Information and Process Modeling for Simulation – Part I: Objects and Events

Gerd Wagner
G.Wagner@b-tu.de
Dept. of Informatics, Brandenburg University of Technology, Cottbus, GERMANY

ACM Subject Categories

- `Computing methodologies~Modeling methodologies`

Keywords

Conceptual Model, Design Model, Information Model, Process Model

Abstract

In simulation engineering, a system model mainly consists of an *information model* describing a system's state structure and a *process model* describing its dynamics, including its state changes. In the fields of *Information Systems* and *Software Engineering* (IS/SE) there are widely used standards such as the *Class Diagrams* of the *Unified Modeling Language (UML)* for making information models, and the *Business Process Modeling Notation (BPMN)* for making process models. This tutorial presents a general approach, called *Object-Event Modeling (OEM)*, for Discrete Event Simulation modeling using UML class diagrams and BPMN-based process diagrams at all three levels of *model-driven simulation engineering*: for making conceptual domain models, for making platform-independent simulation design models, and for making platform-specific, executable simulation models. In our approach, object and event types are modeled as special categories of UML classes, random variables are modeled as a special category of UML operations constrained to comply with a specific probability distribution, and queues are modeled as ordered association ends, while *event rules* are modeled both as BPMN-based process diagrams and in pseudo-code. In Part II, we will discuss the more advanced OEM concepts of *activities* and GPSS/SIMAN/Arena-style *Processing Networks*, while in Part III we will further extend the OEM paradigm towards agent-based modeling and simulation by adding the concepts of *agents* with *perceptions*, *actions* and *beliefs*.

1 Introduction

The term *simulation engineering* denotes the scientific engineering discipline concerned with the development of computer simulations, which are a special class of software applications. Since a running computer simulation is a particular kind of software system, we may consider simulation engineering as a special case of *software engineering*.

Even though there is a common agreement that modeling is an important first step in a simulation project, at the same time it is thought to be the least understood part of simulation engineering (Tako, Kotiadis, & Vasilakis, 2010). In a panel discussion on conceptual simulation modeling (Zee, et al., 2010), the participants agreed that there is a lack of "standards, on procedures, notation, and model qualities". On the other hand, there is no such lack in the field of *Information Systems and Software Engineering (IS/SE)* where widely used standards such as the *Unified Modeling Language (UML)* and the *Business Process Modeling Notation (BPMN)* and various modeling methodologies and model quality assurance methods have been established.

The standard view in the simulation literature, see, e.g., (Himmelspach, 2009), is that a *simulation model* can be expressed either in a general purpose programming language or in a specialized simulation language. This means that the term "model" in *simulation model* typically refers to a low-level computer program and not to a model expressed in a higher-level diagrammatic modeling language. In a *modeling and simulation* project, despite the fact that "modeling" is part of the discipline's name, often no model in the sense of a conceptual model or a design model is made, but rather the modeler jumps from her mental model to its implementation in some target technology platform. Clearly, as in IS/SE, making conceptual models and design models would be important for several reasons: as opposed to a low-level computer program, a high-level model would be more comprehensible and easier to communicate, share, reuse, maintain and evolve, while it could still be used for obtaining platform-specific implementation code, possibly with the help of *model transformations* and *code generation*.

Due to their great expressivity and their wide adoption as modeling standards, *UML* and *BPMN* seem to be the best choices for making information and process models in a model-based simulation engineering approach. However, since they have not been designed for this purpose, we may have to restrict, modify and extend them in a suitable way.

Several authors, e.g. (Wagner, Nicolae, & Werner, 2009), (Cetinkaya, Verbraeck, & Seck, 2011), and (Onggo & Karpat, 2011), have proposed to use BPMN for Discrete Event Simulation (DES) modeling and for agent-based modeling. However, process modeling in general is much less understood than information modeling, and there are no guidelines and no best practices how to use BPMN for simulation modeling. Schruben (1983), with his *Event Graph* diagram language, has pioneered the research on process modeling languages for DES based on the modeling concept of *event types* and the operational semantics concept of *event scheduling* with a *future events list*. Remarkably, Event Graphs correspond to a fragment of BPMN (without Activities and Pools), which indicates the potential of BPMN as a basis of a general process modeling language for DES.

This tutorial article extends and improves the modeling approach presented in (Wagner, 2017b). In particular, the BPMN-based process design modeling approach has been revised and refined, inspired by the *Event Graph* diagrams of (Schruben 1983). The resulting variant of BPMN, called *Discrete Event Process Modeling Notation (DPMN)*, allows making computational (platform-independent) process design models for DES.

The tutorial first provides short introductions to *model-driven engineering*, to *information modeling* with UML class diagrams, and to *process modeling* with BPMN process diagrams, and then shows how to use the approach of *Object-Event Modeling (OEM)* for developing basic discrete event simulations. In a forthcoming Part II of this tutorial, we will discuss the more advanced modeling concepts of *activities* and GPSS/SIMAN/Arena-style *Processing Networks* where work objects "flow through the system" by entering it through an *arrival event* at an *entry node*, then passing one or more *processing nodes*, participating in their processing activities, and finally leaving it through a *departure event* at an *exit node*. Finally, in Part III, we will show how to add the modeling concepts of *agents* with *perceptions*, *actions* and *beliefs* resulting in a general agent-based DES modeling framework.

In our OEM approach, *object types* and *event types* are first modeled in an information model for describing the state structure of the system under investigation, and then used 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. *Queues* are not modeled as objects, but rather as ordered association ends, which can be implemented as collection-valued reference properties. Finally, *event rules*, which include *event routines*, are modeled both as BPMN process diagrams and in pseudo-code such that they can be implemented in the form of special *onEvent* methods of event classes.

The OEM approach results in a simulation design model that has a well-defined operational semantics, as shown in (Wagner, 2017a). Such a model can, in principle, be implemented with any OO simulation technology. However, a straightforward implementation can only be expected from a technology that implements the *Object-Event Simulation (OES)* paradigm proposed in (Wagner, 2017a), such as the *OES JavaScript (OESjs)* framework presented in (Wagner, 2017c).

There are two examples of systems, which are paradigmatic for DES (and for *operations research*): *service/processing systems* with queues (also called "queuing networks") and *inventory management systems*. However, neither of them has yet been presented with elaborate information and process models in tutorials or textbooks. In this tutorial, we show how to make information and process models of an inventory management system and of a service system, and how to code them using the JavaScript-based simulation framework OESjs.

2 Information Modeling with UML Class Diagrams

Conceptual information modeling is mainly concerned with describing the relevant ***entity types*** of a real-world domain and the relationships between them, while information design and implementation modeling is concerned with describing the *logical* (or *platform-independent*) and *platform-specific* data structures (in the form of ***classes***) for designing and implementing a software system or simulation. The most important kinds of relationships between entity types to be described in an information model are ***associations***, which are called "relationship types" in *ER modeling*, and ***subtype***/*supertype* relationships, which are called "generalizations" in *UML*. In addition, one may model various kinds of *part-whole* relationships between different kinds of aggregate entities and component entities, but this is an advanced topic that is not covered in this tutorial.

As explained in the introduction, we are using the visual modeling language of UML Class Diagrams for information modeling. In this language, an entity type is described with a name, and possibly with a list of ***properties*** and ***operations*** (called *methods* when implemented), in the form of a *class rectangle* with one, two or three compartments, depending on the presence of properties and operations. ***Integrity constraints***, which are conditions that must be satisfied by the instances of a type, can be expressed in special ways when defining properties or they can be explicitly attached to an entity type in the form of an *invariant* box.

An ***association*** between two entity types is expressed as a connection line between the two class rec-

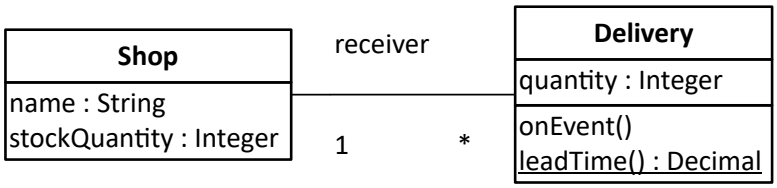

Figure 4. Adding properties and operations.

tangles representing the entity types. The connection line is annotated with *multiplicity* expressions at both ends. A ***multiplicity*** expression has the form `m..n` where *m* is a non-negative natural number denoting the *minimum cardinality*, and *n* is a positive natural number (or the special symbol * standing for *unbounded)* denoting the maximum cardinality, of the sets of associated entities. Typically, a multiplicity expression states an integrity constraint. For instance, the multiplicity expression `1..3` means that there are at least 1 and at most 3 associated entities. However, the special multiplicity expression `0..*` (also expressed as `*`) means that there is no constraint since the minimum cardinality is zero and the maximum cardinality is unbounded.

For instance, the model shown in Figure 3 describes the entity types `Shop` and `Delivery`, and it states that

1. there are two classes: `Shop` and `Delivery`, representing entity types;
2. there is a one-to-many association between the classes `Shop` and `Delivery`, where a shop is the `receiver` of a delivery.

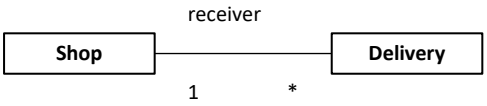

Figure 3. The entity types Shop and Delivery.

Using further compartments in class rectangles, we can add properties and operations. For instance, in the model shown in Figure 4, we have added

1. the properties *name* and *stockQuantity* to `Shop` and *quantity* to `Delivery`,
2. the instance-level operation *onEvent* to `Delivery`,
3. the class-level operation *leadTime* to `Delivery`.

Notice that in Figure 4, each property is declared together with a datatype as its *range*. Likewise, operations are declared with a (possibly empty) list of parameters, and with an optional return value type. When an operation (or property) declaration is underlined, this means that it is class-level instead of instance-level. For instance, the underlined operation declaration `leadTime() : Decimal` indicates that *leadTime* is a class-level operation that does not take any argument and returns a decimal number.

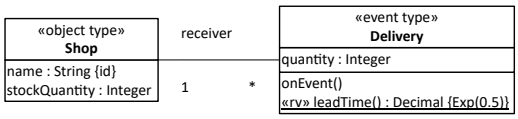

Figure 6. Object and event types as two different categories of entity types.

We may want to define various types of integrity constraints for better capturing the semantics of entity types, properties and operations. The model shown in Figure 5 contains an example of a property constraint and an example of an operation constraint. These types of constraints can be expressed within curly braces appended to a property or operation declaration. The keyword `id` in the declaration of the property `name` in the `Shop` class expresses an ID constraint stating that the property is a standard identifier, or primary key, attribute. The expression `Exp(0.5)` in the declaration of the random variable operation `leadTime` in the `Delivery` class denotes the constraint that the operation must implement the *exponential* probability distribution function with event rate 0.5.

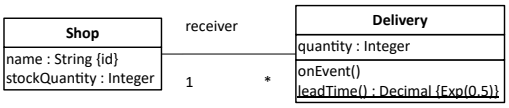

Figure 5. Adding a property constraint and an operation constraint.

UML allows defining special categories of modeling elements called “stereotypes”. For instance, for distinguishing between ***object types*** and ***event types*** as two different categories of entity types we can define corresponding stereotypes of UML classes («object type» and «event type») and use them for categorizing classes in class models, as shown in Figure 6.

Another example of using UML’s stereotype feature is the designation of an operation as a function that represents a *random variable* using the operation stereotype «rv» in the diagram of Figure 6.

A class may be defined as *abstract* by writing its name in italics, as in the example model of Figure 11. An abstract class cannot have direct instances. It can only be indirectly instantiated by objects that are direct in-

stances of a subclass.

For a short introduction to UML Class Diagrams, the reader is referred to (Ambler, 2010). A good overview of the most recent version of UML (2.5) is provided by www.uml-diagrams.org/uml-25-diagrams.html