Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

ADDING AGENT CONCEPTS TO OBJECT EVENT MODELING AND SIMULATION

Gerd Wagner
Luis G. Nardin

Department of Informatics
Brandenburg University of Technology
Konrad-Wachsmann-Allee 5
Cottbus 03046, GERMANY

ABSTRACT

Object Event Modeling and Simulation (OEM&S) is a general Discrete Event Simulation paradigm combining object-oriented modeling with the event scheduling paradigm. We show how to extend OEM&S by adding concepts of agent-based modeling and simulation, resulting in a framework that we call Agent/Object Event Modeling and Simulation (A/OEM&S). The main point for such an extension is to define agents as special objects, which are subject to general (physical) laws of causality captured in the form of event rules, and which have their own behavior allowing them to interact with their inanimate environment and with each other. Because agent behavior is decoupled from physical causality, an A/OE simulator consists of an environment simulator, which simulates the physical world (the objective states of material objects), and agent simulators, which simulate the internal (subjective) states of agents and their behaviors.

1 INTRODUCTION

In the area of modeling and simulation, the term ‘agent-based’ simulation is used ambiguously both for *individual-based* and *cognitive agent* simulation. The former, which is sometimes also called ‘microscopic’ simulation, takes the structure and interactions of individual entities into consideration for modeling complex systems, whereas the latter also models the cognitive state and cognitive operations of an agent. Thus, it seems natural to distinguish between a weak concept of agents, commonly used in individual-based simulation where agents are entities that interact with their environment and with each other, and a stronger concept that is based on modeling the cognitive (or mental) state of agents. In any case, since the interactions of agents are based on discrete perception and action events, it is natural to define an agent-based modeling and simulation approach as an extension of a *Discrete Event Simulation (DES)* approach.

Object-Event Modeling and Simulation (OEM&S), proposed in (Wagner 2017a; Wagner 2018), is a general DES paradigm combining object-oriented state structure modeling with the event scheduling paradigm defined by SIMSCRIPT (Markowitz et al. 1962) and Event Graphs (Schruben 1983). In this paper, we show how to extend OEM&S by adding concepts of agent-based M&S, resulting in a paradigm that we call *Agent/Object-Event Modeling and Simulation (A/OEM&S)*. The main point for such an extension is to define agents as special objects, which are not only subject to general (physical) laws of causality captured in the form of event rules, but which also have their own behavior allowing them to interact with their inanimate environment and with each other. Because agent behavior is decoupled from physical causality, an A/OE simulator consists of an environment simulator, which is in charge of simulating the physical world (the objective states of material objects), and agent simulators, which simulate the internal (subjective) states of agents and their behavior.

Consequently, it seems natural to distinguish between a weak concept of agents, as it is common in individual-based simulation, where agents are entities that interact with their environment and with each other, and stronger concepts that are based on modeling the cognitive (or mental) state of agents. In any

978-1-5386-6572-5/18/$31.00 ©2018 IEEE

case, since the interactions of agents are based on discrete perception and action events, it is natural to define an agent-based M&S approach as an extension of a *Discrete Event Simulation (DES)* approach.

This paper presents the issues and the modeling concepts needed for A/OEM&S. In a follow-up paper, we will present a JavaScript-based implementation of an A/OEM&S framework called *A-OESjs*.

For a general model of the interactive behavior of agents, we need to model the beliefs of an agent about its environment and about itself, which result both from perception and from communication. Beliefs represent the typically partial and sometimes incorrect (subjective) information of agents about their environment. They are the most basic component of the cognitive state of an agent. The simplest model of a cognitive state only consists of beliefs, while more advanced models may also include commitments, goals, intentions, emotions, etc. The beliefs of an agent can be viewed as its information items, such that the term *information state* is synonymous to *belief state*.

In many agent-based M&S projects, it is not relevant to model the incompleteness of beliefs or the possibility of incorrect beliefs, and, consequently, there is no need for representing an explicit belief state as a kind of duplication of the objective information state managed by the simulator. In these cases, we can make the assumption that agents have *perfect information* and short-circuit their information state with the objective information state managed by the simulator.

A concept of belief is needed for modeling both basic forms of interactive behavior: for modeling the interaction with the inanimate environment via a perception-action cycle and for modeling communication between agents. Perception events typically lead to the formation of new beliefs, which are the basis for (re)actions. New beliefs may also result from communication events, especially in communication processes based on tell-ask-reply messages. A concept of belief is needed for modeling both basic forms of interactive behavior: the interaction with the inanimate environment via a perception-action cycle and the communication between agents. Perception events typically lead to the update or formation of new beliefs, which are the basis for (re)actions. Updated and new beliefs may also result from communication events, especially in communication processes based on tell-ask-reply messages.

The beliefs of an agent about objects in the environment (including other agents) cannot be represented in the form of *property-value slots*, like *nameOf007* = "JB", but need to be represented in the form of *object-property-value triples*, like ⟨007, name, "JB"⟩. Therefore, the types of beliefs of an agent cannot be defined by ordinary properties in the definition of an agent type A, but rather by *belief object types* O^b, such that an agent of type A may have beliefs about belief objects of type O^b in the form of object-property-value triples.

In general, communication between agents includes many forms of conversational message exchanges. The most fundamental message types for agent communication are TELL, ASK and REPLY, which are also included in the agent communication language standards FIPA (Foundation for Intelligent Physical Agents) (Fipa 1996) and KQML (Kqml 1993). Making a distinction between facts and beliefs in a simulation framework allows distinguishing between sincere answers and lies in communication.

In general, the behavioural repertoire of an agent may include both *reactive* and *proactive* forms of behavior. The reactive behavior of an agent consists of actions performed in response to events. Proactive behavior is based on declarative tasks or goals that define a certain state of affairs, for which to achieve an agent has to generate and execute an action plan. Whenever an agent commits to execute an action plan, such a plan may be viewed to be a component of the agent's cognitive state in the sense of an *intention*.

There is a popular term, "belief/desire/intention (BDI) architecture", standing for a class of mental agent architectures that have only been sketched, but not well-defined, in (Bratman et al. 1988). In most BDI architectures, the term "desire" is interpreted in the sense of goals. Thus, the term essentially stands for any architecture supporting proactive behavior.

Since reactive behavior is more fundamental than proactive behavior, the latter should be added to a well-defined architecture of the former. In this paper, we are only concerned with reactive behavior. But we plan to add proactive behavior later, when the foundations of A/OEM&S have been established.

As an illustrating example, we use a scenario where a prince, after arriving on an island represented as a 2-dimensional grid, has to collect treasures and find the castle with the princess. On his way, the prince may meet knights who have information about the position of nearby treasures and the direction to the castle with the princess. All objects, including the castles, the princess, the treasures, the prince and the knights, have a position in the form of an (x,y) grid coordinate. These object positions are facts maintained by the environment simulator. As agents, the prince and the knights have self-beliefs about their own position and certain beliefs about the position of treasures and the castle with the princess.

2 RELATED WORK

Scientific communities adopt agent concepts differently. The most prominent communities exploiting these concepts are the Agent-Based Modeling (ABM) and the Multiagent System (MAS) communities. The former uses the concept of agents as an abstraction to model a collection of autonomous decision-making entities to searching for explanatory insights into collective behavior (Bonabeau 2002); while the latter uses the agent abstraction to design systems composed of a loosely coupled network of agents that work together to solve specific practical and engineering problems that are beyond the individual capabilities or knowledge of each agent (Stone and Veloso 2000). Although both use the concept of agents, their goal differences prevent them from agreeing on a standard definition. Still, they agree on some agents capabilities: (1) to perceive the environment (*perceive*), (2) to store and update their subjective view of the environment (*belief*), (3) to change the environment (*act*), and (4) to interact among themselves (*communicate*).

Current simulation tools used by these communities offer different levels of support for modeling these features (see Table 1). Usually platforms used by the ABM community are based on the object-oriented approach and they neither support modeling beliefs or message types, nor clearly distinguish between facts and beliefs.

Table 1: Summary of agent supported features of ABM and MAS platforms.

Platform	**Perceive-Act**	**Fact/Belief**	**Belief Type**	**Message Type**
NetLogo	Yes	No	*property-value*	—
Repast Symphony	Yes	No	—	—
MASON	Yes	No	—	—
Gamma	Yes	No	—	FIPA
AnyLogic	Yes	No	—	—
Jadex BDI	Yes	No	*property-object*	FIPA
Jason	Yes	Yes	*first-order predicate*	partially FIPA/KQML
2APL	Yes	Yes	*first-order predicate*	FIPA

NetLogo (Wilensky 1999) allows the representation of situated agents as turtles in an environment represented as a 2-dimensional space divided up into a grid of patches. Turtles support simple *property-value* variables (or beliefs), some predefined by the system, but they do not distinguish between facts and beliefs. Turtles can act using the command *ask* to sense the environment or to interact with other turtles. Turtles interact via method invocation rather than via the exchange of typed messages.

Repast Symphony (North et al. 2013) and MASON (Luke et al. 2005) are both multiagent simulation toolboxes that provide a set of tools for developing agent-based models in Java. Agents in these toolboxes are defined by extending an abstract Agent class that include default actions and perceptions of the agent. These toolboxes, however, do not support any built-in belief model or message types, leaving their definition to the modeler. Likewise, the GAMA (Drogoul et al. 2013) platform does not define any programming construct to model beliefs; however, GAMA provides a high-level modeling language (GAML) that supports message types, multi-level modeling of agents and very complex environment representations.

AnyLogic is a multimethod simulation modeling tool that supports agent-based, discrete event and system dynamics simulation methodologies. Agents are represented as active objects and modeled as state machines that support sensing and acting on the environment as well as the interaction with other agents. AnyLogic, however, does not neither distinguish between facts and beliefs nor supports belief modeling.

The situation is different to the MAS community in which frameworks, such as Jadex BDI (Pokahr et al. 2005), Jason (Bordini et al. 2007) and 2APL (Dastani 2008), provide programming constructs to specify systems in terms of a set of individual agents and a set of environments in which agents can perform actions. At the individual agent level, these frameworks provide programming constructs to implement cognitive agents based on the BDI architecture, such as beliefs, goals, plans, actions (i.e., belief updates, external actions, or communication actions), and a set of rules through which the agent can decide which actions to perform. They also define or use specific message types through which agents can communicate.

A/OEM&S may bridge the gap between both communities and integrates features important to both in a single formally well-defined simulation platform.

3 INTRODUCTION TO OEM&S

For modeling a discrete dynamic system according to the OEM&S paradigm, we have to

1. Describe its object types and event types;
2. Specify, for any event type, an *event rule* that captures a *causal regularity* responsible for state changes of objects and follow-up events, and that is triggered by events of that type.

In OEM&S, the object types and event types that are relevant for describing the state structure and dynamics of the system under investigation are defined in an information model, which forms the basis for making a process model. Any simulation modeling approach following the OEM&S paradigm (called an *OEM approach*) needs to choose, or define, an information modeling language and a process modeling language. Possible choices are Entity Relationship Diagrams or UML Class Diagrams for information modeling, and UML Activity Diagrams or BPMN Process Diagrams for process modeling. In the OEM approach of Wagner (2018), the choice consists of UML Class Diagrams for conceptual information modeling and information design modeling, as well as BPMN Process Diagrams for conceptual process modeling and DPMN Process Diagrams for process design modeling (DPMN stands for *Discrete Event Process Modeling Notation*, which is a BPMN-based process modeling language).

Object types and event types are modeled as special categories of classes in a UML Class Diagram, which can be implemented with any object-oriented (OO) programming language or simulation library/framework. *Random variables* are modeled as a special category of UML operations constrained to comply with a specific probability distribution such that they can be implemented as methods of an object or event class with any OO simulation technology. *Event rules*, which include *event routines*, are modeled visually in BPMN and DPMN process diagrams and textually in pseudo-code, such that they can be implemented in the form of special *onEvent* methods of event classes.

An OEM approach results in a simulation design model specifying object types and event types and causal regularities for each type of event in the form of event rules. Such a specification has a well-defined operational semantics, as shown in (Wagner 2017a). It can, in principle, be implemented with any OO simulation technology. However, a straightforward implementation can only be expected from a technology that implements the OEM&S paradigm, such as the OES JavaScript (OESjs) framework presented in (Wagner 2017b).

4 BELIEFS

In the simulation of cognitive agents that are situated in an environment consisting of objects and agents, there are two kinds of information items: facts and beliefs. Facts represent the true (objective) state of the

environment, while beliefs represent the typically partial and sometimes incorrect (subjective) information of agents about their environment.

In general, not all beliefs are categorical (having a classical truth value). Certain types of beliefs may be qualified in some way, e.g., in terms of (disjunctive or gradual) uncertainty, lineage or multi-level security. Gradual uncertainty can be expressed with the help of an uncertainty scale such as fuzzy certainty values or probabilities. However, for simplicity, and lack of space, we will not consider any type of belief qualification in this paper.

Beliefs about objects in the environment (including other agents) cannot be represented by simple *property-value slots*, but need to be represented by *object-property-value triples*. Therefore, the definition of an agent type A may include the definition of one or more *belief object types* O^b, such that an agent of type A may have beliefs about belief objects of type O^b in the form of object-property-value triples. These belief triples may be stored in relational-database-like tables or Resource Description Framework (RDF) (W3c 2014) style triple stores as part of the agent's mental state.

The concept of belief object types allows to represent all kinds of beliefs of an agent about its environment, no matter which vocabulary (or ontology) the agent is using. In this way, agents could either use a shared vocabulary, or they could use their own private vocabularies, which would have to be mapped to each other for successful communication. However, in this paper, we do not consider the problems of private vocabularies and ontology/vocabulary mapping. Proposals for solving these communication problems with the help of semantic negotiation, shared and common ontologies or cooperation between agents can be found in (Diggelen et al. 2006; Williams 2004; Garruzzo and Rosaci 2007).

For simplicity, we assume that all agents are using a shared vocabulary, which is the same as the vocabulary used by the environment simulator. This *shared vocabulary assumption* implies that all agents, and the environment simulator, use the same unique names for

1. Object types,
2. Properties of objects,
3. Objects.

Notice that, technically, the unique names for objects are object IDs.

The shared vocabulary assumption does not imply that a belief object type defined for several agents has the same schema (set of properties) as the corresponding object type defined for the environment simulator. As shown in the example below, since beliefs are often incomplete, belief object types may be defined with a subset of the properties defined for the corresponding object type. An additional assumption, called the *shared schema assumption*, is needed for making the schemas of belief object types equal to the schemas of corresponding object types.

For defining an agent type, such as *Prince*, we have to specify its objective properties, like x, y and *wealth* providing its objective position and wealth, its self-belief properties (e.g., *wealth* for representing its believed wealth) and its belief object types, e.g., *Treasure* and *Princess*.

Following the syntax of OESjs, we could have the following code for an A/OES agent type definition in A-OESjs:

```
var Prince = new aGENTtYPE({
  name: "Prince",
  supertype: "GridSpaceObject",
  properties: {"wealth": {range: "NonNegativeInteger"}},
  selfBeliefProperties: ["wealth"],
  beliefObjectTypes: [{"Treasure": ["x","y"]}, {"Princess": ["x","y"]}],
  ...
 });
```

Notice that the agent type *Prince* is defined as a subtype of *GridSpaceObject* such that it inherits the gridspace position properties *x* and *y*. Since there are no corresponding definitions of self-belief properties, this means that a prince agent does not have a belief about (i.e., does not know) its position, but only have beliefs about its wealth and about the positions of the treasure and the princess objects.

The distinction between facts and beliefs requires that a simulator maintains both the objective and the subjective state of an agent in parallel. This can be achieved by forming two classes from an agent type definition: an *agent object* (AO) class and an *agent subject* (AS) class. E.g., the agent type *Prince* could be implemented by forming the *PrinceAO* and *PrinceAS* classes shown in Figure 1.

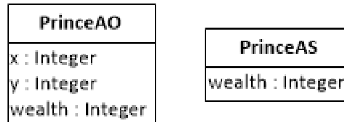

Figure 1: Two sides of the same coin: an *agent object type* and a corresponding *agent subject type*.

While the instances of an agent object class are objects whose states are managed by the environment simulator, the instances of an agent subject class represent agents that are managed by an agent simulator, which may run in a separate thread.

Using the shared schema assumption, the *Prince* agent type definition can be simplified to

```
var Prince = new aGENTtYPE({
  name: "Prince",
  supertype: "GridSpaceObject",
  properties: {"wealth": {range: "NonNegativeInteger"}},
  beliefObjectTypes: ["Treasure", "Princess"],
  ...
 });
```

Based on the shared schema assumption, (1) prince agents now have three self-belief properties (“x”,“y” and “wealth”), defined implicitly, one for each objective property, and (2) the belief object types “Treasure” and “Princess” have the same properties as the corresponding object types.

4.1 Facts and Beliefs as Triples

More precisely speaking, we do not deal with ‘facts’ and ‘beliefs’, but with *atomic fact statements* and *atomic belief statements*, each of them having the form of an *object-property-value triple*. For instance, the atomic fact statement that the wealth of the prince agent with ID 17 is 500, is expressed by the triple

[It’s a fact that] 17 wealth 500

while the corresponding atomic belief statement of agent 17 that its wealth is 650 is expressed by the triple

[Agent 17 beliefs that] 17 wealth 650

and the atomic belief statement of agent 17 that the position of object 23 (the princess) is given by x = 35 and y = 42 is expressed by the triples

[Agent 17 beliefs that] 23 x 35, 23 y 42

In standard predicate logic syntax, such a triple corresponds to an atomic sentence where the property of the triple statement would be used as a predicate, and the object identifier and the property’s value would be the arguments of this predicate, resulting in an expression like:

[Agent 17 beliefs that] wealth(17, 650)

The formal logic language RDF defined by the W3C supports the use of triples along with multiple vocabularies. The RDF query language SPARQL (W3c 2013) allows expressing queries about information triples. It seems to be a natural choice for expressing queries about the beliefs of other agents in ASK messages.

4.2 Perfect Information Agents

A *perfect information (PI)* agent has complete and correct information about itself and about all objects that it knows.

The *Prince* agent type can be defined as a PI agent type in the following way:

```
var Prince = new aGENTtYPE({
  name: "Prince",
  supertype: "GridSpaceObject",
  properties: {"wealth": {range: "NonNegativeInteger"}},
  hasPerfectInformation: true,
  ...
 });
```

This definition implies that all objective properties are duplicated as self-belief properties and all self-belief slots have the same values as the corresponding fact slots, which requires to synchronize the information state of a PI agent with its objective state maintained by the environment simulator.

4.3 Discrepancies between Facts and Corresponding Beliefs

In general, we can have various types of discrepancies between facts and corresponding beliefs. The first issue is the possibility to use different vocabularies to express belief statements about the same fact. Assuming shared vocabularies, we still have the possibility of discrepancies between facts and corresponding beliefs with respect to (1) the completeness of beliefs and (2) the actual and the believed value of a property.

There are several types of possible discrepancies arising from different vocabularies being used. Agents may use different names for object types and/or properties, and they may use different identifiers for objects. There are also the issues of *schema incompleteness* and *non-correspondence*. Schema incompleteness concerns the possibility that not all object types and properties (as defined objectively for the environment of a simulation model) have a corresponding name in the vocabulary of an agent. Non-correspondence refers to the possibility that some of the object type and/or property names used by an agent do not correspond to a real object type or property.

However, in many cases, models abstract away from these possible discrepancies, using the perfect information assumption, the shared schema assumption or at least the shared vocabulary assumption.

5 THE PERCEPTION-ACTION CYCLE

An agent may perceive objects or events in its environment. One may distinguish between a passive and an active form of perception. Passive perception takes place asynchronously while an agent is busy or idle, while active perception requires the agent to perform a perceptive action or activity (like issuing a query about its neighborhood).

The A/OEM&S paradigm is based on a simple model of perception where passive perceptions take the form of (instantaneous) perception events and active perception consists of querying the environment simulator. In general, agents are exposed to perception events and react in response to them.

Conceptually, an incoming message event, caused by another agent sending the message, can be viewed as a special type of perception event. Likewise, a communication action, which is an outgoing message event, can be viewed as a special type of action event. However, for simplicity, in the A/OEM&S language,

we will neither subsume incoming message events and non-communicative perception events under a general concept of perception events, nor outgoing message events and non-communicative action events under a general concept of action events. In A/OEM&S, the term “perception event” refers to non-communicative perception events, and the term “action event” refers to non-communicative action events (see Figure 2).

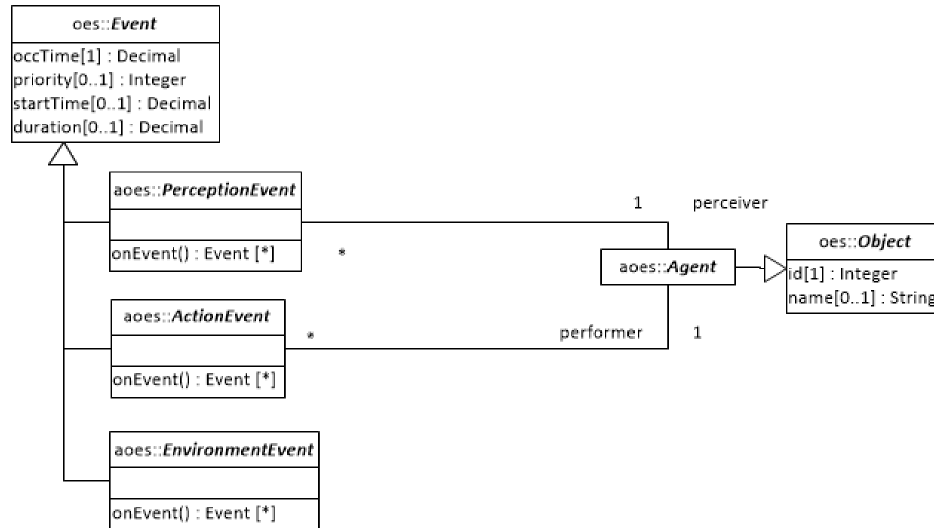

Figure 2: Elements of perception and action.

In the case of physical agents, like robots, (passive) perception events are created by a sensor containing an event detector, such as a *Proximity Infra-Red (PIR)* sensor, which has its output pin going high whenever it detects an infra-red emitting object (e.g., a human or a medium to large sized animal) in its reach. Active perceptions are created by querying the measurement value of a sensor containing a quality detector, such as a *DHT22* temperature and humidity sensor. The distinction between event detectors and quality detectors has been proposed in (Diaconescu and Wagner 2015).

Perception events are internal events created by the environment simulator and transmitted to an agent simulator that processes them by invoking the agent’s *perception event rule* method for the type of the perception event, which represents the agent’s reaction rule for this type of event.

Action events are external events created by an agent simulator and transmitted to the environment simulator, which processes them by invoking the action event type’s onEvent method representing the environment simulator’s event rule.

In our example scenario, the prince agent may perceive a treasure object when moving forward. In the simplest model, perception would be limited to the grid cell to which the prince has moved. When the prince agent has discovered a treasure object, it reacts by picking it up. We could have the following A-OESjs code for defining the action event type *PickUpTreasureObject* with the property *treasureObject* and an event rule:

```
var PickUpTreasureObject = new aCTIONeVENTtYPE({
  name: "PickUpTreasureObject",
  properties: {"treasureObject": "TreasureObject"},
  onEvent: function (e) {
    e.performer.wealth += e.treasureObject.value;
    sim.objects.remove( e.treasureObject.id); // destroy treasure object
  }
 });
```

The event rule of an action event type is applied by the environment simulator when it receives an action event of that type from an agent simulator. In this case the environment simulator would execute the onEvent method such that the prince agent's *wealth* attribute is incremented by the value of the treasure object, which is then destroyed (removed from the simulation).

In general, perception event types may be shared among several agent types, but in the following example *TreasureObjectRecognition* is defined as an internal perception event type within the agent type definition.

```
var Prince = new aGENTtYPE({
  name: "Prince",
  supertype: "GridSpaceObject",
  properties: {"wealth": {range: "NonNegativeInteger"}},
  beliefObjectTypes: ["Treasure", "Princess"],
  ...
  perceptionEventTypes: {
    "TreasureObjectRecognition": {
      properties: {"value": {range: Integer}}
    }
  },
  perceptionEventRules: [
    {"TreasureObjectRecognition": function (e) {
      var actionEvents = [];
      e.perceiver.wealth += e.value;
      actionEvents.push( new PickUpTreasureObject());
      return actionEvents;
    }}
  ],
 });
```

The prince agent perceives the value of a discovered treasure object and increments its *wealth* self-belief property accordingly. The simulation model may create a discrepancy between the perceived and the actual value of a treasure object by defining an event rule that creates *TreasureObjectRecognition* events with a value that deviates from the actual value.

6 COMMUNICATION

In general, communication between agents includes many forms of conversational message exchanges. The most fundamental type of a conversational message exchange is tell-ask-reply communication.

There are two kinds of message types:

- **Ad-hoc** message types: the structure and semantics of these messages types are defined for specific simulation models in the form of rules governing the processing of messages of such a type.
- **Generic** message types: Tell/Untell and Ask/Reply are examples of generic message types, which should be provided as built-ins in a cognitive agent simulation framework. As they refer to general speech acts, the semantics of these message types, which implies a corresponding communication protocol, should be based on speech act theory.
  It is an option to define an ad-hoc message type as an extension of a generic message type.

There are two prominent agent communication language standards: FIPA and KQML, both defining standard message types with a semantics based on the philosophical *speech act* theory of Austin and Searle.

Figure 3 shows the elements of A/OEM&S message-based communication. For simplicity, we assume that all agents are using a shared vocabulary and a shared schema, as discussed in Section 4.

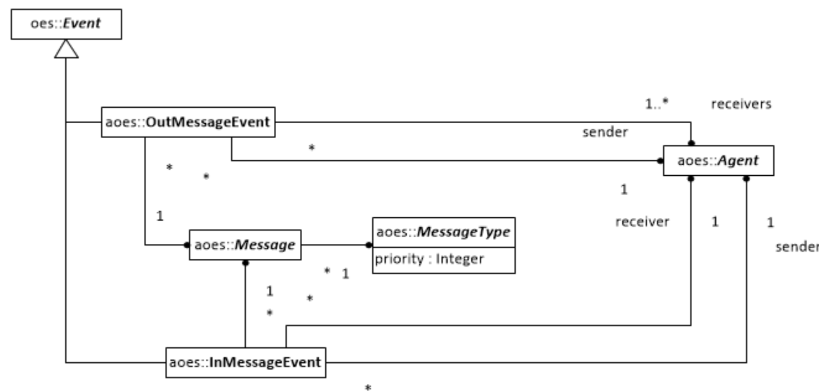

Figure 3: Elements of message-based communication.

Out-message events are external events transmitted by the sender's agent simulator to the environment simulator, which transforms them to corresponding in-message events that are transmitted to the receivers' agent simulators.In-message events are internal events transmitted by the environment simulator to the receiver's agent simulator that processes them by invoking the agent's *in-message event rule* method for this message type.A communication event (see Figure 4) is a composite event consisting of an out-message event and one or more corresponding in-message events.

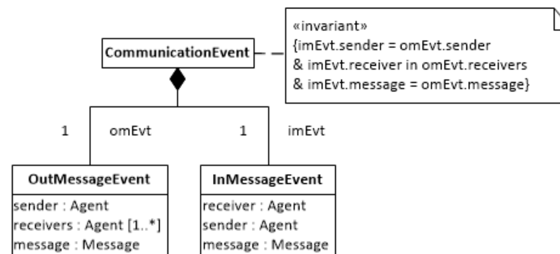

Figure 4: Communication as a composite event.

In our example scenario, the prince agent has to ask knights about the location of treasures, and knights have to reply with zero or more answers. Using the ad-hoc message types *AskAboutTreasureLocations* and *ReplyTreasureLocations*, we could have the following A-OESjs code:

```
var AskAboutTreasureLocations = new mESSAGEtYPE({
  name: "AskAboutTreasureLocations",
  properties: {"myTreasureLocations": {range: Array}}
});
var Prince = new aGENTtYPE({
  name: "Prince",
  supertype: "GridSpaceObject",
  properties: {"wealth": {range: "NonNegativeInteger"}},
  beliefObjectTypes: ["Treasure", "Princess"],
  outMessageTypes: ["AskAboutTreasureLocations"],
  ...
});
```

The agent type *Knight* would have to process an incoming message of type *AskAboutTreasureLocations* and respond with an outgoing message of type *ReplyTreasureLocations* according to the agent's in-message event rule for *AskAboutTreasureLocations* in-message events:

```
var ReplyTreasureLocations = new mESSAGEtYPE({
  name: "ReplyTreasureLocations",
  properties: {"treasureLocations": {range: Array}}
 });
var Knight = new aGENTtYPE({
  name: "Knight",
  supertype: "GridSpaceObject",
  beliefObjectTypes: ["Treasure", "Princess"],
  inMessageEventRules: [
    {"AskAboutTreasureLocations": function (e) {
      var actionEvents = [], treasureLocations = ...;
      actionEvents.push( new OutMessageEvent({
        message: new ReplyTreasureLocations( treasureLocations),
        receivers: [e.sender]
      }));
      return actionEvents;
    }}
  ],
  outMessageTypes: ["ReplyTreasureLocations"]
 });
```

The responding knight agent retrieves the nearby treasure locations from its belief base after it has dropped its out-dated beliefs about the treasure locations that the prince agent has already found and communicated to the knight via the *myTreasureLocations* message property.The agent's simulator transmits the *ReplyTreasureLocations* out-message event to the environment simulator, which creates a corresponding in-message event and transmits it to the asking knight agent's simulator.

7 CONCLUSIONS

We have shown how to define an agent-based simulation paradigm on top of the general DES paradigm OEM&S by accommodating the three most fundamental agent features: beliefs, interaction with the inanimate environment via perception and (re-)action, and inter-agent communication.In future work we plan to fully implement the A/OEM&S paradigm on the basis of the OESjs simulation framework. We also plan to extend the formal semantics of OEM&S such that it accounts for A/OEM&S.

REFERENCES

Bonabeau, E. 2002. "Agent-Based Modeling: Methods and Techniques For Simulating Human Systems". *Proceedings Of The National Academy Of Sciences Of The United States Of America* 99(3):7280–7287.

Bordini, R. H., J. F. HÜbner, and M. J. Wooldridge. 2007. *Programming Multi-Agent Systems In Agentspeak Using Jason*. 1st ed. Wiley Series In Agent Technology. Chichester: Wiley & Sons.

Bratman, M. E., D. J. israel, and M. E.Pollack. 1988. "Plans and Resource-Bounded Practical Reasoning". *Computational Intelligence* 4(4):349–355.

Dastani, M. 2008. "2apl: A Practical Agent Programming Language". *Autonomous Agents and Multi-Agent Systems* 16(3):214–248.

Diaconescu, I.-M., and G. Wagner. 2015. "Modeling and Simulation Of Web-Of-Things Systems As Multi-Agent Systems". In *Proceedings Of The 2015 German Conference On Multiagent System Technologies (Mates)*, Volume 9433 of *Lecture Notes In Artificial Intelligence*, 137–153, edited by P. Muller et al.: Springer Verlag.

Diggelen, J. V., R. J. Beun, F. Dignum, R. M. V. Eijk, and J. J. C. Meyer. 2006. "Anemone: An Effective Minimal Ontology Negotiation Environment". In *Proceedings Of The International Joint Conference On Autonomous Agents and Multi-Agent Systems (Aamas)*, 899–906. Hakodate, Japan: Acm.

Drogoul, A., E. Amouroux, P. Caillou, B. Gaudou, A. Grignard, N. Marilleau, P. Taillandier, M. Vavasseur, D. A. Vo, and J. D. Zucker. 2013. "Gama: Multi-Level and Complex Environment For Agent-Based Models and Simulations". In *Proceedings Of The International Joint Conference On Autonomous Agents and Multiagent Systems*, 1361–1362. Ifaamas.

Fipa 1996. "Foundation For Intelligent Physical Agents". Accessed July 20th, 2018.

Garruzzo, S., and D. Rosaci. 2007. "Ontology Enrichment In Multi-Agent Systems Through Semantic Negotiation". In *Proceedings Of The International Conference On Cooperative Information Systems (Coopis)*, Volume 4803 of *Lecture Notes In Computer Science*, 391–398. Vilamoura, Portugal: Springer.

Kqml 1993. "Knowledge Query and Manipulation Language (Kqml)". Accessed July 20th, 2018.

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. "Mason: A Multi-Agent Simulation Environment". *Simulation: Transactions Of The Society For Modeling and Simulation International* 82(7):517–527.

Markowitz, H., B. Hausner, and H. Karr. 1962. "Simscript: A Simulation Programming Language". Memorandum Rm-3310-Pr, The Rand Corporation, Santa Monica, California.

North, M. J., N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. 2013. "Complex Adaptive Systems Modeling With Repast Simphony". *Complex Adaptive Systems Modeling* 1(3):1–26.

Pokahr, A., L. Braubach, and W. Lamersdorf. 2005. "Jadex: A Bdi Reasoning Engine". In *Multi-Agent Programming*, 149–174, edited by R. Bordini et al. New York, Ny: Springer Science+Business Media Inc., Usa.

Schruben, L. 1983. "Simulation Modeling With Event Graphs". *Communications Of The Acm* 26:957–963.

Stone, P., and M. Veloso. 2000. "Multiagent Systems: A Survey From A Machine Learning Perspective". *Autonomous Robots* 8(3):345–383.

W3c 2013. "Sparql 1.1 Query Language". Accessed July 20th, 2018.

W3c 2014. "Rdf Schema 1.1". Accessed July 20th, 2018.

Wagner, G. 2017a. "An Abstract State Machine Semantics For Discrete Event Simulation". In *Proceedings Of The 2017 Winter Simulation Conference*, 762–773, edited by W. K. V. Chan et al. Piscataway, New Jersey: IEEE.

Wagner, G. 2017b. "Sim4edu.Com – Web-Based Simulation For Education". In *Proceedings Of The 2017 Winter Simulation Conference*, 4240–4251, edited by W. K. V. Chan et al. Piscataway, New Jersey: IEEE.

Wagner, G. 2018. "Information and Process Modeling For Simulation Part I: Objects and Events". *Journal Of Simulation Engineering* 1:1:1–1:25.

Wilensky, Uri 1999. "Netlogo". Accessed July 20th, 2018.

Williams, A. B. 2004. "Learning To Share Meaning In A Multi-Agent System". *Autonomous Agents and Multi-Agent Systems* 8(2):165–193.

AUTHOR BIOGRAPHIES

GERD WAGNER is Professor of Internet Technology at Brandenburg University of Technology, Germany. His research interests include (agent-based) modeling and simulation, foundational ontologies, knowledge representation and web engineering. His e-mail address is wagnerg@b-tu.de.

LUIS G. NARDIN is Assistant Professor at Brandenburg University of Technology, Germany. His research interests are agent-based modeling and simulation, multiagent systems, and simulation data analysis. His email address is nardin@b-tu.de.