Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin and R. G. Ingalls, eds.

EXTENDING DISCRETE EVENT SIMULATION BY ADDING AN ACTIVITY CONCEPT FOR BUSINESS PROCESS MODELING AND SIMULATION

Gerd Wagner
Oana Nicolae
Jens Werner

Department of Informatics
Brandenburg University of Technology
Cottbus, Germany

ABSTRACT

We show how a basic discrete event simulation language can be enabled for business process modeling and simulation by adding an activity construct. While activities are often not considered at all or not treated in a conceptually satisfactory way in the discrete event simulation literature, the great majority of business process modeling languages are based on an activity construct. However, unlike a simulation language, the predominant business process modeling languages, including *UML Activity Diagrams* and the *Business Process Modeling Notation (BPMN)*, are not executable. So, the challenge for business process modeling is to define an executable semantics for activities, while the challenge for discrete event simulation is to find a way how to introduce an activity construct on top of the basic discrete event simulation concepts of *objects* and *events*. The main idea is to define an activity as a complex event having a *start event* and an *end event*. This idea is well-known from the business process modeling literature, e.g. from the *Business Process Definition Metamodel* (Bock 2008), but it has also been used in a rudimentary way in some discrete event simulation approaches, e.g. in (Fishman 2001). Our research contribution consists of two achievements: 1) we define a conceptual model of activities for discrete event simulation and implement it in our ER/AOR simulation language; 2) we show how to use BPMN for the purpose of simulation modeling.

1 INTRODUCTION

In this paper we present a solution for modeling activities, and business processes, on top of the basic discrete event simulation (DES) concepts of *objects* and *events*. We develop our solution by extending an open source DES framework, called *ER/AOR* simulation, which is an agent-based DES framework available from <www.AOR-Simulation.org>. However, we argue that our approach is generic and could also be applied to other DES languages.

The paper is structured as follows. In the introductory subsection 1.1, we briefly discuss the concept of activities being used in DES and in agent-based simulation, and then present our own definition of activities. In subsections 1.2 and 1.3, we introduce the *ER/AOR* simulation framework and the *Business Process Modeling Notation (BPMN)*. In section 2, we show how to model and simulate a simple service queue system without using activities. Then, in section 3, we explain how to add the construct of *activity types* to the ER/AOR simulation language. In section 4, we get back to the simple service queue system and show how to model it with the help of an activity type. Then, in section 5 we compare two versions of another, more complex model (of a drive-thru restaurant): one without and the other one with agents and activities. Finally, in sections 6 and 7, we briefly discuss the merits and shortcomings of BPMN as a simulation modeling language and summarize our conclusions.

1.1 Activities in Discrete Event Simulation

In the DES literature, there is no agreement on how to define activities. This may be due to the fact that activities do not form a basic, but rather an advanced concept of DES.

Ingalls states in (Ingalls 2008), that “Activities are processes and logic in the simulation” and that “There are three major types of activities in a simulation: *delays, queues* and *logic*”. For a business process modeler, these statements must sound enigmatic. In business process modeling (BPM), activities are not processes, and one would never think that a delay or a queue

is an activity. What Ingalls really seems to mean is that in simulation the only relevant feature of many activities is the delay they create in the flow of events (for the sake of using a clear and ontologically faithful terminology, however, one should not classify a delay, or a queue, as an activity).

Also in (Banks et al. 2005), a severely limited definition of the activity concept is used: an activity is defined to be a "duration of time of specified length". Again, this definition represents a projection of the general concept to a specifically biased view. It excludes all those activities whose duration is not already known at their start time. And conceptually, an activity is not a duration, but it rather has a duration.

In (Fishman 2001), an activity is defined as "a pair of events, one initiating and the other completing an operation that transforms the state of an entity". This definition comes much closer to a general definition of activities that is compatible with the concept of activities as being used in BPM.

In *Artificial Intelligence (AI),* very rich concepts of activities, related to motives, goals and belief-based planning, have been proposed for modeling situated cognitive systems (see, e.g., Clancey 2002). An interesting AI-oriented simulation system based on a rich concept of activities, is *Brahms* (Sierhuis 2001) `<www.agentisolutions.com>`. In Brahms, an activity is intended to represent a real-life action and is started by a *situation-action rule*. It has a (fixed or random) duration and may be associated with *resource objects*. Atomic actions (or action events) are modeled as primitive activities in Brahms, and there are three pre-defined primitive activities: communicate beliefs, create objects, and move to a specified location Since Brahms does not use an explicit event concept, it does not relate an activity to a start and an end event, like in a DES treatment of activities.

In our DES approach, we define activities as complex events having a start event, an end event, an optional association with an actor (being an agent that plays the role of a special resource) and zero or more associations with other objects playing the role of further resources used by the activity. This definition is graphically expressed in the conceptual UML class diagram shown in Figure 1. It accommodates both basic and agent-based activity modeling. In some models we may want to abstract away from the agent that performs an activity, in others we may prefer a closer-to-reality model of an activity being performed by an agent. In any case, an activity has a *start time*, a *duration* and an *occurrence time*, which is identified with the end time of the activity.

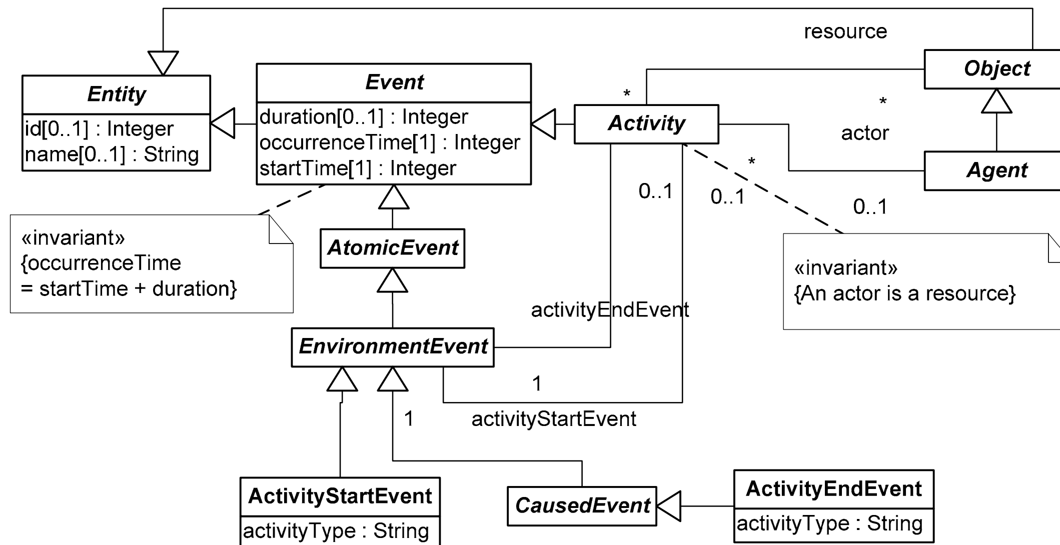

Figure 1: Activities have a start event, an end event, and possibly associated resources (including an actor)

Our approach allows an activity to be started, and ended, in two ways. Either it is started by an instance of the pre-defined event type `ActivityStartEvent` which refers to the type of the activity (by means of its `activityType` attribute) or it is started by some environment event whose user-defined type has been defined as the activity's start event type. The same holds for the activity's end event.

1.2 From Basic Discrete Event Simulation to Agent-Based Discrete Event Simulation

The *ER/AOR Simulation* framework was proposed in (Wagner 2004). It supports both basic discrete event simulation models without agents, also called *Entity-Relationship (ER)* simulations, and complex agent-based simulation models with agents having (possibly distorted) perceptions and (possibly false) beliefs, called *Agent-Object-Relationship (AOR)* simulations. No-

tice that we do not share the widely held view that DES is based on the concept of "entities flowing through the system". Rather, we understand DES to be based on the concept of an "event flow" that is changing the state of objects.

A simulation scenario is expressed with the help of the XML-based *AOR Simulation Language (AORSL)*. The scenario is then translated to Java source code, compiled to Java byte code and finally executed, as indicated in Figure 2.

Figure 2: From AORSL to Java byte code

Distinctive features of the *ER/AOR Simulation* framework are: (1) its high-level rule-based simulation language *AORSL*, (2) an abstract simulator architecture and execution model.

A ***simulation scenario*** essentially consists of a *simulation model* and an *initial state*. An ***ER simulation model*** consists of: (1) a set of *entity type* definitions, including different categories of *event* and *object* types; and (2) a set of *environment rules*, which define causality laws governing the state changes of the environment and the flow of event causation. An ***AOR simulation model*** consists, in addition, of a set of *message* types and *agent* types included in the entity type definitions.

An ***entity type*** is defined by means of a set of properties and a set of functions. There are two kinds of properties: attributes and reference properties. *Attributes* are properties whose range is a data type; *reference properties* are properties whose range is another entity type.

The upper level ontological categories of AOR Simulation are ***objects*** (including *agents*, *physical objects* and *physical agents*), ***messages*** and ***events***, as depicted in Figure 3 (according to this upper-level ontology of AOR Simulation, agents are special objects; for simplicity it is common, though, to say just 'object' instead of using the unambiguous but clumsy term 'non-agentive object'). Notice that only objects, but neither events nor messages, have a state that may change over time.

Both the behavior of the environment (its causality laws) and the behavior of agents are modeled with the help of *rules*, which support *high-level declarative simulation modeling*.

1.2.1 Entity-Relationship Simulation

In basic discrete event simulation, which we also call *Entity-Relationship (ER)* simulation, we deal with two basic categories of entities: *objects* and *events*. A simulation model defines a number of object types and event types, each of them with one or more properties and zero or more functions (to be used for various kinds of computations such as for computing pseudo-random numbers following an empirical distribution). There are two different kinds of event types: those that define *exogenous* events (typically with some random periodicity) and those that define *caused* events that follow from the occurrence of other events.

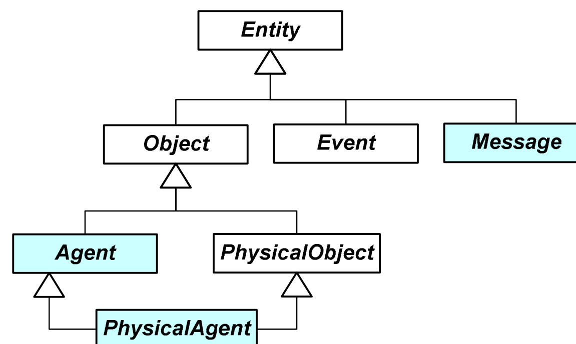

Figure 3: The upper-level ontological categories of ER/AOR simulation (agent-related categories in blue)

The state of the environment (i.e. the system state) is given by the combination of the states of all objects. Environment rules define how the state of objects is changed by (and which caused events result from) the occurrence of events.

An ***environment rule*** is a 6-tuple

<WHEN, FOR, DO, IF, THEN, ELSE>

where: (1) the mandatory `WHEN` element denotes the type of event that triggers the rule; (2) an optional block of `FOR` elements allows to declare variables, such that each variable is bound either to a specific object or to a set of objects; (3) the optional `IF` element is a logical formula (allowing for variables) expressing a state condition; and (4) the optional `DO`, `THEN` and `ELSE` elements are containers for an optional `UPDATE-ENV` element specifying an update of the environment state followed by an optional `SCHEDULE-EVT` element specifying a list of resulting future events.

In each simulation step, all those rules are fired whose triggering event types are matched by one of the current events. The firing of rules may lead to updates of the states of certain objects and it may create new future events to be added to the future events list. After this, the simulation time is incremented to the occurrence time of the next future event (if no continuous changes have been defined for the given model), and the evaluation and application of rules starts over.

1.2.2 Agent-Object-Relationship Simulation

In the form of agent-based discrete event simulation, which we call *Agent-Object-Relationship (AOR)* simulation, we deal with three basic categories of entities: *objects*, *agents* and *events*. When we introduce agents, we have to make further distinctions between different types of events, as depicted in Figure 4. In particular, we need to consider *perception* events and *actions* events in order to account for the perception-action cycle defining the foundation of agent behavior.

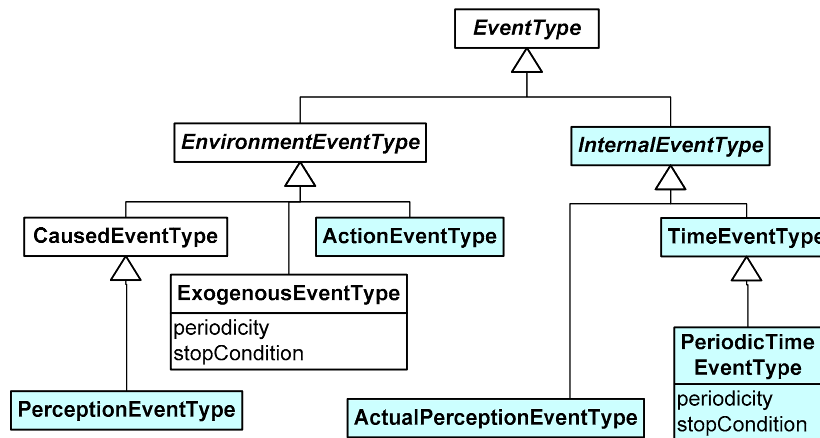

Figure 4: The event type constructs of the ER/AOR simulation language (agent-related elements in blue)

An ***agent type*** is defined by means of: (1) a set of (objective) properties; (2) a set of (subjective) self-belief properties as well as an optional set of (subjective) belief entity types; and (3) a set of *agent rules*, which define the agent's reactive behavior in response to perception events (and internal time events).

1.3 Introduction to the Business Process Modeling Notation

The *Business Process Modeling Notation* (BPMN), see <http://www.omg.org/spec/BPMN/1.2/>, has become today's de-facto BPM standard. BPMN is considered to be understandable by all kinds of users, from the business analysts who create the initial drafts of the business processes, to the technical developers responsible for writing the software applications that will perform those processes, and finally, to the business people who will manage and monitor them. BPMN can be used both for making intuitive, non-executable business process models and for making executable models, such as needed for business process simulation. In the sequel, we provide a brief description of the essential BPMN constructs (which are summarized in the metamodel shown in Figure 5).

BPMN defines *Business Process Diagrams*, which may contain several *processes* of different types. We focus on the representation of private business processes, i.e. processes that allow a complete specification of their workflow. A process may be associated with a *pool*, which is visually rendered as a rectangular container and corresponds to the business actor "owning" the process. A pool may be compartmented into several *lanes* each representing a process part associated with a specific actor under the same domain of control. A process essentially consist of 'flow objects' (events, activities and gateways), 'connectivity objects' (sequence flows, message flows and associations) and 'artifacts' (e.g. data objects).

Events (rendered as circles) and activities (rendered as rectangles with rounded corners) are sequenced with the help of *sequence flows* (rendered as solid arrows) and *gateways* (rendered as diamonds), which allow for AND, XOR as well as OR splits and joins of event flow branches.

Activities subsume *tasks*, which represent *atomic activities* (defined with the help of an attribute called `taskType`), and *sub-processes*. Tasks can be assembled into sub-processes, for allowing reuse. Pools are connected through *message flows* that represent business partner communication.

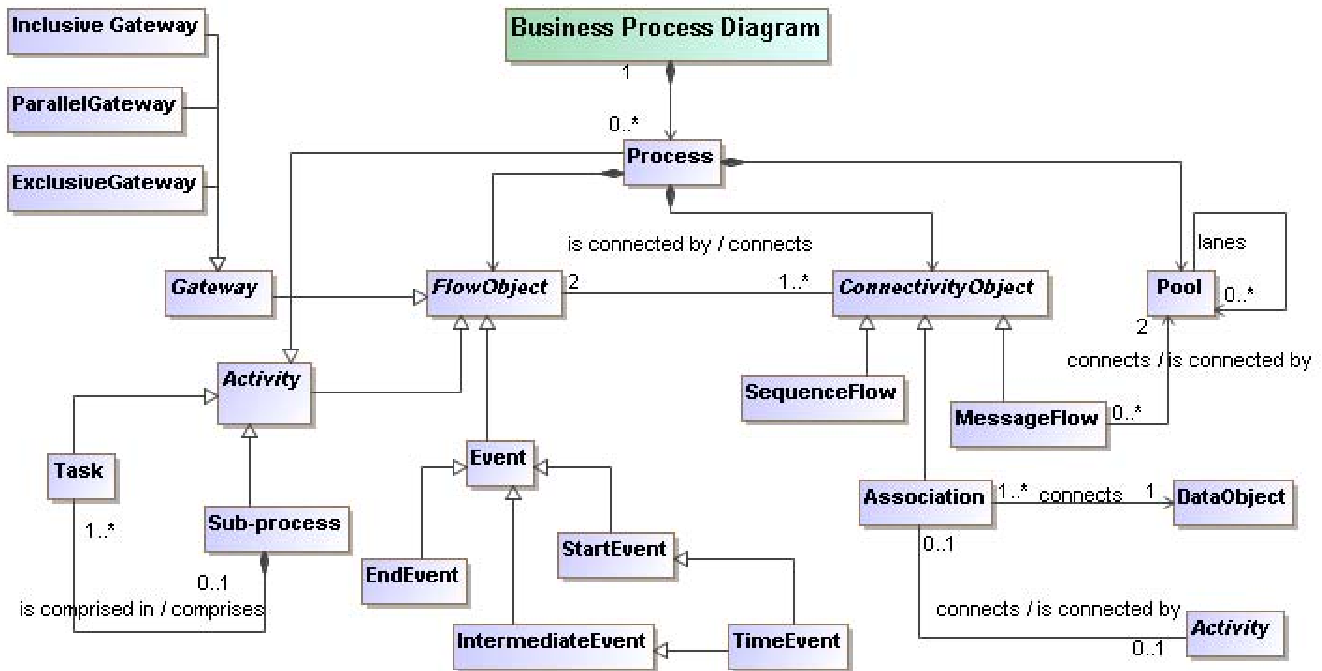

Figure 5: BPMN core elements

In the sequel we use BPMN as a simulation modeling language for expressing the process model that defines the flow of events. A full simulation model can be specified by combining an information model (e.g. a UML class model) with a BPMN process model.

2 MODELING A SERVICE QUEUE SYSTEM WITHOUT ACTIVITIES

In this section, we show how to model and simulate a service queue system without using activities. We use BPMN for expressing the process model for our simulation scenario.

Customers arrive at random times at a service desk where they have to wait in a queue when the service desk is busy. The time between two customer arrivals is uniformly distributed between 1 and 8 minutes. The time for completing a service is, for simplicity, also uniformly distributed between 1 and 6 minutes. An arriving customer is represented by a newly generated object, which is destroyed again, when this customer leaves the system. The customer in service is represented by the first item in the First-In-First-Out (FIFO) queue, while the remaining items of the queue represent the waiting line.

The information model for this scenario includes a global Boolean variable `serviceDeskBusy`, an object type `Customer` with an attribute `arrivalTime`, an exogenous event type `CustomerArrival` and the two caused event types `StartService` and `EndService`. In addition, we also define a first-in-first-out queue `CustomerQueue`. The process model is defined by the BPMN diagram shown in Figure 6. For simplicity, we have discarded the code needed for defining the statistics computations in this example. For the complete model, please check out <http://oxygen.informatik.tu-cottbus.de/aors/examples/Management/ServiceQueue_withoutActivities/scenario.xml>.

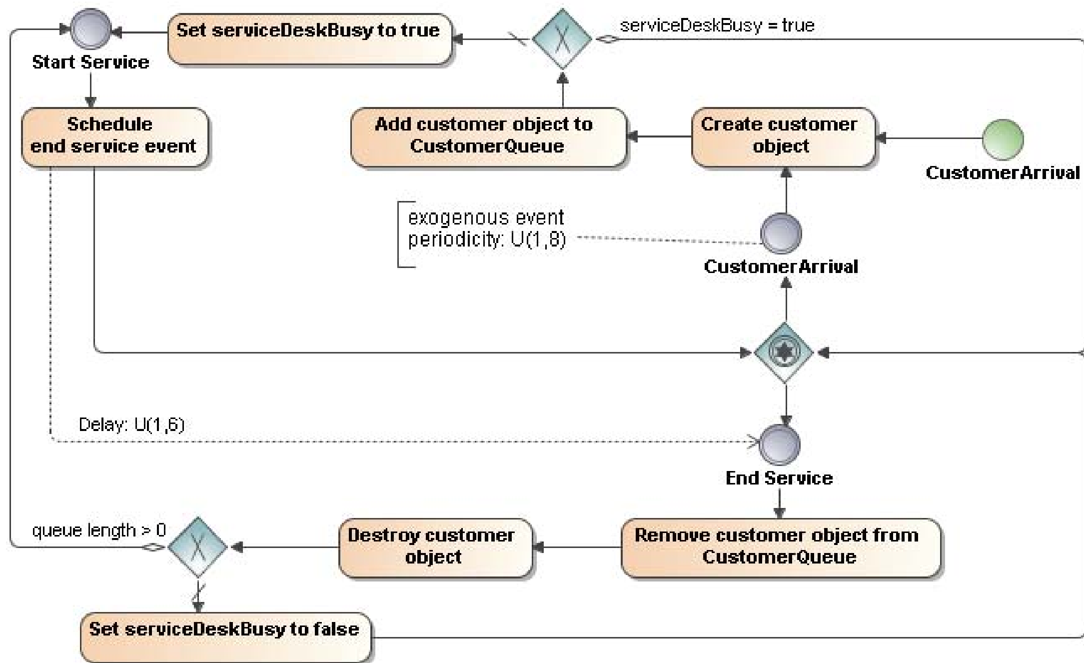

Figure 6: The BPMN diagram for the service queue system modeled without activities

The process starts with a `CustomerArrival` event. In ER/AOR simulation, events may trigger rules that handle them (so, rules can be viewed as *event handlers*). The `CustomerArrival` event triggers the following rule:

```
<EnvironmentRule name="CustomerArrivalRule">
  <WHEN eventType="CustomerArrival" eventVariable="e"/>
  <DO>
    <UPDATE-ENV>
      <Create>
        <Object type="Customer" addToCollection="CustomerQueue">
          <Slot property="arrivalTime">
            <ValueExpr> e.getOccurrenceTime() </ValueExpr>
          </Slot>
        </Object>
      </Create>
    </UPDATE-ENV>
  </DO>
  <IF> ! Global.isServiceDeskBusy() </IF>
  <THEN>
    <UPDATE-ENV>
      <UpdateGlobalVariable name="serviceDeskBusy" value="true"/>
    </UPDATE-ENV>
    <SCHEDULE-EVT>
      <CausedEventExpr eventType="StartService"/>
    </SCHEDULE-EVT>
  </THEN>
</EnvironmentRule>
```

The `StartService` event resulting from the execution of the CustomerArrivalRule triggers the following rule, which schedules an `EndService` event with a delay according to the random duration of the service:

```
<EnvironmentRule name="StartServiceRule">
  <WHEN eventType="StartService"/>
  <SCHEDULE-EVT>
    <CausedEventExpr eventType="EndService">
      <Delay> Global.randomServiceTime() </Delay>
    </CausedEventExpr>
  </SCHEDULE-EVT>
</EnvironmentRule>
```

The global function `randomServiceTime` implements the probability distribution defined for the service duration. It corresponds to the annotation `Delay: U(1,6)` that is attached to the association between the *Schedule end service event* task and the *end service* event, denoting a uniform distribution with lower bound 1 and upper bound 6.

Finally, the `EndService` event resulting from the `StartServiceRule` triggers the following `EndServiceRule`:

```
<EnvironmentRule name="EndServiceRule">
  <WHEN eventType="EndService"/>
  <DO>
    <UPDATE-ENV>
      <RemoveObjectFromCollection collectionName="CustomerQueue" itemObjectVariable="o"/>
      <DestroyObject objectType="Customer" objectVariable="o"/>
    </UPDATE-ENV>
  </DO>
  <FOR objectType="Collection" objectVariable="q" objectName="CustomerQueue"/>
  <IF> q.size() == 0 </IF>
  <THEN>
    <UPDATE-ENV>
      <UpdateGlobalVariable name="serviceDeskBusy" value="false"/>
    </UPDATE-ENV>
  </THEN>
  <ELSE>
    <SCHEDULE-EVT>
      <CausedEventExpr eventType="EndService">
        <Delay> Global.randomServiceTime() </Delay>
      </CausedEventExpr>
    </SCHEDULE-EVT>
  </ELSE>
</EnvironmentRule>
```

Clearly, the service provision in this system is most naturally viewed as an activity, which is modeled implicitly in the BPMN diagram in Figure 6 by including a service start event and a service end event. After introducing an activity concept to the ER/AOR simulation language in the next section we will be able to model this service provision activity explicitly.

3 EXTENDING THE ER/AOR SIMULATION LANGUAGE BY ADDING ACTIVITY TYPES

For using activities in an ER/AOR simulation model, suitable activity types have to be defined as special `EntityTypes`. As shown in Figure 7, an `ActivityType` may define a start event type such that any event of that type triggers the creation of a new activity instance. If an `ActivityType` does not specify a start event type, then new activity instances may be created on the occurrence of an `ActivityStartEvent` with the `activityType` attribute set to the name of the activity type. (see Figure 1).

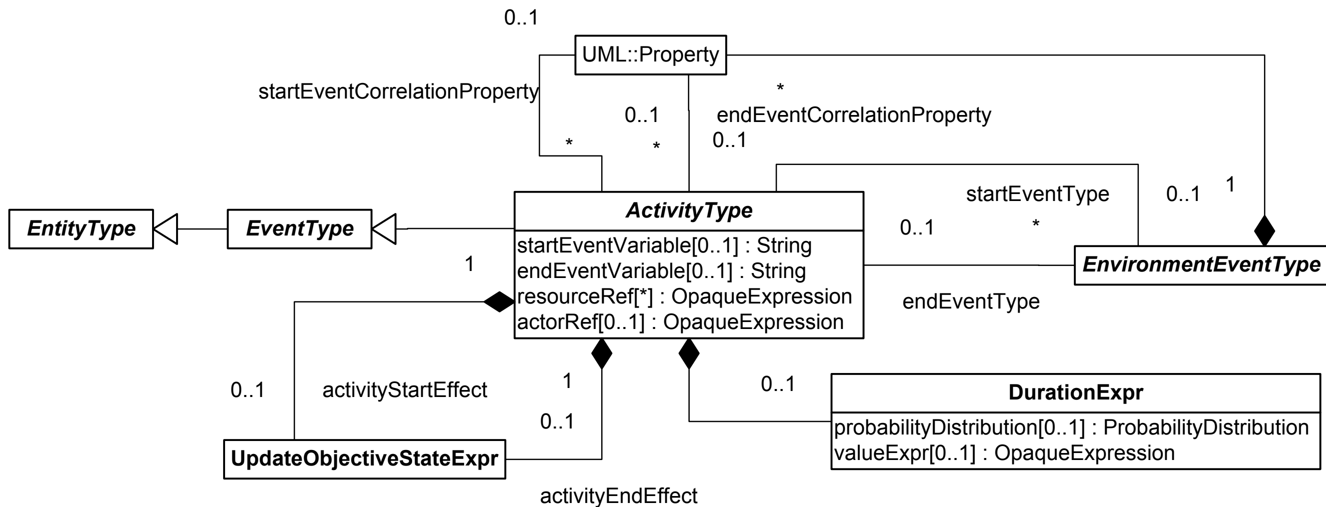

Figure 7: The ActivityType element

As defined in the conceptual UML class diagram shown in Figure 1, activities are complex events having a start event, an end event, an optional association with an actor (being an agent that plays the role of a special resource) and zero or more as-

sociations with other objects playing the role of further resources used by the activity. Such an activity concept has two main benefits:

1. It binds the two events "start activity" and "end activity" implicitly together in a new construct that replaces them. This has the advantage of decreasing the complexity of the model: the event constructs and the four sequence flow arrows attached to them are being replaced by one activity construct and the two sequence flow arrows attached to it. This means that six modeling elements are replaced by three.
2. Associating the resources used with the activity allows the simulator to automatically compute the utilization statistics for these resource objects.

We allow for activity types to define either a pre-set activity duration (constant or random), or an end event type such that any event of that type triggers the completion of the activity. A pre-set random duration is specified with the help of a probability distribution and corresponding parameters. It will be used on the creation of a new activity instance for scheduling an `ActivityEndEvent`.

If an `ActivityType` is defined with a *start event type* and an *end event type*, it is necessary to correlate the end event with the start event by defining a correlation property both for the start event type and the end event type. The value of the *start event correlation property* provides an identifier (relative to the start event type name) for the activity started by the start event. When an event of the end event type occurs, the simulator has to compare the value of the *end event correlation property* with this identifier in order to determine if this event terminates the activity.

An `ActivityType` definition may specify an *activity start effect* and an *activity end effect* expression corresponding to the UPDATE-ENV element of an environment rule. This allows, for instance, to set and reset global flags on the start and end of an activity.

An `ActivityType` definition may specify, with the help of the `resourceRef` attribute, in the form of opaque expressions (typically Java expressions) which resources are allocated to an activity. These expressions will normally use the `startEventVariable` for referring to objects from the context of the activity start event.

4 MODELING A SERVICE QUEUE SYSTEM WITH AN ACTIVITY

In this section we show how to model the service queue system with the help of an activity. The two caused event types `StartService` and `EndService` are now implicitly subsumed (and, hence, replaced) by the activity type `PerformService`, which is defined in the following way:

```
<ActivityType name="PerformService">
  <Duration probabilityDistribution="Uniform">
    <LowerBound value="1"/>
    <UpperBound value="6"/>
  </Duration>
  <ActivityStartEffect>
    <UpdateGlobalVariable name="serviceDeskBusy" value="true"/>
  </ActivityStartEffect>
  <ActivityEndEffect>
    <UpdateGlobalVariable name="serviceDeskBusy" value="false"/>
  </ActivityEndEffect>
</ActivityType>
```

Notice that the activity type `PerformService` is defined as a random duration activity with probability distribution U(1,6). Whenever an activity of that type is started, the global variable `serviceDeskBusy` is set to true, and it is reset to false, when the activity completes.

The new process model of the service queue system, with the activity type `PerformService` is shown in Figure 8. As before, whenever a customer arrival event occurs, a new customer object is created and added to the queue; then, if the service desk is not busy, a `PerformService` activity is started, involving the first customer from the queue; in both cases control returns to the DES loop waiting for the next event, which may be another customer arrival or the implicit `PerformService` end event.

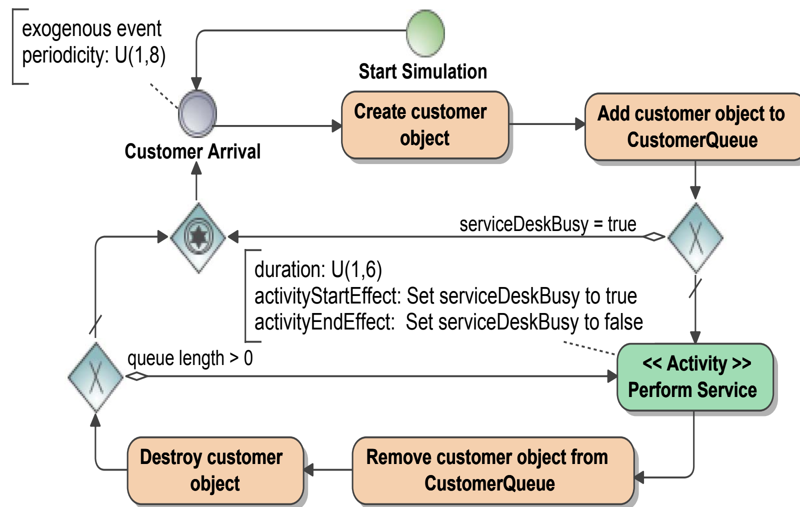

Figure 8: The BPMN diagram for the service queue system modeled with an activity

As described before in section 2, the process starts with a `CustomerArrival` event. that triggers the following rule:

```
<EnvironmentRule name="CustomerArrivalRule">
  <WHEN eventType="CustomerArrival" eventVariable="e"/>
  <DO>
    <UPDATE-ENV>
      <Create>
        <Object type="Customer" addToCollection="CustomerQueue">
          <Slot property="arrivalTime">
            <ValueExpr> e.getOccurrenceTime() </ValueExpr>
          </Slot>
        </Object>
      </Create>
    </UPDATE-ENV>
  </DO>
  <IF> ! Global.isServiceDeskBusy() </IF>
  <SCHEDULE-EVT>
    <ActivityStartEventExpr activityType="PerformService"/>
  </SCHEDULE-EVT>
</EnvironmentRule>
```

Notice how the `CustomerArrivalRule` has been simplified: it does no longer have to take care for updating the global variable `serviceDeskBusy` as this is done implicitly now through invoking the `PerformService` activity in the `SCHEDULE-EVT` element via creating a pre-defined `ActivityStartEvent`.

Finally, the `EndService` event resulting from the `StartServiceRule` triggers the following `EndServiceRule`:

```
<EnvironmentRule name="EndServiceRule">
  <WHEN eventType="ActivityEndEvent" activityType="PerformService"/>
  <UPDATE-ENV>
    <RemoveObjectFromCollection collectionName="CustomerQueue" itemObjectVariable="o"/>
    <DestroyObject objectType="Customer" objectVariable="o"/>
  </UPDATE-ENV>
  <FOR objectType="Collection" objectVariable="q" objectName="CustomerQueue"/>
  <IF> q.size() > 0 </IF>
  <SCHEDULE-EVT>
    <ActivityStartEventExpr activityType="PerformService"/>
  </SCHEDULE-EVT>
</EnvironmentRule>
```

For inspecting the complete model, please check out the following link: <http://oxygen.informatik.tu-cottbus.de/aors/examples/Management/ServiceQueue_withActivity/scenario.xml>.

5 MODELING A DRIVE-THRU RESTAURANT WITHOUT AND WITH AGENTS AND ACTIVITIES

In this section we illustrate the gain obtained by using the higher-level modeling constructs of *agents* and *activities* in combination. Like for higher-level programming constructs, they are not really needed, but they help a lot to make a model more well-structured and more readable. Using agents in DES corresponds to using pools (and lanes) in BPMN. Since the use of pools (and lanes) greatly enhances the readability of a BPMN model, this suggests that using agents in DES also leads to better models.

For our comparison, we are using the drive-through restaurant example from (Ingalls 2008). The scenario is described as follows. As a car enters from the street, the driver, who is called Fred, decides whether or not to get in line. If Fred decides to leave the restaurant, he leaves as a dissatisfied customer. If Fred decides to get in line, then he waits until the menu board is available. At that time, Fred gives the order to the order taker. After the order is taken, then two things occur simultaneously:

- Fred moves forward if there is room, otherwise he has to wait at the menu board he can move forward.
- The order is sent electronically back to the kitchen where it is prepared as soon as the cook is available.

As soon as Fred reaches the pickup window, then he pays and picks up his food, if it is ready. If the food is not ready, then Fred has to wait until his order is prepared.

We present the model without agents and activities in Figure 9, providing its code at the download link <http://oxygen.informatik.tu-cottbus.de/aors/examples/Management/DriveThruRestaurant-withoutAgentsAndActivities/scenario.xml>.

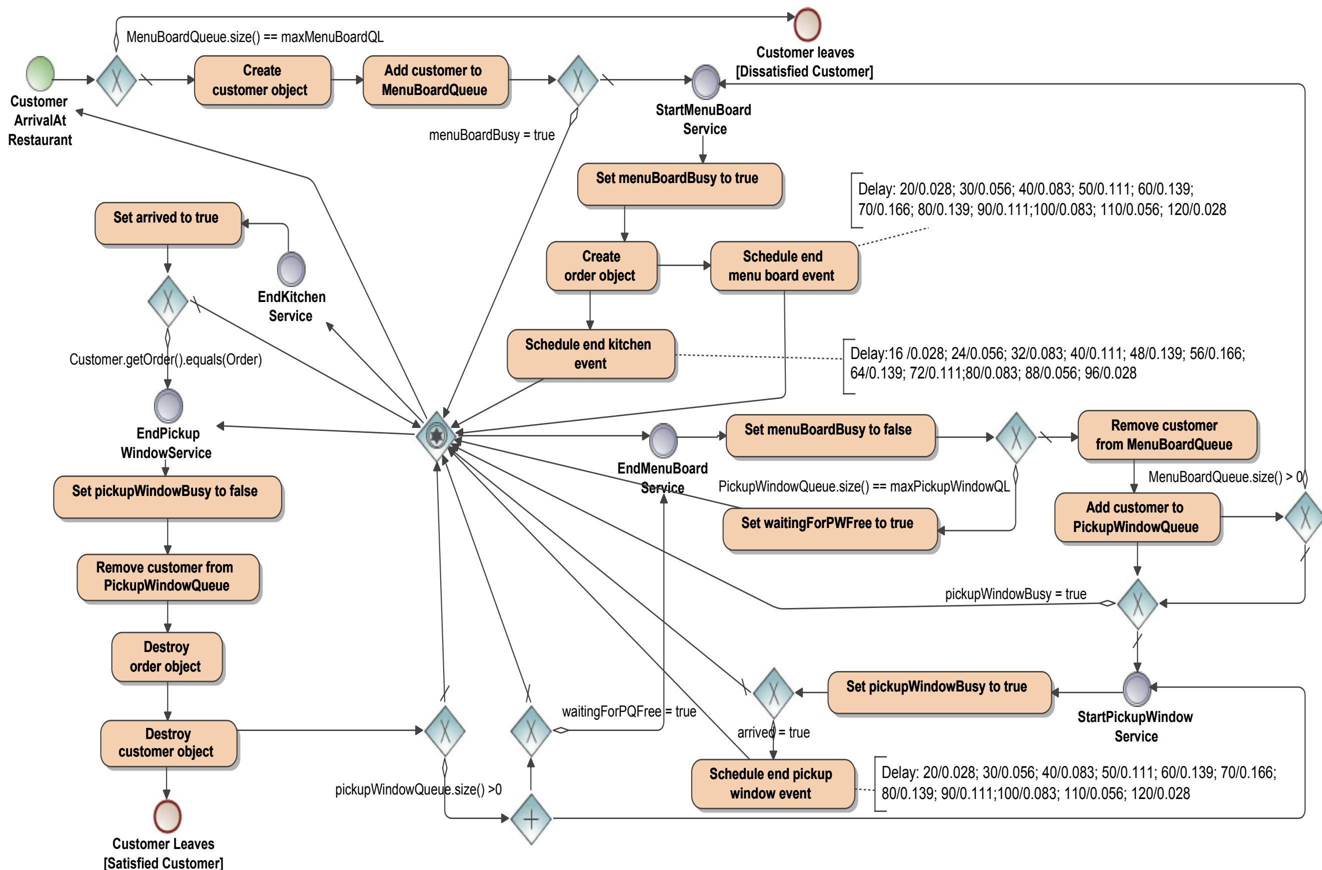

Figure 9: The BPMN diagram for the drive-thru restaurant modeled *without* using agents and activities

We present the model with agents and activities in Figure 10, providing its code at the download link <http://oxygen.informatik.tu-cottbus.de/aors/examples/Management/DriveThruRestaurant-withAgentsAndActivities/scenario.xml>. In Table 1, we compare the two models in terms of their complexity measured by the number of model elements. The first BPMN model does neither use any «Activity» tasks, nor any pools (representing agents) nor any message flows (representing inter-agent communication).

Table 1: Comparing the complexity of both models

BPMN model element	*Model element count* ***without*** *agents and activities*	*Model element count* ***with*** *agents and activities*
Events	8	7
Gateways	11	3
Tasks	17	16
«Activities»		3
Pools and lanes		5
Sequence flows	49	28
Message flows		6
total	**85**	**68**

One gain in this example is a 20% decrease in complexity by using the higher-level modeling constructs of agents and activities. Another gain is the increased structural clarity achieved by encapsulating all agent-related elements in an agent pool.

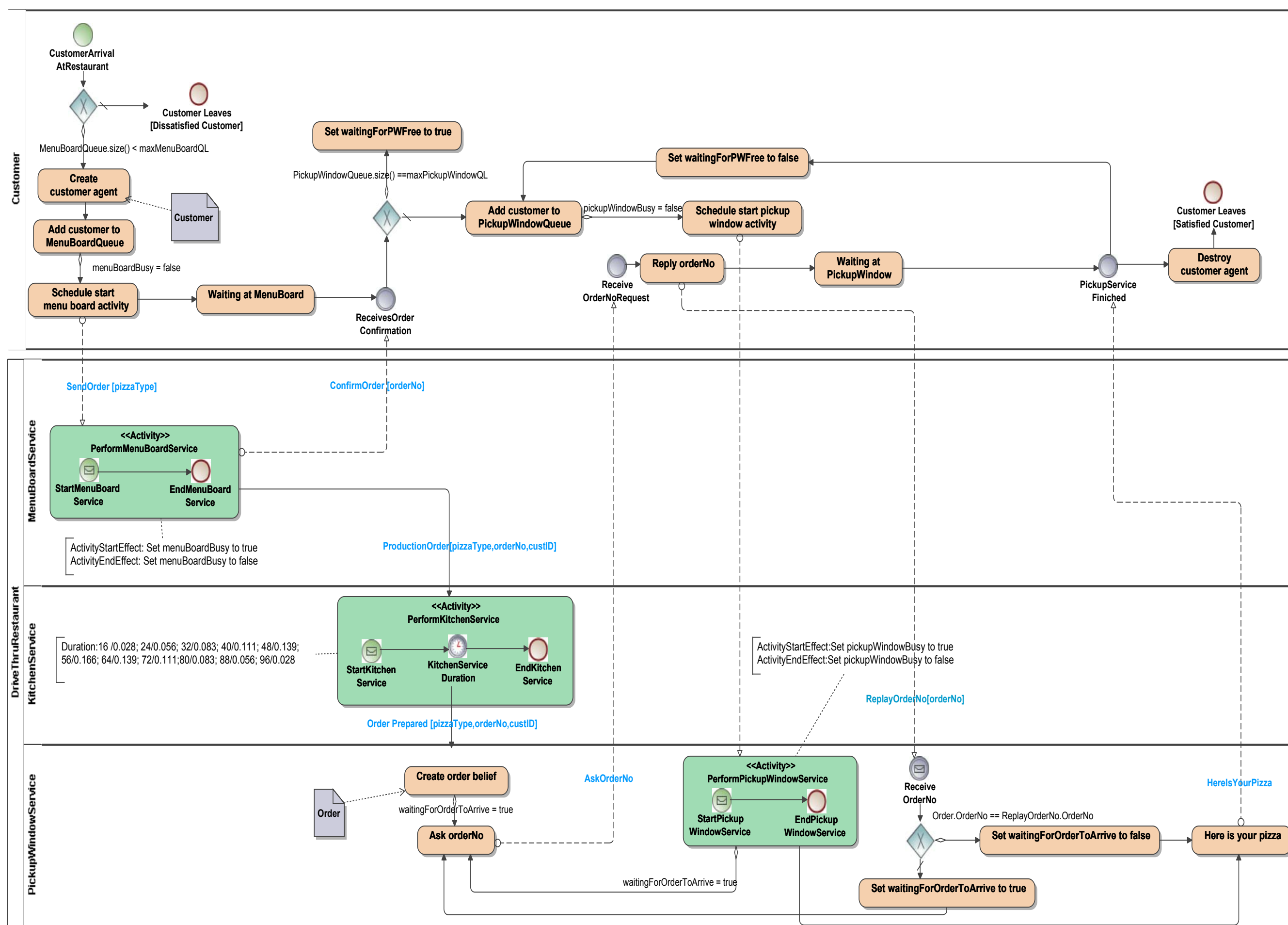

Figure 10: The BPMN diagram for the drive-thru restaurant modeled with agents and activities

6 MERITS AND SHORTCOMINGS OF BPMN AS A SIMULATION MODELING LANGUAGE

BPMN is very well able to represent the event flow dynamics of ER/AOR simulation models. In particular, it supports the expression of environment rules, which are triggered by events, using a data-based XOR gateway for branching to the `IF` and

ELSE parts. For representing a simulation activity, we have used a stereotyped task rectangle (with a green background color).

There are two shortcomings of BPMN with regard to simulation modeling, however. For representing the main loop of DES, which takes care of processing the next event, we use an event-based gateway. However, such a gateway is defined to express a deferred *exclusive* choice in BPMN, while in DES we need to be able to express a deferred *inclusive* choice with this gateway in order to allow for the possibly parallel occurrence of two or more events. For indicating our departure from the standard BPMN semantics for these event-based gateways, we could stereotype the gateway as *"inclusive event-based gateway"*. Moreover, we impose a restriction on the event-based inclusive gateway: it may be followed only by intermediate events, while in standard BPMN an event-based gateway may be followed by *receive tasks*.

Another departure from standard BPMN is created by our usage of a BPMN 'association' between a SCHEDULE-EVT task and the corresponding scheduled event. We annotate this 'association' with the scheduling delay (typically expressed in the form of a probability distribution). Standard BPMN does not support such an association.

7 CONCLUSIONS

We have shown how to extend a discrete event simulation (DES) language by adding a construct for modeling activities having a start event and an end event. Activities can be used both in basic DES and in agent-based DES. Using activities in DES has at least two advantages: it helps reducing the complexity of a simulation model and it allows to automatically compute utilization statistics for resources used by an activity.

We have also presented our preliminary results about how to use BPMN for the purpose of simulation modeling. In this ongoing work we attempt to identify the minimal changes and extensions of BPMN needed for making DES models. We also pursue the goal of identifying the BPMN patterns that are useful for obtaining sound and complete simulation models. As a side effect of this work, we expect to obtain an operational semantics for a large class of BPMN models.

REFERENCES

Banks, J. Carson, J.S. Nelson, B.L. and Nicol, D.M. 2005. *Discrete-Event System Simulation*. Pearson Prentice Hall.

Bock, C. 2008. Tutorial: Introduction to the Business Process Definition Metamodel. <http://www.omg.org/cgi-bin/doc?omg/08-06-32>

Clancey, W.J. 2002. Simulating Activities: Relating Motives, Deliberation, and Attentive Coordination. *Cognitive Systems Research*, Vol. 3, 471–499.

Ingalls, R.G. 2008. Introduction to Simulation. In Mason, S.J. Hill, R.R. Mönch, L. Rose, O. Jefferson, T. Fowler, J.W. (Eds.), *Proceedings of the 2008 Winter Simulation Conference*, 17–26.

Fishman, G.S. 2001. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, Berlin.

Sierhuis, M. 2001. Modeling and Simulating Work Practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design, Ph.D. thesis, Social Science and Informatics (SWI), University of Amsterdam, SIKS Dissertation Series No. 2001-10, Amsterdam, The Netherlands, ISBN 90-6464-849-2.

Wagner, G. 2004. AOR Modeling and Simulation – Towards a General Architecture for Agent-Based Discrete Event Simultion. In *Agent-Oriented Information Systems*, Lecture Notes in AI, volume 3030, 174–188. Springer-Verlag.

AUTHOR BIOGRAPHIES

GERD WAGNER is Professor of Internet Technology within the Department of Informatics, Brandenburg University of Technology. His research interests include agent-oriented modeling and agent-based simulation, foundational ontologies, (business) rule technologies and the Semantic Web. In recent years, he has been focusing his research on the development of an agent-based discrete event simulation framework, called *AOR Simulation*. He can be reached at <http://www.informatik.tu-cottbus.de/~gwagner/>.

OANA NICOLAE is currently a PhD student within the Department of Informatics, Brandenburg University of Technology Cottbus, Germany. Her research interests comprise: Modeling and Simulation of Business Processes, Orchestrations and Choreographies of Business Processes, Conceptual Modeling with BPMN / BPDM / UML.

She can be reached at <http://oxygen.informatik.tu-cottbus.de/~nicolae/>.

JENS WERNER is currently a MSc student within the Department of Informatics, Brandenburg University of Technology Cottbus, Germany. He is one of the main developers of the agent-based discrete event simulation framework *AORS*.

His email address is <wernejen@tu-cottbus.de>.