Meta-level Architecture for Executing Multi-agent Scenarios

Zhiqiang Gao[1], Tomoyuki Kawasoe[1], Akishige Yamamoto[2], and Toru Ishida[3]

[1] Department of Social Informatics, Kyoto University, Japan,
{gaoz, kawasoe}@lab7.kuis.kyoto-u.ac.jp
[2] Mathematical Systems Inc. Tokyo, Japan,
yamamoto@msi.co.jp
[3] Department of Social Informatics, Kyoto University, Japan,
ishida@i.kyoto-u.ac.jp

Abstract. Scenarios, which constrain the behavior of agents, can be the interface between computer experts (agent system developers) and application designers (scenario writers), as well as the interface between scenario writers and agents. It raises a number of challenging issues to execute multi-agent scenarios: *1) How can scenario writers generate customized scenarios easily? 2) How can scenario writers monitor and control scenario execution so as to debug errors in scenarios? 3) How can agents negotiate for scenarios to achieve robust behavior against scenario errors?* So, in this paper, we provide a *web style GUI* (Graphical User Interface) for scenario writers to customize scenarios. We propose a *meta-level architecture* for scenario writers to trace and control the execution of scenarios by observing scenarios, and for agents to negotiate with others as well as scenario writers for scenarios. The *meta-level architecture* is illustrated by an experimental multi-agent system of evacuation simulation.

1 Introduction

In order to bridge the gap between multi-agent technologies and their application, we introduce Q, a scenario description language, for designing interaction between agents and humans from the viewpoint of scenario writer [7]. Q does not aim at describing the internal mechanisms of agents, nor the communication and interaction protocols among agents [3,10]. Rather, it is for describing scenarios representing how humans expect agents to behave. Scenarios can also be the interface between application designers and computer experts. Figure 1(a) shows the relations among scenario writer, computer expert and agents. The procedure of creating a new scenario is described as follows. First, scenario writer and agent system developer agree upon *cues* and *actions* (sensing and acting functions of agents) as an interface between them, which are defined by scenario writers using *defcue* and *defaction* for each application. Second, the scenario writer describes a scenario using Q syntax, while the agent system developer implements *cues* and *actions*. The biggest effect of introducing Q is that

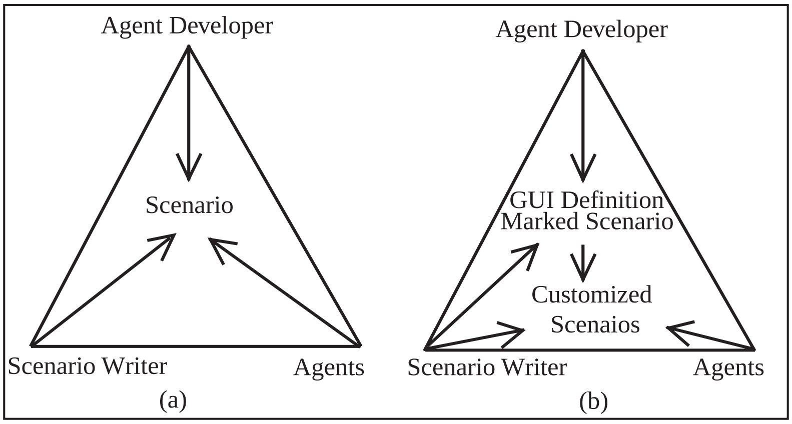


Fig. 1. Relations among scenario writer, computer expert and agents

it can provide a clear interface between computer professionals and application designers, who have totally different perspectives. Since it is developed, Q has been applied in several projects, such as information retrieval *Venus & Mars* [9], social psychological study of agents [5, 14], crossculture experiment *FreeWalk* and digital city [6, 7, 11].

Those who are not computer experts, such as sales managers, often write scenarios. Although they can write scenarios under the help of computer experts, the point is how to customize scenarios for open and dynamically changing applications. Multi-modal interface has made great progress recently [2, 12], however, it seems impossible to generate customized scenarios by gesture or conversation. Fortunately, what application designers often modify in domain specific applications are *the value of arguments in scenarios*, rather than *the sequence of cues and actions*. So, our approach is to provide a *web style GUI* for application designers to change the value of arguments.

A widely accepted methodology for multi-agent monitoring and controlling is that agents monitor the internal state and behavior of other agents as well as their interactions so as to complete tasks efficiently [1, 4]. However, in scenario execution, what scenario writer cares is how to make sure that scenarios are executed correctly, and to modify scenarios when errors are found. So, scenario writers have to trace the execution of scenarios, rather than the internal state of agents and their interactions.

Negotiation among agents for team working has been widely studied in the past decade, and a large number of communication protocols as well as interaction protocols are proposed [8, 13]. However, in scenario execution, the goal of negotiation is not for cooperation or competition, but for robustness of agents' behavior against scenario error. Negotiation does not happen at the meta-layer of agents, but between the meta-layer of Q and agents. Agents need not to communicate with each other frequently, because their strategies for collaboration are constrained by scenarios.

Table 1. Facilities of scenario description language Q

	Syntax	Example
Cue	(defcue name {(parameter in|out|inout)}*)	(defcue ?hear (:from in))
Action	(defaction name {(parameter in|out|inout)}*)	(defaction speak (:sentence in) (:to in))
Guarded Command	(guard {(cue {form}*)* [(otherwise{form}*)]})	(guard ((?hear "Hello" :from Tom) (!speak "Hello" :to Tom)) ((?see :south Kyoto-Station) (!!walk :to Bus-Terminl))
Scenario	(defscenario name ({var}*) (scene1{(cue{form}*)}* [(otherwise{form}*)]) (scene2{(cue{form}*)}* [(otherwise{form}*)]) ...)	(defscenario chat (message) (scene1 ((?hear "hello" :from $x) (go scene2))) (scene2 ((equal? $x Tom) (!say message)) (otherwise(!say "Hello"))))
Agent	(defagent name :scenario scenario-name {key value}*)	(defagent guide :scenario 'sightseeing)
Avatar	(defavatar name)	(defavatar 'Tom)
Crowd	(defcrowd name :scenario scenario-name :population number {key value}*)	(defcrowd pedestrian :scenario 'sightseeing :population 30)

So, the rest of this paper is organized as follow: we first introduce scenario description language Q, and discuss about the requirements for executing multi-agent scenarios. Then we provide a *web style GUI* for customizing scenarios, and propose a *meta-level architecture* for scenario monitor, control and negotiation. At last, we show how the *meta-level architecture* works by a multi-agent system for evacuation simulation.

2 Backgrounds

2.1 Scenario Description Language

We extend *Scheme* by introducing sensing/acting functions and guarded commands to realize scenario description language. A sensing function is defined as a *cue*. There is no side effect to sensing. An acting function is defined as an *action*, which may change the environment of agent system. Some *actions* allow other *actions* to be executed in parallel, which is noted as *!!walk*. This significantly extends the flexibility available to describe scenarios. A guarded command is used for when multiple *cues* are waited simultaneously. Each scenario defines several states (scene1, scene2, etc). Each state is defined as just a guarded command. The scenario is written in the form of state transition. However, since any *Scheme* form can be called in state descriptions, any scenario can be called in *Scheme* forms and scenarios can be nested, it is easy to create fairly complex scenarios. Agents are defined with the scenario in which they are to be executed. Avatars controlled by humans do not require any scenarios. At times, the number of agents may be too numerous to deal with them individually, and it is more convenient to deal with them collectively as a society of agents. The facilities of Q language are summarized in table 1.

2.2 Requirements for Executing Multi-agent Scenarios

In order to execute multi-agent scenarios, a scenario *interpreter* is required to extract *cues* and *actions* from scenarios. Also, an *interface* is needed for requiring *cues* and *actions* and retrieving their execution results. Challenge to the *interpreter* is that the agent system may host a large number of agents. For example, in the virtual space platform of *FreeWalk*, more than 1000 agents operate independently at the same time. In order to control this parallelism, Q *interpreter* should execute 1000 scenarios simultaneously. So, we use *Scheme*'s *continuation* to control process switching. The interface between scenario interpreter and agent system is implemented in COM and CORBA for standardization. However, there are much more requirements for executing multi-agent scenarios.

How Do Scenario Writers Generate Customized Scenarios? Multi-agent scenarios are not necessarily specified by computer professionals, and are instead often written by application designers such as sales managers, publicity officers, and social psychologists, who are not computer experts. So, the challenge is how to have ordinary people generate customized scenarios easily.

How Do Scenario Writers Monitor and Control Scenario Execution? The *interpreter* and *interface* is good enough for running scenarios, if there are no errors in scenarios. However, there are often scenario errors, because error-prone human beings construct them. In addition, it is impractical to assign scenarios to agents, and agents execute them until they are finished. Because multi-agent systems are intrinsically dynamic: what an agent assumes to be true may become false as a consequence of the *actions* of other agents in the system. Therefore, scenario writers have to monitor the execution of scenarios, and change scenario execution when errors occur.

How Do Agents Negotiate for Scenarios? Scenarios assigned by humans may not match the goal of autonomous agents, and agents may not have the ability to execute certain *cues* or *actions*. Furthermore, in an open, complex and dynamic environment, agents have to adjust their goals, skip unachievable *cues*/*actions*, or modify the value of arguments frequently. So, we need a mechanism for agents to negotiate for scenarios at execution time.

3 Meta-level Architecture for Scenario Execution

3.1 Web-Style GUI for Customizing Scenarios

Two approaches have been made to simplify scenario writing for those who are not computer experts. One approach is to use the technology of IPC (Interaction Pattern Card) for those who are familiar with *MS Excel* to generate scenarios from cards. We think that each scenario, for example, scenario for fire drill, lecture or selling, has different interaction pattern. The format of the cards is defined according to these patterns, and humans can describe scenarios by filling in the blanks of cards and combining these cards.

Another approach is to provide *web style GUI* for scenario writers to generate customized scenarios. In the application of evacuation simulation [7], we have found that more than one hundred of parameters are specified in scenarios. For example, if an agent hopes to *walk to Kyoto station*, we have to specify its *walking speed, direction, acceleration, angle speed, angle acceleration, orientation of body, orientation of head, and so on.* So, it is rather difficult to customize scenarios. After the agents are re-designed, they can use default values of parameters and adjust these values under different situations, so the number of parameters is reduced greatly. Thus, on the basis of the simplified scenario, we implement a *web-style GUI* for scenario writers to customize scenarios.

The process for customizing scenario is as follow: scenario writer and computer expert cooperate to prepare a marked scenario, put @ mark before the atoms (*Scheme* object) need to be customized, and define a GUI structure by *defgui* syntax. Given both *defgui* definition and marked scenario, HTML form is calculated and displayed on web browser. Then scenario writer can input and submit the form. CGI engine written in *Scheme* parses the input, rewrites the scenario and start to execute the customized scenario. Figure 1(b) illustrates the process of generating multi-agent scenarios. Figure 2 presents an example of scenario customization.

3.2 Meta-level Architecture

A *meta-level architecture* is proposed to cope with the challenge issues for scenario execution discussed in section 2. Q *interpreter, controller, monitor and analyzer*, handle scenarios, as shown in Figure 3. When scenarios are given to a particular agent, a Q *messenger* is created and coupled with the agent system including the particular agent. The interaction between Q *messenger* and agents is divided into two layers: an *execution layer* and a *meta-layer*. In the *execution layer*, Q *interpreter* requires *cues* and *actions* to agent system, and retrieve their execution results. In the *meta-layer*, scenario writer monitors and controls scenario execution, and agents negotiate with others as well as humans for resolving scenario errors.

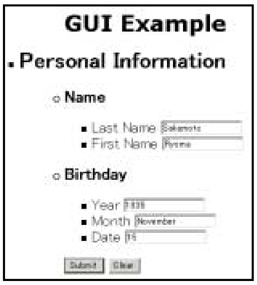
Monitor and Control of Scenario Execution by Scenario Writers. In order to find errors in scenarios, such as *an agent always collides onto the wall*,

```
(defscenario chat ()
 (scene1 ((?hear :name Kaishuu :word “name”)
          (!speak :sentence (string-append “My name is
          ”@”Ryoma” “ “ @”Sakamoto” “.”) :to Kaishuu))
         ((?hear :name Kaishuu :word “birthday”)
          (!speak :sentence (string-append “My birthday is
           “@”November” “ “ @”15” “, “ @”1835” “.”) :to Kaishuu))))
```

(a) Marked Scenario

```
(defgui example-gui “GUI Example”
 (“Personal Information”
          (“Name”
            (“Last Name” @”Sakamoto”)
            (“First Name” @”Ryoma”))
          (“Birthday”
            (“Year” @”1835”)
            (“Month” @”November”)
            (“Date” @”15”))))
```

(b) GUI Definition



GUI Example

- **Personal Information**
  - **Name**
    - Last Name Sakamoto
    - First Name Ryoma
  - **Birthday**
    - Year 1835
    - Month November
    - Date 15

Submit Clear

(c) Genertated GUI

Fig. 2. Customizing scenarios by *web-style GUI*

scenario writers have to observe agent system at running time. They try to find which agent is executing which scenario, and which *cue/action* is being executed when errors occur. So, *monitor*, a component of *messenger*, is realized. Scenario writers choose the agent to be monitored from the menu item of *messenger* or by clicking right button of mouse in agent system. Then the name of agents and scenarios are shown in the title bar of the *monitor/controller* window. Execution history of scenarios is indicated by different background colors of *cues/actions* and ordinal numbers attached, as shown in Figure 4. Finished *cues/actions* are shown in yellow background, while the one being executed is shown in green. Ordinal numbers are attached to finished *cues/actions* to indicate their execution sequence.

Several control primitives are provided for scenario writers to debug errors and control scenario execution in emergency. They are *start, stop, suspend, resume, step-by-step, change scenario and add new scenario.* Facilities of the first four primitives are start, stop, suspend and resume the execution of scenarios, respectively. *Step by step* means after the execution of one *cue/action*, agents wait for further direction from scenario writers. Thus, scenario writers can debug scenario errors step by step. *Change scenario* means stopping execution of the original scenario, and starting a new one. Scenario writers use this facility to change scenario execution in emergency without negotiation with agents. *Add*

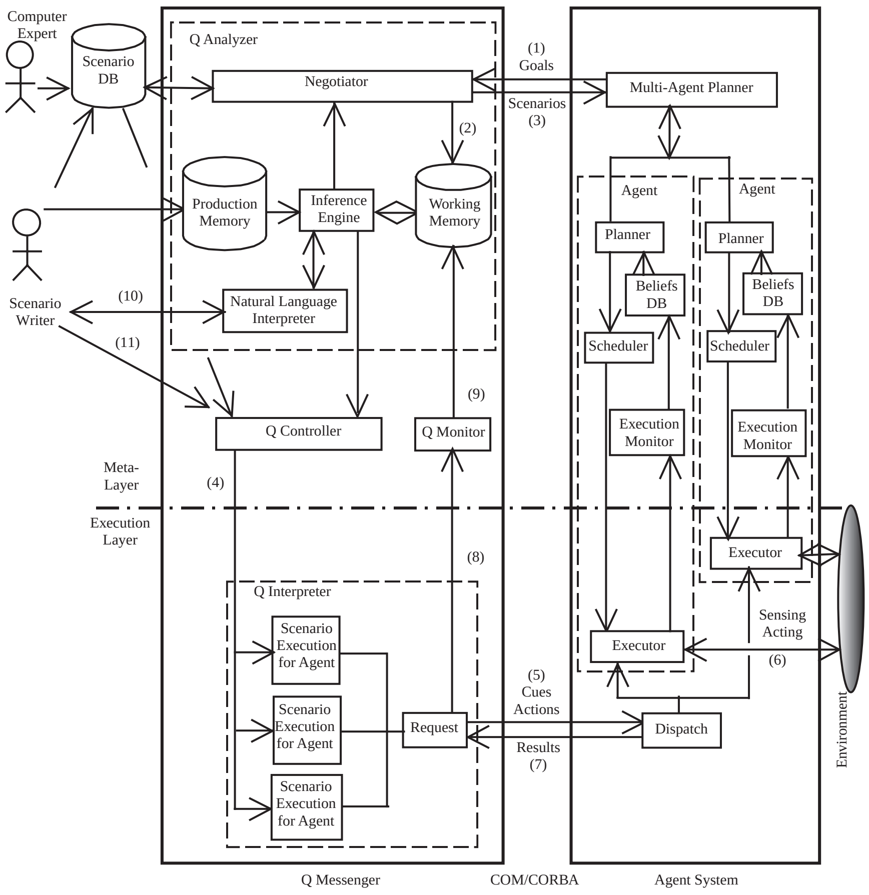
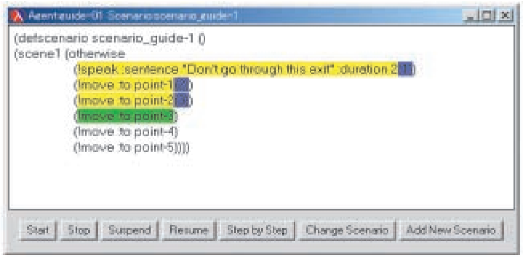


Fig. 3. *Meta-level architecture* for scenario execution



```
(defscenario scenario_guide-1 ()
(scene1 (otherwise
        (!speak :sentence "Don't go through this exit" :duration 2 1)
        (!move :to point-1 2)
        (!move :to point-2 3)
        (!move :to point-3)
        (!move :to point-4)
        (!move :to point-5))))
```

Fig. 4. Monitor and control of scenario execution by scenario writers

new scenario means starting the execution of a new scenario; meanwhile the original one is being executed concurrently.

Table 2. Different scenario errors found by agents and corresponding solutions

Scenario Errors	Solutions	
Scenario	Require for new scenario	Matching scenarios in scenario DB
		Decomposition or merging scenarios
		Negotiation with agents
	Require for changing scenario execution	Autonomous behavior
		Rules in production memory
		Conversation with scenario writers
Cue/Action	Autonomous behavior	
	Rules in production memory	
	Conversation with scenario writers	
Argument	Autonomous behavior	
	Conversation with scenario writers	

Negotiation for Scenarios among Agents and Scenario Writers. Negotiation happens when agents, not scenario writers, find scenario errors. For example, *in an information retrieval scenario agents may fail to connect to certain web site at execution time.* Agents might resolve scenario errors autonomously, or inform scenario *messenger* to change scenario execution according to rules stored in *production memory.* Agents might also start negotiation with others or conversation with scenario writers. Scenario errors are classified into three levels, which are scenario level, *cue/action* level and argument level. If the goal of an agent is changed, the agent requires for new scenario or changing execution of the original one. *Negotiator*, other agents or scenario writers might satisfy scenario requirement. Scenario error at *cue/action* level might be caused by that agents have not inherently the ability, or its precondition is not satisfied due to the change of internal state of agents or environments. Thus, agents send requirements to scenario writer for further direction, such as skipping it or waiting until its precondition is satisfied. Scenario error at argument level happens when an agent intends to modify the value of arguments. Possible scenario errors and corresponding solutions are summarized in Table 2.

In order to communicate with scenario *messenger* and other agents, the message format has to be specified. Because agents can interpret *cues/action*, so our approach is to use *action*-like messages for scenario negotiation, which are classfied into inform, require, reply and accept. Some examples are given below.

```
                        INFORM
(!inform-scenario :for 'stop|start :agent agent
         :scenario scenario)
(!inform-cue-action :for 'skip|wait :agent agent
         :scenario scenario :cue-action cue-action)
(!inform-argument :for value :agent agent :scenario scenario
         :cue-action cue-action :argument argument)
```

```
                        REQUIRE
(!require-scenario :for scenario :agent agent :goal goal)
(!require-scenario :for 'start|stop|pause|resume
         :agent agent :scenario scenario)
(!require-cue-action :for 'skip|wait :agent agent
         :scenario scenario :cue-action cue-action)
(!require-argument :for value :agent agent :scenario scenario
         :cue-action cue-action :argument argument)
                        REPLY
(!reply-scenario :for scenario  :by agent :to agent :goal goal))
                        ACCEPT
(!accept :scenario scenario :agent agent)
```

Agents interact only for scenarios, and the semantics of *action*-like message is self-explained, so no negotiation protocol is needed. For example, if an agent requires for a new scenario, it sends the requirement to Q *messenger*. According to rules stored in *production memory*, the requirement is sent to other agents, who may provide one scenario. At last the agent accepts the scenario and Q *messenger* starts its execution. An example of negotiation for scenarios between two agents are given below.

```
Agent A: (!require-scenario :for 'scenario :agent Agent-A
         :goal ``visit Kyoto")
Agent B: (!reply-scenario :for 'sight-seeing :by Agent-B
         :to Agent-A :goal ``visit Kyoto"))
Agent A: (!accept :scenario 'sight-seeing :agent Agent-A)
```

A conversation interface is provided for scenario writers to interact with agents to resolve scenario errors. During the conversation, agents use *action*-like utterances, which is easy to be understood by scenario writers. Meanwhile, Scenario writers use natural language, which is translated into *action*-like utterances by *natural language interpreter*. Utterances committed by scenario writers are about agents, scenarios, *cues/actions* and arguments, and might fall into a fixed format, such as which agent executes which scenario. So simple patterns exist that indicate key pieces of information to fill in templates of scenario execution control task. Therefore, We define the patterns of utterances to identify different slots in the template, as shown in Figure 5. If the utterance of scenario writers is abbreviated, such as only one verb phrase of *stop* or noun phrase of *Kyoto Station* appears, then the context knowledge of agents' utterance is used to fill in the template slots. Exmaples of conversation are given below.

```
Agent: (!require-scenario :for 'stop :agent Agent
       :scenario 'sight-seeing)
Human: (!command :level 'scenario :for 'stop
       :agent Agent :scenario 'sight-seeing)
Agent: (!require-cue-action :for 'skip|modify
       :agent Agent :scenario 'sight-seeing
```

```
         :cue-action '(!walk :to Kyoto-Station))
Human: (!command :level 'cue-action :for 'skip
         :agent Agent :scenario 'sight-seeing
         :cue-action '(!walk :to Kyoto-Station))
Agent: (!require-argument :for value
         :agent Agent :scenario 'sight-seeing
         :cue-action '(!walk :to Kyoto-Station)
         :argument 'to)
Human: (!command :level 'argument :for
         'Bus-terminal :agent Agent :scenario
         'sight-seeing :cue-action '(!walk :to
        Kyoto-Station :argument 'to)
```

AGENT	e.g., guide-01
SCENARIO	e.g., sight-seeing
CUE/ACTION	e.g., (!walk : to Kyoto-station)
ARGUMENT	e.g., to
ARGUMENT VALUE	e.g., Kyoto-station
COMMAND NAME	e.g., stop, skip, wait, modify

(a) Template of utterances

(1) AGENT start|stop|pause|resume SCENARIO
(2) AGENT stop|skip|wait CUE|ACTION in SCENARIO
(3) AGENT modify|change value of ARGUMENT as ARGUMENT VALUE
in CUE|ACTION of SCENARIO

(b) Examples of utterance pattern

Fig. 5. Template and patterns of utterances for scenario writers to control scenario execution

3.3 Sequence of Scenario Execution

Let us assume a simple case, in which computer expert and scenario writer have created a set of scenarios, and one of which is called *go-to-Kyoto-statio*. An agent, named *Judy*, now wants to go to Kyoto station, and it has the capability of speaking, hearing, walking, taking the train, and taking the bus, etc. (1) The *planner* of *Judy* notifies the *planner* of multi-agent system, and the latter advertises *Judy*'s goal to Q *analyzer*. (2) Q *negotiator* matches *Judy*'s goal by searching in scenario DB, and then obtains the scenario *go-to-Kyoto-station*. (3)

Q *negotiator* sends scenario *go-to-Kyoto-station* and a list of *cues/actions* to *Judy*. *Judy* checks them with its goal and ability, and then adopts the scenario. (4) Q *analyzer* sends a command to Q *controller*, and the latter starts to execute scenario *go-to-Kyoto-station*, which is actually interpreted and executed by Q *interpreter*. (5) Corresponding *cues* and *actions* in scenario *go-to-Kyoto-station* are sent to *request* of Q *interpreter*. (6) *Judy* gets *cues* and *actions* from *dispatch* of agent system, and executes them by sensing and acting, provided they are not conflict with its schedule. (7) Executing results including scenario errors are sent back to *request*, and Q *interpreter* continues executing the scenario. (8) Meanwhile, executing results are also sent to Q *monitor* for Q *analyzer* to monitor scenario execution (9) Scenario errors are put on *working memory* of Q *analyzer*, and corresponding actions are taken by *inference engine*, such as to communicate with scenario writer. (10) Conversation between scenario writer and agents is started through *natural language interpreter*. (11) Scenario writer can even control scenario execution directly by using Q *controller*. The scenario execution sequence is shown in Figure 3.

4 Experiments on Meta-level Architecture by Evacuation Simulation

Evacuation simulation can be created by having pedestrian agents act as humans running around trying to escape. Such simulation results will help a crisis management center accumulate experience and to make correct decisions, since simulations exhibit the mistakes typical of humans. Before tackling large space, such as Kyoto station, we conducted more restricted experiments, which shows 2D simulation of how humans behave when a crisis occurs in a small room. It is the virtual evacuation space designed for the study of evacuation method [15].

4.1 Procedure of Experiment and Scenarios for Agents

We take the experiment of *Four-leader Follow Directions* to evaluate the *meta-level architecture*, which could be summarized as: Subjects and leaders entered the experimental room through Exit C, as shown in Figure 6. All lights in the basement suddenly turned off, and loud emergency alarm sounded for 20s. After the alarm had finished alarming, Exits A and B were opened from the outside. The leader at Point A called out “Don’t go through this exit”, while standing in the way of subjects who were trying to leave through Exit A. Subsequently, the leader at Point B called out “Go in that direction”, while pointing at point P1 with a waving arm. Leaders at Points C and D then began to call out the same phrase, while pointing toward Exit B. Thereafter, all of the leaders proceeded toward Exit B while calling out directions so that all of the subjects escaped by following their direction. Figure 6(a) shows the arrangement of leaders and evacuees at the beginning of evacuation. Figure 6(b) presents the positions and directions of different agents during evacuation simulation.

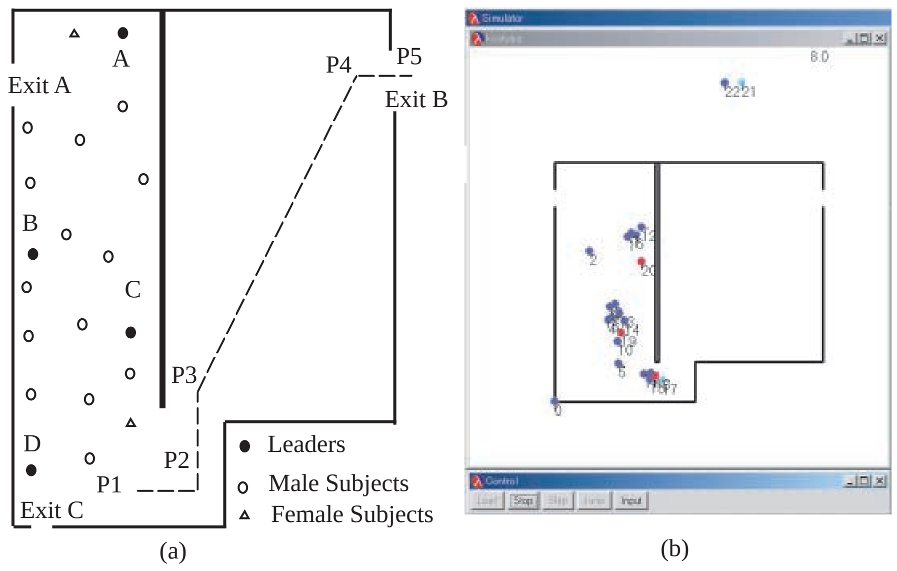


Fig. 6. Evacuation simulation with multi-agents

In order to simplify scenario writing, we first write a scenario *scenario-escape-to-Exit-B*, which provides a course of walking actions to escape to Exit B. Scenario *scenario-leader-1* is designed for the leader at point A, who speaks "Don't go through this exit" just after the experiment starts. Scenarios for leaders at point B, C and D are omitted in this paper, and scenario *scenario-subjects* is assigned to subjects. Examples of scenario definition are given below.

```
(defscenario scenario-escape-to-Exit-B()
 (scene1(otherwise
         (!move :to Point-1)(!move :to Point-2)
         (!move :to Point-3)(!move :to Point-4)
         (!move :to Point-5))))
(defscenario scenario-leader-1()
 (scene1 (otherwise
   (!speak :sentence ‘‘Don’t go through this exit" :duration 6)
   (scenario-escape-to-Exit-B self))))
(defscenario scenario-subjects()
  (scene1 ((?hear :sentence ‘‘Go in that direction")
           (scenario-escape-to-Exit-B self))
          (otherwise (go scene1))))
```

4.2 Test for Meta-level Architecture

The first test is designed for scenario writers to find errors in scenarios and modify it. Scenario *move-around-Exit-A* is assigned to subject *female-01*, which cannot find scenario errors by itself or under the help of other agents. During evacuation simulation, scenario writers find out that *female-01* always moves around Exit A, and cannot get out of the room. He clicks on it, and Q *monitor* shows that *female-01* is executing the scenario *move-around-Exit-A*. So, scenario writer assigns correct scenario *escape-to-Exit-B* to it by using the button of *change scenario*.

The second test is designed for agent to modify scenario errors autonomously through negotiation with others. Scenario *move-around-Exit-C* is assigned to *male-04*. After the error is found by it, agent *male-04* sends "(!require-scenario :for scenario :agent male-04 :goal "escape to Exit B")" to Q *messenger*. *Working memory* of Q *analyzer* gets this message and broadcasts it to all leaders. Agent *leader-01* replies with "(!reply-scenario :for 'scenario-escape-to-Exit-B :by leader-01 :to male-04 :goal "escape to Exit B"))". *Male-04* then accepts the sceanrio and Q *interpreter* starts its execution.

So, the requirements for executing multi-agent scenarios are satisfied by the *meta-level architecture*. However, performance of the *meta-level architecture* is not evaluated in this experiment. we have used legacy systems, such as *FreeWalk* and *Microsoft Agent* to validate multi-agent scenario execution in real applications, and the *execution layer* works well in these systems after the legacy agents are *wrapped* to be scenario enabled. However, the *meta-layer* has not yet been tested in real applications.

5 Conclusions

In this paper, we have introduced Q, a scenario description language, for designing the interaction between agents and humans from the viewpoint of scenario writers. Q does not aim at describing the internal mechanisms of agents, nor the communication and interaction protocols among agents. Rather, it is for describing scenarios representing how humans expect agents to behave. Scenarios can be the interface between application designers and computer experts. After they agree upon *cues* and *actions*, scenario writers describe scenarios using Q syntax, while agent system developers implement *cues* and *actions*. So, Q provides a clear interface between computer professionals and application designers, who have totally different perspectives.

In order to execute multi-agent scenarios, we have realized scenario *interpreter*, which extracts *cues* and *actions* from scenarios and retrieves their execution results from agent system. COM/CORBA are used as the *interface* between Q *messenger* and agent system. Two approaches have been made to simplify scenario writing, IPC for those who are familiar with *MS Excel*, and *a web style GUI* for customizing scenarios. Because there may exist errors in scenarios, and multi-agent systems are intrinsically dynamic, a *meta-level architecture* is proposed for scenario writers to debug errors and agents to negotiate for scenarios. Contributions of this paper are summarized as:

(1) We have developed *a web-style GUI* for scenario writers to generate customized scenarios easily. Combining marked scenarios and GUI definition generates HTML forms for scenario writers to input. After scenario writer submits the input, the CGI engine written in *Scheme* parses the input, rewrites the scenario and begins to execute it.

(2) A *meta-level architecture* for scenario execution has been proposed, with which scenario writers can monitor and control scenario execution to debug scenario errors. When they find errors in agent system, scenario writers can get scenario execution history by different background colors of *cues/actions* and ordinal numbers attached. Scenario writers can debug scenario errors step by step and change scenario execution by control primitives of *start, stop, suspend, resume, step by step, change scenario and add new scenario.*

(3) With the *meta-level architecture*, agents can resolve scenario errors robustly by negotiation with others and conversation with scenario writers. Scenario errors are classified into scenario, *cue/action* and argument levels. Requirement for new scenarios might be satisfied by matching scenarios stored in *scenario DB*, decomposition and merging available scenarios, or negotiation with other agents who have local knowledge. A conversation interface is provided for scenario writer to negotiate with agents to tackle scenario errors at *cue/action* level and argument level. Agents use *action*-like messages to negotiate for scenarios, and scenario writers use natural language to commit commands. Utterance template and patterns are used to interpret the intention of scenario writers.

References

1. Bojinov, H., Casal, A., Hogg, T.: Multiagent Control of Self-reconfigurable Robots. In Proceedings of Fourth International Conference on Multiagent Systems (ICMAS 2000). pp143-150, 2000
2. Cassell, J., Bickmore, T., Billinghurst, M.: Embodiment in Conversational Interfaces: Rea CHI-99. pp520-527, 1999
3. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an Agent Communication Lnguage. International Conference on Information and Knowledge Management (CIKM-94), 1994
4. Huber, M.J., Durfee, E.H.: Deciding When to Commit to Action During Observation-Based Coordination. In Proceedings of First International Conference on Multiagent Systems (ICMAS 1995). pp163-170, 1995
5. Isbister, K., Nakanishi, H., Ishida, T., Nass, C.: Helper Agent: Designing an Assistant for Human-Human Interaction in a Virtual Meeting Space. CHI-00, pp.57-64, 2000
6. Ishida, T., Isbister K. (ed.): Digital Cities: Experiences, Technologies and Future Perspec-tives. Lecture Notes in Computer Science, 1765, Springer-Verlag, 2000
7. Ishida, T., Fukumoto M.: Interaction Design Language Q: The Initial Proposal. Transactions of JSAI, Vol 17, No. 2, pp. 166-169, 2002
8. Jonker, C.M., Treur, J.: An Agent Architecture for Multi-Attribute Negotiation. In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 01). pp.1195-1201, 2001

9. Kitamura, Y., Yamada, T., Kokubo, T., Mawarimichi, Y., Yamamoto, T., Ishida, T.: Interactive Integration of Information Agents on the Web. Klusch, M., Zambonelli, F. (ed.): Cooperative Information Agents V, Springer-Verlag, pp. 1-13, 2001
10. Kuwabara, K., Ishida, T. and Osato, N.: AgentTalk: Describing Multi-agent Coordination Protocols with Inheritance. IEEE Conference on Tools with Artificial Intelligence (TAI-95), pp.460-465, 1995
11. Nakanishi, H., Yoshida, C., Nishimura, T., Ishida, T.: FreeWalk: A 3D Virtual Space for Casual Meetings. IEEE Multimedia. Vol.6, No.2, pp.20-28, 1999
12. Oriatt, S.: Mutual Disambiguation of Recognition Errors in a Multimodal Architecture. CHI-99. pp576-583, 1999
13. Pitt, J., Mamdani, A.: A Protocol-Based Semantics for an Agent Communication Language. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99). pp.486-491, 1999
14. Reeves, B., Nass, C.: The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places. Cambridge University Press, 1996
15. Sugiman, T., Misumi, J.: Development of a New Evacuation Method for Emergencies: Control of Collective Behavior by Emergent Small Groups. Journal of Applied Psychology, Vol. 73, No. 1, pp.3-10, 1988