

TUTORIAL: INFORMATION AND PROCESS MODELING FOR SIMULATION

Gerd Wagner

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

ABSTRACT

Like a system model in *Information Systems* and *Software Engineering* (IS/SE), a system model in simulation engineering mainly consists of an *information model* and a *process model*. In the fields of 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 shows how to use UML class diagrams and BPMN process models 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.

1 INTRODUCTION

We use the term *simulation engineering* for denoting the engineering activity of developing a computer simulation, which is a piece of software that can be executed with different settings for performing a simulation study. 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 engineering project, at the same time it is thought to be the least understood part of simulation engineering (Tako et al. 2010). In a recent 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.

Discrete Event Simulation (DES) is concerned with the simulation of real-world systems that are conceived as *discrete event systems* (or ‘discrete dynamic systems’). Such discrete system *abstractions* are immaterial entities that only exist in the mind of the user (or a community of users) of a language. In order to be documented, communicated and analyzed they must be captured, i.e. represented in terms of some concrete artifact with the help of a language. The representation of an abstraction in a language is called a *model* and the language used for its creation is called a *modeling language*.

A discrete event system model may be expressed at different levels of abstraction. The system’s state structure can be described by means of a set of variables and/or objects of some type. For the purpose of simulation, the goal is to develop an executable simulation model that allows simulating the successive

occurrence of events that may change the values of variables and the states of objects as well as create follow-up events.

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* rather 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 with the help of *model transformation*.

Unlike the IS/SE research community, which developed many modeling standards, such as UML and BPMN, the *modeling and simulation* research community did not develop any general simulation modeling language that could be used as a standard. There are only vendor-specific simulation modeling languages (such as the graphical models of ARENA) developed by simulation tool vendors and embedded in their tools.

Due to their great expressivity and their wide adoption as modeling standards, *UML Class Diagrams* and *BPMN* seem to be the best choices for making information and process models in a model-driven 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. In fact, both an analysis of UML with respect to its suitability for conceptual modeling in (Guizzardi 2005) and an analysis of BPMN with respect to its suitability for agent-based simulation modeling in (Guizzardi and Wagner 2011) have revealed a number of ambiguities and shortcomings that should be resolved and fixed for increasing the suitability of these languages for simulation modeling.

Several authors, e.g. Wagner et al. (2009), Cetinkaya et al. (2011) and Onggo and Karpal (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 develop a simulation of a service desk with the help of UML class diagrams and BPMN diagrams.

1.1 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 and events. In the DES literature, it is often stated that DES is based on the concept of “entities flowing through the system”. E.g., this is the paradigm of an entire class of simulation software in the tradition of the classical ARENA system <www.arenasimulation.com>. However, the loose metaphor of a “flow” only applies to entities of certain types: events, messages, and certain material objects may, in some sense, flow, while many entities of other types, such as buildings or organizations, do not flow in any sense. Also, subsuming these three different kinds of flows under one common term “entity flow” obscures their real meanings. It is therefore highly questionable to associate DES with a “flow of entities”. Rather, one may say that DES is based on a *flow of events*.

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 *qualities*, *dispositions* and *relationships*, and
3. *events*.

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

Clearly, *objects* and *events* are the most basic categories for DES. There is an interesting, and fundamental, duality between them: while all temporal properties of objects are defined in terms of the events they participate in, all spatial properties of events are defined in terms of the spatial properties of their participants, as has been noted in (Masolo et al. 2003).

While the concept of an event is typically 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

1. *objects* (of various types),
2. *events* (of various types),
3. *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 affected 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 laws* (*disposition types*) e.g., in the form of *transition rules*.

Any *discrete event simulation* (DES) formalism has one or more language elements allowing to specify, at least implicitly, *transition functions*, or *transition rules*, that allow representing causal laws. 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. All DES systems implement their internal logic using this basic modeling approach, regardless of the worldview that they present to the user.

Process worldview: The system is described as a process flow where a set of passive entities flow through the system and are subject to a series of process steps (e.g. seize, delay, release) that model the state changes that take place in the system.

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.

The process worldview is still widely used in practice and many simulation systems based on it have incorporated some form of support for objects and *object-oriented (OO)* programming. According to Pegden (2010), agent based modeling is typically implemented with the object world view. So, today’s DES landscape is largely based on the process worldview and object worldview, and the fundamental concept of events is hardly considered anywhere.

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 does neither support 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 *transition rules*.

1.2 Do We Model Entities or Entity Types?

When we model a particular discrete system, including the many different things that make up the system, do we model these particular things and this particular system or do we model this type of system with all the types of things that make up such a type of system? The answer to this question seems to vary from case to case. In the case of modeling a machine, we are typically interested in a model of this type of machine, and not in a model of a particular machine. But in the case of modeling an organization, we typically want to make a model of this particular organization only, and we are typically not really interested in considering the more general case of the type of organization that is instantiated by this particular organization.

However, we argue that we should always model types, and not individuals. Even in the case when what we really need is a model of a particular organization o or of a specific unique machine m , only, we should adopt the view that there are not only o and m , but there are also corresponding types of organizations O and of machines M , which are instantiated by o and m , and the real goal of our modeling project is to model O or M , and not o or m . The additional cost of modeling types (O and M) instead of instances (o and m) is negligible, and the potential benefit is that the resulting models can be reused more easily.

In the case of process modeling, the widely used terminology of “modeling a business process” may obscure the fact that we don't really model a particular *process*, but rather a particular *process type*, which is instantiated many times (each day) by many different processes of that type.

2 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.

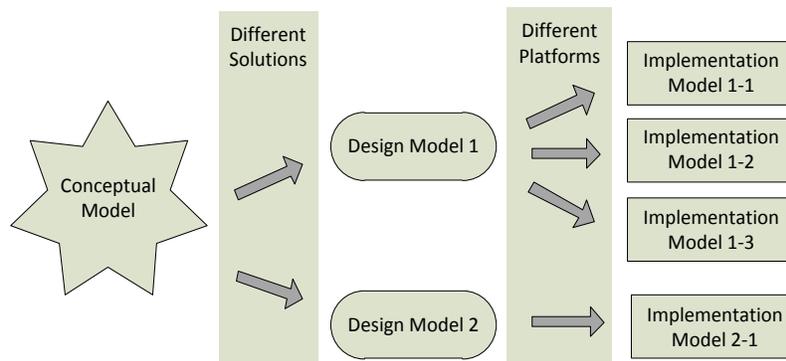


Figure 1: From a conceptual model to design models to implementation models.

In the implementation phase, an implementation model is encoded 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**. So, from a given information design model, we may derive an SQL data model, a Java data model and a C# data 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 a person, 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 *simulation design model* to a *simulation 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 may affect the modeling languages to be used.

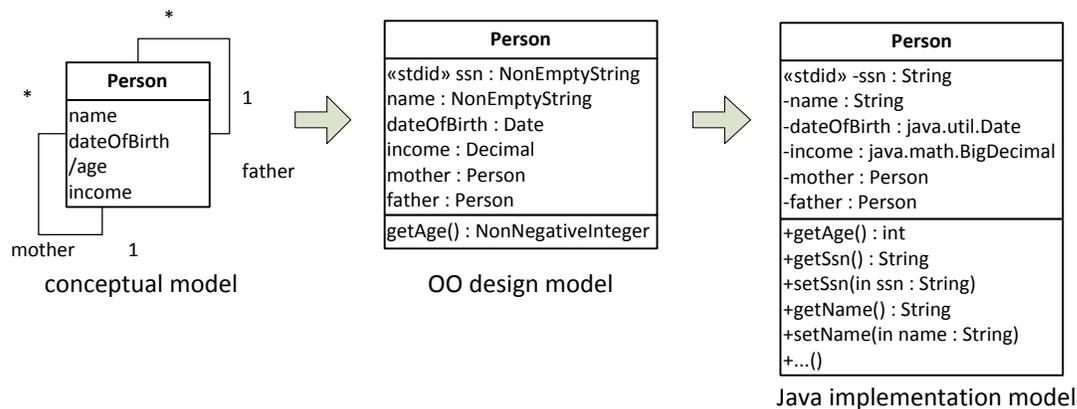


Figure 2: From a conceptual information model via a design model to a Java implementation model.

3 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 use 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 2 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;

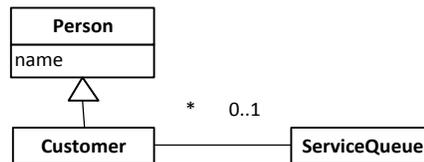


Figure 3: Describing the entity types `Person`, `Customer` and `ServiceQueue`.

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*).

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.

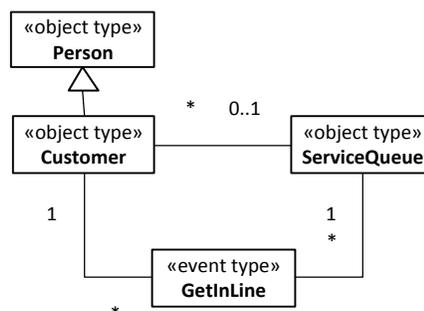


Figure 4: Distinguishing between object types and event types as two different categories of entity types.

UML also allows defining class-level attributes in the form of ‘tagged values’, which is shown in Figure 5 below, where we define the class-level attribute `occurrenceTime`, representing a random variable, in the form of a *tagged value*, and assign it the value $U(1,8)$ denoting the uniform distribution with lower bound 1 and upper bound 8.

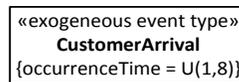


Figure 5: A class with a class-level attribute in the form of a tagged value .

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

4 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.

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 <i>Start Event</i> is drawn as a circle with a thin border line, while an <i>Intermediate Event</i> has a double border line and an <i>End Event</i> has a thick border line.	 Start  End Event
Activity (Task, Sub-Process)	Work that is performed within a process. A <i>Task</i> is an atomic Activity, while a <i>Sub-Process</i> is a composite Activity. A Sub-Process can be either in a <i>collapsed</i> or in an <i>expanded</i> 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 <i>Exclusive OR-Split</i> (if there are 2 or more output flows) or an <i>Exclusive OR-Join</i> (if there are 2 or more input flows). A gateway with a plus symbol denotes an <i>AND-Split</i> (if there are 2 or more output flows) or an <i>AND-Join</i> (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.	

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.

For an introduction to BPMN, the reader is referred to Mancarella (2011). 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).

5 EXAMPLE: MODELING A SERVICE DESK

In the *service queue system* example, as implemented in the *Simurena Library* (Simurena 2012), customers arrive at random times at a service desk where they have to wait in a queue when the service desk is busy. Otherwise, when the queue is empty and the service desk is not busy, they are immediately served by the service clerk. Whenever a service is completed (assuming some random duration), the next customer from the queue, if there is any, will be invited for the service.

5.1 Information Modeling

5.1.1 Making a Conceptual Information Model

It is straight-forward to extract four entity types from the problem description above by analyzing the noun phrases, so a naïve conceptual information model of this system may look as shown in Figure 6. There are one-to-one associations between the entity types *ServiceDesk* and *ServiceClerk* and between *ServiceDesk* and *ServiceQueue*. The fact that a service queue consists of zero or more customers is modeled with the help of a one-to-many association between *ServiceQueue* and *Customer*.

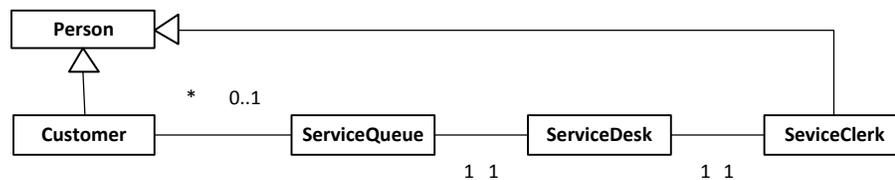


Figure 6: An information model of a service queue system.

Typically, in a simulation design model we would make several simplifications based on our simulation research questions, and, for instance, abstract away from the entity type *ServiceClerk*. But in a conceptual system model, we include all entity types that are relevant for understanding the real-world system, independently of the simplifications we may later make in the solution design and implementation. This approach results in a model that can be re-used in other simulation projects (with different research questions).

Notice that we modeled *Customer* and *ServiceClerk* as subclasses of *Person*. This follows a general pattern analyzed by Guizzardi (2005) who recommends adding a *base type* (or *kind*), such as *Person*, for all *role* classes in a model, such as *Customer* and *ServiceClerk*. One of the benefits of applying this pattern is that we can see that a service clerk may also be a customer, which is a special case of the general possibility that an employee of an organization may also be a customer of it.

After modeling all relevant object types in the first step, we model the relevant event types in a second step, as shown in Figure 7 below. The main type of association between events and objects is *participation*. When adding event types to the object types in our conceptual information model, we therefore also model the participation types between them. For instance, in Figure 7, we have modeled that a customer arrival event has exactly one customer and one service desk as its participants.

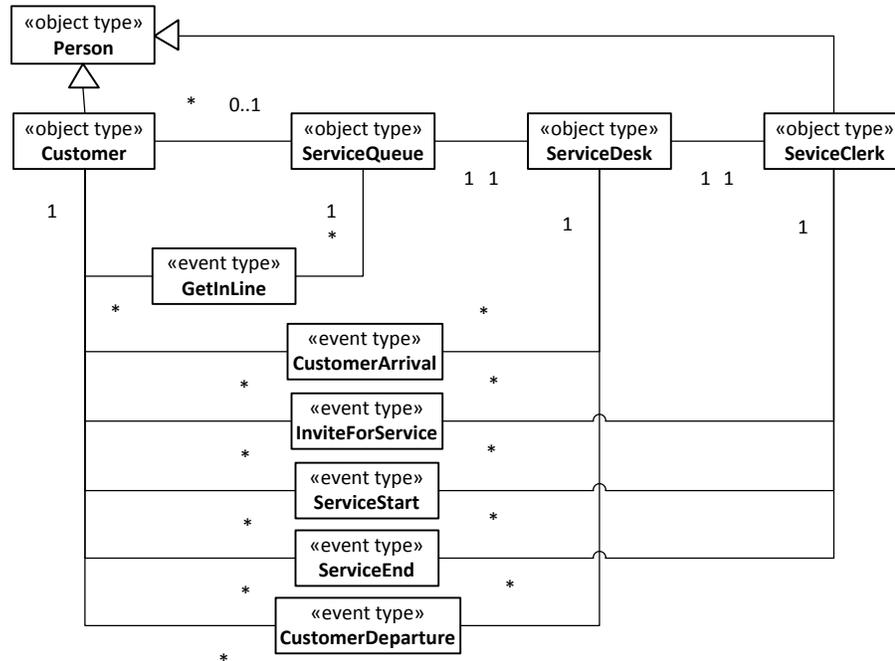


Figure 7: Adding event types.

From an analysis of the problem description, we may infer that there are six potentially relevant types of events in this system: customers arriving, customers getting in line, customers being invited by the clerk for being served, start of service, end of service and customers departing. Since the service queue system is an example of a business system, it is not surprising that all these events are *actions* (or *action events*) performed by human actors. In fact, we obtained this list of event types by asking ourselves: what are the relevant actions of the two actors of the system, *customers* and *service clerks*?

A pattern of two coupled start and end events, like a *ServiceStart* and a *ServiceEnd* event, indicates that there is actually an *activity* that is temporally framed by these two events. So, in our example, these two event types indicate that we may model a *GetService* activity type that would replace the *ServiceStart* and *ServiceEnd* event types. Notice that also a *GetInLine* event may be considered to start an activity of waiting for a *InviteForService* event, as modeled by the *Wait* activity in the conceptual process model shown in Figure 11 below.

In order to complete the model of Figure 7, we may add attributes that help describing objects and events of these types.

5.1.2 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 "mean response time" statistics as the average length of time a customer spends in the system from arrival to departure, which is the average waiting time plus the average service duration.

Our research question requires that we model individual customers, since for being able to compute the time a customer spends in the system we need to know which customer is next for getting the service and what is their arrival time. For knowing which customer is next, we need to model the service queue as a First-In-First-Out (FIFO) queue, which can be expressed in the UML class diagram with the help of the ordered association end `waitingCustomers`. Notice that by placing a dot on the line at this end of the association, and not on the other end as well, we make the association uni-directional implying the design decision that it will be represented by a reference property with name `waitingCustomers` in the `ServiceDesk` class. For being able to easily retrieve the arrival time of a customer, which is an information item coming from the `CustomerArrival` event, we record it along with the customer data, so we add a corresponding attribute to the `Customer` class.

In an information design model we distinguish between two kinds of event types: *exogenous event types* and *caused event types*. While exogenous events of a certain type occur again and again with some *random periodicity* that can be modeled with a probability distribution assigned to the event type, caused events occur at times that result from the internal causation dynamics of the simulation. So, for any event type adopted from the conceptual model, we have to make a decision if we model it as an exogenous or as a caused event type, and for any exogenous event type, we have to specify a probability distribution for its periodicity in our simulation's information design model.

In our example model, shown in Figure 8 below, we define `CustomerArrival` as an exogenous event type with the class-level attribute `occurrenceTime` (in the form of a 'tagged value') having the value $U(1,8)$ denoting the uniform distribution with lower bound 1 and upper bound 8, while we define `CustomerDeparture` as a caused event type.

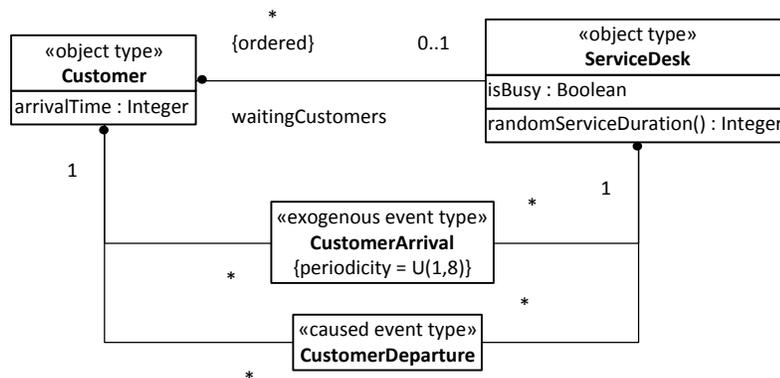


Figure 8: An information design model.

Notice that we have modeled the random duration of a service with the help of the operation `randomServiceDuration()` shown in the third compartment of the `ServiceDesk` class. In this operation, we can either define an empirical distribution function or invoke a theoretical standard distribution.

5.1.3 Deriving an Object-Oriented Design Model

In certain cases, we can make a generic OO design model that can be re-used for, or encoded in, several OO programming languages. E.g., the OO design model shown in Figure 10 below, which is derived from the information design model shown in Figure 9, could be transformed to more specific C++, Java or JavaScript data models. In each OO target language, there would be two foundation classes `Object` and `Event`, each with suitable properties, implementing the two stereotypes `<<object type>>` and `<<event type>>`. Then in the data model, classes stereotyped as *event type* or *object type* in the design model extend the pre-defined foundation classes `Object` and `Event`. This is shown for the two event types `CustomerArrival` and `CustomerDeparture` in Figure 9.

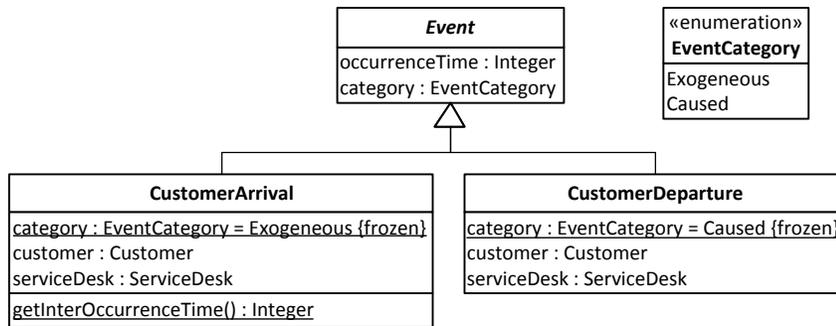


Figure 9: An OO data model for the event types `CustomerArrival` and `CustomerDeparture`.

Notice that an OO design model does no longer contain associations. In our example the associations between `CustomerArrival` and `Customer` and between `CustomerArrival` and `ServiceDesk` have been replaced with the reference properties `customer` and `serviceDesk`, and similarly for `CustomerDeparture`.

5.1.4 Deriving a Platform-Specific Data Model

As an example of a platform-specific data model for a standard programming language, we present a data model for a simple Java simulator similar to the simple Java simulator discussed in (Banks et al. 2005), but with classes for representing object types and event types. In a data model for this platform, as shown in Figure 10, the reference property `waitingCustomers` would have the parameterized type `Queue<Customer>` as its range, where `Queue` is a Java built-in interface that extends `Collection` and that can be implemented, e.g., with the pre-defined library class `LinkedList`.

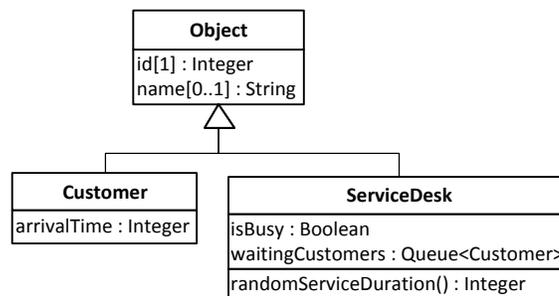


Figure 10: A Java simulation data model with the parameterized type `Queue<Customer>`.

5.2 Process Modeling

5.2.1 Making a Conceptual Process Model

In a conceptual process model we prefer modeling activities instead of corresponding start and end events. Consequently, for the service desk example modeled with 6 event types in Figure 7, we merge the `ServiceStart` and `ServiceEnd` event types into a `GetService` activity type. In addition, we also use a `Wait` activity type replacing the `GetInLine` event type, which is its implicit start event type. This results in a model with the event types `CustomerArrival`, `CustomerDeparture` and `InviteForService`, and the activity types `Wait` and `GetService`, as shown in Figure 11.

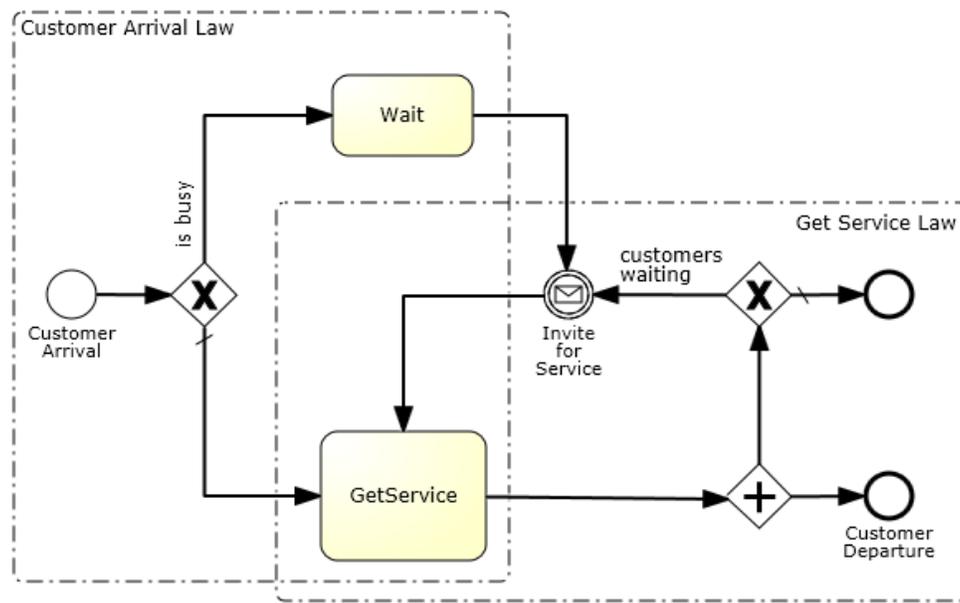


Figure 11: A conceptual process model of the service queue system describing two causal laws.

After modeling the relevant activity types and event types, we identify those types of events that account for the causation of relevant state changes and follow-up events. They trigger a *causal law*. Any event type modeled in the information model could potentially trigger a causal law. For simplicity, however, we omit event types that classify events, which can be considered to temporally coincide with events of another type. Thus, we consider only two laws: one for `CustomerArrival` events and one for `ServiceEnd` events. We omit service start events, since they can be viewed to coincide either with customer arrival events, when the service queue is empty, or with customer departure events, when the queue is non-empty. In a similar way, service end events can be viewed to coincide with customer departure events.

In the conceptual process model, we use a BPMN element group rendered with a dashed line for visualizing a causal law. In this way we get the two BPMN groups `Customer Arrival Law` and `Get Service Law` with the following intended meanings:

1. The `Customer Arrival Law` states that when a customer arrives, she either gets in line and waits, if the service desk is busy, or else gets the service.
2. The `Get Service Law` states that when a service ends, the customer immediately departs, and the service clerk checks if there are still waiting customers. If there are still waiting customers, the next customer is invited for service, so this customer moves forward to the service desk and the next service starts.

In the process design model shown in Figure 12, we turn the causal laws defined in the conceptual process model into corresponding *transition rules* described as BPMN sub-processes with tasks expressed as statements having a clear computational meaning and with a binding to objects. For instance, when variables are used in a task statement, such as the object variables c and sd in the task statement “Add c to $sd.waitingCustomers$ ”, then they need to be bound to an object type with the help of a suitable BPMN data object, like the ones shown on the left hand side of Figure 12.

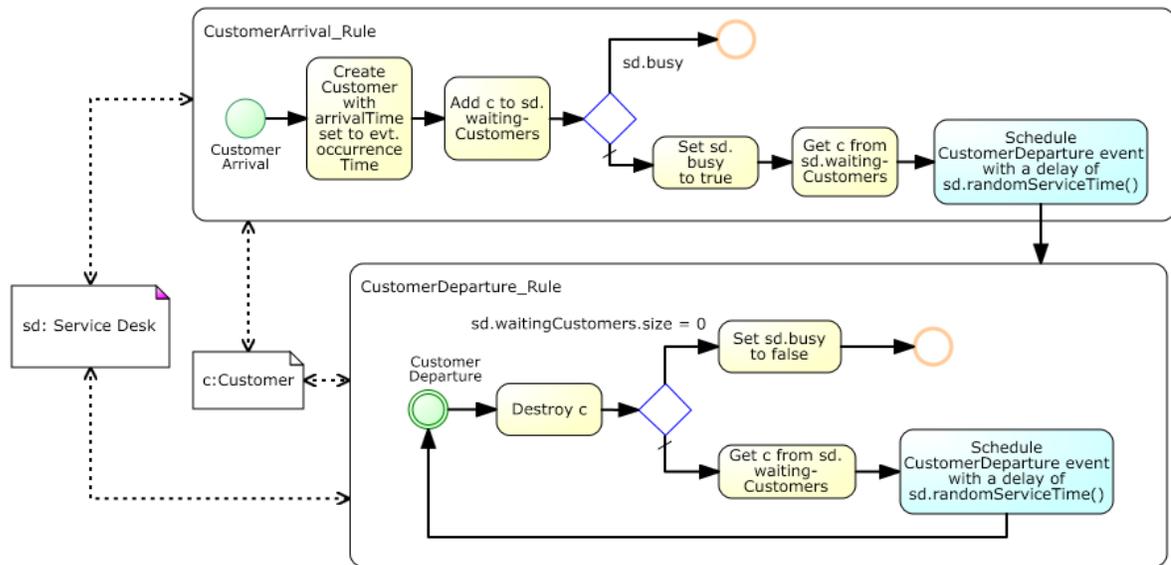


Figure12: A process design model.

6 CONCLUSIONS

In this tutorial, we show how to use UML class diagrams and BPMN process diagrams in a model-driven simulation engineering approach. While the use of UML class diagrams for modeling the (information) state structure of a system is well-understood, this is not the case for the use of BPMN diagrams for modeling the dynamics (or process type) of a system.

For further reading on *modeling for simulation*, see www.modeling-for-simulation.info

REFERENCES

- Ambler, S.W. 2010. UML 2 Class Diagrams. <http://www.agilemodeling.com/artifacts/classDiagram.htm>. Accessed July 25, 2012.
- 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." *Proceedings of 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.
- Cetinkaya, D., A. Verbraeck, and M. D. Seck. 2011. "MDD4MS: A Model Driven Development Framework for Modeling and Simulation." *Proceedings of the 2011 Summer Computer Simulation Conference (SCSC 2011)*. The Hague, Netherlands.
- Guizzardi, G. 2005. *Ontological foundations for structural conceptual models*. PhD thesis, University of Twente, Enschede, The Netherlands. CTIT Ph.D.-thesis series No. 05-74 ISBN 90-75176-81-3.
- Guizzardi, G. and G. Wagner. 2011. "Can BPMN Be Used for Making Simulation Models?" In: J. Barjis, T. Eldabi and A. Gupta (Eds.). *Enterprise and Organizational Modeling and Simulation*. Springer Lecture Notes in Business Information Processing, vol. 88.
- Guizzardi, G. and G. Wagner. 2010a. "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. 2010b. "Towards an Ontological Foundation of Discrete Event Simulation." *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain,

- J. Montoya-Torres, J. Hukan, and E. Yücesan, 652–664. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.
- Guizzardi, G. and G. Wagner. 2012. “Tutorial: Conceptual Simulation Modeling with Onto-UML.” *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. G. Wagner, R. Falbo, J.P. Almeida and R. Guizzardi. 2013. “Towards Ontological Foundations for the Conceptual Modeling of Events.” *Proceedings of 32nd International Conference on Conceptual Modeling (ER'2013)*. Hong Kong, November 11-13. Springer-Verlag, Lecture Notes in Computer Science.
- Guizzardi, G. and G. Wagner. 2013. “Dispositions and Causal Laws as the Ontological Foundation of Transition Rules in Simulation Models.” *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.
- Himmelspach, J. 2009. “Toward a Collection of Principles, Techniques and Elements of Modeling and Simulation Software.” *Proc. of the 2009 International Conference on Advances in System Simulation*. IEEE Computer Society, 56–61.
- Mancarella, S. 2011. *Business Process Modelling Notation – A Tutorial*. Accessed July 25, 2012. <http://www.omg.org/news/meetings/workshops/HC-Australia/Mancarella.pdf>.
- Masolo, C., S. Borgo, A. Gangemi, N. Guarino and A. Oltramari, 2003. “Ontology Library,” WonderWeb Deliverable D18, LOA-ISTC, CNR, Tech. Rep. Accessed August 19, 2014 at <http://wonderweb.man.ac.uk/deliverables/documents/D18.pdf>.
- Onggo, B. S. S. and O. Karpat. 2011. “Agent-Based Conceptual Model Representation Using BPMN.” *Proceedings of 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. 2010. “Advanced Tutorial: Overview of Simulation World Views.” *Proceedings of Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hukan, and E. Yücesan, 643–651. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.
- Robinson, S. 2013. “Conceptual Modeling for Simulation.” *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.
- Tako, A.A., K. Kotiadis, and C. Vasilakis. 2010. “A Participative Modeling Framework For Developing Conceptual Models in Healthcare Simulation Studies.” *Proceedings of Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hukan, and E. Yücesan, 500–512. 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.” *Proceedings of 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.
- Zee, D.-J. van der, K. Kotiadis, A.A. Tako, M. Pidd, O. Balci, A. Tolk, and M. Elder. 2010. “Panel Discussion: Education on Conceptual Modeling for Simulation – Challenging the Art.” *Proceedings of Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hukan, and E. Yücesan, 290–304. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

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