and how we define and execute robotic agents as MRL programs. Section 5 summarizes the usefulness of multi-agent programming to construct the smart office, based on our experiences in realizing delivery tasks. Section 6 reports observations on how human beings work with the robots in the smart office, based on our experiences working everyday. Section 7 describes how multi-agent programming can be applied in many cases to our everyday activities. Section 8 offers our conclusions

2. Related work

Research into computer systems that support our everyday activities is very important in terms of our efficiency and the augmentation of our abilities. This research reveals that there are many devices embedded in computers in our environment; devices that monitor

[4] KL1 is a parallel logic programming language designed in Japanese Fifth Generation Computer Systems Project [36].

our activities and work as mediators in our collaboration work. Several researches into support of everyday activities have been proposed. For Brooks' The Intelligent Room Project [1], several cameras monitor human beings and recognize our speech and gestures, and therefore the system can provide office information and enable an office guidance service. In this project, we do not access computer terminals; the system monitors the states of humans by human tracking, for a human–computer interaction.

The Ubiquitous Computing Project at Xerox PARC [37] designed hardware using hundreds of computers wirelessly connected with each other to support office activities. The group meeting tools, CoLab [34] and Liveboard, and the hand-held tools, ParcTab and the Pad, were the developments in this project. The goal of this project is to develop invisible devices for us by expanding these tools. These devices are things that we can access everyday and everywhere to support individual or group activities.

These projects have many factors in common with our smart office; the motivation, i.e. support for our everyday lives, is the same. However, we regard the robots and sensors as agents in the smart office and place our emphasis on creating intelligent agents. Our agents have a set of rules that describe the services that the agent can provide, and the agent decides whether to collaborate with other agents according to the service specified by the rule set. If an agent can request a task to several agents, the agent finds other agents to collaborate with each other by negotiation and decision making based on the rule set.

Intelligent behavior by agents is obviously important to support our activities. However, one characteristic of a smart office is that it realizes such behavior using AI technologies. Although the above projects could realize computer devices that behave intelligently, a symbol processing approach based on AI is necessary to clearly describe the collaboration among agents. Jini [35], proposed recently by Sun Microsystems, utilizes a look-up function to report available services, and the integration among devices is based on the framework of *grouping*. Our framework is based on agent negotiation and the agents realize collaboration by providing the available services to each other. We used multi-agent programming language to design the smart office, and this enables us to clearly express the way of collaboration among agents. The smart office was designed to prove the hypothesis that AI technologies can make the information appliances around us intelligent and can provide the means to support our everyday activities.

3. The smart office

The smart office is a future office where various information appliances, such as robots, camera, sensors, projectors, blinds, screens and lights are connected with each other by a network, and we can control these information appliances in the same way. This office is an experimental environment to clarify how we should make these information appliances cooperate with each other to support our activities. We constructed some prototype systems and conducted empirical studies in the smart office. Among the systems, the document delivery and printed out document delivery robot system work everyday in this environment. In this section, we describe the environment and the hardware structure for the delivery task. After that, we explain how we realize the delivery task.

3.1. Environment

To facilitate our experiments, a smart office environment was constructed on the second floor of the Information Media Center, Science University of Tokyo. The floor is normal and there is no unusual architecture. Fig. 1 shows the Information Media Center building. The building has an overall area of 2047 square meters and consists of three floors. The first and second floors are the experimental environment; the delivery task is executed on the second floor. Fig. 2 shows a map of the second floor.

There are eleven personal laboratories on the north side, and a meeting room and lounge for discussions on the south side. There are four mobile robots that deliver documents among the laboratories and deliver printed papers from two printers to the laboratories. The arc in Fig. 2 represents a path for a mobile robot. The mobile robots move along the arcs; they are programmed to avoid collisions by communicating with each other.

Fig. 1. The front view of Information Media Center which aims to develop advanced Artificial Intelligence software.

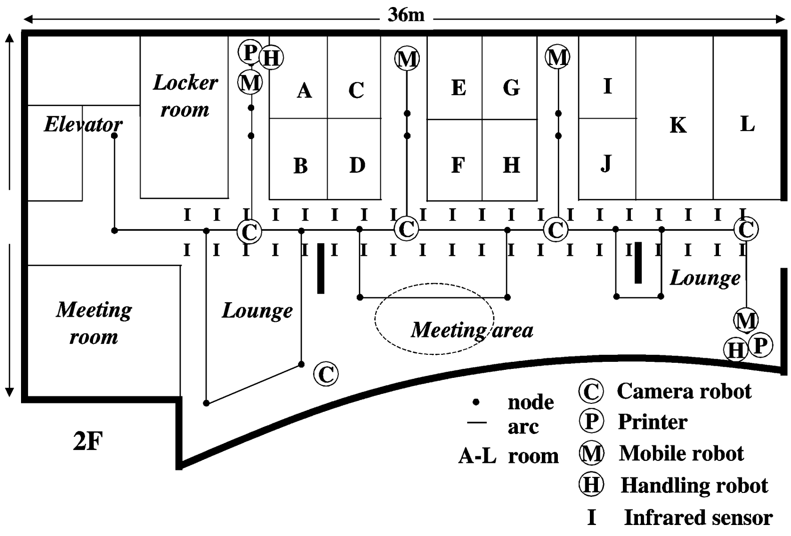

Fig. 2. The map of the smart office.

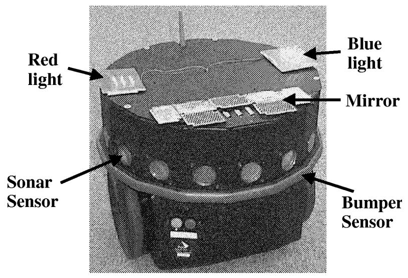

Fig. 3. Delivery robot. This is an extension of Nomad Scout developed by Nomadic company. The two lights on Nomad Scout are for image processing, and the mirrors are used for infrared sensors to enable the robots to assess their exact position. A user touches the bumper sensors to receive documents. Sonar sensors are used for collision avoidance.

Four delivery robots have trays on which to carry the documents and printed papers, and they also have mirrors and lights, as shown in Fig. 3. The mirrors are used to localize the robot by forty infrared sensors on the ceiling and the lights facilitate accurate recognition of the robot's position through image processing. This allows the mobile robots to receive printed papers from the handling robots.

The mobile robots are connected to the workstation through a wireless LAN; the camera and the handling robots are connected through an RS-232C interface. The infrared sensors are connected to the distributed device control network, and communication with the workstation is through a gateway. Printing jobs are monitored by a light sensor that detects an LED signal from the printer. The light sensor also communicates with the workstation through the distributed devices' control network. This allows synchronization between the printers and handling robots.

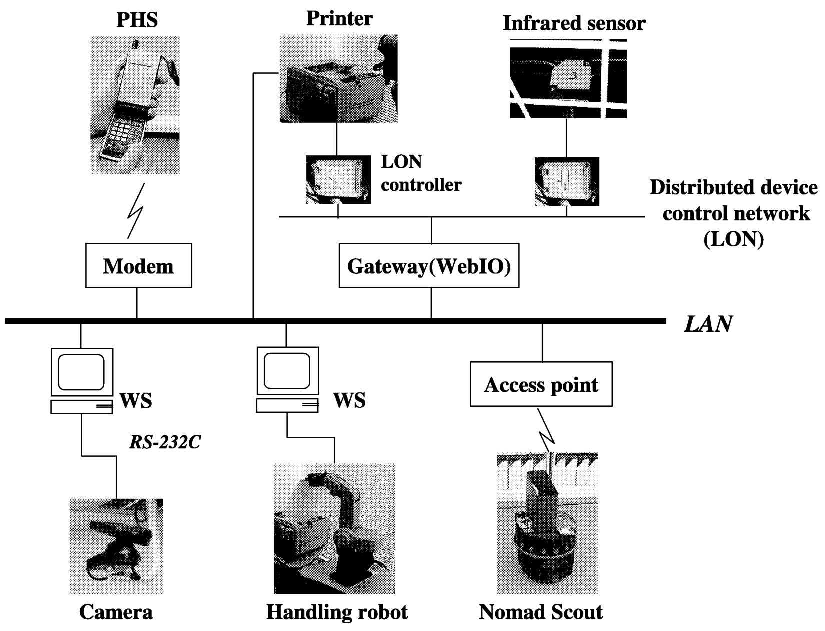

Fig. 4. Hardware at the smart office. Infrared sensors and printer monitoring sensors are connected through LON network developed by Echelon company. These sensors are also connected to LAN through a gateway called WebIO, the product of Echelon.

All robots and sensors in the smart office are connected to workstations and they are each controlled by robotic agents working directly or indirectly on several workstations. Users can access the delivery robot system running on a workstation through a personal computer or a cellular phone (PHS) and request delivery tasks to the system. Fig. 4 shows the hardware in the smart office.

These robots and sensors are not specially designed for a smart office; they are all commercial products. We use device drivers that are developed using Java by us or are provided by their production companies. Such drivers can be used by using socket and serial communications. This indicates that the smart office can be constructed using standard devices and software; therefore, a special infrastructure is not necessary for the smart office design.

3.2. Document (printed out) delivery robot system

The delivery robot system performs the task of carrying documents and printed papers in the experimental environment described above. For documents, a delivery robot goes to a person who requests the delivery task and receives the document in the tray. The robot then delivers the document to the specified place. For printed papers, a handling robot picks up the printed papers from a printer that is not busy and puts the printed papers in the tray on a delivery robot. The delivery robot then delivers the printed papers to the client.

The robotic agents utilized to realize the above tasks are as follows.

– *Delivery task agent*. This agent executes the delivery tasks. All delivery tasks are requested to this agent.
– *Sensor monitoring agent*. This agent monitors the sensor states of all the infrared sensors and the light sensor that detects printer jobs.
– *Handling robot agent*. This agent picks up printed papers from a printer and puts them in a delivery robot's tray.
– *Mobile robot agent*. This agent delivers documents and printed papers in its tray to users.
– *Camera robot agent*. This agent navigates the mobile robots to the position of the printer.
– *Printer agent*. This agent monitors the state of the printer.

When the delivery task agent receives a delivery task request, the agent requests each sub-task to all other agents and they collaborate with each other to achieve the delivery task. As for task complexity, the document delivery task becomes a subclass of the printed paper delivery task and we will now describe the process of the printed paper delivery task step by step; we classified the processes into four types, *communication*, *negotiation*, *decision*, and *execution*.

Step 1. Communication
The delivery task agent receives a task request from a user.

Step 2. Negotiation
The delivery task agent requests a printed out task to all printer agents.

Step 3. Negotiation and decision
The delivery task agent receives accept messages from the printer agents. The delivery task agent selects one printer agent and commits the printed out task to that printer agent. In addition, the delivery task agent requests a delivery task to all mobile robot agents.

Step 4. Execution
The committed printer agent starts printing the document.

Step 5. Negotiation and decision
The delivery task agent receives accepts messages from the mobile robot agents, and selects one mobile robot agent, then commits that mobile robot agent to the delivery task.

Step 6. Communication and execution
The committed mobile robot agent accurately moves to the position of the printer while communicating with a camera robot agent.

Step 7. Communication
After the mobile robot agent arrives at the printer, the mobile robot agent requests a pickup-and-place task to the handling robot agent.

Step 8. Execution
The handling robot agent puts the printed papers in the tray on the mobile robot, after determining that the printing has ended.

Step 9. Execution
The mobile robot agent delivers the printed papers to the user.

Communication is the sending and receiving of a message between two agents, and *negotiation* is the process of broadcasting a message from one agent to many agents. *Negotiation* includes three types of messages (*request*, *accept*, and *commit*). A task is committed to one agent from among several agents.

The *decision* is made to select one agent from among several agents, depending on the services that the agents provide. *Execution* involves sending commands to robots and sensors for control. Thus, *negotiation*, *decision*, *communication*, and *execution* are combined in the delivery task. These steps must be integrated in order to achieve a delivery task. In addition to the above example, when an agent receives a task request, a decision is required to determine whether the agent can perform the task. Therefore, an agent must have a rule set regarding tasks that the agent can do by itself; the agent is aware of the services it can provide. Such decisions are made in the process of agent negotiation.

Fig. 5 shows a graphical user interface to request a delivery task. Five camera images from camera robot agents are displayed in the upper part and a map of the second floor is provided in the lower part. Users can select document delivery or printed paper delivery using this interface. If document delivery is selected, the user specifies the objective room and a mobile robot comes to pick up a document, and then delivers the document to the specified room. If printed paper delivery is required, users do not need to specify a specific

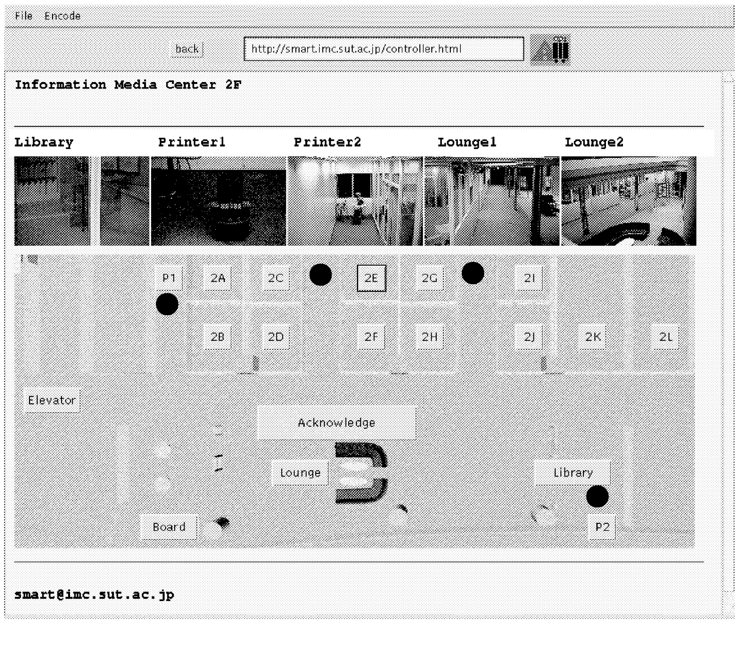

Fig. 5. Web-top controller for mobile robots. Five camera views from the camera agents are shown. The lower GUI is used to request a delivery task. A map is displayed on the GUI. We can request a task by pushing the buttons associated with a worker's room. The black circles indicate the positions of four robots.

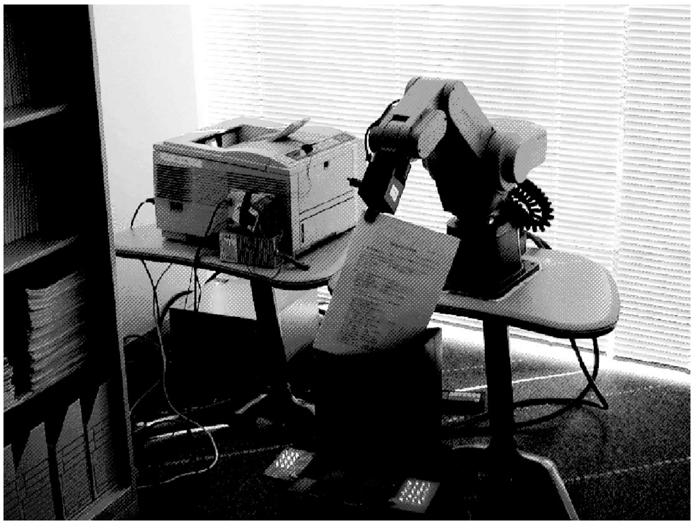

Fig. 6. Cooperation between a handling robot and a mobile robot.

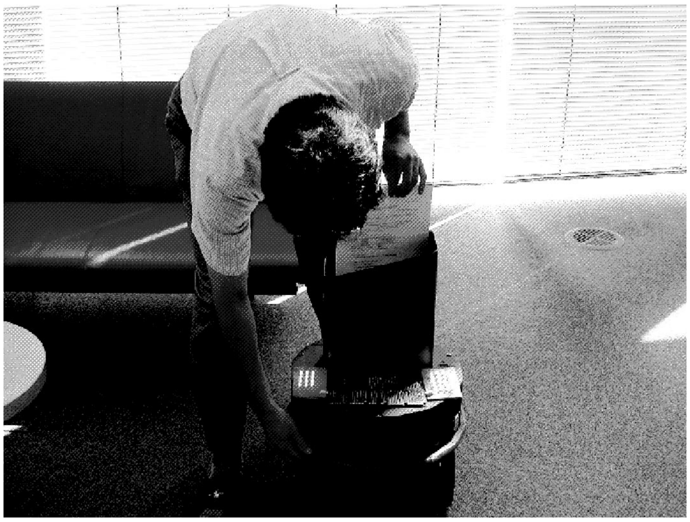

Fig. 7. Interaction between the mobile robot and a user.

printer. Papers are printed out from one of two printers and a mobile robot delivers the printed papers.

Figs. 6 and 7 show the execution of the delivery task. In Fig. 6, a handling robot grasps the printed papers and succeeds in putting the papers in the tray on the mobile robots. In Fig. 7, the mobile robot has arrived at the user who requested the delivery task. The user has received the papers and he is pushing the bumper to signify the end of the delivery task. The mobile robot executes the next task after finishing such a series of motions.

One obvious characteristic of the delivery robot system is the integration of heterogeneous agents and homogeneous agents. An increase in agents for situations requiring cooperation among homogeneous agents allows us to dispatch many tasks to many agents and results in an improvement to the agent system. If cooperation is required among heterogeneous agents, each agent must function appropriately to execute a task. In contrast, only one agent can function appropriately to execute a task among homogeneous agents. This is a distinctive feature of multi-agent collaboration in the smart office.

4. Agent collaboration based on multi-agent programming

Multi-agent programming provides a means to achieve a given task by collaboration among agents. Since each robot and sensor works individually when there is no cooperation nor competition, the robotic agents controlling each robot and sensor basically work in parallel.

In order to realize collaboration among agents that work in parallel, communication among the agents is necessary. We capture the communication as the following two concepts.

Cooperation = Action-level communication among robots and devices.
Several robots and sensors work concurrently based on synchronization.

Collaboration = Negotiation-level communication among robotic agents.
One-to-one or one-to-many communication for task commitment.

Furthermore, cooperation and collaboration are refined by the following functions.

Cooperation:

– *Cooperation by synchronization*. Synchronization is required when several robots work cooperatively as for matching the start and end of action executions of each agent. For example, if a camera which has a pan-tilt function is to navigate a mobile robot to an accurate position, the mobile robot and the camera must be synchronized with each other. Such synchronization is also required when a handling robot puts printed papers into the tray on a mobile robot.
– *Priority handling for emergent situations*. Sometimes a requested task should be canceled and a new task requested with higher priority. In this case, the current task being executed should be stopped. Such an interruption occurs when an emergent task is given. This also occurs when a robot fails to execute an action. In this case, the agent deals with the error and announces the status to other agents working cooperatively. Unlike a single agent, a multi-agent system requires a mechanism to cope with emergent situations.

Collaboration:

– *Task commitment by negotiation*. If there are several agents which can perform a given task, it is necessary to select one of the agents through negotiation. More specifically, the task is announced to all the agents using a broadcast message, and one agent is committed to perform the task. This type of negotiation is derived from the contract net protocol proposed by Smith [33].

When a conflict occurs among several agents, the conflict should be resolved by negotiation. When two mobile robots pass on the same path at the same time, one robot should vacate the path to avoid collision. This means mutual exclusive control. In this case, a supervisor agent playing a role of mediator is necessary to select one agent.

– *Decision*. We can regard it as decision making based on a rule set in which the agent judges whether the agent can perform a requested task and the agent selects one agent of all the agents that accept the task. In particular, non-trivial tasks require planning function. Multi-agent programming should cover such an intelligent function.

These functions are necessary to achieve a delivery task in the smart office mentioned in the previous section. In order to implement these functions, we developed MRL as an instance of an important class of multi-agent programming languages. MRL is based on concurrent logic programming, and supports agent negotiation, decision making and control. In the following sections, we present how to describe robotic agents constructing the smart office and how the robotic agents perform their actions after we give an outline of MRL.

4.1. MRL

MRL extends concurrent logic programming language to make it suitable for multi-agent systems. Concurrent logic programming is logic programming that allows reactive execution in the event of environment changes and concurrent execution of multiple processes. Typical languages in concurrent logic programming are Concurrent Prolog [29], Parlog [3], GHC [36] and CP [28]. These languages provide synchronization function by employing *don't care* nondeterminism, whereby once a computation transition is committed to a selected clause, no backtracking occurs to explore other alternative clauses. Therefore, they are called committed-choice languages (CCL). A guarded Horn clause has the following form:

$$H \texttt{ :- } G \mid B_1, \ldots, B_n, \tag{1}$$

where H, G, and B_i are atomic formulas, H is *Head*, G is *Guard* and $B_1, \ldots, B_n$ are *Body*. *Guard* is a condition to select the clause. This is similar to a guarded command introduced by Dijkstra [5]. A robotic agent consists of a set of guarded Horn clauses, and an MRL program consists of a set of robotic agents.

An agent is invoked by giving a set of goals being executed. These goals are replaced by the clause (1). Goal A is replaced when goal A matches *Head* of the clause (1) by the substitution θ and $G\theta$ is true. When goal A is replaced by clause (1), the replaced goals are $B_1\theta, \ldots, B_n\theta$. Generally, when the goals $A_1, \ldots, A_i, \ldots, A_m$ and A_i are replaced by the clause (1), the new goals are $A_1\theta, \ldots, B_1\theta, \ldots, B_n\theta, \ldots, A_m$.

If there are multiple guarded Horn clauses that can replace a goal, one of the clauses is selected. If there is no guarded Horn clauses that can replace a goal, however, the replacement of the goal is suspended. This represents a situation in which $G\theta$ does not become true even though goal A matches *Head*. This suspension is released when other goals are substituted for the variable to make $G\theta$ true. MRL realizes synchronization among agents using this function of concurrent logic programming.

We can regard clause (1) as a rule representing a state transition of an agent. If the current state of an agent matches H and the condition of the guard is satisfied, $B_i (1 \leqslant i \leqslant n)$ becomes the goals the agent must execute. This rule is explained by the value of n as follows. If $n = 0$, there is no action the agent must execute, and this means that the agent disappears. If $n = 1$, the current state of an agent transits to state B_1. If $n > 1$, multiple processes are running concurrently in an agent.

A state transition of an agent in MRL is usually defined in the following form:

$$\texttt{run(State):-}\ \texttt{G} \mid \texttt{B}_1, \ldots, \texttt{B}_{n-1}, \texttt{run(NewState)}. \tag{2}$$

where the predicate `run` is defined recursively. When guard `G` becomes true, the state transition occurs, and goals $\texttt{B}_1, \ldots, \texttt{B}_{n-1}$ and `run(NewState)` are executed concurrently. The variables `State` and `NewState` represent the states of the agent when a new state (`NewState`) is generated from the current state (`State`) within the goals $\texttt{B}_1, \ldots, \texttt{B}_n$.

Once a clause is selected for goal reduction, other alternatives are never explored. This indicates that concurrent logic programming inhibits backtracking, and there is no proof procedure built into logic programming. In contrast, MRL introduces a sort of theorem prover as a built-in function so as agents make decisions from a set of logical rules. A model generation theorem prover developed by Fujita and Hasegawa provides an efficient implementation using a concurrent logic programming language [11]. We incorporate this type of theorem prover into MRL. Underlying the theorem proving function, an agent possesses a set of logical rules, and each rule has the following implication:

$$A_1, \ldots, A_n \rightarrow B_1; \ldots; B_m \tag{3}$$

where A_i and B_j are atoms, "," is the *and* connective and ";" is the *or* connective. The left and right hand side of "$\rightarrow$" is the antecedent and consequent, respectively. If the antecedent is empty, we put $A_1 = \mathit{true}$ and such implication is called a positive clause. If the consequent is empty, we put $B_1 = \mathit{false}$ and the implication is called a negative clause. The theorem prover of MRL generates a set of candidate models starting from the positive clauses and eliminates irrelevant models by checking the negative clauses. In MRL, a query is put in guards, and the theorem prover is invoked in checking the guard.

MRL supports agent generation at run time, and the built-in agent generation procedure is called in the body part of a guarded Horn clause. The calling pattern has the following form:

$$\texttt{\#AgentClass} : \texttt{new}(\texttt{p}_1, \ldots, \texttt{p}_n) \tag{4}$$

where `AgentClass` is the name of the agent class, and $\texttt{p}_i$ has a parameter value for agent generation.

The agent calling goal (4) can be regarded as the super-agent of a newly generated agent. In contrast, the new agent is a sub-agent, and a set of agents that have the same super-agent is regarded as sibling agents. A sub-agent can also generate its sub-agents. This means that the agent generation framework in MRL constructs a hierarchical structure of agents. MRL supports a simple realization of agent hierarchy suggested by Minsky [21].

As already mentioned about multi-agent programming, a given task is broadcast to all agents that can perform the task. An agent transmits the task to other agents, receives a set of accept messages from the agents, and commits the task to just one agent. In this message passing framework, multi-agent programming should support communication to multiple agents and a particular single agent. Based on the agent hierarchy, each agent communicates with its neighboring agents (super-agent, sub-agent and sibling agents). To handle such agent communication, MRL provides the following built-in message passing statements:

– `A.Msg` (communication to a specific agent A),
– `^Msg` (communication to the super-agent),
– `*Msg` (communication to the sibling agents),
– `!Msg` (communication to the sub-agents)

where `Msg` represents the message content. For a guarded Horn clause, the guard is checked from the current state of computation, and thus the guard is executed in a passive fashion. In contrast, atoms in the body part are used for state transition, and thus the body is executed in an active fashion. In this sense, a message statement in the guard is used for message receiving, while a message statement in the body is used for message sending. This treatment is consistent with the framework of concurrent logic programming.

For example, the following guarded Horn clause states that if the underlying agent receives a message (`Receive`) sent from one of the sibling agents, it then sends the message `Send` to the sub-agents.

```
    p(Receive, Send):- *Receive | !Send.
```

Fig. 8 shows a simple MRL program which describes a camera agent. This agent performs a task in which the agent focuses on the specified node. The declaration of the agent starts with the statement `:- agent(AgentClass)`. The program consists of two agents that are defined as `camera_agent` and `camera_controller`. While the `camera_agent` agent negotiates with other robotic agents mentioned as normal agents, the agent `camera_controller` calls libraries to control the associated camera. The latter agent plays the role of an interface between robotic agents and existing control software.

The sub-agent of a robotic agent calls libraries for each device control. We call this type of agent *robot controller* to distinguish the agents with their robotic agents. In Fig. 8, the `camera_controller` agent is a robot controller.

The predicate `new` is called to generate an agent, and the clause in line (2) is called to generate a `camera_agent` agent. Usually, the body in a `new` predicate definition includes goals for sub-agent generation (`#camera_controller:new` in the program) and for calling the `run` predicate describing agent state transitions. Since these goals are called in parallel, the camera agent and the camera controller are executed concurrently.

The predicate `run` is defined recursively, and its argument represents the state of an agent. In the above program, the initial state of the agent is `null`, and the next state will be `doing(Task)` if the agent has the task being performed. The first clause (line (3)) of the `run` predicate definition states that the agent receives the

```
(1)  :- agent(camera_agent).
(2)   new:- #camera_controller:new, run(null).

(3)   run(null):- *task(focus_on(Node)), ?visible(Node)  |
(4)         !do(focus_on(Node)),
(5)         run(doing(focus_on(Node))).

(6)   run(doing(Task)):- !done(Task) | run(null).

(7)   visible_region(R), Node ∈ R → visible(Node).

(8)  :- agent(camera_controller).
(9)   new:- camera_library:open(1), run(null).

(10)  run(null):-^do(focus_on(Node)) |
(11)        camera_library:do(focus_on(Node), End),
(12)        run(doing(focus_on(Node),End)).

(13)  run(doing(Task,End)):- End = ok |^done(Task), run(null).
```

Fig. 8. The definition of an camera agent in MRL.

task `task(focus_on(Node))` sent from one of the sibling agents. The task is a request for the agent `camera_agent` to focus on the specified node. The statement `?visible(Node)` is a query to confirm that the camera can focus on the node. Whether the query is true or false depends on the logical rule in line (7). If the condition of the rule is true, the query `?visible(Node)` becomes true. The theorem proving function of MRL is used to handle this query.

Once the query is proven true, MRL executes the goals in lines (4) and (5). In line (4), the camera agent sends the command `do(focus_on(Node))` to its sub-agent `camera_controller`. The command is handled within the guard in line (10) and the existing library is called in line (11). In calling the library, the variable `End` is unbound. After the camera executes the command, `End` is instantiated to the constant `ok`.[5] Line (13) reports the end of the execution to the super-agent.

As shown in the camera controller, the function of the robot controller is to call the library to control the target device. The timing of calling the library is determined by messages sent from the robotic agents. The library controls the device and reports the end of the execution to the robot controller. Since existing libraries are not defined like this, we should wrap the libraries to communicate with MRL programs. This is one of the

[5] The constant `ok` means that the action is executed normally. Another constant `no` will be used as an abnormal execution in emergent message handling.

three methods (transduce, wrap and rewrite) proposed by Genesereth to agentify existing software [14].

The program in Fig. 8 shows that an agent is defined by combining a set of guarded Horn clauses with logical rules in MRL. A guarded Horn clause deals with agent communication and specifies from action-level to negotiation-level interaction between agents. In contrast, logical rules specify what capabilities or services an agent provides, and are used within negotiation. In the following, we will describe the multi-agent programming functions supported within MRL and clarify how the delivery task is realized by the MRL program.

4.2. Synchronization

Synchronization is used to specify cooperative actions of robots and sensors where the associated robot controllers (not robotic agents) communicate with each other. Most existing libraries provide stand-alone control of robots and sensors in which communication is not supported. We regard such actions that are executed by calling the library as *atomic* actions, and there is no communication between other robot controllers within an atomic action. For example, a mobile robot has atomic actions such as rotation, forward and backward, and performs a given task (e.g., goto-node) by combining such atomic actions. We wrap this atomic action library to allow communication between atomic actions. This enables us to invoke atomic actions by sending and receiving messages, resulting in synchronization between robot controllers. An atomic action is called in the body of a clause within a robot controller like this:

```
library:do(AtomicAction,Start,End)
```

where `Start` and `End` are variables. When `Start` is instantiated to the constant "`ok`", the action specified at `AtomicAction` starts and `End` will be instantiated after the action finishes. In general, other robot controllers and goals (calling library) share the variables `Start` and `End`. The robot controllers communicate with each other by the shared variables. A robot controller also controls synchronization between atomic actions using the shared variables.

A robot controller reduces an action command sent from its robotic agent to a series of atomic actions. This enables the robotic agent to execute macro commands. For example, we specify the following command to pick-up print-out paper:

```
do(pickup_paper,Start,End)
```

The handling robot agent sends the command to the handling robot controller. The following guarded Horn clause is used to handle this command:

```
%handling_robot controller
 run:-^do(pickup_paper,ok,End) |
    handling_robot_library:do(move(printer),ok,C1),
    handling_robot_library:do(grasp,C1,C2),
    handling_robot_library:do(move(home),C2,End).
```

The assignment of shared variables to the body part indicates that the handling robot moves to the printer (by `move(printer)`), grasps the print-out paper (by `grasp`), and moves to the home position (by `move(home)`) sequentially. After finishing the series of actions, the variable `End` is instantiated to the constant `ok`.

The synchronization between the printer and the handling robot for the delivery task can be done by such a shared variable assignment. To implement this kind of synchronization, we have the following agents:

```
%handling_robot agent
 run:-^do(print_out_handling, ok, End) |
   *do(print_out_check, ok, C1),
   !do(pickup_paper, C1, C2),
   !do(put_paper, C2, End).

%printer agent
 run:- *do(print_out_check, ok, End) |
   !do(print_out_check, ok, End).

%printer controller
 run:-^do(print_out_check, ok, End) |
   printer_library : do(print_out_check, ok, End).
```

After receiving the command `print_out_handling`, the handling robot agent sends the command `print_out_check` to the printer agent (one of the sibling agents), and sends the commands `pick_up_paper` and `put_paper` to the handling robot controller (the sub-agent). These commands are sent in parallel. According to the order of the instantiation of the variables, the sub-agents (the robot controller and the print controller) call the libraries to check whether the paper is printed out, pick-up the paper and put the paper on the mobile robot.

By combining multiple atomic actions and synchronization based on shared variables, it is possible to construct composite actions such as `print_out_handling`. Such actions are abstract and close to the intention-level commands we often use to communicate with other people. As Rosenschein and Shoham proposed, the basic idea of agentifying devices is to reduce an intention-level command to a set of action-level commands that are executed by a machine [27,30]. Logic-style programming based on the guarded Horn clause allows us to construct higher-level action commands that are executed concurrently within multiple devices by combining atomic actions and synchronization. This function provides a means of making the devices intelligent.

4.3. Emergent event handling

An emergent event occurs when a user and other agents cancel a previously assigned task or errors occur in controlling a device. Since such an event occurs after committing the task to another agent or during action execution, it is difficult to handle the event based on the synchronization function mentioned so far. Suppose that an abstract command given to a robotic agent is reduced to a set of atomic actions that are not executed yet. In this case, we need a function to inform these actions of the emergent event. This function allows dynamic control of robotic agents and provides a basis to support situation-oriented action execution. It makes the devices more intelligent so they can cope with emergent events in executing abstract commands.

In order to inform atomic actions of an emergent event, we prepare a variable representing the occurrence of the emergent event. An atomic action is specified as follows:

```
do(AtomicAction, Start, End) + E
```

where the variable `E` indicates whether the atomic action should be canceled or not. If the variable is instantiated to the constant `cancel`, some atomic actions that are not executed yet can be canceled.

The program of the printer controller defined so far is rewritten as follows:

```
%printer controller
 run:-^do(print_out_check, ok, End) + E |
    do(print_out_check, ok, End) + E.

 do(Action, Start, End) + cancel:- true | End = no.
alternatively.
 do(Action, ok, End) + E:- true | printer_library :
    do(Action, ok, End).
 do(Action, no, End) + E:- true | End = no, E = cancel.
```

where the predicate `do` is introduced to recognize whether the variable `E` is instantiated to `cancel` or not. The first clause of this predicate definition instantiates the variable `End` to the constant `no`, when `E` is bound to `cancel`. This situation means that the action `Action` was executed abnormally. The second clause states a rule when there is no emergent event. The rule calls the associated library for action execution. The third clause is a rule when the variable stating the start of action execution is bound to `no`. This case occurs when the previous action was executed abnormally. The rule cancels the execution of the next action.

The `alternatively` statement under the first clause specifies the priority relationship between clauses. Upper clauses over the statement have higher priority than lower clauses. [6] A clause with higher priority can be selected in MRL. In contrast, a clause with the same priority is selected non-deterministically. In the above program, this priority control selects the first clause for emergent event handling.

An emergent event source is either a task cancelled by a user or an error during action execution. Task cancellation is propagated from a robotic agent to its sub-agents, while the error is propagated from the robot controller to the robotic agent. The second clause in the `do` predicate calls the library for the printer control. If an error occurs within the library, the variable `End` is instantiated to `no`. The third clause recognizes this error, and the emergent

[6] Such priority is directly used for KL1 language [2].

event variable E is instantiated to `cancel`. After that, the error can be propagated to the robotic agent.

Although an emergent event is propagated bi-directionally, both cases can be handled using the same set of clauses. This is because event occurrence is represented as a logical variable, and the event is propagated done by unification in logic programming.

In the above program, there is only one atomic action. If a given task is composed of multiple atomic actions, an emergent event variable is shared within the actions, and its event propagation can be broadcast to all the actions. The handling robot controller illustrates this case and is defined by the clause

```
%handling_robot controller
 run:-^do(pickup_paper, ok, End) + E |
    do(move(printer), ok, C1) + E,
    do(grasp, C1, C2) + E,
    do(move(home), C2, End) + E.
```

where the variable E is shared within the three actions, we can cancel actions that are not executed yet. This function is based on the expressive power of logic programming, and our logic-based agent programming enables us to design a sophisticated message passing protocol.

4.4. Agent negotiation

The purpose of agent negotiation is twofold. The first is to select an agent that can perform a given task. The other is to resolve the conflict between agents that execute different actions for different tasks. These situations occur simultaneously in most multi-agent systems. It is important to design negotiation protocols to deal with this problem.

The process of selecting an agent committed to the task consists of task request, task acceptance, and task commitment. To perform the task, the agent broadcasts a task request to all agents, receives several accept messages, and sends a commit message to one of the agents that reply with the accept messages.

Conflict between agents is resolved by their super-agent. Each sibling agent requests the permission to execute an action from its super-agent. A *before* message is used for this purpose, and an unbound variable is attached for reply to this message. After executing the action, the selected agent reports the end of the action to the super-agent via an *after* message. The super-agent checks whether a conflict occurs or not, when receiving the before message.

Table 1 shows the vocabulary for agent negotiation, and includes message formats and contents. The item Send/Receive specifies that the message is for sending or receiving. A sending message is put in the body of a clause, and a receiving message is put in the guard. The emergent variable E in a `commit` message is used to cancel a task commitment. Suppose that agent a_1 sent a `commit` message to agent a_2 and then received another `accept` message from agent a_3. To commit the task to agent a_3, we can cancel the task commitment to a_2 by instantiating the variable E to the constant `cancel`. This is the same framework as in emergent event handling.

Table 1
The vocabulary of Negotiation protocol. `+` and `-` indicate input and output

Send/receive	Recipient	Message
Send/receive	Any	`request(+Sender,+Task)` is used to send a task (`Task`) to agents.
Send/receive	Any	`accept(+Sender,+Task,+Cost)` is used to reply to a request message and sends the agent (`Sender`) the cost of task execution.
Send/receive	Any	`commit(+Task)+E` is used to reply to an accept message and indicates the commitment of a task execution. `E` is used to cancel a previously committed task.
Send	Super	`before(+Action,-Return)` requests permission of an action from its super-agent. The result is reported on `Return`.
Send	Sub	`do(+Action,+Start,-End)` performs an action using `Start` and `End` variables.
Send	Super	`after(+Action)` reports the end of an action.

The variables `Return` and `End` in the `before` and `do` messages are used for reply and are unbound in sending the messages. These variables will be instantiated by an agent who receives the messages. The way to send messages with unbound variables and reply to the message by instantiating the variables is novel in logic programming. This type of a message is called an *incomplete message*, supporting bi-directional communication by sending only one message.

Fig. 9 implements a negotiation protocol using MRL. The program consists of defining the delivery task agent and the mobile robot agent. If a task is received from the super-agent, the delivery task agent broadcasts a `request` message in line (2). The message includes the task information and the agent itself (represented by `#self`). After receiving this message in line (7), the robotic agent sends an `accept` message to `Sender`. The delivery task agent receives the `accept` message in line (4), and then replies with a `commit` message. When it receives the `commit` message in line (10), the robotic agent starts the task execution. In line (11), the robotic agent requests permission to execute the task to the super-agent and sends a `do` message to the robot controller (sub-agent). After finishing the task execution, the robotic agent sends an `after` message to the super-agent in line (15). Fig. 10 illustrates the flow of agent negotiation.

The negotiation protocol in Fig. 9 is general and is used for any collaboration between all robotic agents in the delivery task. For example, the delivery task agent commits the document delivery to a mobile robot agent by using the three messages. A conflict example occurs when two delivery tasks have the same destination room, and only one mobile robot can go to the room. In this case, the super-agent chooses one of the mobile robots agent using the `before` and `after` messages.

Although the negotiation protocol in Fig. 9 is simple for explanation purposes, we can extend it to deal with both negotiation between robotic agents and control by robot controllers at the same time. This extension realizes the concurrent execution of negotiation and control using the MRL concurrent programming function. The extension is based on dividing an agent state into a negotiation-level state and an action-level state. We can define state transition rules for the negotiation-level and action-level state independently. Using these rules, the negotiation process and the action process do not interfere with each other.

```
     %delivery_task agent
 (1) run(null):-^task(Task) |
 (2)   *request(#self, Task),
 (3)   run(requesting(Task)).

 (4) run(requesting(Task)):- *accept(Sender, Task) |
 (5)   Sender.commit(Task) + E,
 (6)   run(requested(Task, E)).

     %robotic agent
 (7) run(null):- *request(Sender, Task) |
 (8)   Sender.accept(#self, Task),
 (9)   run(accepting(Task)).

(10) run(accepting(Task)):- *commit(Task) + E |
(11)   ^before(Task, Return),
(12)   !do(Task, Return, End) + E,
(13)   run(doing(Task, End)).

(14) run(doing(Task, End)):- End = ok |
(15)   ^after(Task), run(null).
```

Fig. 9. Implementing a negotiation protocol in MRL.

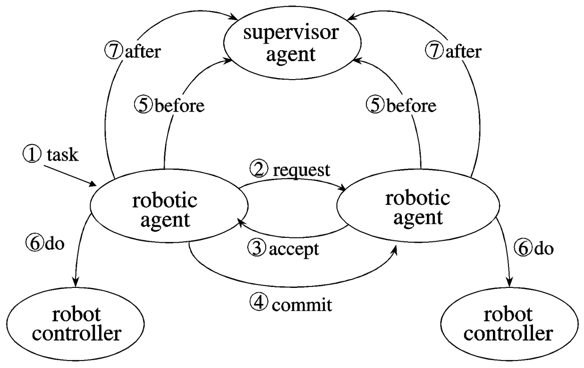

Fig. 10. Flow of agent negotiation. An arrow represents a message. A circled number indicates the order of message arrival.

We can interleave the action execution with the negotiation process, and vice versa. Section 4.7 describes how to realize such a concurrent function.

4.5. Decision making

Agent negotiation needs two types of decision making: one to determine whether an agent can accept a requested task or not; the other to select one of the agents that accept the requested task. In both cases, we pose a query for decision making based on a given set of logical rules an agent possesses. The theorem proving function in MRL is used for reply to the query.

Decisions are made in all cases of agent negotiation for the delivery task. Typical cases are in task acceptance and commitment in deciding a mobile robot agent for the task execution. A mobile robot agent has the following rule set:

$$\texttt{reachable}(\texttt{Node}) \rightarrow \texttt{acceptable}(\texttt{move\_to}(\texttt{Node})).$$

$$\texttt{staying\_at}(\texttt{N}) \rightarrow \texttt{reachable}(\texttt{N}).$$
$$[\texttt{move}(\texttt{N}_2)] : \texttt{reachable}(\texttt{N}_1), \texttt{link}(\texttt{N}_1, \texttt{N}_2) \rightarrow \texttt{reachable}(\texttt{N}_2).$$

These logical rules are used to determine whether the task `move_to(Node)` is acceptable or not. The predicate `reachable` represents the ability of the mobile robot agent to reach a specified node. If `reachable(Node)` is true, the mobile robot agent accepts the task. Here, the predicate `reachable` implements the transitivity law, and the predicate `link` means the connectivity between nodes. The left-hand side before the colon represents a sequence of actions. The rule says that if the antecedent part is true and the agent executes the actions, the consequent part will be true. Since `staying_at(N)` states that the current position of the robot is node `N`, `reachable(N)` is true with no action execution. In contrast, the last rule indicates that the robot will arrive at node $\texttt{N}_2$ after executing the action $\texttt{move(N}_2\texttt{)}$.

Given the above rule set, a query whether the task is acceptable or not is posed in the following guarded Horn clause:

```
%robotic agent
 run(null):- *request(Sender, Task),
     ?acceptable(Task) = Plan |
          Sender.accept(#self, Task, Plan),
          run(accepting(Task, Plan)).
```

where the second condition of the guard is the query part where `acceptable(Task)` is indeed a query and the variable `Plan` is introduced to return a sequence of actions. This query form is used not only for checking the task acceptability of the agent but also for producing the plan to perform the task.

Situation calculus [20] and Kowalski's formulation [16] are typical applications of theorem proving to planning. Such approaches represent the problem state explicitly and represent actions using function symbols in first-order logic. In our approach, actions are represented in the same manner, but the actions are separated from the relationships between state transitions. This description is close to STRIPS [7].

Another example of decision making is to select an agent from the agents that have accepted the task. As an agent selection criteria, we adopted the expiration-based selection for the delivery task. This selection is summarized as follows. A candidate agent returns an `accept` message within a prespecified time interval. If the message arrives after that, we assume that the agent can not accept the task and ignore the message. This selection is based on the expiration time proposed in the contract net. Expiration time avoids the situation in which the agent must wait for the replies from all the agents. Even if some agents have difficulties and can not reply to the messages, the total agent system is not influenced by those difficulties.

To compare the produced plans, we assume that a better plan has lower execution cost. A reasonable plan can be stated by the following rule:

```
plan(Sender, Plan), cost(Plan, Cost), reasonable(Cost) →
  reasonable_plan(Sender, Plan, Cost).
```

where `Sender` is an agent who produced the plan. The cost of the plan is estimated by the goal `cost(Plan,Cost)` and is checked to determine whether the cost is reasonable or not by the goal `reasonable(Cost)`. If these goals are true, the consequent part is true.

Based on the above rules, the following guarded Horn clause implements the expiration-based selection.

```
%task_management agent
  run(null):-^task(move_to(Room)) |
    *request(move_to(Room), #self),
    run(requesting(move_to(Room), #timer(500))).

  run(requesting(Task, Timer)):-
        *accept(Sender, Task, Plan),
        ?assert(plan(Sender, Plan)) |
    run(requesting(Task, Timer)).

  run(requesting(Task, ok)):-
        ?min(Cost, reasonable_plan(Sender, Plan, Cost)),
        ?retract(plan(_, _)) |
    Sender.commit(Task),
    run(null).
```

where `#timer` is unified with a variable. If the specified time passes, the variable is instantiated to the constant `ok`. This enables us to construct a program in which synchronization can be done at the specified time. The second clause of the predicate run is applicable when the variable Timer is unbound. In this case, the agent returns an accept message and asserts the plan the sender produced. The third clause can be selected after the specified time passed. A reasonable plan which is the minimum cost is selected by invoking the min predicate.

Such decision making can not be achieved by using guarded Horn clause only, as Kowalski claimed [17]. MRL solves this problem by combining guarded Horn clauses and logical rules agents possess. This combination allows us to realize intelligent and reactive agents.

4.6. State transition of agents

As shown in the previous sections, robotic agents in the smart office do not only execute actions but also negotiate with each other to perform a given task. It is possible for MRL to specify such functions of the robotic agents, and we model the behavior of the robotic agents as a set of state transition rules. In this modeling, we should record negotiation processes with other agents, in addition to the state of the action execution. To realize collaboration between agents, we classify an agent state into the following three states:

(1) *Task-level state*. This is a state related to task execution. If no task is executed, the state is null. Otherwise, the state is a requested task or its sub-tasks.
(2) *Action-level state*. This is a state related to action execution. If no action is executed, the state is null. If the agent waits for synchronization with other agents, the state is waiting. Otherwise, the state is the currently executing action such as moving and printing.
(3) *Negotiation-level state*. This is a state related to negotiation. If the agent does not negotiate with others, the state is null. When the agent requests other agents to perform the task, the state of the agent is requesting. When the agent has accepted a requested task, the state of the agent is accepting.

Table 2 shows the possible states of all robotic agents in the delivery task. The delivery task agent has given tasks as its task-level state. Since the agent does not execute actions, the action-level state is immaterial. The robotic agents for mobile robots, printers, handling robots and cameras have three levels of states. The task-level state includes move_to(Node) and monitor(print_out) for the mobile robot agent. The state monitor(print_out) means that the mobile robot agent monitors the print-out status using the print-out sensor. This is a sub-task of performing the delivery task. In the action-level state, a sequence of actions is executed for a task. For example, the two actions (waiting and moving) should be combined to perform the task move_to(node). In the negotiation-level state, the two states (requesting and accepting) are generally used. An exception is in the handling robot agent. Since this agent does not request others to perform a task, no requesting state exists. However, this agent is requested to perform a task (grasp the paper), the accepting state exists in the handling robot agent. The sensor-monitoring agent does not have the action-level state because the agent can execute only one action (sensor monitoring). The agent checks the negotiation-level state and performs the associated task (localize).

Table 2
The states of all agents for the delivery task

Agent	Task-level state	Action-level state	Negotiation-level state
Delivery task	null		null
	request(print_deliver)		requesting
	request(document_deliver)		
Mobile robot	null	null	null
	move_to(node)	waiting	requesting
	carry_to(node)	moving	accepting
	monitor(print_out)	moving(camera)	
	monitor(acknowledge)	moving(infrared)	
Printer	null	null	null
	print	waiting	requesting
		printing	accepting
Handling robot	null	null	null
	transfer	waiting	accepting
		transferring	
Camera	null	null	null
	navigate	moving	accepting
Sensor	null		null
monitoring	localize		accepting

The above three states constitute an agent's states. A transition rule refers to these state and specifies the behavior of the agent. According to the three levels of states, state transitions are invoked in the following situations:

– A task-level state is changed when the agent receives a task request or performs his own task (sub-tasks).
– An action-level state is changed when the agent sends commands to its sub-agents for action execution.
– A negotiation-level state is changed when the agent sends or receives messages.

The state change of an agent is determined by the three-levels of states and messages sent from other agents. This indicates that we put the three states and message receiving statements in the guard part and put a new state description in the body to define the agent behavior. The following clause specifies that a mobile robot agent commits a `localize` task to the sensor monitoring agent. The `localize` task is performed by cooperation among the mobile robot and the infrared sensors to localize the position of the mobile robot.

```
run(Task, Action, Negotiation):-
    Task = move_to(Goal),
    Action = moving(N₁, N₂),
```

$$\begin{aligned}
&\texttt{Negotiation} = \texttt{requesting}(\texttt{localize}(\texttt{N}_1, \texttt{N}_2)),\\
&*\,\texttt{accept}(\texttt{localize}(\texttt{N}_1, \texttt{N}_2), \texttt{infrared})\ |\\
&*\,\texttt{commit}(\texttt{localize}(\texttt{N}_1, \texttt{N}_2)),\\
&!\texttt{do}(\texttt{move}(\texttt{Goal}), \texttt{infrared}),\\
&\texttt{Action}' = \texttt{moving}(\texttt{infrared}, \texttt{N}_1, \texttt{N}_2),\\
&\texttt{Negotiation}' = \texttt{null},\\
&\texttt{run}(\texttt{Task}, \texttt{Action}', \texttt{Negotiation}').
\end{aligned}$$

The mobile robot agent has three state variables (Task, Action and Negotiation). The guard part states that the agent is currently executing the task move_to(Goal), that the agent sent the localize task request to other agents, and that one of the agents accepted the localize task. In the body part, the mobile robot agent commits the localize task to the agent who accepted the task. The action-level state is changed to moving(infrared,$\texttt{N}_1$,$\texttt{N}_2$), and the negotiation-level state is changed to null. The variable Task is not updated because the localize task is not achieved yet.

Every robotic agent in the smart office is defined as a set of the above transition rules. By combining possible states shown in Table 1, we find there are possibly 2800 rules for the print-out paper delivery. However, some rules are meaningless, for example, the situation in which the task-level state is null and the action-level state is not null. As the result, the print-out paper delivery consists of about 50 state transition rules.

Table 3 shows the process of the state transitions of a mobile robot agent for the printed-out paper delivery. Given a request from the delivery task agent, the mobile robot agent controls the associated robot so that the robot moves to the position of the printer (np), receives the printed-out paper, and then moves to the worker's room (n3). The task is completed by the user who touches the bumper sensor for acknowledgment. In this table, passing messages are described as from $\Rightarrow$ msg or msg $\Rightarrow$ to.

State (1) indicates that the delivery task agent sends the task delivery(n3) to the mobile robot agent. Given the task by commitment, the mobile robot agent changes the state (2) where the task-level state becomes move_to(n1). To perform this task, the mobile robot agent sends the command do(move(n1)) to the mobile robot controller and sends the request message request(localize) to the sensor monitoring agent. After that, the action-level state becomes moving and the negotiation-level state becomes requesting(localize). After receiving an accept message for the localize task from the sensor monitoring agent, the mobile robot agent commits the localize task to the sensor monitoring agent and sends the mobile robot controller the command do(move(n1),infrared). This command causes the mobile robot to move to the node in cooperation with the infrared sensor. Since the negotiation with the sensor monitoring agent ends at this stage, the negotiation-level state becomes null. When receiving the message done(move(n1)) reporting the end of arrival at the node, the agent recognizes the end of the execution of the task move_to(n1).

Continued state (3) is the mobile robot moving to the position of the printer. State (4) is the process of receiving a printed-out paper from the printer. States (5) and (6) are moving

Table 3
State transition of a mobile robot agent in the printed-out delivery task (mobile robot agent:a_r, handling robot agent:a_h, camera agent:a_c, sensor monitoring agent:a_s, delivery task agent:a_d, node:n_i, printer node:n_p)

No.	Task	Action	Negotiation	Messages
(1)	null	null	null	$a_d \Rightarrow$ request(deliver(n3))
				accept(deliver(n3)) $\Rightarrow a_d$
			accepting	$a_d \Rightarrow$ commit(deliver(n3))
(2)	move_to(n1)		null	do(move(n1)) $\Rightarrow r_c$
				request(localize) $\Rightarrow a_s, a_c$
		moving	requesting	$a_s \Rightarrow$ accept(localize)
				commit(localize) $\Rightarrow a_s$
				do(move(n1),a_s) $\Rightarrow r_c$
		moving(a_s)	null	$r_c \Rightarrow$ done(move(n1))
(3)	move_to(np)	null		do(move(np)) $\Rightarrow r_c$
				request(localize) $\Rightarrow a_s, a_c$
		moving	requesting	$a_c \Rightarrow$ accept(localize)
				commit(localize) $\Rightarrow a_c$
				do(move(np),a_c) $\Rightarrow r_c$
		moving(a_c)	null	$r_c \Rightarrow$ done(move(np))
(4)	monitor(print)	null		request(transfer) $\Rightarrow a_h$
			requesting	$a_h \Rightarrow$ accept(transfer)
				commit(transfer) $\Rightarrow a_h$
		waiting	null	$a_h \Rightarrow$ done(transfer)
(5)	carry_to(n2)	null		do(move(n2)) $\Rightarrow r_c$
				request(localize) $\Rightarrow a_s, a_c$
		moving	requesting	$a_s \Rightarrow$ accept(localize)
				commit(localize) $\Rightarrow a_s$
				do(move(n2),a_s) $\Rightarrow r_c$
		moving(a_s)	null	$r_c \Rightarrow$ done(move(n2))
(6)	carry_to(n3)	null		do(move(n3)) $\Rightarrow r_c$
				request(localize) $\Rightarrow a_s, a_c$
		moving	requesting	$r_c \Rightarrow$ done(move(g))
(7)	monitor(ack)	null	null	wait(bumper) $\Rightarrow r_c$
		waiting		$r_c \Rightarrow$ done(bumper)
(8)	null	null		

processes to node n2 and n3, respectively. In state (7), a user receives the paper and touches the bumper to acknowledge the delivery.

Note that there is no printer agent in Table 3. This agent negotiates with a handling robot agent, not a mobile robot agent.

Different state descriptions allow us to represent the state transitions for actions and negotiations independently. As the result, negotiation can be handled during action execution, and actions can be executed during negotiation. For example, in state (2), the mobile robot agent invokes an action of the robot controller $\mathtt{R}_c$ ($\mathtt{do(move(n1))} \Rightarrow \mathtt{r}_c$), and at the same time negotiates with the sensor monitoring agent ($\mathtt{a}_s$). After negotiation, the agent invokes an action of the robot controller ($\mathtt{do(move(n1,a}_s\mathtt{))} \Rightarrow \mathtt{r}_c$), and changes the state from `moving` to $\mathtt{moving(a}_s\mathtt{)}$.

Note that the mobile robot can move to the node `n1`, even if the mobile robot agent does not receive an accept message from the sensor monitoring agent. This is because the robot controller can execute the command `do(move(n1))` without the assistance of the sensors. In this stage, the negotiation-level state remains `requesting(localize)`. However, when the mobile robot agent receives the message `done(move(n1))` from the robot controller, the mobile robot agent recognizes the end of the task execution and makes the negotiation-level state null. Such a situation appears in state (6). The concurrent execution of action and negotiation is due to the concurrency in MRL.

5. The result

In the previous section, we presented how to describe the delivery task in an MRL program and how the MRL program was executed. We clarified that multi-agent programming functions of MRL allowed various robots and sensors to cooperate with each other. In this section, through our experience, we will clarify the usefulness of multi-agent programming for smart office design from a general viewpoint. This indicates the advantage of the multi-agent approach for designing cooperative systems for various information appliances including robots and sensors.

5.1. Agent approach to information appliances

In order to create intelligent information appliances, we used two features of multi-agent programming. One is that multi-agent programming allows constructing macro and abstract commands by combining atomic actions and synchronization. In a guarded Horn clause, an upper-level command is put in the head, and a set of lower-level commands are put in the body. Logical variables to control the command execution are appended to each command, so we can specify the execution order of lower-level commands by the way of assigning the logical variables. For example, the following commands are put in the body of a guarded Horn clause.

$$!\mathtt{do}(\mathtt{a}_1, \mathtt{ok}, \mathtt{C}_1), !\mathtt{do}(\mathtt{a}_2, \mathtt{ok}, \mathtt{C}_2), !\mathtt{do}(\mathtt{a}_3, \mathtt{C}_1, \mathtt{C}_2).$$

The clause to process the third command is defined as follows.

$$\mathtt{run\text{:-}\hat{}do}(\mathtt{a}_3, \mathtt{ok}, \mathtt{ok}) \mid \ldots$$

This is executed after the end of both $\mathtt{a}_1$ and $\mathtt{a}_2$. In the following case, this is executed after the end of either $\mathtt{a}_1$ or $\mathtt{a}_2$.

$$\mathtt{run\text{:-}\hat{}do}(\mathtt{a}_3, \mathtt{ok}, \mathtt{C}_2) \mid \ldots$$

$$\mathtt{run\text{:-}\hat{}do}(\mathtt{a}_3, \mathtt{C}_1, \mathtt{ok}) \mid \ldots$$

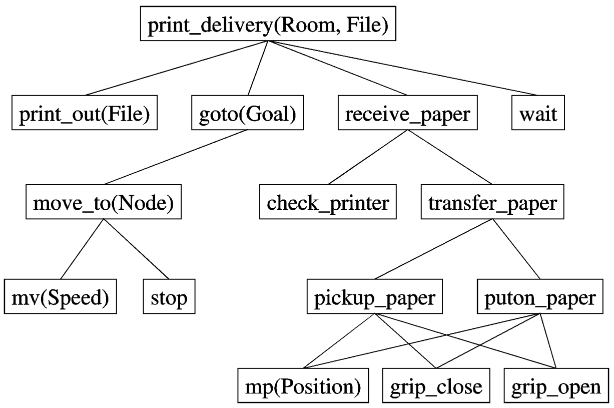

Fig. 11. Hierarchical structure of commands for the delivery task.

These examples indicate that the order of commands can be described using an AND-OR relation.

It is possible to define the hierarchical relation of commands and the relation of the execution order of commands independently. Fig. 11 shows the hierarchical relation of commands in the delivery task. An abstract command is hierarchically constructed from commands that control robots directly. A command is a combination of the lower-level commands, and the AND-OR relations of the execution order are specified within the lower-level commands. For example, `move_to(Node)` is defined by the combination of logical variables assigned to `mv(Speed)` and `stop`. The hierarchy and execution order of commands are defined declaratively, and a complex command executed concurrently can be easily constructed. Such a command execution can be controlled by any number of variables shared with other commands, resulting in inter-command synchronization. As is similar to the task net in a robot programming literature [8,13,31], MRL allows a network structured representation of a task.

The second way to create intelligent agents is supporting decision making during negotiation. The decision is made in two situations, when an agent judges whether the agent can accept a requested task and when an agent selects one of the agents that accepted the task. This means deciding which agent to collaborate with and allows finding the best agent.

The decisions are made based on the logical rule set given in the definition of an agent. The theorem proving function of MRL deals with a query for decision making based on the rule set. This query is used to check whether an agent can provide a service in response to a requested task. It also used to find an agent providing the best service.

A fact derived from the rule set can be interpreted as a capability of an agent. Though Jini [35] proposed by Sun Microsystems looks up services an agent can provide, our approach is different in that the service of an agent is derived by theorem proving from the rule set. This is a novel feature of the agent approach based on AI. For example, the queries shown in Table 4 are available for the delivery task. These are not only the service list of the agent, but they are queries for judgment using logical reasoning to determine whether services are available or not.

Table 4
Queries for the delivery task. "·" means an abbreviation of n-ary arguments

Query	Meaning
`acceptable(move_to(Node))`	Confirms whether a mobile robot agent can reach node `Node`.
`acceptable(focus_on(Node))`	Confirms whether a camera agent can focus on node `Node`.
`acceptable(print_out(File))`	Confirms a printer agent can print out file `File`.
`min(Cost, reasonable_plan(·))`	Selects a mobile robot agent that has the best plan.
`max(Area, focus_on(·))`	Selects a camera agent whose visible scope is the widest.
`min(Distance, distance(·))`	Selects a printer agent that has the minimum cost to perform the `print_delivery` task.

5.2. Dynamic system configuration of information appliances

The primary characteristic of distributed problem solving based on the contract net protocol is that agents are loosely coupled. This allows dynamically changing agent structures [4].

Our multi-agent programming also has this feature, and it is possible to add or delete robotic agents in run time. In order to confirm this feature, we developed an agent server which can generate and delete robotic agents in run time. This server receives a request to generate and delete an agent from not only users but also other agents and this server sends the request to the specified robotic agent. For example, the following program handles the command to generate a new mobile robot agent.

```
:- agent(mobile).
 run:- *create_agent(mobile_robot,Host,Agent)|
     #mobile_robot:new(Host,Agent),
     run.
```

where the agent class `mobile` is the super-agent of mobile robot agents. The first parameter of the message `create_agent` indicates the class (here, `mobile_agent`) of the agent to be generated and the agent is generated by calling the `new` predicate. The agent name is stored in the variable `Agent`, and the agent runs on the machine specified by `Host`.

An agent is deleted by the following clause.

```
run:- *terminate_agent(mobile_robot,Agent)|
    !terminate(Agent),
    run.
```

The class and name of the agent to be deleted are received in the guard. A `terminate(Agent)` message is sent to the sub-agent which receives the message and terminates itself by the following clause.

```
%mobile_robot agent
 run(Agent, null):-^terminate(Agent) | true.
 run(Agent, Task + E):-^terminate(Agent) | E = cancel.
```

The first clause means the case in which there is no executing task. The `run` predicate is not recursively called in the body, and all streams to communicate with other agents are closed automatically before the agent terminates itself. Since the terminations of the super-agent is reported to lower agents, the lower agents are deleted recursively. The second clause is the case in which the agent has an executing task. In this case, since it is likely to cooperate with other agents, the variable for emergent events is used to notify the associated agent that will terminate all the actions concerning the executing task.

In a class of agents which may be generated and deleted, the above clauses are added to the agent definition. Since mobile robots and handling robots for the delivery task must be turned off periodically, we incorporate the above clauses into the mobile robot and handling robot agent definitions, so that the agents can be dynamically generated and deleted.

An action cancellation for emergent situations is sent to a robot controller which is the sub-agent of a robotic agent. Postprocesses for the action cancellation should be added to the definition of the robot controller. For example, when a handling robot controller receives a cancellation while grasping printed papers, the handling robot should put the printed papers on the printer and move the hand to the home position. The handling robot controller must include such postprocesses. If appropriate postprocesses are prepared for all actions, no problem happens even if agents are deleted.

However, there is the problem that an agent engaged in negotiation may be removed during the negotiation. In the contract net protocol, the processes of manager and contractor for task sharing are executed locally, and it was claimed that this makes it easy to generate and delete agents. However, if an agent which requests a task is deleted, the receiver for an the accept message is lost. Similarly, if an agent which accepts the task is deleted, the receiver for a commit message may be lost. This occurs because the agent does not know that the other agent is deleted. Therefore, the agent deletion should be prohibited or postponed until the task commitment is finished. In a multi-agent setting, it is easy to change a system configuration dynamically, but we need a careful treatment to solve the problem.

5.3. Agent negotiation using various communication patterns

Broadcast and incomplete messages are the distinct features of communication in concurrent logic programming. A shared variable for multiple goals is regarded as a communication channel, and a message is broadcast to the goals by instantiating the shared variable. If a goal receives a message with an unbound variable and the goal instantiates the variable, the goal can return a new message to the original goal via the variable. This means that an incomplete message realizes a bi-directional communication with only one transmission. Such an incomplete message is useful for realizing task commitment, synchronization and emergent event processing in multi-agent programming.

From a viewpoint of collaboration between agents, the above communication facilities are used to realize information sharing among agents. As an example, we will focus on synchronization among heterogeneous agents (different robot controllers). Some variables for synchronization are assigned to each robot controller, and the robot controllers execute actions by instantiating the variables. At this time, the robot controller does not know which controller to cooperate with and does not know to which controller it should report the end of an action. By accessing the variable given by a super-agent (robotic agent), the robot controller can cooperate with other controllers.

We consider a similar example in which a super-agent serving as a supervisor for multiple robotic agents resolves conflicts among robotic agents. In the case of mobile robots, a conflict occurs when several robots move on the same path. Each robot agent has its own plan to achieve the delivery task. Each plan is locally consistent, but includes conflict (collision in this case) in a global sense. In order to solve this problem, the super-agent detects conflicting actions of sub-agents and gives the sub-agents shared variables that are used to exchange information to cope with the conflict actions.

Thus, it is possible to consistently collaborate among all agents by sharing information including logical variables. A logical variable serves a bridge between agents, and an agent can coordinate own behavior consistently, without information about other agents. Unlike the blackboard model [15], there is no global variable in logic programming; shared information is locally accessible in the same way that the agent accesses the state variables. Such information sharing realized in MRL is used to provide a basis for agent collaboration.

5.4. Flexible invocation processes

Pattern-directed invocation is one of the most important properties of AI programming. Multi-agent programming realizes goal-directed invocation and data-directed invocation based on concurrent processes, synchronization and asynchronization. Task request from external agents is processed by a goal-directed invocation, and an event from sensors is processed by a data-directed invocation. Guarded Horn clauses support such bi-directional invocation where the invocation occurs when the condition of a guard is satisfied.

A variable for emergent situations is used for both goal-directed and data-directed invocation. Task cancellation by a user is reported to the actions for the task by instantiating the variable in the task. When an error happens during action execution, the error is reported to the task and to the user, resulting in the task cancellation. Both bottom-up and top-down computations are realized in the same program. This is due to unification of the logical variable.

A robotic agent has two functions, negotiation and execution, that are processed asynchronously by concurrent processing. There exists state transition rules for negotiation and for action execution, and one of the rules is selected non-deterministically. This allows us to interleave the action execution with the negotiation process. Since the ordering relation for goal execution is partial in concurrent logic programming, we can realize flexible invocation processes during state transitions.

Incomplete messages are also useful for realizing flexible invocation. We adopted the expiration-based selection as a task commitment strategy for agent negotiation in Section 4. This strategy ignores `accept` messages arriving after a specified time. The strategy is simple, but we can realize a sophisticated strategy by distinguishing task acceptability from a planning process. In this new strategy, a candidate agent returns an `accept` first. However, before producing the plan, if the agent receives another `accept` message from another agent, the selection is based on comparing the plans of the current candidate and the new agent. This process repeats until a plan is completely produced. In general, the plan is not generated faster than the `accept` message is received, so this strategy is reasonable.

We combined this strategy with the expiration-based selection. The original program was modified as follows. After receiving an `accept` message, the agent waits for producing a plan and also waits for other `accept` messages within a specified time interval. Such a mixed strategy is realized using incomplete messages, and the non-determinacy of guarded Horn clauses. For the delivery task, the best mobile robot agent was successfully selected in real time by the strategy. Such a complex collaboration implies the novel feature of multi-agent programming.

6. Observation

We summarize observations when we used the delivery robot system everyday in our smart office experimental environment. The delivery robot system works for more than three hours everyday, and the system handles an average of thirty delivery task requests in one day. A maximum of twenty users works in the smart office; they request the delivery with their computers or cellular phone.

From a viewpoint of robotics, our research is close to the topic of the cooperative control of distributed robots. Communication and control are the most important themes in this topic, which deals with speeding up communications, resolving time delays, and ensuring precision control. However, we think that such problems are not critical for everyday support like the delivery task. Actually, the delivery task does not require high speed. For printed out delivery, there is no problem if a mobile robot arrives at the printer before the printer finishes printing out. In a localization task, high accuracy is not required, because we succeeded in keeping the localization error to less than 4 inches with the assistance of infrared sensors. This error is not a problem for mobile robots to deliver among rooms. In contrast, high accuracy is required when the handling robot puts printed paper into the tray on the mobile robot. In this case, we tried mobile robot navigation by a camera. As the result, the success rate reached 95%. Communication among robot controllers allows the delivery robot system to accomplish everyday tasks.

When an accident happens to robots and sensors, the executed task can be cancelled by the emergent event handling of MRL. However, MRL can not deal with several critical errors. A software approach alone can not cope with situations in which the printers run out of paper or when a mobile robot's battery is discharged. Thus, we should periodically care the critical cases of robots and ensure that the robots work normally. This is why we

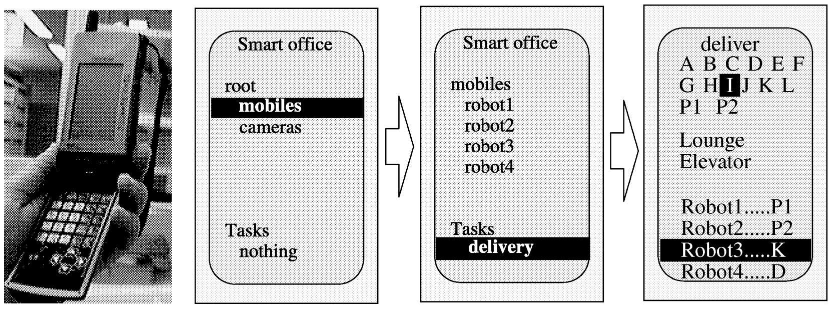

Fig. 12. Controlling by a cellular phone. This cellular phone has 1 MB of memory, so we can download programs. It is requesting a delivery task to mobile agents. The alphabets indicate the rooms. The positions of the robots are displayed in the lower portion.

use mobile robots for three hours a day. We have to care for the agents so they can support our activity.

The research of the smart office requires not only constructing an electronic environment, but also dynamically changing the environment supporting our activities through interactions between human and robotic agents. This is because the environment always changes and we always seek to improve smart office tasks in order to make our everyday work go smoothly. Furthermore, we do not use one information appliance for one purpose, but rather we pursue efficiency and economy by multiple use of one information appliance. Robotic agents thus require multiple dynamic functions. Actually, our mobile robots can not only deliver documents but can also guide visitors, and we use cameras for not only navigation of mobile robots, but also for monitoring human beings. The smart office environment will change as agents perform such multiple functions.

We can give abstract commands to information appliances based on agent modeling. However, a big gap was created between abstract commands and action-level commands. For example, it became hard for users to know which robotic agent is performing the users' task. Furthermore, it became difficult to monitor the current states of robotic agents and control robotic agents using low-level commands. In contrast, we do not have to give detailed instructions and thus, we successfully developed a very small controller which enables us to request tasks on the cellular phone as shown in Fig. 12. This controller can provide the minimum information for a user to know the state of robotic agents on a small display. The support by such a user interface becomes more necessary as the cooperation among robotic agents becomes more complicated.

7. Other applications

We will now turn to applications in the smart office other than delivery robot system and will describe how to use multi-agent programming in those applications. The applications are a smart TV conference system and a security monitoring system. Both applications are realized using multiple cameras.

The smart TV conference system displays the presenter, the questioner, and participants at remote sites on the screen using multiple cameras. The display system displays several images from the cameras on the screen. Participants have cellular phones, and questioners access the smart TV conference system with a cellular phone. One of the cameras focuses on the questioner.

In a TV conference, an image from one camera is sent to a remote site and displayed on a screen at the remote site so participants can interact with each other. In this case, the camera normally focuses on a small number of participants. In contrast, multiple cameras and multiple projectors allow a TV conference with many people participating. Moreover, when there are many questioners, the system displays images of the questioners on smaller frames. We can thus see the number of questioners and know who wants to ask a question. After one questioner is finished, the image of the next questioner expands. We can regard such support by the camera agents and projector agents as intelligent support of our meeting activity.

When the smart conference agent receives a request for a question from a user through the cellular phone, the agent transmits the task to all camera agents. A camera agent collaborates with a projector agent for focusing and image sending to the display system.

The judgment of whether the camera agent can provide the focusing service depends on the relative position between the camera and the questioner. The relative position appended to an accept message is a criteria to select the best camera agent of all agents that send the accept message. The calculation of the relative position and criterion for selecting the best agent are stored in the rule set on both the camera agent and the smart conference agent. Such a cooperation procedure is almost the same as the delivery task.

The security monitoring system works when there is no person in the office rather than in the day time. There are 10 human detection sensors on the ceiling in our experimental environment. When these sensors detect a human, the fact is reported to a sensor monitoring agent. Eight cameras are also installed on the ceiling. Several camera agents are selected through negotiation with the sensor monitoring agent and focus on the human detected by the infrared sensor. The difference from the delivery task is that the sensor monitoring agent does not commit the focusing task to only one camera agent. The task is committed to all camera agents which are not performing other tasks.

Since camera agents perform delivery tasks in the day time, the camera agents have multiple functions. Therefore, the rule set for task acceptance should include a time constraint. In order to use robotic agents for multiple purposes, we need to extend the rule set on the agent and describe the available services for every time zone.

Although these two examples are constructed using camera agents, other collaboration tasks among various information appliances are realized in the smart office. The basic way to realize such collaboration is to describe which services each information appliance can provide. We can derive how information appliances realize the services by using the planning function of AI and can integrate each service by negotiation. This allows the environment itself with its embedded information appliances to provide the services to support our activities. The proposed multi-agent programming can make information appliances useful to us based on the AI approach.

8. Conclusions

In this paper, we designed a smart office environment that supports our activities by cooperation among information appliances including robots and sensors from the viewpoint of AI for everyday activities. We modeled these appliances as robotic agents realized by the multi-agent language MRL we developed. Robotic agents cooperate with each other to achieve the document printing and delivery task, which is a typical task in the smart office. Although distributed devices have usually been controlled through an engineering approach, the framework of collaboration based on AI allows integrating the services each agents provides and supporting our everyday activities.

We shall design small or embedded agent systems to advance the agentification of information appliances on an everyday and everywhere level. For example, if such systems work well on hand-held computers, it will be possible to use the system for outdoor activities. In addition, the plug-and-play function enables us to swap any intelligent appliances and thus realize various functions, further expanding the appliance's functions for human support. It is important to clarify which agents are prepared, which functions are realized in an agent, and how they can interact with users in the real world. We must investigate the degree to which these agent-oriented information appliances support us as tools for everyday use.

MRL supports important functions of multi-agent programming. However, in order to design intelligent agent community as advanced collaboration, agent negotiation should be specified implicitly by constraints between shared variables. Bobrow proposes concurrent constraint programming as a means of intelligent computation, wherein its declarative nature allows software development in a compositional manner [10,32]. This compositionality enables us to plug in some components. Our approach exploits concurrency and various communication patterns in concurrent logic programming. The introduction of constraints into MRL enhances its declarative programming ability and enables us to describe the relationships between agents as implicit constraints.

We have designed intelligent information appliances base on agent collaboration. However, it is also important that an agent itself becomes intelligent through a learning function. In particular, it is necessary that personalized systems be designed based on user preferences for support. We believe that an agent learns inductively using past experience and will try to apply inductive logic programming (ILP) [23,25] to agent design. Such agentification which personalizes information appliances can be regarded as a key technology for advancing AI for everyday tasks.

References

[1] R.A. Brooks, The Intelligent Room Project, in: Proc. Second International Cognitive Technology Conference (CT-97), 1997.
[2] T. Chikayama, A KL1 implementation for Unix systems, New Generation Computing 12 (1993) 123–124.
[3] K.L. Clark, PARLOG: Parallel programming in logic, ACM Trans. Program. Lang. Syst. 8 (1) (1986) 1–49.
[4] R. Davis, R.G. Smith, Negotiation as a metaphor for distributed problem solving, Artificial Intelligence 20 (1983) 63–109.
[5] E.W. Dijkstra, Guarded commands, nondeterminacy, and formal derivation of programs, Comm. ACM 18 (8) (1975) 453–457.

[6] E. Ephrati, J.S. Rosenschein, The Clarke Tax as a consensus mechanism among automated agents, in: Proc. AAAI-91, Anaheim, CA, 1991, pp. 173–178.
[7] R.E. Fikes, P.E. Hart, N.J. Nilsson, Learning and executing generalized robot plans, Artificial Intelligence 3 (1972) 251–288.
[8] R.J. Firby, Task networks for controlling continuous process, in: Proc. Second International Conference on AI Planning Systems, 1994.
[9] M.S. Fox, An organizational view of distributed systems, IEEE Trans. Systems Man Cybern. 11 (a) (1981) 70–80.
[10] M.P.J. Fromherz, V.A. Saraswat, Model-based computing: Using concurrent constraint programming for modeling and model compilation, in: U. Montanari, F. Rossi (Eds.), Principles and Practice of Constraint Programming (CP-95), Lecture Notes in Computer Science, Vol. 976, Springer, Berlin, 1995, pp. 629–635.
[11] H. Fujita, R. Hasegawa, A model-generation theorem prover in KL1 using ramified stack algorithm, in: Proc. Eighth International Conference on Logic Programming, Paris, 1991, pp. 535–548.
[12] L. Gasser, Social conceptions of knowledge and action: DAI foundations and open system semantics, Artificial Intelligence 47 (1991) 107–138.
[13] E. Gat, ESL: A language for supporting robust plan execution in embedded autonomous agents, in: Proc. IEEE Aerospace Conference, 1997.
[14] M.R. Genesereth, S.P. Ketchpel, Software agents, Comm. ACM 37 (7) (1994) 48–53.
[15] B. Hayes-Roth, Architectural foundations for real-time performance in intelligent agents, Real-Time Systems International Journal of Time-Critical Computing Systems 2 (1990) 99–125.
[16] R. Kowalski, Logic for Problem Solving, Elsevier/North-Holland, Amsterdam, 1979.
[17] R. Kowalski, Using metalogic to reconcile reactive with rational agents, in: K. Apt, F. Turini (Eds.), Meta-Logics and Logic Programming, MIT Press, Cambridge, MA, 1995.
[18] S. Kraus, J. Wilkenfeld, G. Zlotkin, Multiagent negotiation under time constraints, Artificial Intelligence 75 (1995) 297–345.
[19] P. Maes, Agents that reduce work and information overload, Comm. ACM 37 (7) (1994) 30–40.
[20] J. McCarthy, Programs with common sense, in: M. Minsky (Ed.), Semantic Information Processing, MIT Press, Cambridge, MA, 1968, pp. 403–418.
[21] M. Minsky, The Society of Mind, Simon and Schuster, New York, 1986.
[22] T. Mitchell, R. Caruana, D. Freitag, J. McDermott, D. Zabowski, Experience with a learning personal assistant, Comm. ACM 37 (7) (1994) 80–91.
[23] F. Mizoguchi, H. Ohwada, Constrained relative least general generalization for inducing constraint logic programs, New Generation Computing 13 (1995) 335–368.
[24] F. Mizoguchi, H. Ohwada, Personalized mail agent using inductive logic programming, in: K. Furukawa, D. Michie, S. Muggleton (Eds.), Machine Intelligence, Vol. 15, Oxford University Press–USA, 1999, pp. 154–175.
[25] S. Muggleton, Inductive logic programming, New Generation Computing 8 (1991) 295–318.
[26] D.A. Norman, The Psychology of Everyday Things, Basic Books, 1988.
[27] S.J. Rosenschein, Formal theories of knowledge in AI and robotics, New Generation Computing 3 (4) (1985) 345–357.
[28] V.A. Saraswat, Concurrent Constraint Programming, MIT Press, Cambridge, MA, 1993.
[29] E. Shapiro, The family of concurrent logic programming language, ACM Computing Surveys 21 (1989) 413–510.
[30] Y. Shoham, Agent-oriented programming, Artificial Intelligence 60 (1993) 51–92.
[31] R. Simmons, D. Apfelbaum, A task description language for robot control, in: Proc. Intelligent Robots and Systems (IROS-98), 1998, pp. 1931–1937.
[32] M.P. Singh, D.G. Bobrow, M.N. Huhns, M. King, H. Kitano, R. Reiter, The next big thing: Position statements, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 1511–1524.
[33] R. Smith, The contract net protocol: High-level communication and control in a distributed problem solver, IEEE Trans. Comput. C-29 (12) (1980) 1104–1113.
[34] M. Stefik, G. Foster, D.G. Bobrow, K. Kahn, S. Lanning, L. Suchman, Beyond the chalkboard: Computer support for collaboration and problem solving in meetings, Comm. ACM 30 (1) (1987) 32–47.
[35] Jini$^{\text{TM}}$ Lookup Service Specification, Revision 1.0, Sun Microsystems, Inc., 1999.

[36] K. Ueda, T. Chikayama, Design of the kernel language for the parallel inference machine, The Computer Journal 33 (6) (1990) 494–500.
[37] M. Weiser, Some computer science issues in ubiquitous computing, Comm. ACM 36 (7) (1993) 74–84.
[38] M. Wooldridge, N.R. Jennings, Agent theories, architectures, and languages: A survey, in: Proc. ECAI Workshop on Agent Theories, Architectures, and Language, 1994, pp. 1–39.