Introduction to Scenario Description Language *Q*

Toru Ishida and Shohei Yamane
Department of Social Informatics, Kyoto University
ishida@i.kyoto-u.ac.jp, yamane@ai.soc.i.kyoto-u.ac.jp

Abstract

One of the most important attributes of social agents is the ability to fluently interact with human communities. Interaction scenarios are seldom designed by computer professionals, and instead are most often specified by application designers such as sales managers, travel agencies, and school teachers. Our challenge is to extend agent technologies to ensure the social acceptance of agents in various applications. To this end, we have developed Q, a language for designing interaction scenarios among agents and human communities. Q can also act as an interface between computer professionals and application designers. This paper details the specifications of Q, and shows how to design and execute scenarios for social agents.

1. Introduction

The research community tackling agents and multiagent systems has studied and developed various types of software agents. Typically, *personal agents* belong to humans and help their owners to operate or engage in complex computer and communication systems. In this paper, however, we focus on *social agents,* which can be members of a human community: social agents support human-human interaction, while personal agents support human-computer interactions. To understand the nature of social agents, we need a research platform that can support experiments with them. This paper describes a scenario description language for designing interactions among social agents and human communities.

Many of the languages proposed for describing agent behavior are based on agent internal mechanisms. For social agents, however, we should consider protocols among agents and humans. *Q* is a language for describing interaction between agents and humans based on agent external roles. *Q* does not depend on agent internal mechanisms; its goal is to describe how scenario writers should be able to *request* agents to behave.

The change of focus from agent internal mechanisms to interaction scenarios significantly impacts the language syntax and semantics. For example, if an agent accepts only two requests, "on" and "off," *Q* allows scenario writers to use just two commands, "on" and "off." This does not mean that the agent is not intelligent, only that the agent is not controllable once it is turned on. Furthermore, the semantics of commands cannot be known unless they are actually tried. For example, the semantics of the command "move" depend on whether the agent can run rapidly with a light step or move slowly in a thoughtful manner. Since *Q* cannot control the internal mechanism of the agent, *Q* does not have executable functions, such as *Java function calls*, that are often used to explicitly control agent behavior.

To realize *Q*, we extend Scheme[1] by introducing cues, actions and guarded commands. Since Scheme is the mother language of *Q*, all Scheme functions and forms can be used in *Q* scenarios. The new language facilities are explained in the following sections.

2. Variable and value

Variables in *Q* scenarios are symbols that start with the alphabet other than reserved words. It is necessary to declare variables before using them. There are two ways to declare a variable:

1. Declaration of a global variable by "define."
2. Declaration of a local variable by "let", "letrec" and "let*."

There are three ways to assign a value to a variable:

1. Initialization of a variable by "define," "let," "letrec" and "let*."

[1] Scheme, a dialect of the Lisp programming language, was invented by Guy Lewis Steele Jr. and Gerald Jay Sussman.

2. Assignment of a value to a variable by "set!."
3. Assignment of a value to a variable by executing a *cue* and an *action*. (See also "3. Cue and action.")

Assignment of values to variables by cues and actions is not explicit. Variables assigned within cues and actions are called *pattern variables*. The values of pattern variables are always *reference values*. To handle reference values, the following functions are provided.

1. refval *variable*: Function "refval" gets a value that is referred to by "*variable*."
2. set-ref! *variable value*: Function "set-ref" updates a value that is referred to by "*variable*" to "*value*."
3. make-ref *value*: Function "make-ref" creates a pattern variable to refer to "*value*."

It is recommended to use symbols starting with "$" to describe pattern variables. Since pattern variables are one type of variables, it is necessary to declare them in the same fashion as other variables.

3. Cue and action

An event that triggers interaction is called a *cue*. Cues are used to request agents to observe their environment. No cue is permitted to have any side effect. Cues keep on waiting for the event specified until the observation is completed successfully. Once cues have been successfully performed, they return *#t*. Cues are defined by "defcue" as shown below. It is recommended that each cue name begin with "?."

Syntax 1: Definition of cue
(defcue *name*
{(:*keyword* in|out|inout)}*)

Example 1: Definition of cue
(defcue ?position (:name in) (:distance in) (:angle in))
(defcue ?observe (:name in) (:gesture in))
(defcue ?hear (:from out) (:word in))
(defcue ?see (:object in) (:direction in))

In the above definition of cues, "in" and "out" control the flow of data. Their role is to express data flow between the agent and the scenario interpreter. When "in" is specified, a value of the keyword argument is transferred from the scenario interpreter to the agent. On the other hand, when "out" is specified, a value of the keyword argument is transferred from the agent to the scenario interpreter. When "inout" is specified, a value is transferred from/to both sides.

Comparable to cues, *actions* are used to request agents to change their environment. Two types of actions are prepared. One type is *asynchronous actions*, which can be executed in parallel with other actions, and thus the following actions do not have to wait for completion of asynchronous actions. The other type is *synchronous actions*, which cannot be executed in parallel with other actions, and the following actions have to wait for their completion.

Action is defined by "defaction" shown below. Once actions have been successfully performed, they return *#t*. It is recommended that each synchronous action name begin with "!," and asynchronous actions with "!!." If a particular action needs to have both properties, we define both synchronous and asynchronous versions of the action (i.e. "!walk" and "!!walk").

Syntax 2: Definition of action
(defaction *name*
{(:*keyword* in|out|inout)}*)

Example 2: Definition of action
(defaction !walk (:from in) (:to in))
(defaction !!walk (:from in) (:to in))
(defaction !speak (:to in) (:sentence in))
(defaction !approach (:to in) (:distance in) (:angle in))
(defaction !!approach (:to in) (:distance in) (:angle in))

Keyword arguments are to determine various information for executing cues and actions. They are defined in the form that consists of a *keyword symbol* with ":" (colon) and its value. For example, in the description of ":to Jerry," "to" is the keyword symbol and its value is "Jerry."

In the following example, the agent waits for Jerry hears him say "Hello," walks from a bus terminal to a railway station, says "Hello" to Jerry, and observes whether a railway station can be seen to the south. If we use asynchronous action "!!walk" in the above example, the agent says "Hello" to Jerry, just after he starts walking. Asynchronous actions significantly extend the flexibility (and complexity at the same time) available for describing agent scenarios.

Example 3: Execution of cues and actions
(?hear :from Jerry :word "Hello")
(!walk :from "bus_terminal" :to "railway_station")
(!speak :to Jerry :sentence "Hello.")
(?see :object "railway_station" :direction "south")

Unlike functions in programming languages, the semantics of cues and actions are not defined by *Q*. Since cues and actions are executed differently by different agents, their semantics fully depend on the implementation of the agents.

4. Control construct

All Scheme control constructs such as conditional branches and recursive calls can be used in *Q*. In addition, *guarded commands* are introduced for the situations in which we need to observe multiple cues simultaneously.

Syntax 3: Guarded command
(guard {(*cue* {*form*}*)}*
    [(otherwise {*form*}*)])

A guarded command combines cues and *forms*. All cues are sent to the agent as a package, and the agent observes all cues in parallel. After either cue becomes true, the corresponding forms are evaluated sequentially. This parallel observation finishes when either cue becomes true. If multiple cues are satisfied, one of the satisfied cues is selected by the agent. If no cue is satisfied, the "otherwise" clause is evaluated if it exists. The timing when the "otherwise" clause is executed depends on the agent system. The return value of a guarded command is the value of the last evaluated form.

Example 4: Guarded command
```
(guard
   ((?hear :from $agent :word "Hello")
    (!speak :to (refval $agent) :word "Hello"))
   ((?hear :from $agent :word "Fire")
    (!!approach :to "exit" :distance "near")
    (!speak
       :to $agent :sentence "Here is an exit!")))

(guard
  ((?hear :name $agent :word "What's that?")
   (guard
      ((?position
         :name "Kyoto_station" :distance "near")
       (!speak
         :to (refval $agent)
         :sentence "That's Kyoto station."))
      ((?position
         :name "Kyoto_university" :distance "near")
       (!speak
         :to (refval $agent)
         :sentence "That's Kyoto University"))
      (otherwise
       (!speak
         :to (refval $agent)
         :sentence "I can't see anything.")))))
```

The first example indicates how two sensing functions are performed in parallel: the agent hears "Hello," or hears "Fire." In the second example, a nested structure of guarded commands is used: when the agent hears "What's that?" two sensing functions are performed in parallel to determine whether Kyoto station or Kyoto university is closer to it, and if both landmarks are not observed, agent speaks "I can't see anything."

5. Scenario

A *scenario* is used for describing state transitions, and is defined by "defscenario." Each state is defined as a guarded command, but can also include *conditions* in addition to cues. Scenarios can be called from other scenarios or functions.

Syntax 4: Definition of scenario
(defscenario *name*
  ({*variable*}*
    {&key (*variable value*)}*
    {&pattern (*variable value*)}*
    {&aux (*variable value*)}*)
  {(*state*
    {(*testform* {*form*}*)}*
    {(*cue* {*form*}*)}*
    [(otherwise {*form*}*)])}*)

In each state, "*testform*" or "*cue*" can be specified. The former indicates the condition expressions of Scheme. The latter, together with "otherwise," is performed as a guarded command (see also "4. Control construct"). Therefore, if "*testform*" is true, the subsequent forms are executed. If "*testform*" is false, the forms following the observed "*cue*" are evaluated.

For state transition, "(go *state*)" is used in "{*form*}*." It is recommended to give an informative name to each state. In the case that no state transition happens during the execution of "{*form*}*," the scenario terminates. A scenario returns the value of its last form.

Example 5: Definition of scenario
```
(defscenario chat
   (&key (message "Oh! Jiro, good boy!")
    &pattern ($agent #f))
   (greeting
```

```
    ((?hear :from $agent :word "Hello")
      (go messaging))
    ((?hear :from $agent :word "Bye")
      (go farewell)))
  (messaging
    ((equal? (refval $agent) Jiro)
      (!speak :to Jiro :sentence message))
    (otherwise
      (!speak
        :to (refval $agent) :sentence "Hello")))
  (farewell
    (#t
      (!speak
        :to (refval $agent) :sentence "Bye"))))
```

In the above example, a chat scenario consists of three states; "greeting," "messaging," and "farewell." In the "greeting" state, which is the initial state, the agent waits to hear "Hello" or "Bye" from someone. If the agent hears "Hello," it shifts to the "messaging" state. If the agent hears "Bye," it shifts to the "farewell" state. In the "messaging" state, if it appears that the agent hears "Hello" from "Jiro," it tells him "Oh! Jiro, good boy!" which is assigned to variable "message." On the other hand, if it hears "Hello" from some other agent, it replies "Hello." In the "farewell" state, it replies "Bye."

There are two ways to make an agent execute a scenario. One is to set a scenario to "scenario" keyword argument in the definition of the agent (see also "6. Agent"). The other is to execute a scenario as described below.

Syntax 5: Execution of scenario
(*scenario_name agentID arguments*)

In the above definition, "*scenario_name*" indicates the scenario to be invoked, "*agentID*" the agent to execute the scenario, "*arguments*" the arguments necessary to execute the scenario.

Example 6: Execution of scenario
```
(chat Taro :message "Hi, Jiro.")
```

The above example shows how to make agent "Taro" execute scenario "chat." In the same way, scenarios can be invoked within any other scenarios or functions.

Example 7: Scenario invocation in scenario
```
(defscenario chat_1 ()
  (small_talk
    (#t
      (chat self :message "Hi, Jiro."))))
```

Example 8: Scenario invocation in function
```
(defscenario chat_2 ()
  (small_talk
    (#t
      (fchat self "Hi, Jiro."))))

(define (fchat agent message)
  (chat agent :message message))
```

Note that it is necessary to specify an agent ID when executing a scenario. This is because there is no way for the scenario interpreter to refer to the agent that is to execute the scenario. In the above example, the function "fchat" receives agent ID as an argument. In both examples, "self" represents the ID of the agent executing the scenario.

Cues and actions can also be invoked in functions. Before executing cues and actions, it is necessary to bind "self" to the agent ID, because "self" is implicitly referred to in the execution of cues and actions.

Example 9: Execution of cues and actions in function
```
(defscenario greeting ()
  (greeting
    (#t
      (fhello self 3)))

(define (fhello agent distance)
  (let ((self agent)
    ($agent (make-ref #f)))
    (?position :name $agent :distance distance)
    (!speak :to (ref-val $agent) :sentence "Hello.")))
```

In the above example, "self" is bound to argument "agent," when starting function "fhello."

6. Agent

Agents and avatars are defined in the following way.

Syntax 6: Definition of agents and avatars
```
(defagent name
  [:scenario value]
    ;; A scenario assigned to the agent.
  [:population value]
    ;; The number of agents
    ;; generated by this definition.
  {:keyword value}*)

(defavatar name
```

[:scenario *value*]
;; A scenario assigned to the avatar.
[:population *value*]
;; The number of avatars
;; generated by this definition.
{:*keyword value*}*)

Two keyword arguments are required to define an agent or an avatar. One is "scenario" to assign a specific scenario to the agent or the avatar, and another is "population" to specify the number of agents or avatars to be generated with the same scenario. Note that those keyword arguments are optional. If keyword argument "scenario" is omitted, it is necessary to explicitly describe scenario invocations (see also "5. Scenario"). If keyword argument "population" is omitted, the number of agents or avatars is set to one.

Example 10: Definition of agents

```
(defagent evacuation_leader
   :scenario fire_navigation
   :population 1
   :location '(100.0 -10.0 5.0 45.0)
      ;; Agent's initial coordinate.
   :shape '("body1.wrl" "body2.wrl"))
      ;; Agent's image data.

(defagent evacuees
   :scenario fire_drill
   :population 10
   :location '((100.0 -10.0 5.0) (50.0 10.0 1.0))
      ;; Area where agents are to show up.
   :shape '(("body1.wrl" "body2.wrl")))
      ;; Agents' image data.
```

The first example defines agent "evacuation_leader" to execute scenario "fire_navigation." Keyword arguments "location" and "shape" specifies details for creating agents. The values of "location" and "shape" represent agent's initial coordinate and image data. The second example defines ten "evacuees," executing the same scenario "fire_drill."

A unique ID is assigned to each agent or avatar. This ID can be acquired by evaluating agent/avatar's name. In the case that "population" is more than two, the list of IDs is bound to the name of agents. By using "list-ref" function, each ID in the list can be obtained. The following example shows how to refer to the ID of the fourth agent in a group of "evacuees."

Example 11: Refer to agent ID in group

```
(list-ref evacuees 3)
```

In most cases, an agent is defined with a scenario to identify how the agent should perform. Even if a crowd of agents executes the same scenario, different agents exhibit different actions, because they interact with their own environment. Avatars controlled by humans usually do not require any scenario. Avatars can have scenarios, however, if it is beneficial to constrain their behaviors.

7. Environment

To define the environment of agents and avatars, "defenv" is used optionally.

Syntax 7: Definition of environment

(defenv *name*
{:*keyword value*}*)

Keyword arguments and their values are predefined by each agent system.

Example 12: Definition of environment

```
(defenv virtual_space
   :max_entry 100
      ;; Maximum number of agents (integer)
   :ip_address "i.kyoto-u.ac.jp"
      ;; IP address to connect (string)
   :port_number 12345
      ;; Port number to connect (integer)
   :vrml_url
      "http://www.ai.i.kyoto-u.ac.jp/shijo_street/"
      ;; URL of VRML source codes (string)
   :vrml_path "data\shijo_street\")
      ;; Relative path to the resource directory (string)
```

In the example above, environment "virtual_space" is defined. This environment can handle up to 100 agents. The IP address and the port number are for connecting to "virtual_space." The URL of the image data is also specified. Finally, a relative path to the local directory is identified.

8. Complete examples

To create complete examples, let us start with Microsoft agents. The action "!gesture" can perform any of the 60 different gestures known to Microsoft agents. Let's assume that we would like to control Microsoft agent Marlin as follows: if the agent finds that the user is looking at "http://www.news.com/," it says "Hello" to him/her with a gesture and shows some of today's topics, and asks him/her to select one of

them. The corresponding scenario in *Q* is given below. The agent keeps waiting until the user makes a choice.

Example 13: Menu selection

```
(defscenario Web_navigation
    (&pattern ($x #f) &aux (time #f))
 (introduction
   ((?watch_web :url "http://www.news.com/")
    (!gesture :animation "Greet")
    (!speak :sentence "Hello, I am Marlin.")
    (!fly :x 50 :y 300)
    (go selection)))
 (selection
    (!display :url "http://www.news.com/topics.html")
    (!speak :sentence "What are you interested in?")
    (!ask :selection '("Japanese Kimono"
                       "Sumo Wrestling"
                       "Sudoku"
                       "Nothing"))
    (?receive :selection $x)
    (go agent-initiative)))
```

Assume that we place the following task on Merlin: if the user wants to learn more about Japanese traditional clothes, Kimono, then visit the Kimono Web site and let the user freely click Web pages in the site. Each time the user visits a new page, summarize its content (see Figure 1). If the user does not react for a while, move to the next subject.

Example 14: Mixed initiative Web navigation

```
 (agent-initiative
   (#t
     (!speak
       :sentence "Hm-hum, you are so enthusiastic.")
     (!speak :sentence "Then, how about this page?")
     (!display :url "http://kimono.com/index.htm")
     (go user-initiative)))
 (user-initiative
   ((?watch_web :url "http://kimono.com/type.htm")
     (!speak :sentence "There are many types of obi.")
     (!speak :sentence "Can you tell the difference?")
     (go user-initiative))
   ((?watch_web :url "http://kimono.com/fukuro.htm")
     (!gesture :animation "GestureLeft")
     (!speak
       :sentence "Fukuro obi is for a ceremonial dress.")
     (!speak :sentence "Use it at a dress-up party!")
     (go user-initiative))
   ((?watch_web :url "http://kimono.com/maru.htm")
     (scenario_for_maruobi self)
     (go user-initiative))
   ((?timeout :time 20)
     (go change-topic)))
 (change-topic
   (otherwise
     (!speak
       :sentence "Did you enjoy Japanese Kimono?")
     (!speak
       :sentence "OK, let's go to the next topic.")
     (go selection))))
```

Figure 1. Microsoft agent introduces Kimono

Another complete example is a scenario for a FreeWalk agent, which navigates a user ("you" in the following example) to the safe exit. After the scenario starts, if you approach the agent (a woman with red clothes), she will approach you and ask "Where is the exit?"

Example 15: Move to talk (see Figure 2)

```
 (defscenario evacuation_guidance()
   (approach
     (#t
       (!walk :route '((1.5 0.0)))
       (?position :name you :distance "near")
       (!!face :to you)
       (!speak :to you :sentence "Where is the exit?")
       (go ask)))
```

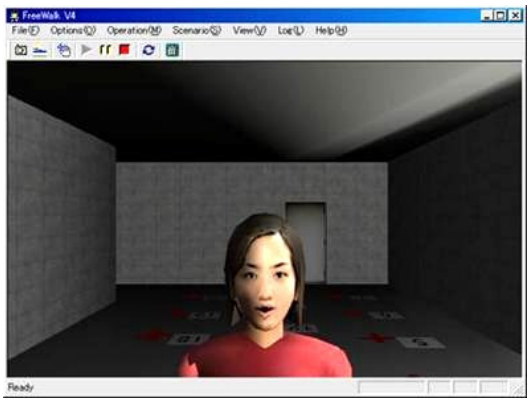

Figure 2. FreeWalk agent talks to you

If you step away from her, she will follow you. When you point at the corner, she will walk to the corner you pointed out.

Example 16: Ask direction (see Figure 3)

```
(ask
  ((?position :name you :angle "rear")
    (!turn :to you)
    (!speak :to you :sentence "Please tell me.")
    (go ask))
  ((?position :name you :distance "far")
    (!turn :to you)
    (!approach :to you :distance "close")
    (!speak :to you :sentence "Where will you go?")
    (go ask))
  ((?position :name you
      :point_angle "angle_to_corner")
    (!!face :to "corner")
    (!turn :to "corner")
    (!speak :to you :sentence "OK.")
    (!walk :route "path_to_corner")
    (!!face :to you)
    (!turn :to you)
    (!speak :to you :sentence "Let's go together!")
    (go together)))
```

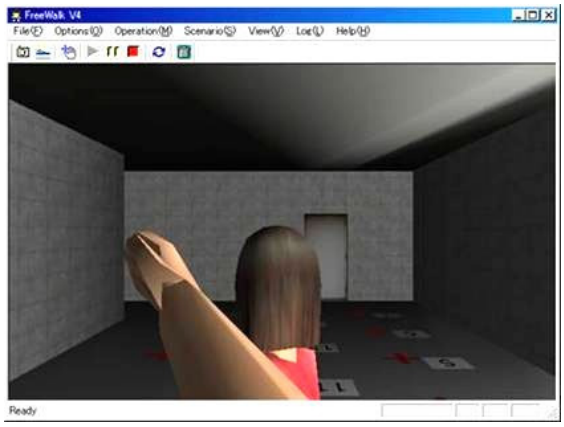

Figure 3. FreeWalk agent asks direction

The agent walks to the corner. When she reaches it, she asks you to go with her. When you reach the agent, she will walk to the exit. If you don't follow her, she will turn to you and ask you to go with her.

Example 17: Go together (see Figure 4)

```
(together
  ((?hear :from you)
    (!speak :to you  :sentence "What happened?")
    (!behave :gesture "Come")
    (!speak :to you :sentence "This way.")
    (go together))
  ((?position :name you :distance "near")
    (!face :to "exit")
    (!turn :to "exit")
    (go exit)))
```

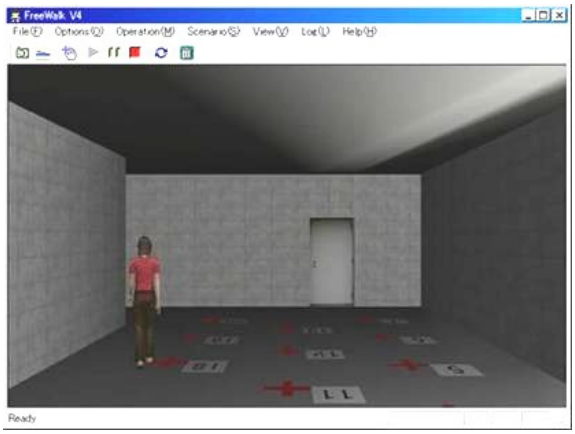

Figure 4. FreeWalk agent goes together

When she reaches the exit, she asks you to go ahead. After you exit the room, the agent follows you.

Example 18: Evacuation guidance (see Figure 5)

```
(exit
  ((?position :name agent
      :from "exit" :distance "near")
    (!turn :to you)
    (!speak :to you
      :sentence "We arrived at the exit. After you."))
  ((?position :name you :distance "near")
    (!walk :route "path_to_exit")
    (go exit))))
```

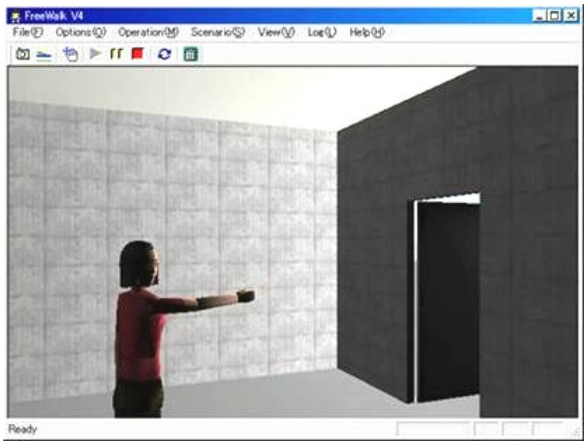

Figure 5. FreeWalk agent shows the exit

9. Conclusion

To create social agents, we need to include views from application designers. To this end, we developed *Q* to describe interaction scenarios among agents and humans [3]. Scenarios also encourage dialogs between agent designers and application designers [14].

Q is a successor of AgenTalk [7]. Compared to previous agent description languages, however, we cannot assume the correctness of *Q* scenarios. Rather we need to provide a training environment for agents to create robust and adaptive agent coordination behavior. In this environment, errors in scenarios are corrected not by debugging codes but by rehearsals [19]. Metaphors of human communities are to be introduced in developing social agents.

Q has been already combined with Microsoft agents for Web navigation [6], FreeWalk [10] for virtual city experiments [4], Cormas for natural resource simulations [1], and Caribbean [18] for massively multiagent simulations. The reason why totally different agent systems can be connected is that *Q* is a language for describing requests to social agents, and does not depend on agent internal mechanisms. FreeWalk/*Q* [12], Cormas/*Q* [17] and Caribbean/*Q* [8] have been used for assisting communication in cyberspace [2], social psychological experiments [11], indoor / outdoor disaster evacuation experiments [9,13], modelling agents for virtual training [15], and simulating agricultural economics [16]. Various applications have confirmed the feasibility and usefulness of *Q* for designing social agents [5].

Acknowledgement

This work would not have been possible without the contributions of K. Isbister, Y. Nakajima, H. Nakanishi, Y. Murakami, D. Torii and A. Yamamoto. *Q* has been developed under the JST CREST digital city project.

References

[1] F. Bousquet, I. Bakam, H. Proton and C. Le Page. Cormas: Common-Pool Resources and Multiagent Systems. *LNAI 1416*, pp. 826-838, 1998.

[2] K. Isbister, H. Nakanishi, T. Ishida, and C. Nass. Helper Agent: Designing an Assistant for Human-Human Interaction in a Virtual Meeting Space. *CHI-00*, pp. 57-64, 2000.

[3] T. Ishida. *Q*: A Scenario Description Language for Interactive Agents. *IEEE Computer*, Vol. 35, No. 11, pp. 54-59, 2002.

[4] T. Ishida. Digital City Kyoto: Social Information Infrastructure for Everyday Life. *Communications of the ACM* , Vol. 45, No. 7, pp. 76-81, 2002.

[5] T. Ishida, Y. Nakajima, Y. Murakami and H. Nakanishi. Augmented Experiment: Participatory Design with Multiagent Simulation. *IJCAI-07*, 2007.

[6] Y. Kitamura, T. Yamada, T. Kokubo, Y. Mawarimichi, T. Yamamoto, T. Ishida. *LNCS*, 2182, pp. 1-13, 2001

[7] K. Kuwabara, T. Ishida and N. Osato. AgenTalk: Describing Multiagent Coordination Protocols with Inheritance. *IEEE Conference on Tools with Artificial Intelligence (TAI-95)*, pp.460-465, 1995.

[8] Y. Nakajima, H. Shiina, S. Yamane, H. Yamaki and T. Ishida. Caribbean/Q: A Massively Multi-Agent Platform with Scenario Description, *International Conference on Semantics, Knowledge and Grid (SKG-06)*, 2006.

[9] Y. Nakajima, H. Shiina, S. Yamane, H. Yamaki and T. Ishida. Disaster Evacuation Guide Using a Massively Multiagent Server and GPS Mobile Phones. *IEEE/IPSJ Symposium on Applications and the Internet (SAINT-07)*, 2007.

[10] H. Nakanishi, C. Yoshida, T. Nishimura and T. Ishida. FreeWalk: A 3D Virtual Space for Casual Meetings. *IEEE Multimedia*, Vol.6, No.2, pp.20-28, 1999.

[11] H. Nakanishi, S. Nakazawa, T. Ishida, K. Takanashi and K. Isbister. Can Software Agents Influence Human Relations? Balance Theory in Agent-mediated Communities. *AAMAS-03,* pp. 717-724, 2003.

[12] H. Nakanishi and T. Ishida. FreeWalk/*Q*: Social Interaction Platform in Virtual Space. *VRST-04*, pp. 97-104, 2004.

[13] H. Nakanishi, S. Koizumi, T. Ishida and H. Ito. Transcendent Communication: Location-Based Guidance for Large-Scale Public Spaces. *CHI-04*, pp. 655-662, 2004.

[14] Y. Murakami, T. Ishida, T. Kawasoe and R. Hishiyama. Scenario Description for Multi-Agent Simulation. *AAMAS-03,* pp. 369-376, 2003.

[15] Y. Murakami, Y. Sugimoto and T. Ishida. Modeling Human Behavior for Virtual Training Systems. *AAAI-05*, pp. 127-132, 2005.

[16] D. Torii, T. Ishida and F. Bousquet. Modeling Agents and Interactions in Agricultural Economics. *AAMAS-06*, pp. 81-88, 2006.

[17] D. Torii, T. Ishida, S. Bonneaud and A. Drogoul. Layering Social Interaction Scenarios on Environmental Simulation. *LNAI*, 3415, pp.78-88, 2004.

[18] G. Yamamoto and Y. Nakamura. Architecture and Performance Evaluation of a Massive Multi-Agent System. *Agents-99*, pp. 319-325, 1999.

[19] S. Yamane and T. Ishida. Meta-level Control Architecture for Massively Multiagent Simulations. *Winter Simulation Conference (WSC-06)*, pp. 889-896, 2006.