SOARS: Spot Oriented Agent Role Simulator – Design and Implementation

Hideki Tanuma[1], Hiroshi Deguchi[2] and Tetsuo Shimizu[1]

[1]Institute of Medical Science, University of Tokyo, 4-6-1 Shirokane-dai, Minato-ku, Tokyo 108-8639, Japan
[2]Department of Computational Intelligence and Systems Science, Tokyo Institute of Technology, 4259 Nagatsuta-cho, Midori-ku, Yokohama, Kanagawa 226-8503, Japan

Summary. In this paper we present the design of an agent-based simulation language called SOARS (Spot Oriented Agent Role Simulator). SOARS is designed to describe agent activities according to roles within social and organizational structures. Role taking processes can be described in our language. SOARS is also designed according to the theory of agent-based dynamic systems. Decomposition of multi-agent interaction is one of the most important characteristics of our framework. The notion of spot and stage gives spacial and temporal decomposition of interaction among agents. We apply our multi-agent framework to policy analysis of emerging virus protection in the case of SARS.

In the latter part of the paper, we explain the implementation of the SOARS simulation platform. The simulation engine and related built-in functional objects are implemented in Java language. An application user can describe the agent-based simulation model only by writing script in SOARS script language without knowledge of Java classes. If needed, the user can customize the function of SOARS by implementing additional Java classes. The easiest way to customize is to develop special functional objects, and the SOARS platform will be the interface between such customized objects.

Key Words. Spot, Agent-Based Social Systems Science, SARS, SOARS, Agent-Based Dynamic System

1 Introduction

We often use simulation analysis in agent-based modeling. These days Ascape[1] by The Brookings Institution and Repast[2] by the University of Chicago and Argonne National Laboratory are becoming popular. In the history of agent-based modeling, Swarm by the Santa Fe research institute has been very influential.

[1] http://www.brook.edu/es/dynamics/models/ascape/default.htm
[2] http://repast.sourceforge.net/

Repast is called the social science version of Swarm. In Japan, MAS (Multi-Agent Simulator)[3] has been developed under the influence of Swarm for educational purposes. Repast, Ascape and MAS are descendants of Swarm which has been developed nor for social sciences but for cellar-type simulation.

In this paper we introduce a new type of modeling framework for the agent-based dynamic system (ABDS) and develop an agent-based simulation language called SOARS (Spot Oriented Action Role Simulator).

SOARS has been developed for infection protection against SARS in hospitals. In hospitals, there are large numbers of agents: doctors, nurses, patients, radiographers, inspectors, office workers and volunteers, who move about among many spots.

We have developed a framework for agent-based modeling not only for simulation but also for theoretical analysis of agent-based dynamic systems. We especially focus on parallel and sequential decomposition of multi-agent processes. Lisp has provided a primitive language for AI. In the same way, we try to provide a primitive language for agent-based modeling depending on the agent-based dynamic systems theory now being developed.

2 Basic Design of SOARS

(1) Three Layer Modeling Framework

SOARS is a modeling framework with three layers. On the bottom layer SOARS is written in Java. We use Java for programming the extension of SOARS, called information objects or resolvers under the regulated interface. On the middle layer of SOARS we provide an original scripting language to describe role behavior of agents. On the top layer of SOARS we provide a model builder with GUI for developing a SOARS application model. A power user can use Java for extending the model components of SOARS. On the bottom layer we can add various types of new functions on agents and spots of SOARS as objects or resolvers by programming in Java. On the middle layer the SOARS scripting language is provided for modeling agent roles. On the top layer the GUI-based model builder helps a user who has enough domain knowledge but lacks appropriate programming skill.

(2) Static Structure of SOARS: Agent, Spot, Resolver, State object

In SOARS we introduce several notions for modeling an agent-based dynamic system; they are the agent, the spot, the resolver, information and the physical object.

---

[3] http://www2.kke.co.jp/mas_e/MASCommunity1.html

The spot is an abstract notion of a localized field or a place where agents interact with each other. Agents move among spots. Each spot and agent has a state that is described by equipped objects. Spots are used to represent not only a concrete physical place but also an abstract place for interaction such as a committee. In the following example we use concrete spots in a hospital, such as a consultation room or a hospital ward. A spot is equipped with a state object that is called an information object or a physical object. A spot is also equipped with a special type of object called a resolver. A resolver on a spot describes an interaction among agents.

In the following example of a hospital infection model, a group of agents consists of office staff, cashier staff, reception staff, nurse, doctor, examiner, visitor, and patient. A group of spots inside a hospital consists of locker room, office, medical office, reception, examination waiting area, examination room, nurses' station, consultation waiting area, consultation room, ward, cashier, and mortuary. Spots which exist outside a hospital are home, business office and school.

(3) Dynamical Structure of SOARS: Step, Stage, Phase, Turn

Scheduling of SOARS is given as follows.

```
Step k start
  Stage i start
      Agent Phase start
          Agent turn j start
              1) General rule execution
              2) Active Rule execution
              3) Self Resolution: if necessary
          Agent turn j end
          Agent turn j+1 start,..., Agent turn j+1 end
      Agent Phase end
      Resolver Phase start
          Resolver turn j start
              Resolution: calculation of interaction
          Resolver turn j end
          Resolver turn j+1 start,..., Resolver turn j+1 end
      Resolver Phase end
  Stage i end
  Stage i+1 start ,...., Stage i+1 end
Step k end
```

At the turn of each agent on agent phase, the rules of the role of an agent and self resolver are executed turn by turn. In each resolver turn the program of resolver is executed turn by turn and the result s are returned to requested agents and spots. The process is shown in Fig. 1.

Step is a basic time unit of a discrete time dynamic system. A step is divided into several stages. Each stage represents a typical interaction or activity among agents and spots in a given dynamic system. Each stage is divided into two phases

called agent phase and resolver phase. In the agent phase, agent rules are executed at each agent turn. In the resolver phase a resolver on a spot calculates the interaction among agents and spots.

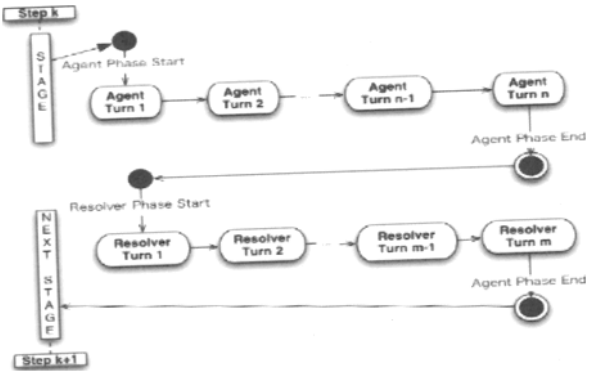

Fig. 1. Structure of Stage

(4) ABDS: Structure of Agent-Based Dynamic System (ABDS)

An agent-based dynamic system (ABDS) is a dynamic system which is an extension of the classic model of dynamic systems such as the distributed parameter system or the lumped parameter system. In ABDS we formulate moving agents and objects of the field expressed by a set of spots.

Let Sate_Space be a state space for an ABDS, Stage_Type be a set of stage type, Agent_Set be a set of agents, Spot_Set be a set of spots respectively. Let num_agent = |Agent_Set| and num_spot =|Spot_Set|. Agent_ID and Spot_ID denote index set as {1,..., num_agent} and {1,..., num_spot} respectively.

Let Fa be a set of one to one function from Agent_ID to Agent_ID. Let Fs be a set of one to one function from Spot_ID to Spot_ID as follows.

Fa={ f | f:Agent_ID->Agent_ID and f is a one to one function }

Fs={ g | f:Spot_ID->Spot_ID and f is a one to one function }

OPA[i]j :Sate_Space --> Sate_Space denotes a state transition operator at agent turn j on agent phase of stage i. OPA[i] = Π{OPA[i]j | j$\in$Agent_ID } denotes a state transition operator at agent turn j of stage i, which is generated as a cartesian product of OPA[i]j. In the same way OPS[i]j :Sate_Space --> Sate_Space denotes a state transition operator at resolver turn j on resolver phase of stage i.

OP[i]=Π { OPS[i]]j | j$\in$Spot_ID }.

Then total state transition operator at stage i is defined as OP[i]=OP[i]$\times$ OPA[i]. We assume the following commutative condition for transition operators OP[i] and OPA[i] at any stage as follows.

(1) $\forall i \in$ Stage_Type $\forall x \in$ Sate_Space $\forall f, g \in$ Fa

Π {OPA[i]f(j) | j$\in$Agent_ID}(x)=Π {OPA[i]g(j) | j$\in$Agent_ID}(x).

(2) $\forall i \in$ Stage_Type $\forall x \in$ Sate_Space $\forall f, g \in$ Fs

Π {OPS[i]f(j) | j$\in$Spot_ID }(x)=Π {OPS[i]g(j) | j$\in$Spot_ID} (x).

Where f and g mean permutation of agent index and spot index.

Then the following commutative property holds.

$\forall i \in$ Stage_Type $\forall x \in$ Sate_Space $\forall f, g \in$ Fa $\forall u, v \in$ Fs

Π {OPS[i]u(j) | j$\in$Spot_ID } $\times$ Π {OPA[i]f(j) | j$\in$Agent_ID}(x)

= Π {OPS[i]v(j) | j$\in$Spot_ID (x) $\times$ Π {OPA[i]g(j) | j$\in$Agent_ID}(x).

In our ABDS we assume the commutative condition for transition operators. If the transition operators satisfy the commutative condition then the result of state transition dose is not affected by the execution turn of agents and resolver at each stage.

(5) Interaction Decomposition

We distinguish the following three types of interaction in ABDS.

1) Inter agents Interaction

2) Interaction between agents and spots

3) Inter spots Interaction

Inter agents interaction should be mediated by a certain resolver on a spot in our modeling framework. Interaction between agents and spots is basic in our framework. The interaction is described by rules of agents or resolvers of spots. Properties of agents and spots change after the interaction.

If we are interested in the propagation process among state quantities of a field, inter spots interaction is required. For describing inter spot interaction we have to introduce a neighborhood model of spots and a two stage interaction model.

For example a two dimensional field interaction is modeled by a distributed parameter dynamic system. It can be approximated by a two dimensional lattice structure of spots. A lattice structure of spots can be constructed by assuming a suitable neighborhood model and interaction among neighborhood spots. In the same way, a network type interaction among spots is modeled by a lumped parameter dynamic system that can be modeled by a network type neighborhood model and interaction among spots.

In agent-based dynamics many agents and spots interact with each other. The interaction decomposition is important for modeling agent-based dynamics. We distinguish two types of decomposition. The one is temporal decomposition of interaction. The other is the spacial decomposition of interaction. The spacial decomposition is done by our spots structure. The commutative condition for transition operators guarantees the execution turn does not affect the result in each agent turn and resolver turn.

To satisfy the commutative condition for transition operators we introduce a two stage model for inter spots interaction. In the resolver phase of the first stage, each resolver notices the state of the spot in relation to the neighborhood spots. In the second stage, each resolver calculates its change. Then, propagation of the state of spot can be executed without being affected by a calculation turn. A step is divided into several stages depending on the ABDS model, as is shown in the previous two stage model. The stage model is very important for temporal decomposition of interaction. We investigate the stage model in the concrete ABDS modeling as follows.

3 Infection Model in a Hospital and City

In a hospital, many agents such as staff and patients move and interact at many spots. The hospital is forced to take prompt measures in the case of an emerging virus infection like SARS. There are many organizational measures for virus protection. We have to distinguish the effective measures of high cost performance. We present a prototype model of a SARS virus infection with a hospital organization model and a simple city model.

(1) Spot Structure

1) Hospital Model: Hospital consists of the following spots; Locker room, Office, Medical office, Reception, Examination waiting area, Examination room, Nurses' station, Consultation waiting area, Consultation room, Ward, Cashier, Mortuary.
2) Home: Agents go back home after finishing their work in hospital or to an office or to school.
3) Business Office: Adult agents who are not hospital staff or patients work in a business office in the daytime.
4) School: Students go to school in the daytime. Fig. 2 and 3 show patient and doctor view of the hospital respectively.

(2) Role Structure

Rules of agents are described as roles of agents. In our hospital model the following roles are introduced: office staff, cashier staff, reception staff, nurse, doctor, examiner, visitor, and patient.

(3) Stage Structure

We introduce the following stage structure.
1) Role Taking stage: An agent takes and changes his role at a suitable situation.

2) Moving stage: An agent moves from spot to spot depending on his role.
3) Infection stage: Spot contamination from infected agents and agent infection from contaminated spot and infected agents are treated in this stage.
4) Measure stage: In our prototype model we only compare "no policy model" with "trace and isolation without vaccination policy" where SARS carrier's contacts are traced and isolated.
5) Business Stage: Results of examination and consultation are reported and treated.
6) Data collection stage: At this stage simulation data are collected.
7) Progression stage: State transition of infection and contamination level is treated.

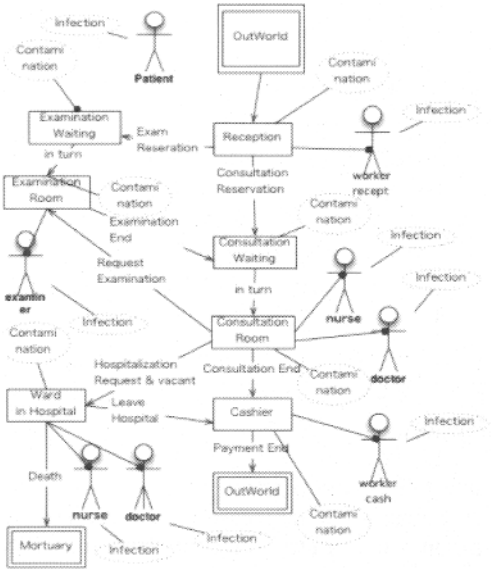

Fig. 2. Patient View of Hospital

(4) Infection Resolution Process

At the infection stage an agent notifies his infection level at the spot where he is staying at his turn of the agent phase. The resolver calculates contamination level of the spot and infection level of agents at resolver turn of the infection stage.

(5) Model Assumption

We assume 800 agents consisting of 200 families in our model. One family consists of two adults and two students. In hospital there are six doctors, six nurses, one examiner, one office staff, one cashier staff and one reception staff. Other adults work in offices. Infection and contamination will happen at the hospital, at the office and at home. We assume an organizational structure only in hospital. Each agent acts depending on the rules of the role.

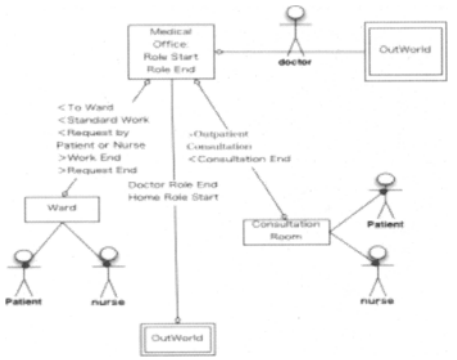

Fig. 3. Doctor View of Hospital

(6) Measures to prevent infection

There are several types of measures to prevent infection.

1) Sterilization of spots: Sterilization is a basic measure against infection. We can add level property of sterilization, which affects contamination probability.

2) Infection protection for agents: Protection level property can be added to doctor, nurse, examiner and office staff of the hospital, which affects infection probability. The level property can be added to any agents for public health protection policy.

3) Isolation policy for agents who had been in close contact with the SARS patient: Trace and isolation are very effective, even if there is no vaccination. This has been pointed out by Epstein [Epstein 2002]. In our model, the trace and isolation policy is very powerful in preventing infection. We introduced a trace and home isolation policy. We simulate a 3-day 100% isolation case where people who have contacted SARS patients over the past three days are traced and 100% captured and ordered to remain isolated at home. Family members are also isolated for 14 days. If imperfect virus inspection is available then we can evaluate the effect of the isolation after imperfect virus inspection by simulation.

4) Gatekeeper policy: A gatekeeper policy is a very important measure to prevent infection. In hospital or at airports, a fever check is a basic gatekeeper measure.

(7) Simulation Scenario

In this prototype modeling we only compare the no policy case with the trace and home isolation case. The result is shown as follows. Figure 3.3 shows the no policy case. In this case all agents are infected and recovered agents have immunity. Figure 3.4 shows the trace and home isolation policy case. The policy is effective. Infection is controlled and the number of recovered agents with immunity is less than a hundred.

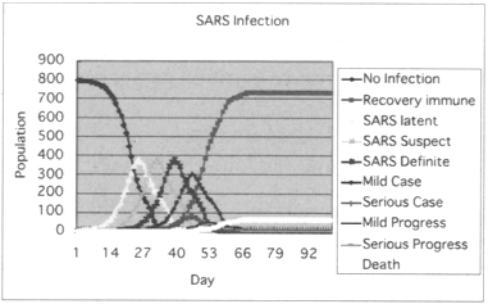

Fig. 4. No Policy Case

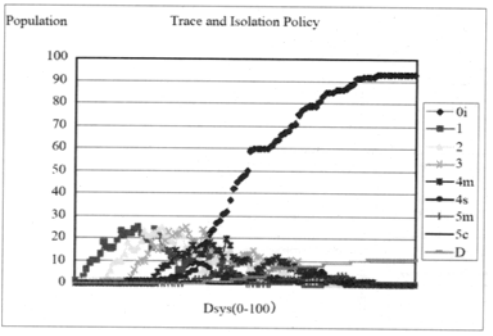

Fig. 5. Pursuit and Isolation Policy Case

5 Implementation of Simulation Platform

In the SOARS environment, the application users write the simulation model in a spreadsheet based script language, and can use some additional functional objects implemented in Java classes if needed. The simulation engine is also implemented in Java language, and skilled users can easily expand the function of the SOARS library by implementing subclasses.

(1) SOARS Library

The SOARS library is implemented in Java language and composed of the following packages and the belonging classes.

env package – the simulation model components and the engine classes
main package – the simulation entry main function classes
role package – the role related classes implementing the rule methods
script package – the script parser and the command formula parser classes
time package – the simulation time model classes
util package – the functional objects and the related interfaces
view package – the Swing GUI components and the logging output classes

Usually the SOARS library is served as an executable JAR file which starts up the main GUI console. The application users can specify the class path URL in the SOARS scripts, and then the additional functional objects are dynamically loaded by the SOARS engine.

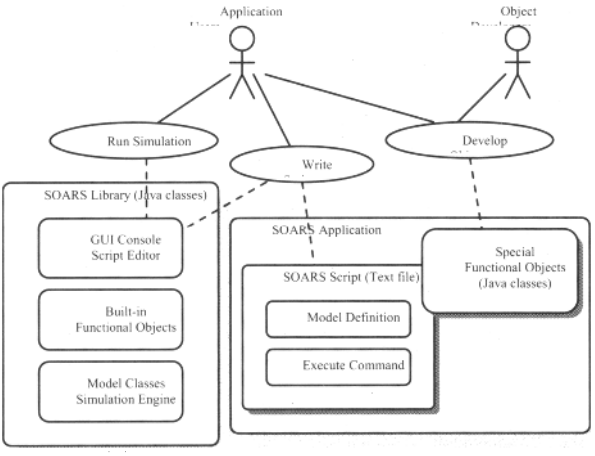

Fig. 6. System Overview of SOARS Platform and Related Use Cases

(2) Structure of SOARS Script Language

The SOARS script language consists of the following parts;

Model definition – the definition of the simulation model elements,
Execution command – the description of the running parameters,
Other information – some additional information e.g. class path URL.

The script language has spreadsheet-based format and is saved as a tab-separated text file. The basic unit of the script is a group of row lines separated by some blank lines called 'paragraph'. For example, the following paragraph defines three spots named 'Home', 'School', and 'Office'.

spot
Home
School
Office

The first row of each paragraph specifies the paragraph method. In the above case, the keyword 'spot' specifies the spot definition paragraph method and the following rows are passed as the method parameter. Deep into the Java implementation, the paragraph method named 'spot' is implemented as the function which has the following signature and is actually invoked using the Java reflection.

```
public void spot(Iterator paragraph);
```

The definition of the agents and the rules needs many parameters and applying only the paragraph method is insufficient. So, the special paragraph method 'itemData' leads to the secondary unit of the script language called 'item'. For example, the following paragraph defines five agents named 'Worker1', 'Worker2', 'Student1', 'Student2', and 'Student3'.

itemData			
agentNumber	agentName	agentCommand	agentCreate
2	Worker	<Office>moveTo	
3	Student	<School>moveTo	

The second row of the 'itemData' paragraph specifies the extra script methods called 'item method'. All contents of the third row or below are passed as the parameters of the item method of same column. In the case, the item methods have the following signatures in the Java implementation.

```
public void agentNumber(String item);
public void agentName(String item);
public void agentCommand(String item);
public void agentCreate(String item);
```

The standard script methods are implemented in the following Java class inheritance chain.

```
class script.ScriptParser;
```

This class implements the Java reflection mechanism.

`class env.ObjectLoader extends script.ScriptParser;`
This class implements the model definition methods.
`class main.MainGUI extends env.ObjectLoader;`
This class implements the execution command methods.

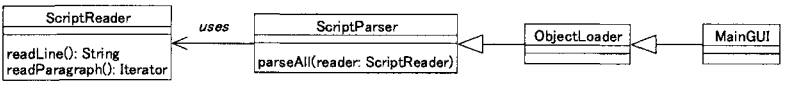

Fig. 7. Script Method Related Implementation Classes

(3) Description of Rule Action Formula

In the agent definition paragraph, the item method 'agentCommand' gets the initialization command formula as a parameter string. For the previous example, the command formula '<Office>moveTo' means that the agents move to the spot named 'Office' at first. Also, the item method 'ruleCondition' takes the rule condition formula and the method 'ruleCommand' takes the rule command formula. Each action rule of the agents is expressed as some condition formulae and a command formula and then form an if-then rule action formula.

The fundamental component of the rule action formula is a rule method. The rule methods are implemented in the subclasses of role.Role and invoked by the Java reflection. The standard rule methods implemented in the class role.RoleStructure are the following:

String property related – keyword, set, is, get, put,
Functional object related – askEquip, setEquip, isEquip, getEquip, putEquip, printEquip,
Role related – activateRole, setRole, isRole,
Time related – setTime, isTime,
Spot related – moveTo, isSpot,
Debug tracing related – trace, traceOff,
Other utilities – TRUE, FALSE, doSelfResolution.

In the rule action formula, the rule method is sometimes decorated with a string parameter and some operational prefixes. For example, the method moveTo needs a spot designation prefix operator written in angle brackets like this: <School>moveTo. The spot designation operator adds the Spot object parameter in the Java implementation. The previous formula invokes the Java method which has the following method signature.

```
public void moveTo(Spot spot);
```

Some rule methods take a string parameter suffix. The method 'keyword' defines the agent's string property keyword. For example, the rule action formula 'keyword FLAG' invokes the Java method below.

```
keyword("FLAG");
```

Also, the composition of the prefix operator and the string parameter makes different context. The rule action formula '<School>keyword FLAG' defines the string property keyword 'FLAG' of the spot named 'School'. This formula invokes the Java method which has the following signature.

```
public void keyword(Spot spot, String parameter);
```

A rule method which returns a boolean value is called 'rule condition method'. In the standard rule methods, the following rule condition methods are defined.

```
is, askEquip, isEquip, isRole, isTime, isSpot, trace,
traceOff, TRUE, FALSE
```

The rule condition methods can be used as a parameter of the 'ruleCondition' item method. For example, the formula '<Office>isSpot' means whether the agent is at the spot 'Office' or not. The condition methods can also be composed by some binary operators to make more complex formula. For example, the next formula describes the rule actions that if the agent is at the spot 'Office' or the spot 'School', then move to the spot 'Home'. We call such a composed formula 'command formula', and call the formula which returns a boolean value 'condition formula'.

```
<Office>isSpot || <School>isSpot ? <Home>moveTo
```

The parsing process of the rule action formula is implemented in the Java abstract class script.CommandParser. The abstract methods of parsing command and condition terms are implemented in the classes role.RuleCommandParser and util.AskCommandParser. The former class interprets the spot designation prefix and the latter does not. The rule action formula is converted to the Condition class instance by the parser classes and wrapped in the Rule class instance with a debugging information string. These Rule instances are stored in the RoleType class instance, and then executed by the SOARS simulation engine.

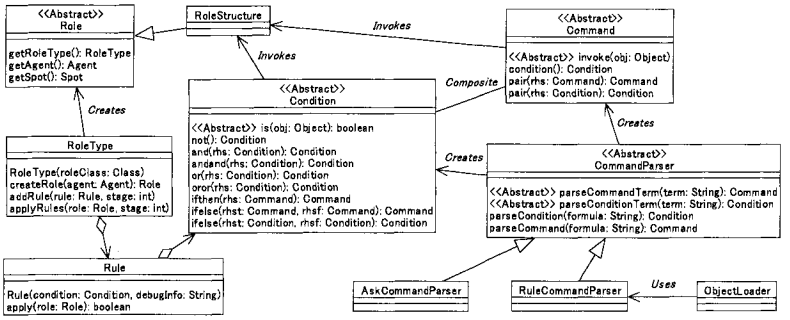

Fig. 8. Rule Action Formula Related Implementation Classes

(4) Simulation Model Objects and Functional Objects

The SOARS simulation model is mainly composed of the following elements;

Agent – the subject of the rule action,
Spot – the place where the agents move to,
Role type – the kind of agent rule action pattern,
Stage – the action sequence slot which the simulation time step is divided into,
Rule – the agent action command set described in if-then formula.

Each agent and spot has two types of attributes. One is literal information called 'property' which is a dictionary-like data structure formed of 'key=value' string to string mappings. This information can be used as the rule action branching flags and the logging variables. Another is object information called 'equipment' which is also a dictionary-like data structure, but formed of 'key=object', and the object can be any Java class instance. The functional objects can be stored in the equipment, and usually these objects implement some interfaces to communicate with the SOARS script language. These attributes are stored in the instance of the common subclass env.EquippedObject of the agent class env.Agent and the spot class env.Spot.

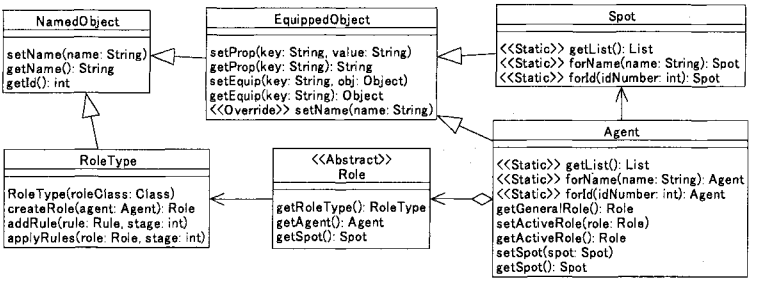

Fig. 9. Simulation Model Implementation Classes

6 Conclusion

We have developed a new type of agent-based simulation language called SOARS which depends on the concept of agent-based dynamic systems. Agent-based modeling is becoming more important for evidence-based policy making. Agent-based simulation will provide traceability of the evidence for economic, social and organizational planning. ABM is becoming a core of developing agent-based social systems sciences. Classic agent-based simulation tools are strongly affected by cellar type simulation. We insist that new ABM tools should be based on a new dynamic systems concept for agent-based modeling. We have developed both the theory of an agent-based dynamic system and a spot oriented agent role simulator in our SOARS project. The SOARS project is an open project. If you are interested in SOARS language please contact deguchi@dis.titech.ac.jp. Updated information on SOARS can be found at <http://degulab.cs.dis.titech.ac.jp/soars/index.html>.

References

[Deguchi 2004] H. Deguchi, Economics as an Agent Based Complex System, SpringerVerlag, 2004

[Epstein 2002] Joshua M. Epstein, Derek A. T. Cummings, Shubha Chakravarty, Ramesh M. Singa, and Donald S. Burke, "Toward a Containment Strategy for Smallpox Bioterror: An Individual-Based Computational Approach" Center on Social and Economic Dynamics Working Paper, No. 31, Brookings Institution, December 2002