Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

INTRODUCTION TO INFORMATION AND PROCESS MODELING FOR SIMULATION

Gerd Wagner

Brandenburg University of Technology
Department of Informatics
P. O. Box 101344
03013 Cottbus, GERMANY

ABSTRACT

In simulation engineering, a system model mainly consists of an *information model* and a *process model*. 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 how to use UML class diagrams and BPMN process diagrams at all three levels of *model-driven simulation engineering*: for making conceptual simulation 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 stereotyped classes and random variables are modeled as stereotyped operations constrained to comply with a specific probability distribution, while event rules/routines are modeled both as BPMN patterns and in pseudo-code.

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*.

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 specifically designed for this purpose, we may have to restrict, modify and extend them in a suitable way.

Several authors, e.g. Wagner et al. (2009) and Onggo and Karpat (2011), have proposed to use BPMN for discrete event simulation 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.

In this tutorial, we provide short introductions to *model-driven engineering*, to *information modeling* with UML class diagrams, and to *process modeling* with BPMN diagrams, and then show how to use our general model-based simulation engineering approach for developing a simulation of an inventory management system. In our approach, object and event types are modeled as stereotyped classes that can be implemented with any object-oriented programming language or simulation library/framework. Random variables are modeled as stereotyped operations constrained to comply with a specific probability distribution, and event rules/routines are modeled both as BPMN patterns and in pseudo-code.

An extended version of this tutorial is available as (Wagner 2017).

2 WHAT IS DISCRETE EVENT SIMULATION?

The term *Discrete Event Simulation (DES)* has been established as an umbrella term subsuming various kinds of computer simulation approaches, all based on the general idea of modeling entities/objects and events. In the DES literature, it is often stated that DES is based on the concept of "entities flowing through the system" (more precisely, through a "queueing network"). E.g., this is the paradigm of an entire class of simulation software in the tradition of GPSS (Gordon 1961) and SIMAN/Arena (Pegden and Davis 1992). However, this paradigm characterizes a special (yet important) class of DES only, it does not apply to all discrete dynamic systems.

In *Ontology*, which is the philosophical study of what there is, the following fundamental distinctions are made:

- there are ***entities*** (also called *individuals*) and ***entity types*** (called 'universals' in philosophy);
- there are three fundamental categories of entities:
  1. ***objects***,
  2. *tropes,* which are existentially dependent entities such as the *qualities* and *dispositions* of objects and their *relationships* with each other, and
  3. ***events***.

These ontological distinctions are discussed, e.g., in Guizzardi and Wagner (2010, 2013).

While the concept of an event is often limited to instantaneous events in the area of DES, the general concept of an event, as discussed in philosophy and in many fields of computer science, includes composite events and events with non-zero duration.

A *discrete event system* (or *discrete dynamic system*) consists of

- ***objects*** (of various types),
- ***events*** (of various types),
- ***dispositions*** of objects triggered by events,

such that the states of objects may be changed by events occurring at times from a discrete set of time points, according to the dispositions of the objects triggered by the events.

For modeling a discrete event system as a state transition system, we have to describe its

1. ***object types***, e.g., in the form of *classes* of an object-oriented language;
2. ***event types***, e.g., in the form of *classes* of an object-oriented language;
3. ***causal regularities*** (*disposition types*) e.g., in the form of *event rules*.

Any *discrete event simulation* (DES) formalism has one or more language elements allowing to specify, at least implicitly, ***event rules*** that allow representing causal regularities. These rules specify for any event type, the ***state changes*** of objects and the ***follow-up events*** caused by the occurrence of an event of that type, thus defining the dynamics of the transition system. Unfortunately, this is often obscured by the standard definitions of DES that are repeatedly presented in simulation textbooks and tutorials. It is common to call the different computational paradigms, on which simulation languages and systems are based, "worldviews".

According to Pegden (2010), a *simulation modeling worldview* provides "a framework for defining a system in sufficient detail that it can be executed to simulate the behavior of the system". It "must precisely define the dynamic state transitions that occur over time". Pegden continues saying that "Over the 50 year history of simulation there has been three distinct world views in use: event, process, and objects":

Event worldview: The system is viewed as a series of instantaneous events that change the state of the system over time. The modeler defines the events in the system and models the state changes that take place when those events occur. According to Pegden, the event worldview is the most fundamental worldview since the other worldviews also use events, at least implicitly.

Processing network worldview: The system under investigation is described as a processing network where "entities flow through the system" (or, more precisely, work objects are routed through the network) and are subject to a series of processing steps performed at processing nodes through processing activities, possibly requiring resources and inducing queues of work objects waiting for the availability of resources (processing networks have been called "queueing networks" in Operations Research). This approach allows high-level modeling with semi-visual languages and is therefore the most widely used DES approach nowadays, in particular in manufacturing industries and service industries. Simulation platforms based on this worldview may or may not support object-oriented modeling and programming.

Object worldview: The system is modeled by describing the objects that make up the system. The system behavior emerges from the "interaction" of these objects.

All three worldviews, and especially the process and object worldviews, which dominate today's simulation landscape, lack important conceptual elements. The event worldview does not support modeling objects with their (categorical and dispositional) properties. The process worldview neither supports modeling events nor objects. And the object worldview, while it supports modeling objects with their *categorical* properties, does not support modeling events. None of the three worldviews does support modeling the *dispositional* properties of objects with a full-fledged explicit concept of *event rules*.

The event worldview and the object worldview can be combined in approaches that support both objects and events as first-class citizens. This seems highly desirable because (1) objects (and classes) are a must-have in today's state-of-the-art modeling and programming, and (2) a general concept of events is fundamental in DES, as demonstrated by the classical event worldview. We use the term ***object-event worldview*** for any DES approach combining object-oriented (OO) modeling and programming with a general concept of events.

3 MODEL-DRIVEN ENGINEERING

Model-Driven Engineering (MDE), also called *model-driven development*, is a well-established paradigm in IS/SE, see, e.g., the *Model-Driven Architecture* proposal of the Object Management Group (MDA 2012). Since simulation engineering can be viewed as a special case of software engineering, it is natural to apply the ideas of MDE also to simulation engineering. There have been several proposals of using an MDE approach in Modeling and Simulation (M&S), see, e.g., the overview given in Cetinkaya and Verbraeck (2011).

In MDE, there is a clear distinction between three kinds of models as engineering artifacts resulting from corresponding activities in the analysis, design and implementation phases:

1. ***domain models*** (also called ***conceptual models***),

2. platform-independent ***design models***,
3. platform-specific ***implementation models***.

Domain models are solution-independent descriptions of a problem domain produced in the analysis phase of a software engineering project. We follow the IS/SE usage of the term 'conceptual model' as a synonym of 'domain model'. However, in the M&S literature there are diverging proposals how to define the term 'conceptual model', see, e.g., (Guizzardi & Wagner 2012) and (Robinson 2013). A domain model may include both descriptions of the domain's state structure (in conceptual *information models*) and descriptions of its processes (in conceptual *process models*). They are solution-independent, or 'computation-independent', in the sense that they are not concerned with making any system design choices or with other computational issues. Rather, they focus on the perspective and language of the subject matter experts for the domain under consideration.

In the design phase, first a platform-independent design model, as a general computational solution, is developed on the basis of the domain model. The same domain model can potentially be used to produce a number of (even radically) different design models. Then, by taking into consideration a number of implementation issues ranging from architectural styles, nonfunctional quality criteria to be maximized (e.g., performance, adaptability) and target technology platforms, one or more platform-specific implementation models are derived from the design model. These one-to-many relationships between conceptual models, design models and implementation models are illustrated in Figure 1.

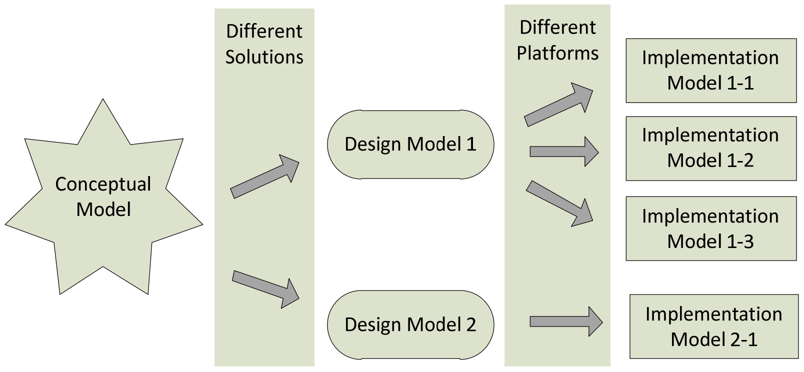


Figure 1: From a conceptual model to design models to implementation models.

In the implementation phase, an implementation model is coded in the programming language of the target platform. Finally, after testing and debugging, the implemented solution is then deployed in a target environment.

A model for a software (or information) system, which may be called a 'software system model', does not consist of just one model diagram including all viewpoints or aspects of the system to be developed (or to be documented). Rather it consists of a set of models, one (or more) for each viewpoint. The two most important viewpoints, crosscutting all three modeling levels: domain, design and implementation, are

1. ***information modeling***, which is concerned with the state structure of the domain;
2. ***process modeling***, which is concerned with the dynamics of the domain.

In the computer science field of database engineering, which is only concerned with information modeling, domain information models have been called 'conceptual models', information design models have been called 'logical design models', and database implementation models have been called 'physical design models'. In the sequel, we call information implementation models *data models* or *class models*. So, from a given information design model, we may derive an SQL data model, a Java class model and a C# class model.

Examples of widely used languages for information modeling are *Entity Relationship (ER) Diagrams* and *UML Class Diagrams*. Since the latter subsume the former, we prefer using UML class diagrams for making all kinds of information models, including SQL database models. Examples of widely used languages for process modeling are *(Colored) Petri Nets*, *UML Sequence Diagrams, UML Activity Diagrams* and the *BPMN*. Notice that there is more agreement on the right concepts for information modeling than for process modeling, as indicated by the much larger number of different process modeling languages. We claim that this reflects a lower degree of understanding the nature of events and processes compared to understanding objects and their relationships.

Some modeling languages, such as UML Class Diagrams and BPMN, can be used on all three modeling levels in the form of tailored variants. Other languages have been designed for being used on one or two of these three levels only. E.g. Petri Nets cannot be used for conceptual process modeling, since they lack the required expressivity.

We illustrate the distinction between the three modeling levels with an example in Figure 2 below. In a simple conceptual information model of people, expressed as a UML class diagram, we require that any person has exactly one mother and one father, expressed by corresponding binary many-to-one associations, while we represent the associations *mother* and *father* with corresponding reference properties in the object-oriented (OO) design model. Also, we may not care about the datatypes of attributes in the conceptual model, while we do care about them in the design model, where we use platform-independent datatype names (such as `Decimal`), and in the Java implementation model, where we use Java-specific datatypes (such as `java.math.BigDecimal`). Finally, in the Java implementation model, we add *get* and *set* methods for all attributes, and we specify the visibility *private* (symbolically -) for attributes and *public* (symbolically +) for methods.

Model-driven simulation engineering is based on the same kinds of models as model-driven software engineering: going from a *domain model* via a *design model* to an *implementation model* for the simulation platform of choice (or to several implementation models if there are several target simulation platforms). The specific concerns of simulation engineering, like, e.g., the concern to capture certain parts of the overall system dynamics with the help of random variables, do not affect the applicability of MDE principles. However, they define requirements for the modeling languages to be used.

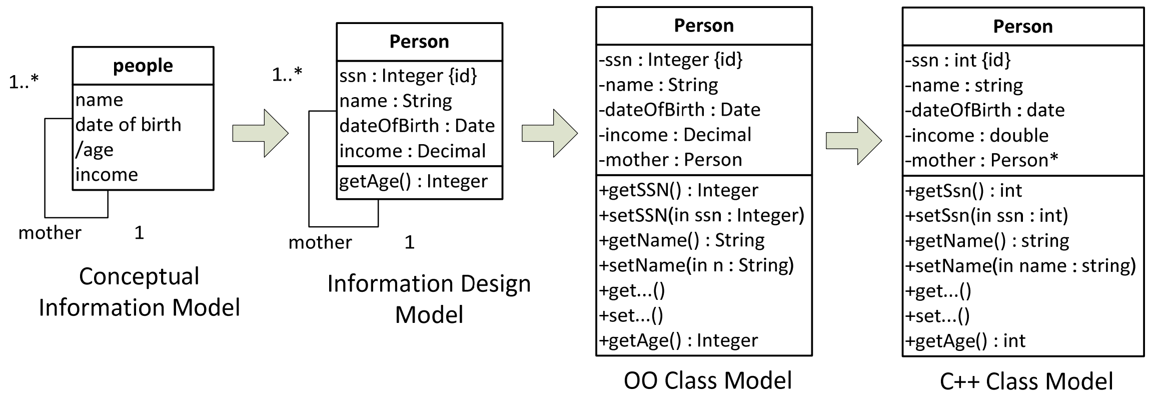


Figure 2: From a conceptual information model via a design model to OO and C++ class models.

4 INFORMATION MODELING WITH UML CLASS DIAGRAMS

Conceptual information modeling is mainly concerned with describing the relevant ***entity types*** of a 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 (called ***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 types and component types, but this is a more advanced topic and cannot be covered in this introductory 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***, 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 rectangles 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 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.

A *subtype* relationship between two entity types is expressed by an arrow with a large arrowhead, as in the example model shown in Figure 3 below, where the entity type `Customer` is a subtype of the entity type `Person`. Generally speaking, when `A` is a subtype of `B`, this means (1) that `A` inherits all properties (and other features) from `B`, and (2) that all instances of `A` are also instances of `B`.

E.g., the model shown in Figure 3 below describes the entity types `Person`, `Customer` and `ServiceQueue`, and states that

1. `Customer` is a subtype of `Person` (and, hence, inherits the property `name`);
2. there is a many-to-one association between `Customer` and `ServiceQueue`, or, more precisely, a service queue (as an entity of type `ServiceQueue`) is associated with any number of entities of type `Customer`, while a customer may be waiting in at most one service queue;

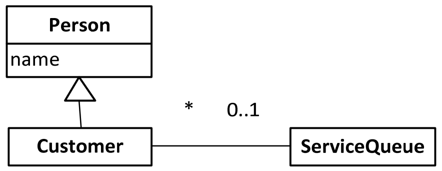


Figure 3: Describing the entity types Person, Customer and ServiceQueue.

UML allows defining special categories (called 'stereotypes') of modeling elements. 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 the model of Figure 4 below, which also describes the event type `GetInLine`. An event of that type involves exactly one customer who gets in line at exactly one service queue (we also say the customer and the service queue *participate* in the event, or: they are its *participants*).

Another example of using UML's stereotype mechanism is the designation of an operation as a function that implements a *random variable* in the diagram of Figure 5, where the operation stereotype «rv» indicates that an operation is categorized as representing a random variable.

The models shown in Figure 3 and 4 are solution-independent conceptual models, which often do not contain attributes, or if they contain attributes, their range (datatype) is typically not specified. Any such conceptual model can be refined into a solution-specific, but platform-independent, design model containing full attribute definitions.

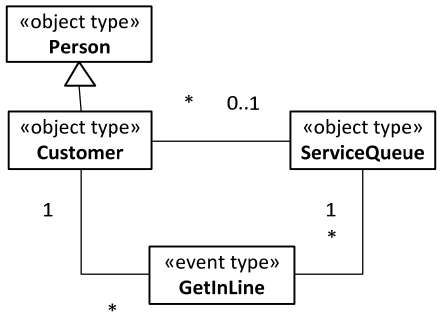


Figure 4: Distinguishing between object types and event types as two different categories of entity types.

The UML also allows defining various types of constraints, which help to capture the semantics of a problem domain. The model shown in Figure 5 contains examples of a property constraint and 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 *serviceDeskNo* 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 *serviceDuration* denotes the constraint that the operation must implement the exponential probability distribution with event rate 0.5.

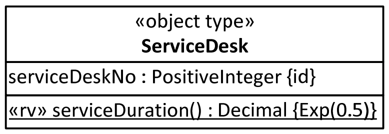

«object type»
ServiceDesk
serviceDeskNo : PositiveInteger {id}
«rv» serviceDuration() : Decimal {Exp(0.5)}

Figure 5: An object type with a property constraint and an operation constraint.

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

5 PROCESS MODELING WITH BPMN

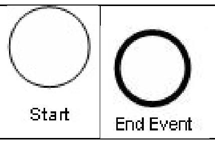
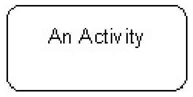
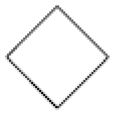
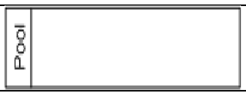
The Business Process Modeling Notation (BPMN) is an activity-based graphical modeling language for defining business processes following the flow-chart metaphor. In 2011, the Object Management Group (OMG) has released version 2.0 of BPMN with a semi-formal *token flow semantics*.

The most important elements of a BPMN process model are listed in Table 1 below.

Ontologically, BPMN activities are special event types. However, the subsumption of activities under events is not supported by the semantics of BPMN. It is one of the issues that require further improvements of BPMN. Another severe issue of the official BPMN (token flow) semantics is its limitation to *case handling* processes. Each start event represents a new case and starts a new process for handling this case in isolation from other cases. This semantics disallows, for instance, to model processes where several cases are handled in parallel and interact in some way, e.g., by competing for resources. Consequently, this semantics is inadequate for capturing the overall process of a business system with many actors performing tasks related to many cases with various interdependencies, in parallel. We do therefore not adopt the official BPMN semantics, but just the visual syntax of BPMN and large parts of the informal semantics of its elements. Defining a more general semantics for BPMN that is adequate for simulation modeling is still an open research issue.

Due to these issues, it is not clear at present how to best use BPMN, and how to adapt its syntax and semantics, for simulation modeling. Our proposal how to use BPMN for simulation modeling is therefore rather experimental and still needs to be evaluated and improved. But we claim that despite these issues, using BPMN as a basis for developing a process simulation modeling approach is the best choice of a modeling language we can make, considering the alternatives, which are either not well-defined (Flow Charts, "Logic Flow Diagrams") or not sufficiently expressive (Petri Nets, UML State Transition Diagrams, UML Activity Diagrams), and which are therefore all inferior compared to BPMN.

Table 1: Basic elements of BPMN.

Name of element	Meaning	Visual symbol(s)
Event	Something that "happens" during the course of a process, affecting the process flow. A ***Start Event*** is drawn as a circle with a thin border line, while an ***Intermediate Event*** has a double border line and an ***End Event*** has a thick border line.	
Activity (Task, Sub-Process)	Work that is performed within a process. A ***Task*** is an atomic Activity, while a ***Sub-Process*** is a composite Activity. A Sub-Process can be either in a *collapsed* or in an *expanded* view. The latter shows its internal process structure.	
Gateway	For controlling how a process flows. A single Gateway could have multiple input and multiple output flows. The plain gateway symbol denotes an ***Exclusive OR-Split*** (if there are 2 or more output flows) or an ***Exclusive OR-Join*** (if there are 2 or more input flows). A gateway with a plus symbol denotes an ***AND-Split*** (if there are 2 or more output flows) or an ***AND-Join*** (if there are 2 or more input flows).	
Sequence Flow	Defines the temporal order of Events, Activities, and Gateways.	
Pool	Represents an agent role (like 'Buyer' or 'Seller') or a specific instance of such a role (like 'Amazon.com").	
Message Flow	Represents a message exchange communication link between two Pools. It's an option to render the message type with a message icon.	

For an introductory BPMN tutorial, the reader is referred to (Camunda). A good modeling tool, with the advantages of an online solution, is the *Signavio Process Editor*, which is free for academic use (www.signavio.com/bpm-academic-initiative).

6 EXAMPLE: MODELING AN INVENTORY MANAGEMENT SYSTEM

There are two examples of systems, which are paradigmatic for DES (and operations research): *service desks* and *inventory management*. Neither of them has yet been presented with elaborate information and process models in simulation tutorials and textbooks. In this section, we use the example of inventory management (see http://sim4edu.com/sims/1/description.html for an elaborate service desk model).

We consider a simple case of inventory management: a shop selling one product type (e.g., one model of TVs), only, such that its in-house inventory only consists of items of that type. On each business day, customers come to the shop and place their orders. If the ordered product quantity is in stock, the customer pays her order and the ordered products are handed out to her. Otherwise, the order may still be partially fulfilled, if there are still some items in stock, else the customer has to leave the shop without any item. If an order quantity is greater than the current stock level, the difference counts as a lost sale.

When the stock quantity falls below the reorder point, a replenishment order is sent to the vendor for restocking the inventory, and the ordered quantity is delivered a few days later.

The purpose of the simulation is to compute the percentage of lost sales, which is an important performance indicator.

6.1 Information Modeling

How should we start the information modeling process? Should we first model object types and then event types, or the other way around? Here, the right order is dictated by existential dependencies. Since ***events existentially depend on the objects that participate in them*** (this is an ontological pattern that is fundamental for DES), we first model object types, together with their associations, and then add event types on top of them.

6.1.1 Making a Solution-Independent Conceptual Information Model in Three Steps

We can extract the following candidates for object types from the problem description above by identifying and analyzing the domain-specific noun phrases: *shops* (for being more precise, we also say *single product shops*), *products* (= items), *inventories*, *customers*, *customer orders*, *replenishment orders*, and *vendors*. Since noun phrases may also denote events (or event types), we need to take another look at our list and drop those noun phrases. We recognize that *customer orders* and *replenishment orders* denote messages or communication events, and not ordinary objects. This leaves us with the five object types described in the diagram shown in Figure 6 below.

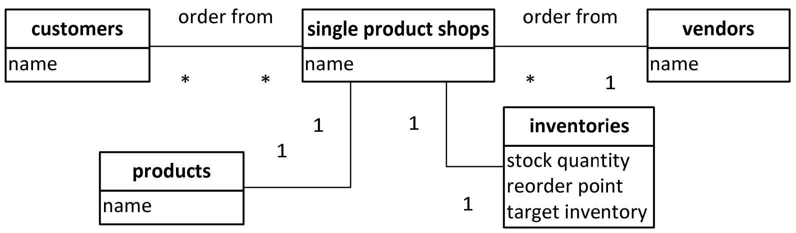


Figure 6: Our first version conceptual information model, describing object types, only.

Notice that we have also modeled the following associations between these five object types:

1. The (named) many-to-many association *customers-**order-from**- shops*.
2. The (un-named) one-to-one association *shops-**have**-products*.
3. The (un-named) one-to-one association *shops-**have**-inventories*.
4. The (named) many-to-one association *shops-**order-from**-vendors*.

The second association is one-to-one because we are assuming that our shops sell only a single product, while the third association is one-to-one because we assume that our shops have only one inventory for their single product.

We have also added some attributes to the object types, such as a *name* attribute for *customers*, *shops*, *products* and *vendors*, and a *reorder point* as well as a *stock quantity* attribute for *inventories*. Some of these attributes can be found in the problem description (such as *reorder point)*, while others have to be inferred by commonsense reasoning (such as *target inventory* for the quantity to which the inventory is to be restocked).

In the next step, we add event types. We have already identified *customer orders* and *replenishment orders* as two potentially relevant event types mentioned as noun phrases in the problem description. We can try to extract the other potentially relevant event types from the text, typically by considering the verb phrases, such as "pay order", "hand out product", and "deliver". For getting the names of our event types, we nominalize these verb phrases. So we get *customer payments*, *product handovers* and *deliveries*. Finally, even if this is not mentioned in the business description above, we know, using common sense, that a delivery by the vendor leads to a corresponding payment by the shop, so we also need a *payments* event type.

So we add these six event types to our model, together with their participation associations with involved object types, now distinguishing class rectangles that denote event types from those denoting object types with the help of UML stereotypes, as shown in Figure 7 below. Notice that a participation association between an object type and an event type is typically one-to-many, since an event of that type has typically exactly one participating object of that type, and, vice versa, an object of that type typically participates in many events of that type.

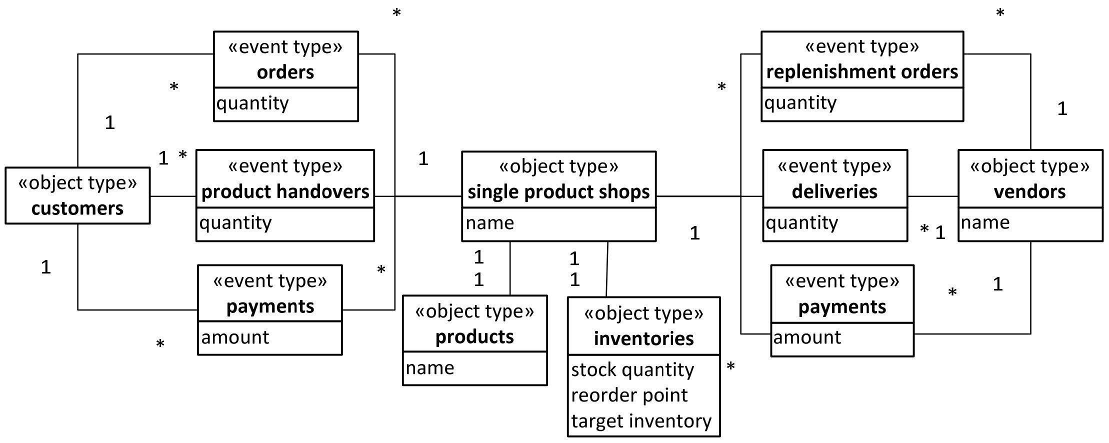


Figure 7: The complete conceptual information model.

Notice that, for brevity, we omitted the event types for the shop declining a customer order and the customer leaving the shop. Even so, the model may seem quite large for a problem like inventory management. However, in a conceptual model, we describe a complete system including all object and event types that are relevant for understanding its dynamics.

Typically, in a simulation design model we would make several simplifications allowed by our research questions, and, for instance, abstract away from the object types *products* and *inventories*. But in a conceptual model of the system under investigation, we include all relevant entity types, independently of the simplifications we may later make in the solution design and implementation. This approach results in a conceptual model that can be re-used in other simulation projects (with different research questions).

6.1.2 Making a Solution-Specific and Platform-Independent Information Design Model

We now derive an information design model from the solution-independent conceptual information model shown in Figure 7 above. Our design model is solution-specific because it is a computational design for the following specific simulation research question: *compute the percentage of lost sales* (as the difference between the total number of ordered items and the total number of sold items divided by the total number of ordered items). It is platform-independent in the sense that it does not use any modeling element that is specific for a particular platform, such as a Java datatype.

In the first step, we take a decision about which object types and event types defined in the conceptual model can be dropped in the solution design model. The goal is to keep only those entity types in the model, which are needed for being able to answer the research question. One opportunity for simplification is to drop *products* and *inventories* because our assumptions imply that there is only one product and only one inventory, so we can leave them implicit and allocate their relevant attributes to the *SingleProductShop* class. As this class name indicates, in the design model, we follow a widely used naming convention: the name of a class is a capitalized singular noun phrase in mixed case.

Further analysis shows that we can drop the event types (customer and vendor) *payments*, since we don't need any payment data, and also *product handovers*, since we don't care about the point-of-sales logistics. This leaves us with three potentially relevant object types: *customers*, *single product shops* and *vendors*, and with three potentially relevant event types: *customer orders*, *replenishment orders* and *deliveries*.

There is still room for further simplification. Since for computing the percentage of lost sales we don't need the order quantities of individual orders, but only the total number of ordered items. It's sufficient to model an aggregate of customer orders like, for instance, the *daily demand*. Consequently, we don't need to consider individual customers and therefore can drop the object type *customers*, and we use the aggregate event type *DailyDemand* instead of *customer orders*. Since we don't need any vendor information, we can also drop the object type *vendors*. Finally, since we can now assume that replenishment orders are placed when a *DailyDemand* event has occurred, implying that any *replenishment order* event temporally coincides with a *DailyDemand* event, we can also drop the event type *replenishment orders*.

Thus, the simplifications of our first design modeling step lead to a model as shown in Figure 8.

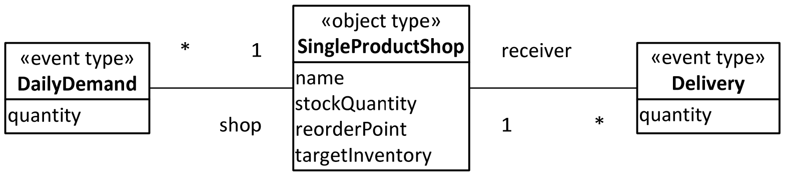


Figure 8: The initial information design model with attributes and associations (Step 1).

Notice that the two associations model the participation of the shop in both *DailyDemand* and *Delivery* events, and the association end names *shop* and *receiver* represent the reference properties *DailyDemand::shop* and *Delivery::receiver*. These reference properties allow to access the properties and invoke the methods of a shop from an event, which is essential for the *event routine* of each event type. Thus, the ontological pattern of *objects participating in events* and the implied software pattern of object reference properties in event types are the basis for defining event routines (and rules) in event types.

In the next step (step 2), we distinguish between two kinds of event types: ***exogenous event types*** and ***caused event types***, and we also define for all attributes a platform-independent datatype as their range, using specific datatypes (such as `PositiveInteger`, instead of plain `Integer`, for the quantity of a delivery), as shown in Figure 9.

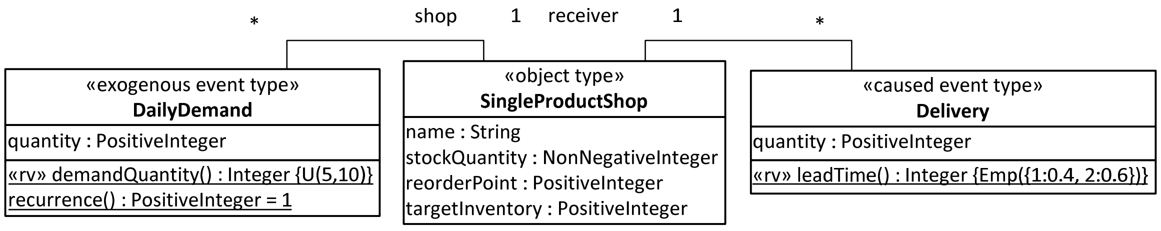

*
shop
1
receiver
1
*
«exogenous event type»
DailyDemand
quantity : PositiveInteger
«rv» demandQuantity() : Integer {U(5,10)}
recurrence() : PositiveInteger = 1
«object type»
SingleProductShop
name : String
stockQuantity : NonNegativeInteger
reorderPoint : PositiveInteger
targetInventory : PositiveInteger
«caused event type»
Delivery
quantity : PositiveInteger
«rv» leadTime() : Integer {Emp({1:0.4, 2:0.6})}

Figure 9: Adding the range of attributes and random variables (Step 2).

While exogenous events of a certain type occur again and again with some (typically *random*) *recurrence*, caused events occur at times that result from the internal causation dynamics of the simulation model. So, for any event type adopted from the conceptual model, we choose one of these two categories, and for any exogenous event type, we add a *recurrence* operation that is responsible for computing the time until the next event occurs. In our example, the recurrence of *DailyDemand* events is 1 ("each day").

In our example model, shown in Figure 9 above, we define *DailyDemand* as an exogenous event type with a recurrence of 1, implying that an event of this type occurs on each day, while we define *Delivery* as a caused event type.

6.1.3 Deriving an OO Design Model from the Information Design Model

An *OO design model* is a type of model that is in-between the platform-independent information design model and the platform-specific (e.g., Java, JavaScript, AnyLogic or Netlogo) class models for OO programming platforms.

In an OO target language, there would normally be two predefined abstract foundation classes, called *oBJECT* and *eVENT* in the model below, each with a set of generic properties and methods, implementing the two stereotypes «object type» and «event type». These two classes are shown with their name in italics in Figure 10 below, indicating that they are *abstract*, which implies that they cannot have direct instances.

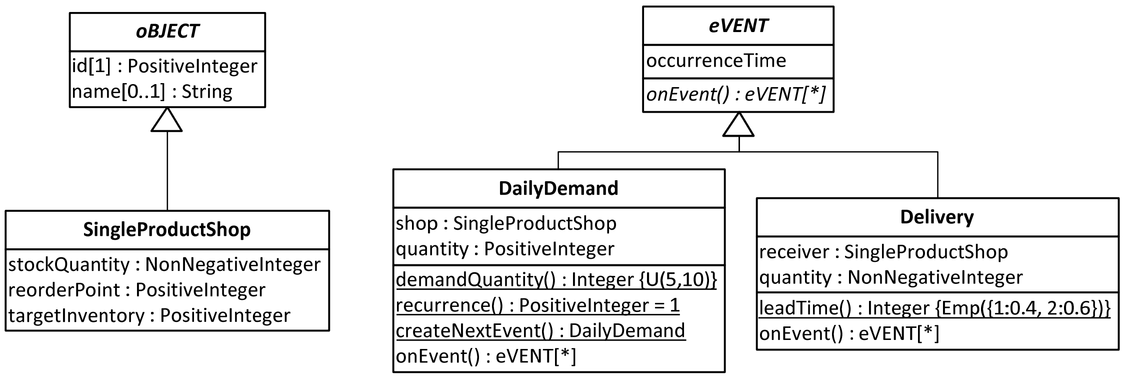

oBJECT
id[1] : PositiveInteger
name[0..1] : String
SingleProductShop
stockQuantity : NonNegativeInteger
reorderPoint : PositiveInteger
targetInventory : PositiveInteger
eVENT
occurrenceTime
onEvent() : eVENT[*]
DailyDemand
shop : SingleProductShop
quantity : PositiveInteger
demandQuantity() : Integer {U(5,10)}
recurrence() : PositiveInteger = 1
createNextEvent() : DailyDemand
onEvent() : eVENT[*]
Delivery
receiver : SingleProductShop
quantity : NonNegativeInteger
leadTime() : Integer {Emp({1:0.4, 2:0.6})}
onEvent() : eVENT[*]

Figure 10: The OO class model.

Notice that our OO class model does no longer contain the two explicit associations, which have been replaced with the reference properties *DailyDemand::shop* and *Delivery::receiver*. This is the way associations are implemented in OO programs.

The abstract *onEvent* operation in the *eVENT* class refers to the event routines triggered by events. These event routines are defined by the *onEvent* methods of the subclasses *DailyDemand* and *Delivery*. Notice that for handling the exogenous events of type *DailyDemand*, we have added a static *createNextEvent* method in *DailyDemand* for creating the next *DailyDemand* event, whenever a *DailyDemand* event has occurred.

6.1.4 Deriving Platform-Specific Models and Code

Finally, we may either make platform-specific (e.g., Java, JavaScript, AnyLogic or Netlogo) class models and then code them, or directly code the OO class model in the chosen target language by applying the same patterns as we would for making the platform-specific class model. In a platform-specific model, we use the specific elements of the chosen platform, such as Java-specific datatypes in the case of the Java platform. For instance, the event class *DailyDemand* can be implemented in the JavaScript simulation platform ΩE provided by sim4edu.com with the following code:

```
var DailyDemand = new cLASS({
  Name: "DailyDemand",
  supertypeName: "eVENT",
  properties: {
    "shop": {range: "SingleProductShop"},
    "quantity": {range: "PositiveInteger", label:"Quantity"}
  },
  methods: {
    "onEvent": function () {...}
  }
});
```

6.2 Process Modeling

We show a conceptual process model and a process design model for our inventory management case, both in the form of BPMN process diagrams.

6.2.1 Making a Conceptual Process Model

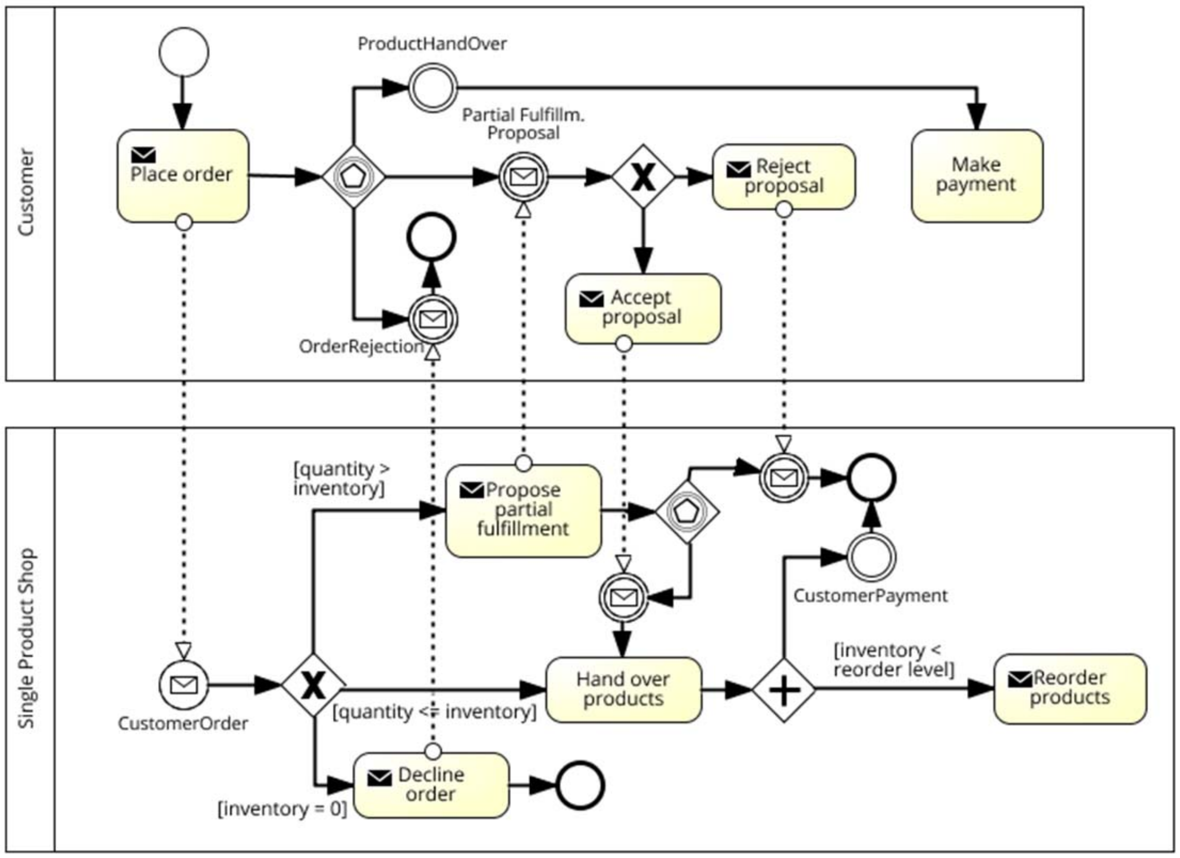


Figure 11: A conceptual process model (where Vendor is omitted for a lack of space).

In the conceptual process model shown in Figure 11, we model the two actors *Customer* and *Single Product Shop*, together with their interactions in the style of a business process model. For an elaborate explanation of this model see (Wagner 2017).

6.2.2 Making a Process Design Model

A process design model does not describe the process of the real world system under investigation, but defines a computational design of the simulation process by describing all event rules of the model to be designed. Consequently, the BPMN tasks of the process design model represent computational (simulator) actions, and not the actions of real-world actors. Since our design for the given research question includes only two event types, *DailyDemand* and *Delivery*, we only need to model two event rules. For a lack of space, we can only show one of them, the *DailyDemand* event rule, in the BPMN diagram shown in Fig. 12. Notice that event rules can also be modeled textually, in an event rule table, by using pseudo-code, as shown in (Wagner 2017).

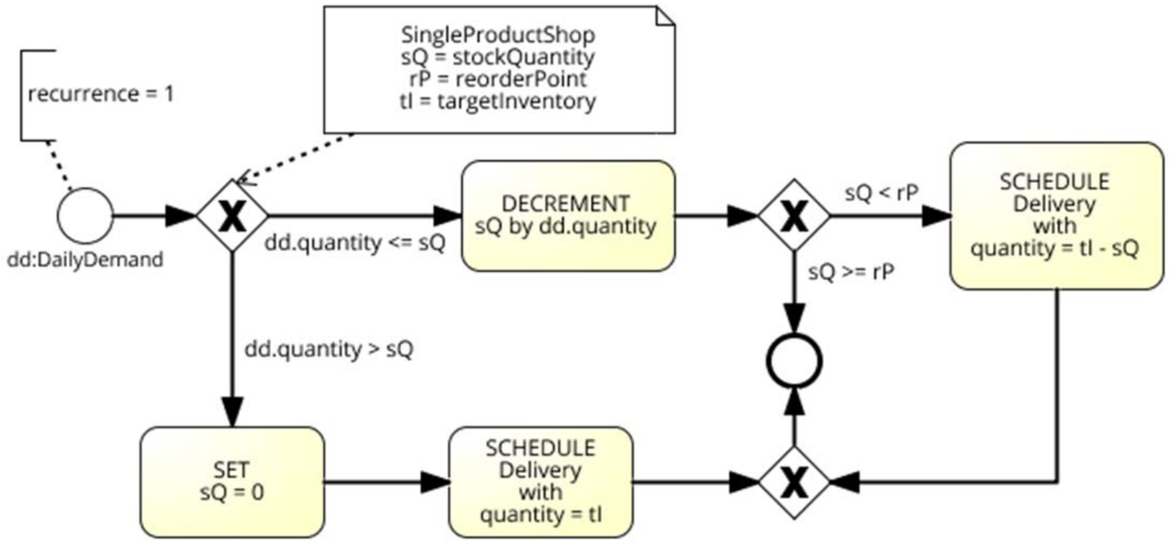


Figure 12: A design model for the *DailyDemand* event rule.

7 CONCLUSIONS

UML class diagrams and BPMN process diagrams allow making visual simulation models that can be coded with any simulation platform supporting objects and events. While using UML and BPMN is not yet common in *modeling and simulation*, both languages are well-established in *information systems* and *software engineering*.

REFERENCES

Ambler, S.W. 2010. UML 2 Class Diagrams. http://www.agilemodeling.com/artifacts/classDiagram.htm

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

Cetinkaya, D., and A. Verbraeck. 2011. "Metamodeling and Model Transformations in Modeling and Simulation". In *Proceedings of the 2011 Winter Simulation Conference*, edited by S. Jain, R.R. Creasey J. Himmelspach, K. P. White, and M. Fu, 3048–3058. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Gordon, G. 1961. "A general purpose systems simulation program". In *Proceedings of the Eastern Joint Computer Conference*, Washington, D.C.

Guizzardi, G., and G. Wagner. 2010. "Using the Unified Foundational Ontology (UFO) as a Foundation for General Conceptual Modeling Languages". In Poli R., M . Healy and A. Kameas (Eds.), *Theory and Applications of Ontology: Computer Applications*., 175–196.

Guizzardi, G., and G. Wagner. 2012. "Tutorial: Conceptual Simulation Modeling with Onto-UML". In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R.

Pasupathy, O. Rose, and A.M. Uhrmacher, 52−66. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Guizzardi, G., and G. Wagner. 2013. “Dispositions and Causal Laws as the Ontological Foundation of Transition Rules in Simulation Models”. In *Proceedings of the 2013 Winter Simulation Conference*, edited by R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, 1335–1346. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. http://informs-sim.org/wsc13papers/includes/files/117.pdf

Himmelspach, J. 2009. “Toward a Collection of Principles, Techniques and Elements of Modeling and Simulation Software”. In *Proc. of the 2009 International Conference on Advances in System Simulation*. IEEE Computer Society, 56–61.

Camunda. “BPMN 2.0 Tutorial”. https://camunda.org/bpmn/tutorial.

Onggo, B. S. S., and O. Karpat. 2011. “Agent-Based Conceptual Model Representation Using BPMN”. In *Proceedings of the 2011 Winter Simulation Conference*, edited by S. Jain, R.R. Creasey J. Himmelspach, K. P. White, and M. Fu, 671−682. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Pegden, C.D. and D.A. Davis. 1992. “Arena: a SIMAN/Cinema-Based Hierarchical Modeling System”. In *Proceedings of the 1992 Winter Simulation Conference*, edited by J.J. Swain, D. Goldsman, R.C. Crain, and J.R. Wilson, 390–399. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Pegden, C.D. 2010. “Advanced Tutorial: Overview of Simulation World Views”. In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, 643−651. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Robinson, S. 2013. “Conceptual Modeling for Simulation”. In *Proceedings of the 2013 Winter Simulation Conference*, edited by R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, 377-388. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Wagner, G., O. Nicolae, and J. Werner. 2009. “Extending Discrete Event Simulation by Adding an Activity Concept for Business Process Modeling and Simulation”. In *Proceedings of the 2009 Winter Simulation Conference*, edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin and R. G. Ingalls, 2951-2962. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Wagner, G. 2017. “Information and Process Modeling for Simulation”. *Journal of Simulation Engineering*, 1:1. Available from: http://JSimE.org.

Zee, D.-J. van der et al. 2010. “Panel Discussion: Education on Conceptual Modeling for Simulation – Challenging the Art”. In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, 290−304. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHY

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