Scenario Mechanism in Agent-Oriented Programming *

Rui Shen, Ji Wang
National Laboratory for
Parallel and Distributed Processing
Changsha, 410073, China
shenrui98@yahoo.com, jiwang@mail.edu.cn

Hong Zhu
Department of Computing
Oxford Brookes University
Oxford, OX33 1HX, United Kingdom
hzhu@brookes.ac.uk

Abstract

Scenario has been used to describe agent behaviors in the context of environment situations in the specification languages for agent-based systems, such as SLABS. It becomes an important language facility in the declaration of an agent for specifying its behaviors in its environment. Therefore, towards agent-oriented programming, it is necessary to introduce and implement scenario mechanism in programming languages. This paper reports our attempts to support the language facility from the view of programming languages, and presents an approach to facilitating the scenario mechanism in agent-oriented programming. The basic idea is to extend object-oriented programming language to support agent-oriented programming, where Java is chosen as the base language. Firstly, the language framework of agent-oriented programming, SLABSp is presented, mostly conforming to SLABS, whose syntax is extended based on Java. Scenario mechanism is introduced as the new feature in the programming language. Secondly, the underlying object models are defined to serve as the semantics of the language, where agents are modeled by a couple of objects. A compiler has been built to compile the agent-oriented programs into Java. A running platform has been constructed as the multi-agent runtime environment of SLABSp.

1. Introduction

Since agent-oriented programming [6, 7, 8] was presented as a new paradigm, many attempts have been made to develop actual, useful agent-oriented systems. It is desired to develop agent-oriented programming languages as well as programming environments [1, 5, 6].

*Supported by the National Science Foundation of China under grant No. 60233020 and 90104007, the National High Technology Development 863 Program of China under grant No. 2002AA116070.

Scenario [3, 9, 10, 12] has been used to describe the agent behaviors in the context of environment situations in the specification languages for agent-based systems, such as SLABS [12]. As described by SLABS, a scenario is a set of typical combinations of the behaviors and states of related agents in the system. Scenarios can be used in modeling agents' behaviors of multi-agent systems. It becomes an important language facility in the declaration of agents for specifying their behaviors in the environment. However, scenario facility has not been supported at the level of programming languages. Due to the lack of facilities which can clearly state how agents' behaviors are related to their environment (related agents), developing agent-oriented applications is still complicated and difficult. Therefore, it is necessary to introduce and implement scenario mechanism in programming languages.

This paper reports our attempts to support the language facilities and features in SLABS from the view of programming languages, and presents an approach to facilitating the scenario mechanism in agent-oriented programming. The basic idea is to support agent-oriented programming by extending object-oriented programming languages, where Java is chosen as the base language. As a result, a compiler is built to compile agent-oriented programs into Java. A simple platform is constructed as the multi-agent runtime environment.

The remainder of this paper is organized as follows. Section 2 gives the overview of SLABS language. Section 3 defines the SLABSp language. Section 4 describes the implementation of SLABSp, including the underlying object model and patterns of the agent system empowered by scenario mechanism. Section 5 reviews the related work on agent-oriented programming. Section 6 concludes the paper with a brief discussion of future work.

2. Overview of SLABS

SLABS is a model-based formal specification language for multi-agent systems [12]. It integrates a number of novel

language facilities to support the development of agent-based systems. The set of language facilities includes a modular structure suitable for the formal specification of multi-agent systems, a scenario description mechanism for defining agents' behaviors in the context of environment situations, and a notion of caste as a collection of agents that have the same behaviors and structural characteristics.

SLABS defines agents as encapsulations of data, operations and behavior rules, and each agent has its own rules that govern its behavior. An agent can have 'visible' and 'invisible' data and operations to other agents.

A caste [11, 12, 13] is defined as a set of agents with the same structural and behavioral characteristics. Similar to the relation between objects and classes, agents are members of castes, and inheritance relationships can be defined between castes.

Since object can be regarded as a degenerate form of agent, a multi-agent system is simply defined as a set of agents [12]. The environment of an agent consists of a number of agents.

A scenario is a set of typical combinations of the behaviors of related agents in a multi-agent system, whose most fundamental characteristic is to put events in the context of the history of behavior. A basic form of scenario description is a set of patterns. Each pattern describes the behavior of an agent in the environment by a sequence of observable state changes and observable action invokings. The SLABS syntax of scenarios [12] in EBNF is as follows.

Scenario ::=
  Agent-Name: Pattern
  | Arithmetic-Relation
  | $\exists_{[number]}$ Agent-Var $\in$ Caste-Name: Pattern
  | $\forall$ Agent-Var $\in$ Caste-Name: Pattern
  | Scenario $\wedge$ Scenario
  | Scenario $\vee$ Scenario
  | $\neg$ Scenario
  | '(' Scenario ')'

There are scenarios focused on a single agent, an arithmetic relation expression, quantity expression about a caste, compound of other scenarios, and bracketed scenarios. For detail information about SLABS, please refer to [12].

3. SLABSp: agent-orient programming with scenario

SLABSp is a programming language presented in this paper for supporting SLABS, which includes the mechanism to support scenarios in agent-oriented programming. Namely, we intend to make scenario a programming language level mechanism. In this section, we sketch the language SLABSp.

The design of SLABSp has been influenced by SLABS and Java. Firstly, because the concept of scenario is not isolated and is a part of behavior rules, it needs the related concepts such as caste, agent, state, action, and behavior rule, as described by SLABS. On the other hand, SLABSp embeds 'agent' description mechanisms in Java programming language. As an experimental language, it is hoped that SLABSp is simple enough to illustrate how scenario mechanism can be applied and implemented.

The structure of agent/caste is shown in Figure 1. Agent and caste have states, actions and behavior rules. An agent has a copy of elements from the castes they join. The environment defines the set of agents in the system that can affect agent's behavior.

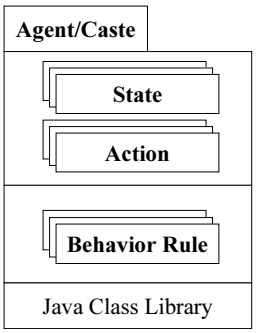

Figure 1. Structure of an agent/caste.

An example program of SLABSp is given in Figure 2. It's a caste named Worker, with one integer state (flag), two actions (sleep and work), and one behavior rule named init which is executed only once when agents of this caste start with the help of the state flag.

When state flag is 0, behavior rule init will be executed, i.e. set state flag to 1 and do action sleep. So behavior rule init will be executed only once at the beginning. Scenarios can be specified in the 'while' statement at the right portion of Figure 2, such as the following scenario which will be satisfied when all agents of caste Worker take action work after action sleep:

```
for all Worker: [sleep(), work()]
```

Corresponding to the SLABS language, the EBNF definition of the syntax elements in SLABSp is given below.

(a) Agent and Caste. Each of them has a name, some elements (state, action and behavior rule), and can join several castes. When an agent joins a caste, it will copy all elements of the caste, including the states, actions, and behavior rules. Currently, when multiple castes are joined, they can not have elements of one type with the same name.

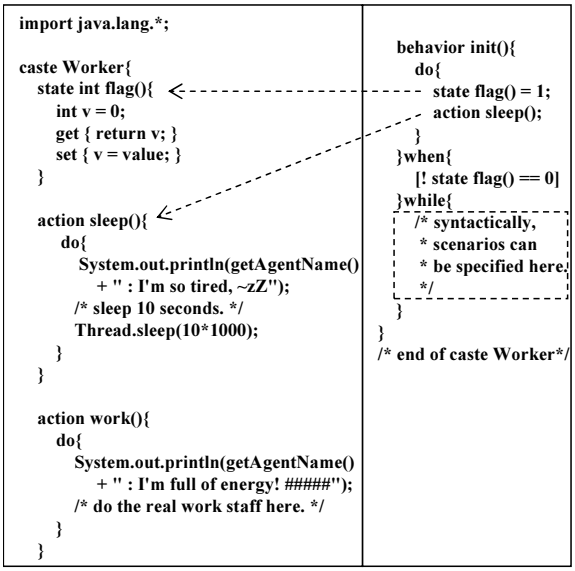

Figure 2. Caste Worker in SLABSp.

```
Agent ::=
   (Java-Import)*
   'agent' name ['extends' Caste-Id (',' Caste-Id)*] '{'
         (Element | Java-Definition)*
   '}'

Caste ::=
   (Java-Import)*
   'caste' name ['extends' Caste-Id (',' Caste-Id)*] '{'
         (Element | Java-Definition)*
   '}'

Element ::=
       State-Element
   |   Action-Element
   |   Behavior-Element
```

(b) State-Element, which can be 'internal' to the agent or be observable for other agents. A State-Element must have 'Getf' and 'Setf' clauses for read and write operations, so it can represent some complex, multi-dimensional values or objects.

The state flag of caste Worker in Figure 2 is not internal, so it is observable for others.

```
State-Element ::=
   ['internal'] 'state' Type id '(' Parameter-List ')' '{'
      ( Java-Definition | Getf | Setf )*
   '}'

Getf ::= 'get' '{' Java-Code '}'

Setf ::= 'set' '{' Java-Code '}'
```

(c) Action-Element, which can be 'internal' to the agent or be observable for other agents. The 'do' clause will be executed when the action is invoked.

In Figure 2, action sleep prints some words and sleeps for 10 seconds, while action work prints some other words.

```
Action-Element ::=
   ['internal'] 'action' id '(' Parameter-List ')' '{'
      'do' '{' Java-Code '}'
      (Java-Definition)*
   '}'
```

(d) Behavior-Element. The 'when' clause contains patterns of the agent itself, and the 'while' clause contains the scenario of the environment of the agent. When they are both satisfied, the 'do' clause will be executed.

In the caste Worker described above, the 'while' clause is empty, which means always being satisfied. The 'when' clause is satisfied when state flag becomes 0.

```
Behavior-Element ::=
   'behavior' id '{'
      'do' '{' Java-Code '}'
      (Java-Definition)*
   '}' 'when' '{'
      [ Pattern ]
   '}' 'while' '{'
      [ Scenario ]
   '}'
```

(e) Scenario, which can observe a single agent, a number of agents of a caste, all agents of a caste, or any relation expression. Scenario can be logical compound of scenarios. Pattern is used to specify the sequence of observable state changes and observable action invokings, defined as Sequence-Unit. A Sequence-Unit is either a State-Assertion, or an Action-Pattern. Once the target agent's states are changed or actions are invoked, the pattern sequence will be processed by an automaton, the Pattern Process Machine.

The atomic action 'any' can be matched by any actions, and the 'id' can be matched by action whose name is the same with 'id'.

```
Scenario ::=
       Agent-Id ':' Pattern
   |   Relation-Expression
   |   'for' (number | 'all') Caste-Id ':' Pattern
   |   Scenario 'and' Scenario
   |   Scenario 'or' Scenario
   |   'not' Scenario
   |   '(' Scenario ')'

Pattern ::= '[' Sequence-Unit (',' Sequence-Unit)* ']'
```

```
Sequence-Unit ::=
      Action-Pattern
   |  '!' State-Assertion

Action-Pattern ::= Atomic-Action ['^' number]

Atomic-Action ::=
      'any' | id
   |  id '(' Parameter-Value-List ')'
```

Note that 'Java-Import' is the same as Java's import declaration, and 'Java-Definition' can be any declaration clause of Java, such as class declaration and method declaration. 'Java-Code' is the sequence of Java statements, with some special token referencing state and action elements of the agent, as used in the 'do' clause in Figure 2.

4. Implementation

4.1. The underlying object model

The underlying object models are defined to serve as the semantics of SLABSp. Programs in SLABSp are compiled to Java source codes, then to Java class files with our libraries for the underlying object model.

When compiling SLABSp programs to Java, a set of underlying objects are needed to implement the Agents, Castes, and etc. The relationship between syntax elements and underlying object classes is given in Table 1. A syntax element in SLABSp programs will be compiled to a subclass of the corresponding underlying object class. These classes, together with their relationships, form the final Java package of classes representing the SLABSp program.

Table 1. Relationship between syntax element and underlying elements

Syntax Element	Underlying Object Class
Caste	JacCaste
Agent	JacAgent
State-Element	JacState
Action-Element	JacAction
Behavior-Element	JacBehavior
Scenario	JacScenario
Pattern	JacPatternProcessMachine
State-Assertion	StateAssertion
Action-Pattern	ActionPattern

Figure 3 is the UML class diagram of the underlying objects. All agents in SLABSp program will be compiled to a Java class inheriting JacAgent, and all castes inheriting JacCaste. They both extend JacUnit, which is an aggregation of JacState, JacActoin and JacBehavior. JacState is super class of all state elements, the same happens to JacAction and JacBehavior.

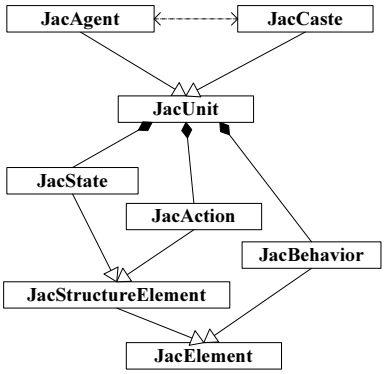

Figure 3. Class diagram of agent and caste.

In this model, an agent's structure can be changed at runtime, which makes dynamic evolving possible. An agent can add new states, actions, or even behavior rules on the fly. The membership can also be changed at runtime: agent can determine the caste to join at appropriate time.

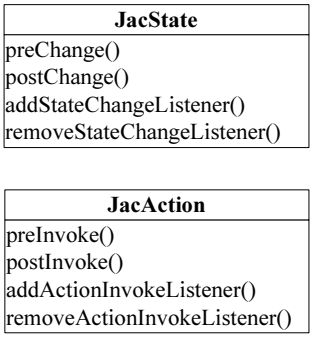

Figure 4. Key methods in JacState and JacAction.

As shown in Figure 4, JacState has preChange and postChange methods, which will be invoked before and after the state changing method respectively. JacAction's preInvoke and postInvoke methods will be respectively invoked before and after the action invoking method. These methods will trigger their listeners, the JacPatternProcessMachine, to process scenarios.

Figure 5 shows that a JacBehavior object has relationship with JacScenario object. The JacScenario is the super class of other concrete scenario classes. AgentScenario cares for

Scenario ::= Agent-Id ':' Pattern

in the syntax definition of scenario. CasteScenario cares for the quantified scenarios:

Scenario ::= 'for' (number | 'all') Caste-Id: Pattern.

Other classes represent compound scenarios. AndScenario reaches the satisfied state if its two sub scenarios are both satisfied. OrScenario reaches the satisfied state if either of its two sub scenarios is satisfied. NotScenario is satisfied when its sub scenario is not satisfied, and it is rejected when its sub scenario is satisfied.

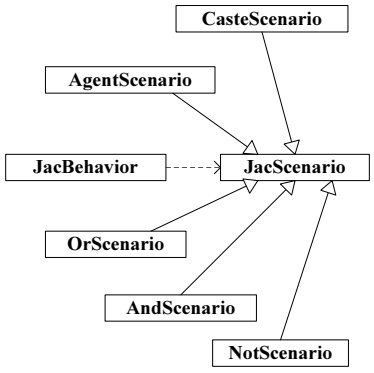

Figure 5. Class diagram of scenarios.

AgentScenario and CasteScenario use an automaton called Pattern Process Machine to process the state transition defined by the pattern. JacPatternProcessMachine has implemented two interfaces, JacStateChangeListener and JacActionInvokeListener. When a state is changed, all registered JacStateChangeListener will be notified, and when an action is invoked, all registered JacActionInvokeListener will be notified. So JacPatternProcessMachine can track the environment's change via the listener interfaces. The Pattern Process Machine only need register to the states and actions it cares, so that it will not be bothered by irrelevant events.

JacPatternProcessMachine has a sequence of unit defined by the pattern in the behavior rule, as shown in Figure 6.

Each sequence unit is either a state assertion or an action pattern, which is represented by subclass of StateAssertion and ActionPattern. They have a match method to determine whether the changing event matches the sequence unit. Each sequence unit has an 'active' flag to sign whether it is the one to be matched next. The sequence is an array of sequence units. When the Pattern Process Machine goes to the end of the sequence, the EndState, it reaches the satisfied state, as shown in Figure 7.

Once a related state is changed or an action is invoked, the Pattern Process Machine will be notified and do the process. If it is an action invoking, all the active ActionPatterns are checked and deactivated, the successor sequence units of the matched ActionPatterns are activated. Then all the active StateAssertions are evaluated. The satisfied StateAssertions are deactivated, and their successor sequence units are activated. At the end of the processing, the first sequence unit is always activated. Whenever the EndState sequence unit is activated, the Pattern Process Machine will enter its satisfied state.

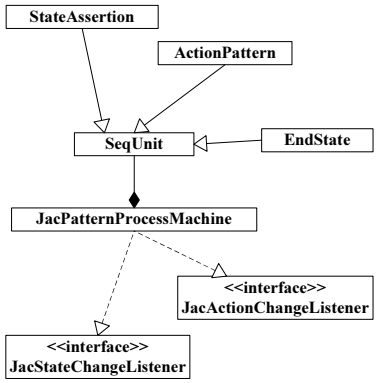

Figure 6. Class diagram of Pattern Process Machine and related classes.

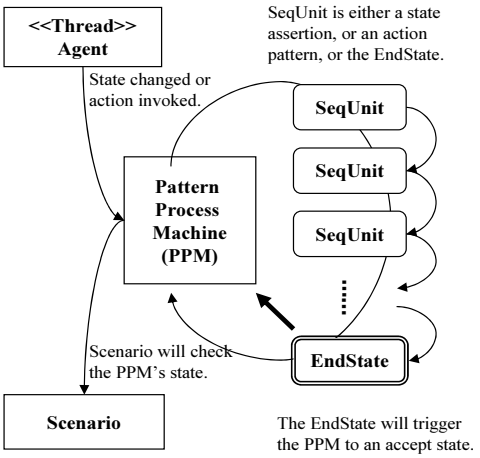

Figure 7. Processing the Pattern Process Machine.

4.2. Scenario processing

With the syntax definition of scenario, there are 4 types of scenarios in SLABSp: scenario about one agent, quantified scenario about one caste, arithmetic relation expres-

sion, and logical compound scenarios. The former two are key scenarios, which are handled respectively by AgentScenario and CasteScenario. Arithmetic relation expression is just a conditional expression, which may use CasteScenario to count the number of specific agent in one caste. Compound scenarios simply connect scenarios with logical conjunctions, and they eventually use scenarios of the former three types.

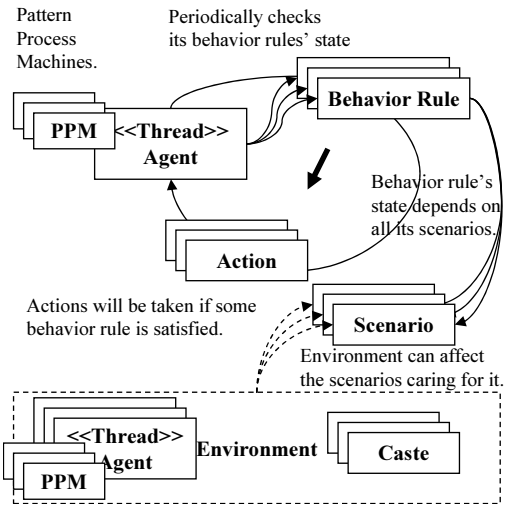

Figure 8. How scenarios are checked.

As shown by Figure 8, the agent periodically checks its behavior rules' states, for each satisfied behavior rule, the corresponding 'do' clause will be executed. A behavior rule's state depends on all its scenarios, and the environment can affect the scenarios caring for it — via the registered Pattern Process Machines.

(a) AgentScenario. Scenario about one agent is handled by AgentScenario, and has the form

Scenario ::= Agent-Id ':' Pattern.

When compiled, a subclass of AgentScenario is generated with the Agent-Id as the parameter, and the pattern is represented by a subclass of JacPatternProcessMachine.

When running, an instance of this subclass of AgentScenario will find the agent with name Agent-Id, and register its JacPatternProcessMachine object to the JacAgent object. The JacPatternProcessMachine object will find the states and actions it cares, and register itself as listeners to them, then wait for notifications. When the caring states are changed or actions are invoked, the JacPatternProcessMachine will process as described above. Once the JacPatternProcessMachine object's state changes, AgentScenario knows about it, and its state will be checked by the owner JacAgent object of the AgentScenario object.

(b) CasteScenario. Quantified scenario about one caste is handled by CasteScenario, and has the form

Scenario ::= 'for' (number | 'all') Caste-Id: Pattern.

When compiled, a subclass of CasteScenario is generated with the Caste-Id and the number as the parameters, and the pattern is represented by a subclass of JacPatternProcessMachine.

When running, an instance of this subclass of CasteScenario will find the caste with name Caste-Id, and register itself to the JacCaste object. The JacCaste object records all the agents who join it, and will instantiate the JacPatternProcessMachine object for every agent it records, and then register the JacPatternProcessMachine object to the corresponding JacAgent object. Every JacPatternProcessMachine object tracks one agent's states and actions. The CasteScenario will be notified when any JacPatternProcessMachine object's state is changed, so it can know the number of the satisfied agents in the caste. Comparing the parameter number with current number of the satisfied agents, the CasteScenario object can represent the quantity of the scenario, and the result will be checked by the owner JacAgent object of the CasteScenario object.

4.3. The runtime environment

We have constructed a basic runtime environment for executing SLABSp programs. It works on a single JVM now, and it can be extended to a distributed platform.

The class diagram of the runtime environment is shown in Figure 9. Platform has two containers, one for agents, and the other for castes. Each JacAgent implements the Java Runnable interface, and has its own thread, as a support to the autonomy characteristic of agent.

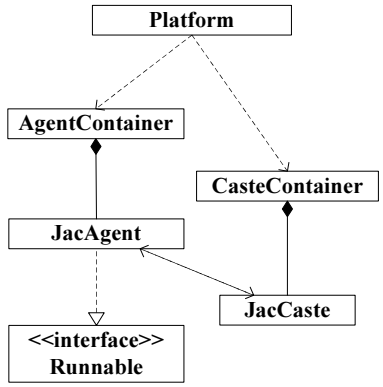

Figure 9. Runtime platform.

The platform loads the specific agent or caste class, does

the initiation, then spawns and starts a new thread for each agent.

The agent periodically checks all its behavior rules to see which behavior rule is satisfied and then executes it. AgentContainer manages agents' lifecycle, such as loading an agent from the compiled Java classes, start and stop the agent, and finally, remove the agent from Platform.

CasteContainer is used to load Java classes of castes, or remove castes from Platform.

JacPlatform also provides the naming service to find an agent or castes by its qualified name.

4.4. The example

With the caste Worker defined in Figure 2, agents named Tom and Jack are defined in Figure 10.

Tom and Jack both are Workers, thus they both have a copy of action work and sleep from caste Worker. Tom has two additional behavior rules (wokeup and gotobed). When Tom has taken action sleep, and he sees Jack has taken action sleep, Tom will take action work; when Tom has taken action work, and he sees Jack has taken action work, Tom will take action sleep.

```
agent Tom extends Worker{
      behavior wokeup(){
        do{
              action work();
        }
      }when{
        [sleep()]
      }while{
        Jack: [sleep()]
      }

      behavior gotobed(){
        do{
              action sleep();
        }
      }when{
        [work()]
      }while{
        Jack: [work()]
      }
}
```

```
agent Jack extends Worker{
      behavior wokeup(){
        do{
              action work();
        }
      }when{
        [sleep()]
      }while{
        Tom: [work()]
      }

      behavior gotobed(){
        do{
              action sleep();
        }
      }when{
        [work()]
      }while{
        Tom: [sleep()]
      }
}
```

Figure 10. SLABSp source code of agents Tom (left) and Jack (right).

In Figure 10, Jack also has two additional behavior rules (wokeup and gotobed). When Jack has taken action sleep, and he sees Tom has taken action work, Jack will take action work; when Jack has taken action work, and he sees Tom has taken action sleep, Jack will take action sleep.

As Workers, Tom and Jack both start with action sleep. So at beginning, Tom's behavior rule wokeup will be taken, and Jack's behavior rule wokeup will be taken next, then Tom's behavior rule gotobed, and so does Jack. And this will loop 'forever'. Figure 11 is a screen shot of the example.

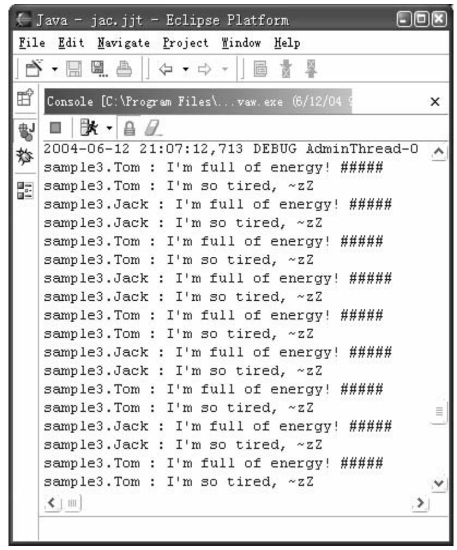

Figure 11. Screen shot of the example.

5. Related work

There are many research works reported in the literature related to the realization of agent-oriented programming and hundreds of agent construction tools have been realized. Many of them are based on Java, such as AgentBuilder [5], JACK [2], JADE [1], ZEUS [4] and etc.

AgentBuilder [5] provides tools for building Java agent systems based on a toolkit and a runtime system. The toolkit includes tools for managing the agent software development process, analyzing the domain of agent operations, defining, implementing and testing agent software. The runtime system provides an agent engine, used as execution environment of agent software. Agents in AgentBuilder are based on a model derived by the AGENT0 [6] agent model. Agents usually communicate through KQML messages and it's possible to define new communication commands to cope with developer's particular needs.

JACK [2] Intelligent Agents is an environment for building, running and integrating commercial Java-based multi-agent software using a component-based approach. JACK incorporates the significant advances in agent research and software engineering. It provides the core architecture and infrastructure for developing and running software agents in distributed applications. The JACK Agent Language extends Java with agent-oriented concepts, such as Agents,

Capabilities, Events, Plans, Agent Knowledge Bases, Resource and Concurrency Management.

JADE [1] is a software framework to help the development of agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. JADE uses an agent model and a Java implementation that offer a good runtime efficiency and software reuse. JADE agent platform tries to optimize the performance of a distributed agent system implemented with the Java language. In particular, its communication architecture tries to offer flexible and efficient messaging, transparently choosing the best transport available and leveraging state of the art distributed object technology embedded within Java runtime environment.

ZEUS [4] provides a library of agent components for the rapid development of Java agent systems, supporting a visual environment for capturing user specifications, an agent building environment that includes an automatic agent code generator and a collection of classes that form the building blocks of individual agents. Agents are composed of five layers: API layer, definition layer, organizational layer, coordination layer and communication layer. The API layer allows the interaction with non-agentized world. The definition layer manages the task the agent must perform. The organizational layer manages the knowledge about the other agents. The coordination layer manages coordination and negotiation with other agents. Finally, the communication layer allows the communication with the other agents.

The scenario mechanism in SLABSp has not supported by these works. When their agents want to know the behavior and state of other agents, they have to communicate with each other directly. However, in SLABSp, with scenario mechanism, an agent can perceive other agents in its environment, rather than communicate directly. Scenario mechanism makes it possible for an agent to perceive other agents' behaviors, which makes agent-oriented programming more flexible.

6. Conclusions and future work

Scenario has been used to describe agent behaviors in the context of environment situations in SLABS. As demonstrated in SLABS, the scenario mechanism provides a language facility for defining agent's behavior under specific environment. This paper presents an approach to the support of scenario mechanism in agent-oriented programming. As an experimental programming language, SLABSp is designed and implemented, where the scenario mechanism is explicitly supported as the distinguished feature.

It is noticed that the scenario mechanism in this paper can be used in many existing agent programming tools. For example, scenario mechanism can be used for an agent to perceive the environment. It can reduce many unnecessary direct communications among agents, so it can ease the development process and improve the runtime performance. So, an agent can sense its environment with the help of scenario mechanism.

At this moment, the pattern used in the scenario is described by the sequence of state assertions and action patterns. We are defining more expressive pattern syntax for more complex environment, such as patterns in regular expressions of state assertions and action patterns.

The other further work includes the support of the dynamic features of agent/caste presented in SLABS, and the implementation of SLABSp over a distributed computing environment, by using Java RMI or CORBA.

References

[1] F. Bellifemmine, A. Poggi, G. Rimassa, and P. Turci. An object-oriented framework to realize agent systems. In *Proceedings of WOA 2000 Workshop*, volume 1195, pages 52–57, 29–30 May 2000.

[2] M. Coburn. *JACK Intelligent Agents: User Guide, version 2.0*. http://www.agent-software.com, 2001.

[3] B. Moulin and M. Brassard. A scenario-based design method and environment for developing multi-agent systems. In *Proceeding of First Australian Workshop on DAI*, volume 1087 of *LNAI*, pages 216–232, 1996.

[4] H. S. Nwana, D. T. Ndumu, and L. C. Lee. ZEUS: An advanced tool-kit for engineering distributed multi-agent systems. In *Proceedings of PAAM98*, pages 377–391, 1998.

[5] Reticular Systems, http://www.agentbuilder.com. *AgentBuilder, An integrated Toolkit for Constructing Intelligence Software Agents*, 1999.

[6] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[7] G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[8] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[9] H. Zhu. Formal specification of agent behaviour through environment scenarios. In *Proceeding of NASA First Workshop on FAABS*, volume 1871 of *LNCS*, pages 263–277, 2000.

[10] H. Zhu. Scenario analysis in an automated requirements analysis tool. *Journal of Requirements Engineering*, 5(1):2–22, 2000.

[11] H. Zhu. The role of caste in formal specification of MAS. In *Proceeding of PRIMA'2001*, volume 2132 of *LNCS*, pages 1–15, 2001.

[12] H. Zhu. SLABS: A formal specification language for agent-based systems. *International Journal of SEKE*, 11(5):529–558, 2001.

[13] H. Zhu and D. Lightfoot. Caste: A step beyond object orientation, in modular programming languages. In *Proceeding of JMLC'2003*, volume 2789 of *LNCS*, pages 59–62, 2003.