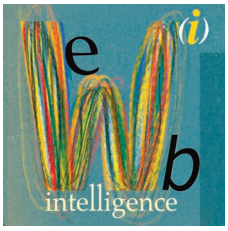

Q: A Scenario Description Language for Interactive Agents

The *Q* language—which describes interaction scenarios between agents and users based on agent external roles—provides an interface between computing professionals and scenario writers.

Toru Ishida
Kyoto University

Agent internal mechanisms form the basis for many of the languages proposed for describing agent behavior and interagent protocols. These mechanisms include Soar,[1] a general cognitive architecture for developing systems that exhibit intelligent behavior, and the Knowledge Query and Manipulation Language,[2] a language and protocol for developing large-scale sharable and reusable knowledge bases.

For the Web, however, we should also consider the needs of application designers such as sales managers, travel agents, and schoolteachers. To this end, we have been developing *Q*, a language for describing interaction scenarios between agents and users based on agent external roles. *Q* can also provide an interface between computing professionals and scenario writers. Rather than depending on agent internal mechanisms, *Q* seeks to describe how scenario writers should request that agents behave.

The change in focus from agent internal mechanisms to interaction scenarios significantly affects the language's syntax and semantics. For example, if an agent accepts only two requests, on and off, *Q* lets scenario writers use only the commands on and off. This does not mean the agent lacks intelligence, only that it is not controllable.

Further, the only way to know the semantics of commands is to try them. For example, the semantics of the move command depend on whether the agent can run rapidly with a light step or move slowly, in a thoughtful manner. Since *Q* cannot control the agent's internal mechanism, it cannot use functions—such as Java function calls—to implement detailed agent behavior.

Scenarios also help establish a bridge between the computing professionals who design agents and the scenario writers who design applications. We can expect an effective dialog to emerge from the interplay between the two different perspectives during the process of formalizing interaction patterns.

DESCRIBING SCENARIOS

Q is an extension of Scheme, a Lisp programming language dialect. We introduce sensing and acting functions and guarded commands in Scheme. Since Scheme is *Q*'s mother language, all Scheme functions and forms can be used in any *Q* scenario.

Cue and action

A *cue* is an event that triggers interaction. Scenario writers use cues to request that agents observe their environment. No cue can have any side effect. Cues keep on waiting for the specified event until the observation successfully completes.

Comparable to cues, *actions* request that agents change their environment. Unlike functions in programming languages, *Q* scenarios do not define the semantics of cues and actions. Since different agents execute cues and actions differently, their semantics fully depend on the agent. The following example shows cues, which start with a question mark, and actions, which start with an exclamation point:

```
(?hear "Hello" :from Jerry)
(!walk :from bus_terminal
       :to railway_station)
(!speak "Hello" :to Jerry)
```


0018-9162/02/$17.00 © 2002 IEEE

```
(?see railway_station
       :direction south)
```

In this example, the agent waits for Jerry to say hello, Jerry walks from the bus terminal to a railway station, the agent says hello to Jerry and then asks whether he can see the railway station to the south. These *synchronous* actions return after completion.

Asynchronous actions, however, execute in parallel. For example, *walk* can be an asynchronous action, since we can speak and walk at the same time. To represent asynchronous actions, we use the notation `!!walk`. If we use `!!walk` in the preceding example, the agent says hello to Jerry just after he starts walking. Asynchronous actions significantly extend both the flexibility and the complexity of agent scenario descriptions.

Guarded commands

Q can use all Scheme control structures, such as conditional branches and recursive calls. In addition, the language introduces guarded commands for use in situations that require observing multiple cues simultaneously. A guarded command combines cues and forms, including actions. After either cue becomes true, the guarded command evaluates the corresponding form. If no cue is satisfied, it evaluates the `otherwise` clause, if present, as follows:

```
(guard
  ((?hear "Hello" :from Jerry)
   (!speak "Hello" :to Jerry) .... )
  ((?see railway_station
          :direction south)
   (!walk :from bus_terminal
          :to railway_station) .... )
(otherwise
   (!send "I am still waiting"
          :to Tom) .... )))
```

In this example, if one of the cues is observed—if the agent hears Jerry say hello—the corresponding forms are performed afterward: The agent says hello to Jerry. If the guarded command does not observe any cue, it performs the `otherwise` clause, and the agent sends the message "I am still waiting" to Tom.

Scenarios

A scenario describes state transitions. Scenarios can be called from other scenarios. The scenario defines each state as a guarded command, but it can include conditions in addition to cues.

Writers draft scenarios in the form of simple state transitions, which can describe fairly complex tasks since any form can be evaluated in the body of states. Scenarios can be called recursively, Scheme functions can be called in scenarios, and any scenario can be called in Scheme functions.

```
(defscenario reception (msg)
  (scene1
    ((?hear "Hello" :from $x)
     (!speak "Hello":to $x)
     (go scene2))
    ((?hear  "Bye")
     (go scene3)))
  (scene2
    ((?hear "Hello" :from $x)
     (!speak "Yes, may I help you?"
                 :to $x))
    (otherwise (go scene3)))
  (scene3 ...))
```

In this example, each scenario defines states as `scene1`, `scene2`, and so on. The same observation yields different actions in different states. In `scene1`, the agent says hello when it hears someone say hello. In `scene2`, however, the agent responds with "Yes, may I help you?" when it hears hello again.

Asynchronous actions extend both the flexibility and the complexity of agent scenario descriptions.

Agents and avatars

Agents, avatars, and a crowd of agents are defined as follows.

```
(defagent Tom :scenario
  'guide_for_sightseeing)
(defavatar Jerry)
(defcrowd pedestrian :scenario
  'sightseeing :population 30)
```

In Q, scenarios specify what the agents must do. Even if a crowd of agents executes the same scenario, the agents exhibit different actions as they interact with their environment, which includes other agents and human-controlled avatars.

MICROSOFT AGENTS EXAMPLE

We can use Microsoft agents to get a feel for how scenarios work. Assume that we assign the following task to the Microsoft agent named Merlin: Let a user who wants to learn more about the traditional Japanese kimono visit the kimono Web site and freely click Web pages. Each time the user visits a new page, the agent summarizes its content. If the user does not react for a while, the agent moves to the next subject.

```
(defscenario card14 ()
  (scene1
    (otherwise
      (!speak "Hm-hum, you are so enthusiastic.")
      (!speak "Then, how about this page?")
      (!display :url "http://kimono.com/index.
        htm")
      (go scene2)))
  (scene2
    ((?watch_web :url "http://kimono.com/type.
      htm")
      (!speak "There are many types of obi.")
      (!speak "Can you tell the difference?")
      (go scene2))
    ((?watch_web :url "http://kimono.com/fukuro.
      htm")
      (!gesture :animation "GestureLeft")
      (!speak "Fukuro obi is for a ceremonial dress.")
      (!speak "Use it at a dress-up party!")
      (go scene2))
    ((?watch_web :url "http://kimono.com/maru.htm")
      (card42 self)
      (go scene2))
    ((?timeout :time 20)
      (go scene3)))
  (scene3
    (otherwise
      (!speak "Did you enjoy Japanese Kimono?")
      (!speak "OK, let's go to the next subject."))))
```

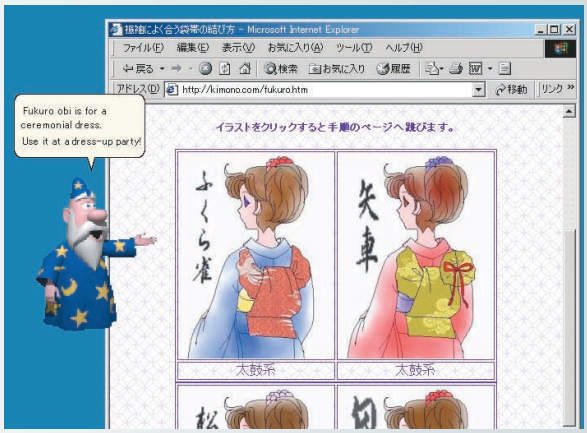

Figure 1. Sample Q scenario. Written by Masahito Fukumoto and Akishige Yamamoto in Q, the scenario uses the agent Merlin to guide users to Web sites and pages with information about the Japanese kimono.

Figure 1 shows this Q scenario and its outcome. The action `!gesture` can perform any of the 60 different gestures that Microsoft agents support.

Although computing professionals can work with Q easily, scenario writers may not be familiar with the Scheme syntax. Further, since Q is a general-purpose scenario description language, it grants too much freedom for describing scenarios for specific domains. We thus introduced interaction pattern cards (IPCs) to capture the interaction patterns in each domain.

Figure 2 shows an IPC equivalent of the Q scenario shown in Figure 1. Scenario writers can use Excel to fill in the card. The IPC translator then generates a Q scenario according to the card's contents and predefined semantics. Note that IPC is not merely an Excel interface to Q. Rather, it provides a pattern language, and so it should be carefully designed by analyzing the interactions in each domain.

We have used Q and Microsoft agents to develop a multicharacter interface for information retrieval in which domain-specific search agents cooperate to satisfy users' queries.[3] Previous research often used blackboard systems to integrate the results from multiple agents. However, given that computing professionals develop search agents independently, attempts to integrate these results are often unsatisfactory.

Thus, we have taken a totally different approach. Instead of integrating the results at the back end, our interface displays the integration process to users as a dialogue involving multiple characters, each of which represents a different search agent. Users can observe the collaboration process among the agents and join the conversation if necessary. This multicharacter interface increases user satisfaction by integrating the results socially at the front end.

DESIGNING SCENARIOS

A diverse variety of applications use interactive agents on the Web.[4] To allow application designers to use fairly complex agents, we use Q in a scenario design process that provides a clear interface between application designers and computing professionals.

Q architecture

Figure 3 shows the Q architecture for handling scenarios. When a particular agent receives a Q scenario, the corresponding agent system creates a Q processor to execute the scenario. This agent system can host multiple agents. For example, hundreds of agents can coexist in the FreeWalk[5] 3D virtual space. Although we implemented Q's processors in Scheme, we provide the program interface to C++

Card ID	14	Card name	Visiting kimono Web site	Card type	User initiative
Opening	Action				
	Hm-hum, you are so enthusiastic. Then, how about this page? http://www.kimono.com/index.htm				
Reactions to users' mouse click repeat	Mouse click	Cue	Action		
		http://kimono.com/type.htm	There are many types of obi. Can you tell the difference?		
		http://kimono.com/fukuro.htm	(GestureLeft) Fukuro obi is for a ceremonial dress. Use it at a dress-up party!		
		http://kimono.com/maru.htmco	(Evaluate card42)		
	No reaction	Seconds	Action		
		20	(End of repeat)		
Closing	Action				
	Did you enjoy Japanese kimono? OK, let's move on to the next subject.				

Figure 2. Interaction pattern card by Yohei Murakami. This example, which displays the same information contained in Figure 1's Q scenario, provides an Excel interface to a pattern language.

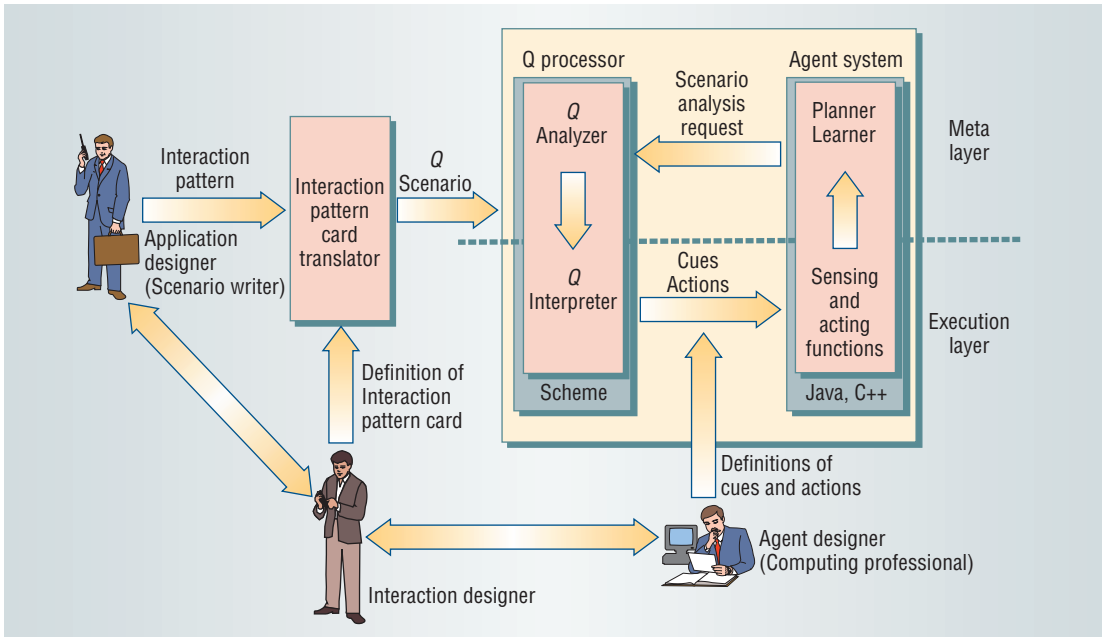

Figure 3. Q architecture. In the execution layer, the Q interpreter asks agents to execute cues and actions. If the agents find a problem in a given scenario, control transfers to the metalayer, where the Q analyzer can investigate the scenario.

and Java software, so we can easily combine Q and legacy agent systems.

The Q interpreter can execute hundreds of scenarios simultaneously. A problem arises when executing concurrent scenarios, however: Execution results can differ with each run. To avoid this, we selected Scheme as Q's mother language because the interpreter can use Scheme's *continuation* to completely control process switching.

The Q architecture consists of two layers. In the execution layer, the Q interpreter asks agents to execute cues and actions. If the agents find any problem in a given scenario, the execution layer transfers control to the metalayer. In the metalayer, agents can request the Q analyzer to investigate the scenario. Based on the report from the Q analyzer, the agents can enter into negotiations with the scenario writers.

The Q architecture has some similarities to the three-layer architectures for robot planning, which consist of a control layer, sequencing layer, and deliberative layer.[6] The first and third layers correspond to Q's execution layer and metalayer, respectively. The Q scenario can be seen as a sequencing layer that scenario writers prepare outside the agent.

Design process

The three-step process of creating a scenario focuses on a dialogue that bridges two perspectives.

- First, a scenario writer and a computing professional agree upon the cues and actions to use as the interface between them. Rather than using cues and actions assigned a priori, the two parties define a scenario vocabulary for

Table 1. Sample cues and actions for FreeWalk agents.

Function	Cue	Action
Motion	?position (get location and orientation)	*Movement*
	?observe (observe gestures and actions)	!walk* (walk along a route)
	?see (see objects)	!approach* (approach other agent)
		!block* (block another agent)
		Rotation
		!turn* (turn body)
		!face* (turn head)
		Gesture
		!point* (point at object)
		!behave* (perform some gesture)
		Appearance
		!appear (show up)
		!disappear (erase self)
Conversation	?hear (hear voice)	!speak* (speak by voice)
	?receive (receive text messages)	!send* (send text messages)
	?answer (receive an answer to questions)	!ask* (ask questions)
Miscellaneous	?finish (finish asynchronous actions)	!change (mode change)
	?input (key input by users)	!finish (stop asynchronous actions)
		!output (output logs)

*An asterisk indicates the action can be asynchronous.

Figure 4. Virtual subway station in Kyoto. Using current technology, developers can populate a virtual city like this one by Hideyuki Nakanishi with hundreds of agents and more than a score of human-controlled avatars.

each application domain through negotiation.

- Second, the scenario writer uses Q to describe a scenario, while the computing professional implements cues and actions.
- Third, the design process can introduce another actor, the *interaction designer*. This third actor observes the patterns of interaction in each domain and proposes IPCs. These cards trigger a dialogue between scenario writers and interaction designers, leading to a better understanding of the interaction patterns in each domain. IPCs also improve a scenario writer's productivity.

Agents can be autonomous or dependent. If autonomous, scenarios can be simple; if not, the scenario writer should specify all details. The granularity of cues and actions depends on two independent factors: the level of agent autonomy and the degree of precision scenario writers require.

APPLYING SCENARIOS

Using Q to create interdisciplinary 3D Web applications presents an interesting challenge. FreeWalk, a video conference tool that supports casual meetings within 3D communities, can create virtual cities that exactly mirror their real-world counterparts.

Table 1 summarizes the cues and actions for FreeWalk agents, and Figure 4 displays a view of such agents that Q scenarios control. Using the power of current technology, we can populate each virtual city with hundreds of agents, all of which walk around in real time.[7]

We are using virtual cities to create crisis management simulations that involve humans and agents. This pilot application links computer scientists, architects, and social psychologists. In these simulations, agents act as pedestrians, security guards, and so on. We can create realistic evacuation simulations by having pedestrian agents act as humans running around trying to escape. Such simulations will contribute to the discovery of typical human mistakes in these situations and will help train people to make correct decisions in real crises.

Figure 5 shows 2D and 3D simulations of how humans behave when a crisis occurs in a small room. A comparison of the results obtained from the ongoing simulation to previous findings[8] confirmed that we have succeeded in making multiagent simulations guided by Q scenarios sufficiently realistic.

Based on this experiment, we plan to conduct crisis management simulations in virtual Kyoto's subway and railway stations. The simulation will include 20 or more people-controlled avatars connected via the Internet and hundreds of agents controlled by Q scenarios.

As more humans and agents coexist in the Internet, describing multiagent scenarios will become essential for describing interaction scenarios and providing an interface between computing professionals and scenario writers. Rather than depending on agent internal mechanisms, the Q scenario description language seeks to describe how scenario writers should request that agents behave. In such experiments, if agents are completely autono-

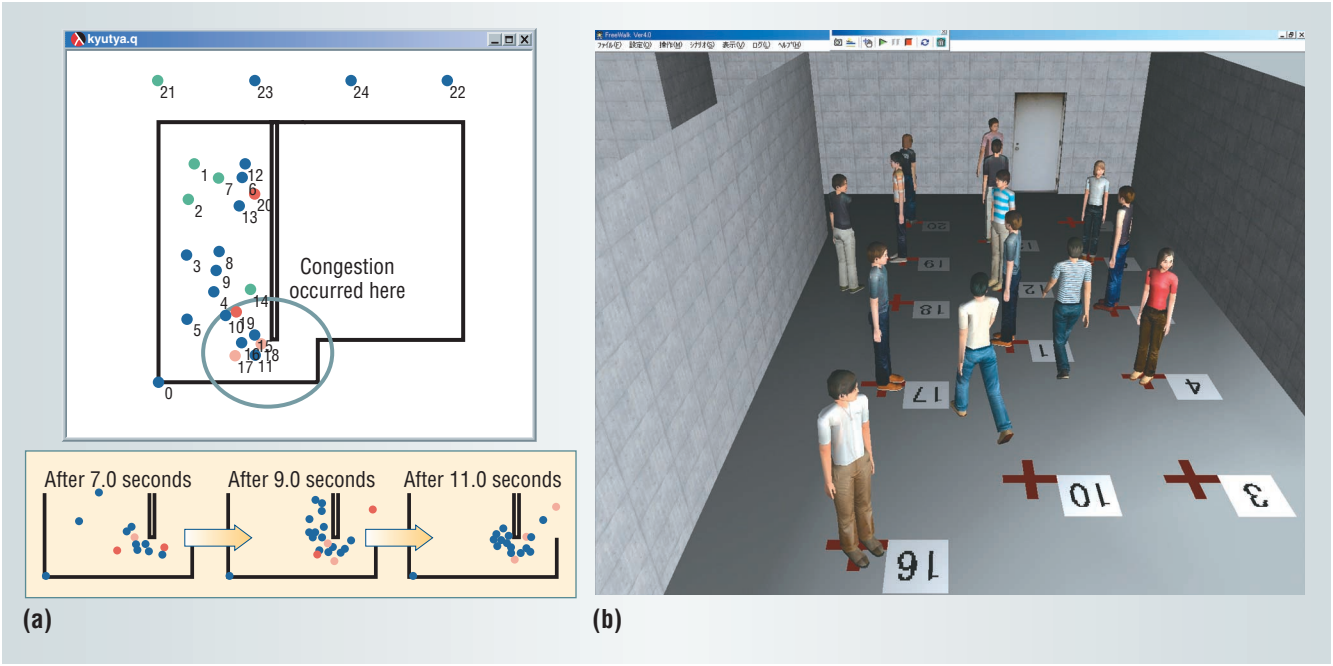

Figure 5. Evacuation simulation. (a) 2D simulation by Tomoyuki Kawasoe and Kazuhisa Minami shows a regularly updated overhead graphic of agents' responses to a small-room crisis, while (b) the 3D simulation generates a real-time animation of the same situation.

mous, controlling the entire system becomes difficult. Thus, *Q* uses scenarios to represent social constraints so that agents can use them to guide their behavior. Future research includes how agents should behave under given social constraints. ■

Acknowledgments

Thanks to Masahito Fukumoto, Reiko Hishiyama, Hideaki Ito, Tomoyuki Kawasoe, Yasuhiko Kitamura, Kazuhisa Minami, Yohei Murakami, Hideyuki Nakanishi, Shiro Takata, Ken Tsutsuguchi, and Akishige Yamamoto for making this work possible. The Department of Social Informatics at Kyoto University and JST CREST Digital City developed Project FreeWalk and *Q*. The source code for both is available at http://www.digitalcity.jst.go.jp/Q/.

References

1. J.E. Laird and P. Rosenbloom, "The Evolution of the SOAR Cognitive Architecture," D.M. Steier and T.M. Mitchell, eds., *Mind Matters: A Tribute to Allen Newell*, Lawrence Erlbaum, Hillsdale, N.J., 1996, pp. 1-50.
2. T. Finin et al., "KQML as an Agent Communication Language," *Proc. Int'l Conf. Information and Knowledge Management*, ACM Press, New York, 1994, pp. 456-463.
3. Y. Kitamura et al., "Interactive Integration of Information Agents on the Web," *Cooperative Information Agents V*, M. Klusch and F. Zambonelli, eds., Springer-Verlag, New York, 2001, pp. 1-13.
4. Y.Y. Yao et al., "Web Intelligence," *Web Intelligence: Research and Development*, N. Zhong et al., eds., Lecture Notes in Artificial Intelligence 2198 (LNAI 2198), Springer-Verlag, Heidelberg, 2001, pp. 1-17.
5. H. Nakanishi et al., "FreeWalk: A 3D Virtual Space for Casual Meetings," *IEEE MultiMedia*, vol. 6, no. 2, 1999, pp. 20-28.
6. E. Gat, "Three-Layer Architectures," *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, D. Kortenkamp, R.P. Bonasso, and R. Murphy, eds., MIT Press, Cambridge, Mass., 1998, pp. 195-210.
7. T. Ishida, "Digital City Kyoto: Social Information Infrastructure for Everyday Life," *Comm. ACM*, vol. 45, no. 7, 2002, pp. 76-81.
8. T. Sugiman and J. Misumi, "Development of a New Evacuation Method for Emergencies: Control of Collective Behavior by Emergent Small Groups," *J. Applied Psychology*, vol. 73, no. 1, 1988, pp. 3-10.

***Toru Ishida** is a professor of social informatics at Kyoto University and a research professor at NTT Communication Science Laboratories. His research interests include multiagent systems, the human-centered semantic Web, digital cities, and community computing. Ishida received a PhD in information science from Kyoto University. Contact him at ishida@i.kyoto-u.ac.jp.*