REMODELING THE BEER GAME AS AN AGENT-OBJECT-RELATIONSHIP SIMULATION

Jeroen van Luin
Florin Tulba
Gerd Wagner
Faculty of Technology Management
Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven,
The Netherlands
{J.v.Luin, F.Tulba, G.Wagner}@tm.tue.nl

KEYWORDS

Supply Chain management, Simulation, Agent Oriented Modeling, Agent Based Simulation

ABSTRACT

In this paper we take the classic MIT Beer Game supply chain simulation and remodel it as a multi-agent based simulation using the *Agent-Object-Relationship* modelling language proposed in [Wag03].

1 INTRODUCTION

Simulation games have a long tradition in management science. Since their main purpose is to simulate interactions between business actors, they are well-suited for agent-based approaches. We will demonstrate this using the classical MIT Beer Games as example. [Ster03].

There are many different formalisms and implemented systems that are all categorized under the title 'agent-based simulation' (ABS). E.g., one of the most prominent ABS systems is SWARM [Swarm04], an object-oriented programming library that provides special support for event management, but does not support any cognitive agent concept.

Like SWARM, many other ABS systems do also not have a theoretical foundation in the form of a *metamodel*. They do therefore not allow the specification of a simulation model in a high-level declarative language, but instead require specifying simulation models at the level of imperative programming languages.

We take a different approach and aim at establishing an ABS framework based on a high-level declarative specification language with a UML-based visual syntax and an underlying simulation metamodel as well as an abstract simulator architecture and execution model. The starting point for this research effort is an extension and refinement of the classical *discrete event simulation* paradigm by enriching it with the basic concepts of the *Agent-Object-Relationship (AOR)* metamodel proposed in [Wag03]. The resulting simulation method is called *AOR simulation (AORS)*. Its principles have been discussed in [WT03].

The paper is structured as following. After introducing the Beer Game and the principles of AOR modeling and simulation in section 2 and 3, we present the AORS model of the Beer Game in section 4. We then discuss the results obtained from running the simulation, our conclusions and possible extensions of the game in section 5 and 6.

2 THE MIT BEER GAME

The MIT Beer Game is a management simulation that was developed by the System Dynamics Group of the Sloan School of Management in the early 1960s.

The game is played by teams of four persons, each person playing one of the four roles of Retailer, Wholesaler, Distributor and Factory. The four roles are arranged in a simple beer supply chain (see figure 2.1) .

The factory has an unlimited supply of raw materials and each of the roles has an unlimited storage capacity. The supply lead times (the time between shipment by the supplier, and arrival at its destination) and order delay times (the time between the sending of an order, and the arrival of the order at its destination) are fixed.

Supplies take two full turns to arrive, the orders for new beer arrive at their destination in the next turn.

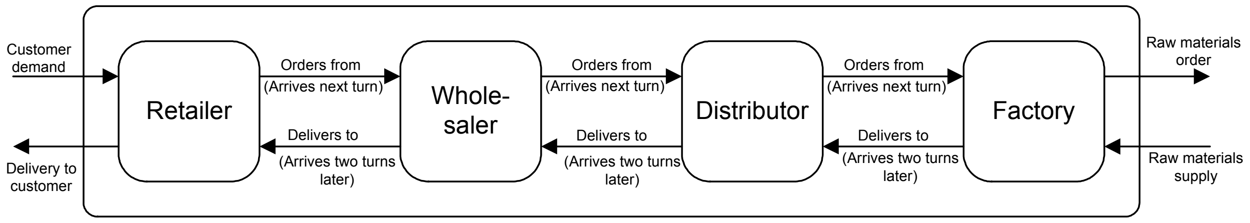


Figure 2.1 The supply chain structure of the MIT Beer Game

Each turn (usually a simulated week), the retailer receives a customer order and tries to ship the requested amount from its inventory. It then orders an amount of beer from its supplier, the wholesaler, which tries to ship the requested amount from its inventory, and so on. Orders that cannot be met are placed in backorder and must be met as soon as possible.

At the end of each week, each role has to calculate that week's costs. Remaining inventory is charged $0.50 per item as holding costs and backorders are charged $1.00 per item as shortage costs. The objective of the game is to be the team with the lowest overall costs or to be the player with the lowest cost within a team, after playing a fixed number of weeks.

The players have complete information about their own role, including a history of their own incoming orders, outgoing orders and backorders. However, they have no information about the other roles. The customer demand is only known to the retailer, and no player knows the inventory size of the other players, or how much the other players have in backorder.

Though the simulation is a very simplified version of the real life situation (each role has unlimited labour, capacity, machines and funds and there are no competitors), the average costs after running 36 weeks is $2000 and the orders between the players show an extremely oscillating pattern. The calculated optimal strategy would have a total cost of only $200 [Ster92].

3 AOR MODELING AND SIMULATION

3.1 AOR Modeling

The AOR modeling language (AORML) is based on an ontological distinction between active and passive entities, that is, between agents and (non-agentive) objects of the real world. The agent metaphor includes *artificial* (software and robotic), *natural* (human and animal) as well as *social/institutional* agents (groups, organizations etc.).

In AORML, an entity is an agent, an event, an action, a claim, a commitment, or an ordinary object. Only agents can communicate, perceive, act, make commitments and satisfy claims. Objects are passive entities without such capabilities. Besides human and artificial agents, AORML also includes the concept of *institutional agents,* which are composed of a number of other agents that act on their behalf. Organizations and organizational units are important examples of institutional agents.

There are two basic types of AOR models: *external* and *internal* models. An external AOR model adopts the perspective of an external observer who is looking at the (prototypical) agents and their interactions in the problem domain under consideration. In an internal AOR model, we adopt the internal (first-person) view of a particular agent to be modeled. While a business domain model corresponds to an external model, a system design model corresponds to an internal model, which can be derived from the external one.

In external AOR modeling, there is a distinction between *action events* and *non-action events*, and between *communicative* action events (or *messages*) and *non-communicative* action events. These distinctions are important for AOR simulation, since they allow a close-to-reality treatment of perception and communication.

The most important behavior modeling element of AORML are ***reaction rules***, which are used to express *interaction patterns*. A reaction rule is visualized as a circle with incoming and outgoing arrows drawn within the agent rectangle whose reaction pattern it represents. Each reaction rule has an incoming arrow with a solid arrowhead, which specifies the triggering event type. In addition, there may be ordinary incoming arrows representing state conditions (referring to corresponding instances of other entity types).

There are two kinds of outgoing arrows: one for specifying mental effects (changing beliefs and/or commitments) and one for specifying the performance of (physical and communicative) actions. An outgoing arrow with a double arrowhead denotes a mental effect. An outgoing connector to an action event type denotes the performance of actions of that type. Figure 4.2 shows an example of an *interaction pattern diagram* describing the interactions of a Beer Game agent. For instance, the reaction rule R3 has a trigger arrow and a mental effect arrow. It is triggered by `Order` messages and leads to a state change that is specified by the OCL postcondition:

```
OrderedQuantity = OrderedQuantity@pre +
                  Order.Quantity
```

In symbolic form, a reaction rule is defined as a quadruple

$$\varepsilon, C \rightarrow \alpha, P$$

where ε denotes an event term (the triggering event), C denotes a logical formula (the mental state condition), α denotes an action term (the triggered action), and P denotes a logical formula (specifying the mental effect or postcondition).

3.2 AOR Simulation

In an Agent Based Simulation it is natural to partition the simulation system into two parts:

1. the ***environment simulator*** responsible for managing the state of all external (or physical) objects and for the external/physical state of each agent;
2. a number of ***agent simulators*** responsible for managing the internal (or mental) state of agents.

The state of an AOR Simulation system consists of:

- the simulated time *t*
- the environment state representing
  - the non-agentive environment (as a collection of objects) and

– the external states of all agents (e.g., their physical state, their geographic position, etc.)

- the internal agent states (e.g., representing perceptions, beliefs, memory, goals, etc.).
- a (possibly empty) list of future events

For each simulation run, a start and an end calendar date-time are specified. The simulated time *t* represents a cycle counter, which can be transformed into a calendar date-time with the help of a cycle duration constant (for which we use 100ms by default). The simulation run stops at the end date-time. By default, both the perception-action and the action-perception delay are 100ms, as illustrated in figure 3.2.1.

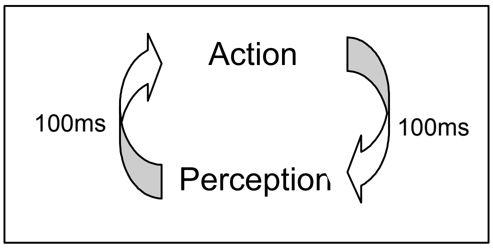


Fig. 3.2.1: The perception-action and action-perception delays

At initialization time, for each exogenous event type, according to its distribution function, events are created and put into the future events list.

A simulation cycle consists of the following steps:

1. At the beginning of a new simulation cycle, at simulated time t, the environment simulator determines the current events (all the events of the future events list whose occurrence time is now).
2. Based on the current environment state and the current events, the environment simulator determines
   a) a new environment state
   b) a set of new caused events (with suitable occurrence time) created by the causality rules of the environment simulator, including
      - newly caused *environment events*
      - next step *receive message events* (with occurrence time *t+1*) derived from the corresponding current send message events
      - next step *perception events* (with occurrence time *t+1*) derived from the corresponding current *environment events* and *non-communicative action events*
   c) The environment simulator then hands over to each agent simulator, the current *receive message events* and *perception events* for the simulated agent.
3. Using its current internal state and its list of current perceptions and incoming messages, each agent simulator computes,
   a) a new internal agent state,
   b) a set of new action events (with occurrence time t+1) created by the behavior rules of the agent simulator (representing the immediate reactions of the simulated agent in response to the delivered perceptions and message receptions)
4. The future events list is updated by removing all the processed events and adding the new caused events from step 2b and the new action events from step 3b.
5. If the simulation run continues, the environment simulator sets the simulated time t
   a) to t+1, if it is in incremental time progression mode, or
   b) to the occurrence time of the next event from the future events list, if it is in event-based time progression mode, and starts over with step 1 of the simulation cycle.

The simulation run ends when the future events list is empty or when the end date-time is reached.

This abstract architecture and execution model can be instantiated by different concrete architectures and systems. In the next section, we present a Prolog program, which instantiates this architecture and implements an AORS system.

3.3 A Prolog Prototype of an AORS System

Implemented as a Prolog program, the AORS simulation cycle yields the following procedure:

```
1: cycle( Max, Max, _, _, _) :- !.
2: cycle( _, _, _, _, []) :- !.
3: cycle( Now, Max, EnvSt, IntAgtSts, EvtList)
   :-
4:    extractCrtEvts( Now, EvtList, CrtEnvEvts,
      CrtPercEvts),
5:    envSimulator( Now, CrtEnvEvts, EnvSt,
      NewEnvSt, TranslCausEvts),
6:    agtsSimulator( Now, CrtPercEvts,
      IntAgtSts, NewIntAgtSts, TranslActEvts),
7:    computeNewEvtList( EvtList, CrtEnvEvts,
      TranslCausEvts, TranslActEvts,
      NewEvtList),
8:    findNextMoment( NewEvtList, NextMoment),
9:    cycle( NextMoment, Max, NewEnvSt,
      NewIntAgtSts, NewEvtList).
```

Lines 1 and 2 represent the exit condition (when the end date-time is reached or when the future events list is empty). In line 4, the current environment events (step 1 of the simulation cycle) and also the current perception events are extracted from the future events list. Lines 5 and 6 simulate the system in the current cycle by first calling the environment simulator and then calling all agent simulators. In line 7, the future events list is updated (step 4). The last two lines update the time and start a new cycle (step 5). `NewEnvSt` and `NewIntAgtSts` represent the new environment state and the new internal states of agents.

We represent physical causality as a transition function, which takes an environment state and an event and provides a new environment state and a set of caused events. This function is specified by a set of reaction rules for the environment simulator in the form of

```
    rrEnv( RuleDescr, Now, Evt, Cond,
           CausEvt, Eff)
```

with obvious parameter meanings. Agent behavior, as a function from a mental state and a perception event to a new mental state and a set of action events, is also specified by a set of reaction rules:

```
    rr( AgentName, RuleDescr, OwnTime,
        Evt, Cond, ActEvt, Eff)
```

For processing these rules we use two meta-predicates:

1. **prove**(X, P), where X is a list of atomic propositions (representing an environment state or an internal agent state) and P is a proposition.
2. **update**(X, P, X′) where X′ is the new state resulting from updating X by assimilating P (in our simple example this means asserting or retracting atomic propositions).

When *E* is a current event, and there is an environment simulator rule whose event term matches *E* such that **prove**(EnvSt, Cond) holds, then the specified CausEvt is added to the future events list as described in step 2 above and the environment state is updated by performing

update(EnvSt, Eff, NewEnvSt)

In a similar way, the reaction rules of each agent are applied, updating its internal state by

update(IntAgtSt, Eff, NewIntAgtSt)

Concerning step 2c, notice that if there are only communication events (messages), then the perceptions of an agent are the messages sent to it.

We now present the environment simulator:

```
1.    envSimulator( Now, CrtEvts, EnvSt,
         NewEnvSt, TranslCausEvts) :-
2.       findall( [CausEvt, Eff],
         (
            member( Evt/_, CrtEvts),
            rrEnv( RuleDescr, Now, Evt, Cond,
            CausEvt, Eff),
            prove( EnvSt, Cond)
         ), ListOfResults),
3.       extractEffects( ListOfResults,
                         Effects),
4.       computeNewEnvState( EnvSt, Effects,
                         NewEnvSt),
5.       extractEvents( ListOfResults,
                        CausEvts),
6.       translateCausEvts( Now, CausEvts,
                            TranslCausEvts).
```

In line 2 all events (and their accompanying effects) that are caused by an event from the CrtEvts list are collected in ListOfResults. Based on the effects of the current environment events (extracted on line 3) the new environment state is determined (line 4). After extracting also the caused events from the ListOfResults (in line 5), their absolute time stamp is computed with respect to the current moment (line 6).

A similar procedure is performed for each agent, as shown in the following code:

```
1.    agtSimulator( AgtName, Now, CrtPercEvts,
        IntAgtSt, NewIntAgtSt, ActEvts) :-
2.      findall( [ActEvt, Eff],
        (
           member( Evt, CrtPercEvents),
           rr( AgtName, RuleDescr, Now, Evt,
               Cond, ActEvt, Eff),
           prove(IntAgtSt, Cond),
        ), ListOfResults),
3.      extractEvents( ListOfResults,
                       ActionEvents),
4.      extractEffects( ListOfResults,
                        Effects),
5.      computeNewState( IntAgtSt, Effects,
                         NewIntAgtSt).
```

4 AN AORS MODEL OF THE BEER GAME

When looking at the Beer Game from an agent-oriented viewpoint, we can identify four agents (the retailer, wholesaler, distributor and factory) that have similar behavior. We exclude the end customer and the raw materials supplier from the simulation and therefore assume that the environment sends beer orders to the retailer, receives beer deliveries from the retailer and supplies the factory with raw materials.

Each agent receives orders from its client. The client is either the end customer, in case of the retailer, or the downstream supply chain node, in the other cases. Each order specifies a quantity of beer, which is added to the order quantity for that week. The agent can also receive a beer delivery from its upstream node or, in case of the factory, a raw materials delivery from the environment. The delivered quantity is added to the quantity in stock.

At the end of the week, the orders for this week and the backorder quantity are added to form the total outstanding order. This total order quantity is compared to the current quantity in stock. If the latter is equal to or greater than the former, the quantity of beer shipped is equal to the total order quantity, and the backorder quantity is set to 0.

If the inventory is insufficient to fulfil the total order, the entire inventory is shipped and the remaining (unfulfilled) part of the total order is recorded as being in backorder. At the end of its turn, the agent decides on the amount of beer it wants to order from its supplier.

Figure 4.1 shows the AOR Interaction Frame Diagram describing the possible interactions between two agents of the Beer Game supply chain.

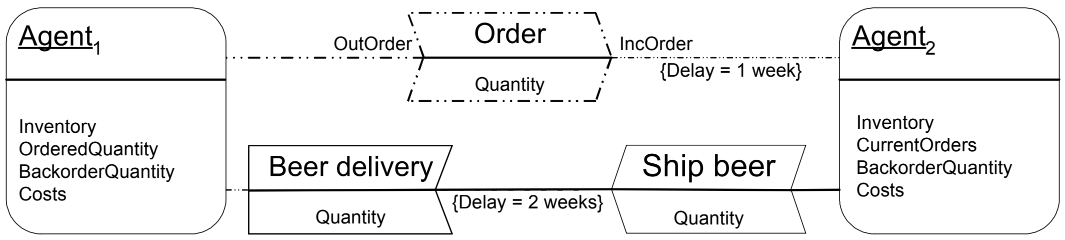


Fig. 4.1 The interaction frame between two agents

The incoming and outgoing orders are modeled as messages, the shipping of beer is modeled as a non-communicative action event and the reception of beer is modeled as a non-action event. The shipment of a quantity of beer by one agent and the reception of this shipment by the other agent are modeled as two different events (with a two week delay) abstracting away from the transportation process.

The reactions of the agents to the incoming events as described in figure 4.1 can be specified and visualised in the form of reaction rules, as shown in the diagram in figure 4.2.

4.1 Reaction Rules

The first reaction rule, R1, is triggered periodically at the end of each business week. It compares the inventory with the total number of outstanding orders. Depending on the outcome of this test, either the entire order or the maximum available quantity of beer is shipped. The quantities of the inventory, backorder and this week's orders and are then adjusted accordingly.

Reaction rule R1: shipping outstanding orders

```
ON      Event    End of Week
IF      Cond     Inventory >=
                 (OrderedQuantity +
                 BackorderQuantity)
THEN    Action   Ship( OrderedQuantity +
                 BackorderQuantity)
        Effect   Inventory = Inventory@pre
                 – (OrderedQuantity
                 + BackorderQuantity@pre)
        Effect   BackorderQuantity = 0
ELSE    Action   Ship(Inventory@pre)
        Effect   BackorderQuantity =
                 (BackorderQuantity@pre +
                 OrderedQuantity) –
                 Inventory@pre
        Effect   Inventory = 0
        Effect   OrderedQuantity = 0
```

The second reaction rule, R2, handles the ordering of new beer from the upstream agent. This action takes place at the end of the week. The trigger for this rule is a perception that the end of the week has been reached.

Reaction rule R2: Ordering stock

```
ON    Event    End of Week
DO    Action   SEND( Order( ComputeQty())
```

The third reaction rule, R3, is triggered by the reception of a new order from the customer. It adds the new amount of requested beer to the amount of beer that had already been ordered this week.

Reaction rule R3: receiving a new order

```
ON    Event    Order
DO    Effect   OrderedQuantity =
               OrderedQuantity@pre +
               Order.Quantity
```

The fourth reaction rule, R4 is triggered by the arrival of a new shipment of beer. The quantity of the beer in the event is added to the existing amount.

Reaction rule R4: receiving new beer

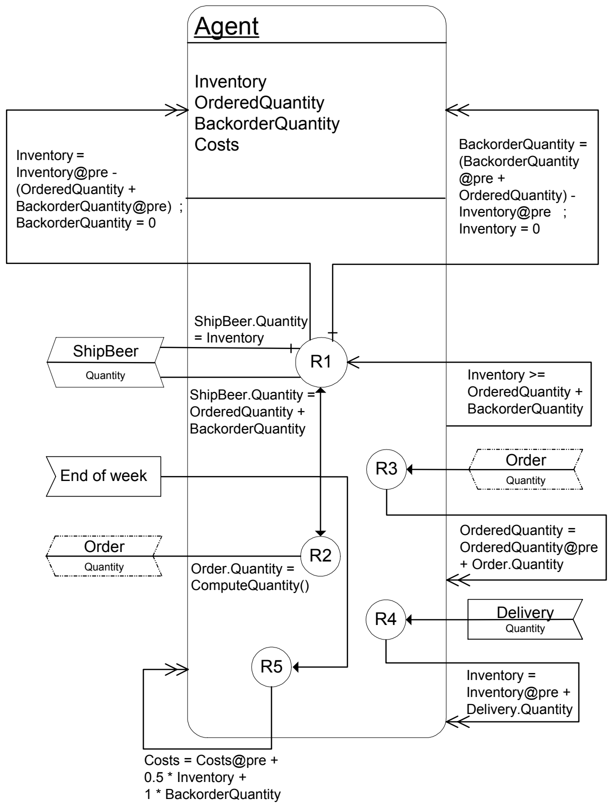
```
ON    Event    Delivery
DO    Effect   Inventory = Inventory@pre +
               Delivery.Quantity
```

The fifth and last reaction rule handles the calculating of the costs of the past week, and adds it to the total costs. Like the shipping of beer and the ordering of new stock, this action takes place at the end of a business week and is triggered by the perception of the end of the week.

Reaction rule R5: Calculating costs

```
ON    Event    End of Week
DO    Effect   Costs = Costs@pre + (0.5 *
               Inventory) + (1 *
               BackorderQuantity)
```

The AOR Interaction Pattern Diagram depicting the reaction rules R1–R5 is shown in figure 4.2.



Agent
Inventory
OrderedQuantity
BackorderQuantity
Costs
Inventory = Inventory@pre - (OrderedQuantity + BackorderQuantity@pre) ; BackorderQuantity = 0
BackorderQuantity = (BackorderQuantity @pre + OrderedQuantity) - Inventory@pre ; Inventory = 0
ShipBeer.Quantity = Inventory
ShipBeer
Quantity
R1
Inventory >= OrderedQuantity + BackorderQuantity
ShipBeer.Quantity = OrderedQuantity + BackorderQuantity
End of week
R3
Order
Quantity
OrderedQuantity = OrderedQuantity@pre + Order.Quantity
Order
Quantity
R2
Order.Quantity = ComputeQuantity()
R4
Delivery
Quantity
R5
Inventory = Inventory@pre + Delivery.Quantity
Costs = Costs@pre + 0.5 * Inventory + 1 * BackorderQuantity

Fig. 4.2 Interaction Pattern Diagram

5 RUNNING THE MODEL

We have transformed the AORS model of the Beer Game into code for the AORS Prolog simulator. We then tested it with different ordering strategies for the agents. The customer demand has been set to a random integer number between 1 and 10. Each simulation was run 1000 times for 12 simulated weeks per run.

The following examples show how the reaction rules R1 and R3 are encoded in our Prolog simulator:

```
rr( _, "R2/endOfWeek", _,
  pE( endOfWeek),
  supplier( Supplier) &
  orderedQuantity( CurOrdered)
  /(
     Quantity is max( 1, CurOrdered)
  ),
  [sM([Supplier], order( Quantity)) / 1 ]
  , none).

rr( _, "R3/order", _,
  rM( _, order( Quantity)),
  orderedQuantity( CurOrdered)
  /(
      NewCurOrdered is CurOrdered + Quantity
  ),
  [],
  not( orderedQuantity( CurOrdered)) &
  orderedQuantity( NewCurOrdered)).
```

The first rule states that at the end of a week, each node in the chain sends a replenishement order to its supplier; the `computeQty()` function provides a value that is equal to that week's ordered quantity, but at least one. The second rule states that when receiving an order, the node just updates the ordered quantity of beer.

We have tested 15 strategies with these settings and rules. These strategies were:

- A constant order of 1 to 10 (each of the possible numbers in the customer order function)
- An outgoing order equal to the incoming order
- An outgoing order equal to the current backorder quantity
- Using the same order function for the agents as for the customer (random between 1 and 10).
- Trying to keep the inventory at least at 10: order = max(1, 10-Inventory)
- Trying to keep the inventory at least at 20: order = max(1, 20-Inventory)
- Constant order of 5, but this time with a fixed customer demand of 5 (to be used as a reference)

The results we found were:

Strategy	**Avg. costs**
Constant order of 1	470.23
Constant order of 2	441.11
Constant order of 3	407.39
Constant order of 4	371.55
Constant order of 5	345.45
Constant order of 6	315.85
Constant order of 7	300.96
Constant order of 8	301.01
Constant order of 9	324.31
Constant order of 10	357.74
Order equal to the incoming order	797.29
Order equal to the backorder quantity	3030.40
Random order between 1 and 10	534.30
Keeping inventory at least at 10	499.53
Keeping inventory at least at 20	494.47
Constant order of 5, demand of 5	300

The results show us that in the case of a customer with a random order between 1 and 10 and the costs for inventory being lower than those of backorders, the optimal strategy is to order 7 or 8 units of beer each turn.

6 CONCLUSIONS

We have chosen the MIT Beer Game as a test case in this paper because it is a very simple but illustrative supply chain simulation and management game. Using the AOR modeling language we were able to create a model for the Beer Game in a high-level declarative specification language with a UML-based visual syntax. The transformation of such a model into executable code for the AORS Prolog simulator was straightforward routine work. We plan to implement an AORS modeling tool that supports automatic code generation.

Agent-based simulation seems to be very suitable for modeling management simulation games. The problem of how to support interactivity in such simulations deserves special attention. The Beer Game is too simple to display the full scope of interactivity issues in simulation. In our future efforts to investigate interactivity in agent-based simulation, we plan to remodel and/or develop more complex games, in which interactivity will be essential for making them an educational tool that is also fun.

Finally, the Beer Game raises the question how a supply chain agent could improve its performance, e.g. by learning from past experience. It may therefore be a suitable case for investigating the possibility to extend the basic AOR agent architecture with advanced behavior modeling constructs for learning, such as utility functions.

7 REFERENCES

[MIT02] Website of the Web-based Beer Game. http://beergame.mit.edu/, 2002.

[Ster92] Sterman, J.D. "Teaching takes off, Flight Simulators for Management Education". *OR/MS Today*, October 1992, 40-44

[Ster03] Sterman, J.D. Website of the Beer Game, http://mitsloan.mit.edu/faculty/r-beergame.php, MIT, 2003.

[Swarm04] Website of Swarm Development Group, http://www.swarm.org/, January 2004.

[Wag03] Wagner, G. "The Agent-Object-Relationship Meta-Model: Towards a Unified View of State and Behavior", *Information Systems* 28:5 (2003), 475-504. See http://aor.rezearch.info/.

[WT03] Wagner, G. & Tulba, F. Agent-Oriented Modeling and Agent-Based Simulation. In Giorgini, P. & Henderson-Sellers, B. (Eds.), *Conceptual Modeling for Novel Application Domains*, volume 2814 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 205-216.