PROCESS DESIGN MODELING WITH EXTENDED EVENT GRAPHS

Gerd Wagner

Department of Informatics
Brandenburg University of Technology
P. O. Box 101344
03013 Cottbus, Germany
G.Wagner@b-tu.de

ABSTRACT

Schruben's *Event Graphs (EGs)*, defining the event types of a simulation model and event scheduling arrows between them, representing causal regularities, provide an elegant visual modeling language and formalism for event-based simulation, which can be viewed as the most fundamental *Discrete Event Simulation (DES)* approach. We show how to extend and visually improve the language of EGs by adding elements of the *Business Process Modeling Notation (BPMN)*: (1) *mini diamonds* for designating conditional control flow arrows, (2) *Gateways* for conditional and parallel branching, (3) typed *Data Objects* for accommodating *object-oriented (OO)* state structure modeling, and (4) *Activities*. The resulting extension of EGs, called *Discrete Event Process Modeling Notation (DPMN)*, is more expressive and visually more clear than traditional EGs, and its visual syntax is harmonized with BPMN process diagrams, thus building a bridge between the DES and the *Business Process Management* research communities.

Keywords: Event Graphs, BPMN, DPMN, Discrete Event Simulation, Object Event Simulation.

1 INTRODUCTION

The NSF workshop on *Research Challenges in Modeling and Simulation for Engineered Complex Systems* (Fujimoto et al. 2016) identified four key research challenges, one of them being "conceptual modeling", where conceptual models are defined as the models that form the language through which individuals with widely different expertise communicate and collaborate. The workshop report states that "advances in conceptual modeling are essential to enable effective collaboration and cost-effective, error-free translation of the model into a suitable computer representation".

As argued by Wagner (2018b), the term "conceptual model" is often used ambiguously in M&S either for the solution-independent *domain model* or for the technology-independent *design model*. While domain models are solution-independent descriptions of a problem domain produced in the analysis phase (mainly for communicating with domain experts), a design model is developed on the basis of the domain model in the design phase as a technology-independent solution design.

In the areas of *Information Systems* and *Software Engineering*, the term "conceptual model" is used as a synonym of "domain model". As opposed to a domain model, a design model is solution-specific because it is a computational design for the particular purpose of a simulation project (defined, e.g., by specific research questions).

The *Event Graph (EG)* diagrams of Schruben (1983) allow defining computationally complete process design models for event-based simulation, which can be viewed as the most fundamental *Discrete Event Simulation (DES)* paradigm. In these diagrams, circles represent event types, and arrows between two event type circles *A* and *B* represent *event scheduling* in the sense that an occurrence of an event of type *A* in a simulation run causes the simulator to schedule a future event of type *B*.

We can understand the simulation concept of event scheduling with a *Future Events List (FEL)* pioneered by SIMSCRIPT (Markowitz 1962) as a computational formalization of *causation*. It allows simulating event causation in the sense that an occurrence of an event of type *A* in a simulation run causes an occurrence of an event of type *B* at a later time point in the simulation run.

While the concept of event scheduling, as captured by EGs, is the basis of a formal semantics of *DES design models*, a philosophical (or *ontological*) semantics of *causation* is needed for the semantics of *conceptual DES models*, because they are not about computational artifacts, but about real world systems. In *conceptual DES modeling*, we can use BPMN-style process models, but we have to deal with *causation arrows* between event type circles instead of event scheduling arrows. However, the present paper is only concerned with DES design models based on extended EGs.

In EGs, event type circles may be annotated with (possibly conditional) variable assignments representing state changes. A (possibly conditional) event scheduling arrow may be annotated with a delay time expression and an assignment for the attributes of the caused event.

A simple EG example, modeling a single service queueing system, is shown in Figure 1 below.

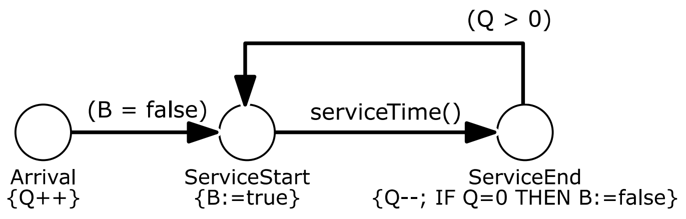

Figure 1: An EG describing a single service queueing system.

In the EG describing a single service queueing system shown in Figure 1, the customer being served is considered to be part of the queue. Whenever an *Arrival* event occurs, the state variable *Q*, representing the queue length, is incremented by 1, as defined by the state change statement *Q*++.

In addition, if the state condition *B* = false holds, this means that the service desk is not busy such that an immediate follow-up event of type *ServiceStart* can be scheduled. This is expressed by the arrow between the *Arrival* circle and the *ServiceStart* circle, which is a conditional event scheduling arrow, as indicated by the arrow's annotation (*B* = false) specifying a condition.

Whenever a *ServiceStart* event occurs, the state variable *B* is set to true and a *ServiceEnd* event is scheduled, with a delay obtained by invoking the *serviceTime* function, which implements a corresponding random variable, as defined by the unconditional event scheduling arrow between *ServiceStart* and *ServiceEnd*. Finally, a *ServiceEnd* event causes (a) the state change that *Q* is decremented by 1, as specified by the state change statement Q--, (b) the state change that *B* is set to false if Q is equal to 0, and (c), if the condition *Q* > 0 holds, that an immediate follow-up event of type *ServiceStart* is scheduled.

Notice that in Figure 1, *ServiceStart* and *ServiceEnd* denote event types for *caused events*, as implied by having incoming event scheduling arrows, while *Arrival* events are not caused, but *exogenous*, which means that they are typically periodically recurring.

EGs allow conditional and parallel branching simply by attaching more than one outgoing arrow to an event circle. However, such a notation is visually less explicit and harder to read compared with using special branching symbols, like the Gateway symbols of BPMN. In Section 3, we therefore propose to extend EGs by adding the Gateway symbols of BPMN for improving their visual readability.

BPMN is a graphical modeling language for defining business processes based on events and activities, following the flow-chart metaphor. It is remarkable that the visual and conceptual overlap between EGs and BPMN diagrams, and possible combinations of the two languages, have not yet been investigated, neither by DES researchers nor by Business Process Management researchers.

BPMN needs to be adapted for the purpose of simulation modeling. For instance, in BPMN, *Data Objects* can be attached to events and activities for describing the inter-dependencies between events, activities and data. However, the syntax and semantics of BPMN Data Objects is not sufficient for representing pre-conditions and state changes.

Consequently, in section 4, we propose to add an extended/improved form of Data Objects to EGs. This is one of the issues that motivate the development of a variant of BPMN, called the *Discrete Event Process Modeling Notation (DPMN)*, defined in (Wagner 2018a), which may be viewed as a BPMN-based extension of EGs or as a an EG-based restriction and formalization of BPMN process diagrams.

Classical EGs represent the state of a system in the form of a simple set of global variables. However, the state variables of a real world system can be more naturally represented by the attributes of programming language objects (or *information objects*) intended to represent real-world objects. This was the fundamental insight that led Dahl and Nygaard (1966) to the development of the simulation language *Simula*, which triggered the development of the *Object-Oriented Programming (OOP)* paradigm in computer science.

Modeling the state structure of a system by modeling the state structure of its objects can be achieved by making a UML class model that defines object types in the form of classes, which can be implemented by corresponding OOP (e.g., Java or C#) classes. Using a UML class model as a basis of an EG also allows defining event types and complex datatypes, in addition to object types. Such a model provides a type system for object-oriented extensions of EGs.

While the addition of BPMN-style mini-diamonds and exclusive/inclusive/parallel gateway diamonds to EGs, discussed in Section 3 and 4, can be viewed as syntactic sugar that does not increase the expressivity of EGs, the addition of (1) object type definitions, discussed in Section 2, (2) Data Objects, discussed in Section 5, and (3) Activities, discussed in Section 6, to EGs does increase their expressivity.

As we show in Section 8, a general formal semantics for extended EGs (or *DPMN process models*) is obtained by mapping an extended EG to a set of event rules, which define an *Abstract State Machine*, as proposed by Wagner (2017).

2 DEFINING OBJECT TYPES AND EVENT TYPES IN A UML CLASS DIAGRAM

Process models, such as EGs or BPMN Process Diagrams, are based on an underlying definition of the types of events, objects and complex data values they are using. In the BPMN 2.0 specification, these type definitions are provided by an XML Schema. However, it is more natural to use a visual form of type definitions, such as provided by UML Class Diagrams.

An example of a class diagram defining the types underlying the EG of Figure 1 is shown in Figure 2. It defines the object type *ServiceDesk* and the three event types *Arrival*, *ServiceStart* and *ServiceEnd*. The associations between these three event types and the object type *ServiceDesk* represent the ontological pattern that *objects participate in events*. Compuationally, they model a reference property *serviceDesk* for all three event types, such that any *Arrival*, *ServiceStart* or *ServiceEnd* event comes with a reference to a specific instance of *ServiceDesk*.

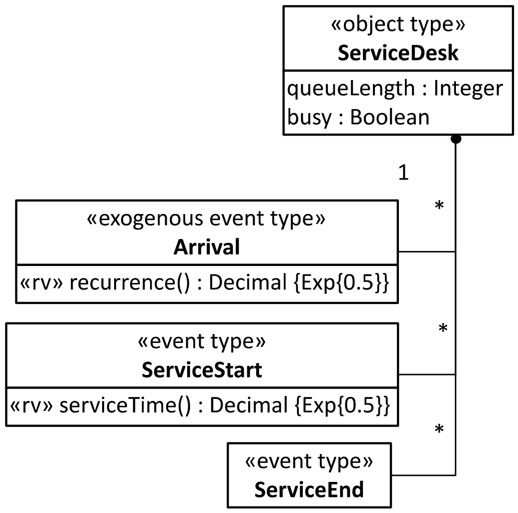

Figure 2: A class diagram defining the object and event types of the process models of Figures 6-9.

Notice that object and event types are modeled as stereotyped classes, using the UML stereotypes «object type» and «event type», and random variables are modeled as stereotyped operations, using the UML stereotype «rv», constrained to comply with a specific probability distribution, here the *Exponential Distribution* with event rate 0.5, symbolically expressed as *Exp(0.5)*.

The event type *Arrival* is an example of a type of *exogenous* events, which are not caused by any causal regularity of the system under investigation and, therefore, have to be modeled with a (typically stochastic) *recurrence* function that allows to compute the time between two occurrences of events of that type. In the original EG approach of Schruben (1983), and also in the pseudo-code of the *Arrival* event routine presented in (Pegden 2010), the event routine of an exogenous event type has to take care of creating the next event of that type. However, it is preferable to assume that this is part of the semantics of DES, that is, a discrete event simulator provides generic support for exogenous event types by means of a built-in mechanism that takes care of creating the next event whenever an event of that type is processed.

3 IMPROVING THE NOTATION OF DELAY EXPRESSIONS AND CONDITIONS

3.1 An Improved Notation for Delay Expressions

We prefer prefixing an event scheduling arrow annotation that denotes a scheduling delay with a "+" sign for providing a visual clue about its meaning, as shown in Figure 3 below. A scheduling delay expression can be formed with the help of functions, which are typically defined for a class in an underlying UML class model, such as *ServiceDesk.serviceTime()* in Figure 3.

3.2 An Improved Notation for Conditional Event Scheduling Arrows

In our notation of extended event graphs, we prefer using brackets, as in [*Q*=1], instead of parenthesis for enclosing a condition as an annotation of a conditional event scheduling arrow, as shown in Figure 3. This harmonizes the syntax of EGs with the syntax of *State Charts* (or "UML State Machines") where conditions ("guards") are expressed for conditional state transitions.

For indicating that an event scheduling arrow is conditional, a *mini-diamond* is added at its origin. This is exemplified by the arrow between the *Arrival* circle and the *ServiceStart* circle in Figure 3. In the original EG proposal of (Schruben 1983), this has been indicated by a wavy line through the middle of the arrow. The mini-diamond notation adopted from BPMN is visually simpler and better readable.

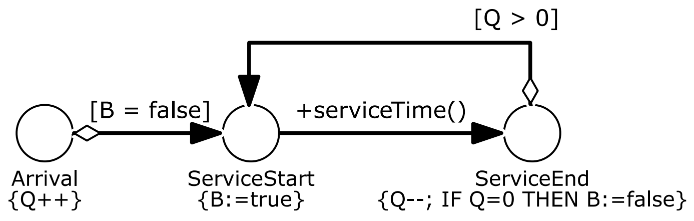

Figure 3: Using the mini-diamond notation of BPMN for conditional event scheduling arrows.

4 ADDING GATEWAYS FOR CONDITIONAL AND PARALLEL BRANCHING

We propose to extend the visual syntax of EGs by adding the Gateway symbols of BPMN for explicitly modeling the conditional and parallel branching of event scheduling.

The simplest case of conditional branching is the IF-THEN-ELSE branching pattern, also called XOR-split, shown in Figure 4. The BPMN Gateway symbol for visualizing an IF-THEN-ELSE branching pattern is called Exclusive Gateway (or XOR Gateway), rendered as a diamond filled with an “x”.

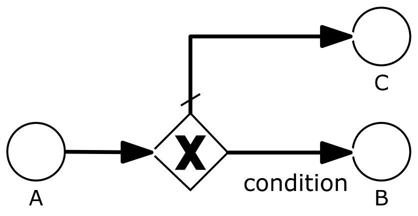

Figure 4: An Exclusive Gateway (XOR-Split).

The BPMN Gateway symbol for visualizing a parallel branching pattern is called Parallel Gateway (expressing an AND-Split), rendered as a diamond filled with a “+” sign, as shown in Figure 5.

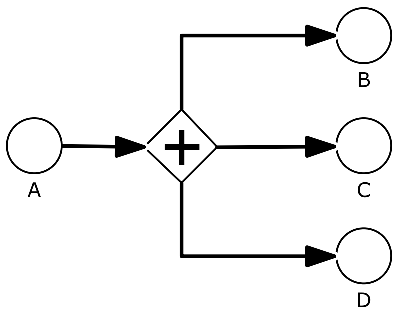

Figure 5: A Parallel Gateway (AND-Split).

In the model of Figure 5, an event of type A causes, resp. schedules, three events: a B event, a C event and a D event, possibly occurring in parallel.

5 ADDING DATA OBJECTS FOR CAPTURING THE SYSTEM STATE

In the EG of Figure 1, we can see that an event occurrence may come with two effects: it may change the system state by changing state variables, and it may lead to the scheduling of follow-up events. In the case of an *Arrival* event, the state change is the incrementation of the state variable *Q* and the follow-up event scheduled is a new *ServiceStart* event.

While modeling the state of a system in the form of a set of mathematical variables may be preferable for mathematical models, due to the simplicity of such an approach, considering the structure of objects that make up a system, and modeling the involved types of objects with their properties (and possibly operations/functions) is preferable for a computational simulation model. Modeling the state structure of a system by modeling the state structure of its objects can be achieved by making a UML class model that defines object types in the form of classes, which can be implemented by corresponding OOP (e.g., Java or C#) classes.

Based on the class model shown in Figure 2, we may attach a corresponding Data Object *sd:ServiceDesk* to the event type *Arrival* in a DPMN diagram, as shown in Figure 6.

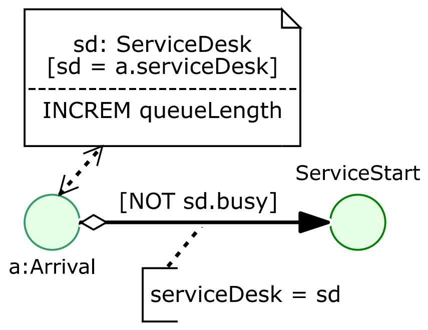

Figure 6: Attaching a Data Object *sd* of type *ServiceDesk* to the event type *Arrival*.

A DPMN Data Object associates an object variable (like *sd*) with an object type (like *ServiceDesk*) such that the object variable can be used in expressions and statements. Attaching a Data Object to an event type means that an object of the specified type participates in an event of that type. This means that in expressions and state change statements used in the event processing logic, which is often called *event routine*, expressed for that event type, the state of participating objects may be queried and changed.

Whenever an event of the given type occurs, the attached Data Object is bound to the specific object participating in the event occurrence as specified by a binding condition expressed in brackets underneath the Data Object title, such as [sd = a.serviceDesk] binding the *sd* object of type *ServiceDesk* to the *serviceDesk* object participating in the *Arrival* event *a*.

The possible state changes are specified by a state change program consisting of one or more state change statements shown in the second compartment of a Data Object rectangle. In the case of the *sd:ServiceDesk* Data Object there is only one state change statement:

INCREM sd.queueLength

This statement is intended to specify that the *queueLength* attribute of the *ServiceDesk* object *sd* is to be incremented by 1.

The diagram shown in Figure 6 represents an *event rule model* since it defines an event rule for the event type *Arrival*. An event rule specifies which state changes and which follow-up events are caused by an occurrence of an event of a certain type.

Since our information model defines three event types, we need to make two more event rule models: one for the event type *ServiceStart*, shown in Figure 7, and one for *ServiceEnd*, shown in Figure 8.

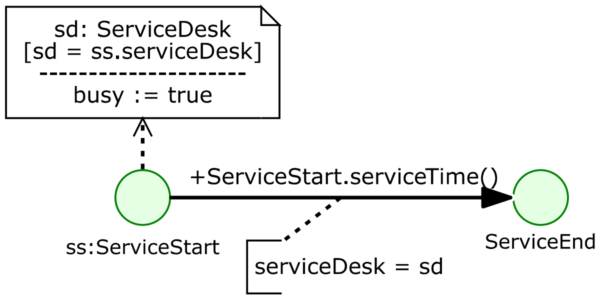

Figure 7: An event rule model for the event type *ServiceStart*.

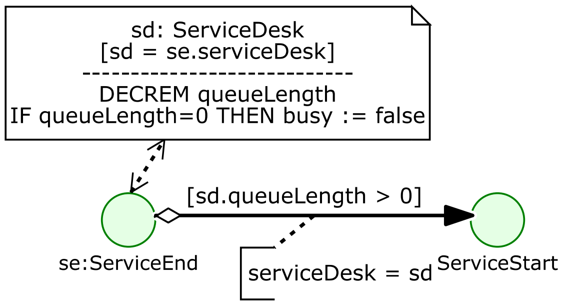

Figure 8: An event rule model for the event type *ServiceEnd*.

The event rule models shown in Figures 6-8 can be merged into a process model, as shown in Figure 9.

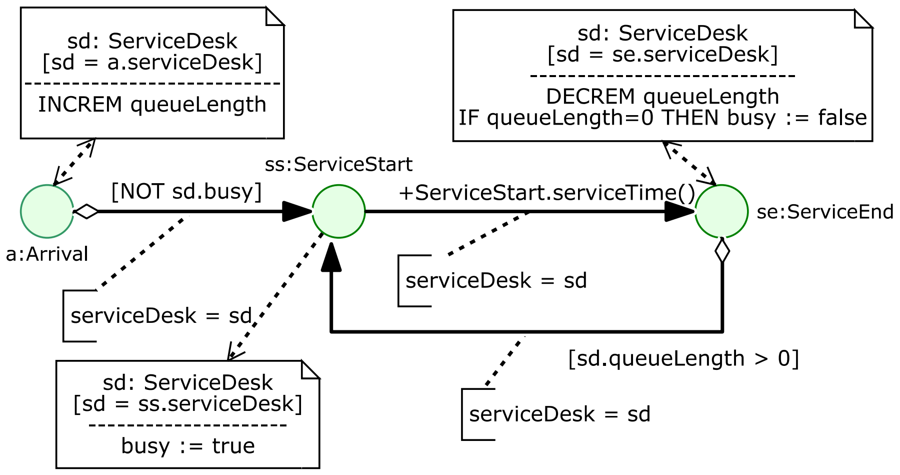

Figure 9: A process design model for the service queuing system.

The combination of a UML class model defining object and event types, such as the model shown in Figure 2, with a DPMN process model, such as the one shown in Figure 9, allows a computationally complete specification of a basic (event-based) DES model.

6 ACCOMMODATING ACTIVITIES

Conceptually, an activity is a composite event that is temporally framed by a pair of start and end events. Consequently, whenever a model contains a pair of related start and end event types, like *ServiceStart* and *ServiceEnd* in the model of Figure 9, they can be replaced with a corresponding activity type, like *ServicePerformance*, as shown in Figures 10 and 11.

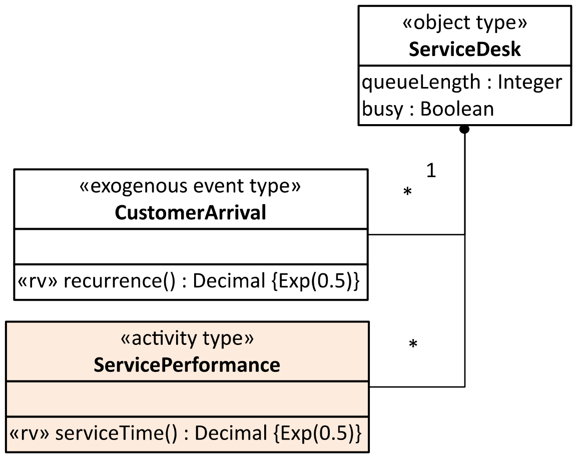

Figure 10: An information design model with an activity type *ServicePerformance*.

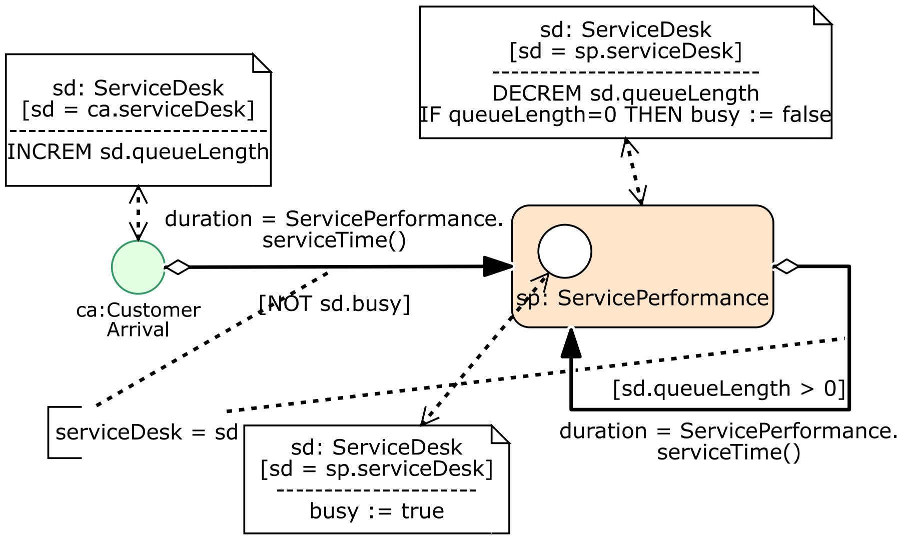

Figure 11: A process design model with an activity type *ServicePerformance*.

Notice that a fixed-duration activity is scheduled by providing an assignment of a value to the implicit activity attribute *duration*. A simulator executes this by scheduling an implicit activity start event and an implicit activity end event that is delayed by the time provided by the *duration* attribute.

Several more advanced issues related to the DPMN concept of activities will be discussed in a follow-up paper: (1) modeling open-duration activities, where the time when an activity ends is not known when it starts, with a special activity end scheduling arrow; (2) defining resource dependencies for activity types and their execution semantics based on resource pools and resource assignments; (3) modeling processing activities in GPSS/SIMAN/Arena-style processing networks.

7 FROM BPMN TO DPMN

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.

The BPMN Process Diagram language has several semantic issues and is not expressive enough for making computational process design models that can be used both for designing DES models and as a general basis for coding platform-specific simulation models. In particular, the modeling element of *Sequence Flow* arrows is semantically overloaded. They do have different meanings depending on which elements ("flow objects") they connect. For instance, a Sequence Flow leading to a Gateway diamond has a completely different meaning than a Sequence Flow leading to an Event circle. And while BPMN Sequence Flow arrows pointing to an Event circle do have an implicit meaning of causation, BPMN does not define any computational semantics for them, as opposed to Event Graphs.

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 need to adapt the language of BPMN Process Diagrams for the purpose of simulation design modeling where a process model must represent a computationally complete process specification. While we can use large parts of its vocabulary, visual syntax and informal semantics, we need to modify them for a number of modeling elements. The resulting BPMN variant is called *Discrete Event Process Modeling Notation (DPMN)*, see (Wagner, 2018a).

DPMN is a BPMN-based diagram language for making (computational) process design models for discrete event simulation. It combines the intuitive flowchart modeling style of BPMN with the rigorous semantics provided by the event scheduling arrows of Event Graphs (Schruben 1983) and the event rules of the *Object Event Modeling and Simulation* paradigm (Wagner 2018b).

DPMN adopts and adapts the syntax and semantics of BPMN in the following way:

1. A DPMN diagram has an underlying UML class diagram defining its (object and event) types.
2. DPMN Sequence Flow arrows pointing to an event circle denote *event scheduling* control flows. They must be annotated by event attribute assignments for creating/scheduling a new event.
3. DPMN has three special forms of Text Annotation:
    1. Text Annotations attached to Event circles for declaring event rule variables,
    2. Text Annotations attached to Sequence Flow arrows pointing to Event circles for the occurrence time or delay of the events to be scheduled,
    3. Text Annotations attached to Sequence Flow arrows pointing to Event circles for event attribute assignments.
4. DPMN has an extended form of Data Object visually rendered as rectangles with two compartments:
    1. a first compartment showing an object variable name and an object type name separated by a colon, together with a binding of the object variable to a specific object;
    2. a second compartment containing a block of state change statements (such as attribute value assignments).

8 OPERATIONAL SEMANTICS OF DPMN DIAGRAMS

An operational (transition system) semantics for DPMN (and EGs) is obtained by decomposing a DPMN process model to a set of event rule models each defining an event rule, which together with the underlying class model define an *Object Event Simulation (OES)* model. An OES model, together with an initial state consisting of initial objects and events, defines a (typically non-deterministic) transition system corresponding to an *abstract state machine (ASM)* in the sense of Gurevich (1985), as shown by Wagner (2017).

A DPMN diagram represents an *event rule model* if

1. it has exactly one Event circle with no incoming arrows (corresponding to what is called a "Start Event" in BPMN),
2. it has no event scheduling path with a length greater than 1, where the length is defined as the number of Event circles in a path minus 1.

Examples of DPMN diagrams representing event rule models are shown in Figures 6-8.

A DPMN process model is a composition of event rule models, like the model of Figure 9, which is composed of the event rule models of Figures 6-8.

For instance, the event rule models of Figures 6-8 define the following event rules:

Rule name	**ON (event expression)**	**DO (event routine)**

r_{Arr}	Arrival(*sd*) @ *t*	E' := {}
		Δ := { INCREM *sd.queueLength*}
		IF *sd.queueLength* = 1
		THEN E' := $E' \cup$ { ServiceStart @ *t*+1}
		RETURN ⟨ Δ, E' ⟩

r_{Start}	ServiceStart(*sd*) @ *t*	E' := { ServiceEnd @ (*t* + ServiceStart.serviceTime())}
		Δ := {}
		RETURN ⟨ Δ, E' ⟩

r_{End}	ServiceEnd(*sd*) @ *t*	E' := {}
		Δ := { DECREM *sd.queueLength*}
		IF *sd.queueLength* > 0
		THEN E' := $E' \cup$ { ServiceStart @ *t*+1}
		RETURN ⟨ Δ, E' ⟩

An event rule associates an event expression with an *event routine F*:

$$\textbf{ON}\ \mathrm{E}(\underline{x})@t\ \textbf{DO}\ F(t, \underline{x}),$$

where the event expression $\mathrm{E}(\underline{x})@t$ specifies the type E of events that trigger the rule, and $F(t, \underline{x})$ is a function call expression for computing a set of *state changes* Δ and a set of *follow-up events E'*, based on the event parameter values $\underline{x}$, the event's occurrence time *t* and the current system state, which is accessed in the event routine *F* for testing conditions expressed in terms of object states.

A *Future Events List (FEL)* is a set of ground event expressions partially ordered by their occurrence times, which represent future time instants either from a discrete or a continuous model of time. The partial order implies the possibility of simultaneous events, such as { ServiceEnd@4, Arrival@4}.

A *system state S* is a set of objects, each in a particular state, where an object state is a set of property-value slots. A *simulation state* is a pair ⟨ *S*, *FEL* ⟩ consisting of a system state *S* and a future events list *FEL*.

A DPMN process model with an underlying UML class model can be directly mapped to an *Object Event Simulation (OES)* model, which is a triple ⟨ *OT*, *ET*, *R* ⟩ where

1. *OT* is a set of object types defining the state structure of the system;
2. *ET* is a set of event types;
3. *R* is a set of event rules (expressed in terms of *OT* and *ET*) defining the dynamics of the system, such that *R* contains a rule for each event type in *ET*.

While *OT* and *ET* are defined by the UML class model, *R* is defined by the DPMN process model.

We show how to express our running example model of a service desk system in the form of an OES model. The set of object types contains just one element with one attribute:

OT = { ServiceDesk(*queueLength*: Integer) }

All three event types have a reference property for referencing the service desk that participates in the event (that is, the service desk where the event occurs):

ET = { Arrival(*sd*: ServiceDesk | recurrence(): Decimal),
ServiceStart(*sd*: ServiceDesk | serviceTime(): Decimal)
ServiceEnd(*sd*: ServiceDesk) }

The set of event rules contains the three rules shown in the table above: R = { r_{Arr}, r_{Start}, r_{End} }.

Such a model, together with an initial simulation state (specifying initial objects and initial events), defines an OES system, which is a transition system where

1. possible *system states* are defined by *OT*: each object type defines a *state space* for its instances (essentially, in the form of a Cartesian Product over the ranges of all properties minus those value combinations that are prohibited by the object type's integrity constraints);
2. *transitions* are provided by event occurrences triggering event rules from *R* that change the simulation state through changing the system state (by changing the states of affected objects) and the FEL (by removing current events and adding follow-up events).

For further explanations how a triggered event rule maps a system state to a set of state changes and a set of follow-up events, and how the event rule set *R* works as a transition function mapping a simulation state to a successor state, see (Wagner 2017).

9 RELATED WORK

As reported in (Rosenthal et al. 2018), the BPM research community has investigated various approaches to *business process simulation* based on BPMN process models, typically requiring to transform a BPMN model to an established simulation formalism/platform, such as *Coloured Petri Nets, DEVS* or *DESMO-J*. However, there has been no attempt yet, neither in BPM research nor in DES research, to combine BPMN with Event Graphs.

10 CONCLUSIONS

We have shown how Event Graphs can be modified and extended by (a) using mini-diamonds for expressing conditional event scheduling arrows, (b) adding Gateways for conditional and parallel branching, (c) adding Data Objects for expressing the state of a system in an object-oriented manner in combination with a UML class model for defining object types and event types, and (d) adding Activities for bundling pairs of activity start and end events. The resulting modeling approach, with its language DPMN, is based on the two modeling standards UML and BPMN and on the classical Event Graph simulation language. It allows making visual object-oriented simulation design models that can be implemented with various (object-oriented) simulation technologies. The potential of DPMN for modeling Arena-style Processing Networks with Entry, Processing and Exit nodes has to be investigated in future work.

REFERENCES

Dahl, O.-J., and K. Nygaard. 1966. "Simula – an ALGOL-Based Simulation Language". *Communications of the ACM* 9(9), pp. 671–678.

Fujimoto, R., S. Cornford, C. Paredis, and P. Zimmerman. 2016. *Research Challenges in Modeling & Simulation for Engineering Complex Systems*. Workshop report, January 13-14, 2016, National Science Foundation, Arlington, Virginia, USA. Available from https://www.imagwiki.nibib.nih.gov/sites/default/files/FullReport-Final.pdf

Gurevich, Y. 1985. "A New Thesis". *American Mathematical Society Abstracts*, page 317, August 1985.

Markowitz, H., B. Hausner, and H. Karr. 1962. *SIMSCRIPT: A Simulation Programming Language*. Report No. RM-3310-PR. Santa Monica, CA: The RAND Corporation.

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. Yucesan, 643−651. Piscataway, New Jersey: IEEE.

Rosenthal, K., B. Ternes and S. Strecker. 2018. "Business Process Simulation: A Review". In *Proc. of Twenty-Sixth European Conference on Information Systems (ECIS2018)*, Portsmouth, UK.

Schruben, L.W. 1983. "Simulation Modeling with Event Graphs". *Communications of the ACM* 26, pp. 957-963.

Wagner, G. 2017. "An Abstract State Machine Semantics for Discrete Event Simulation". In *Proceedings of the 2017 Winter Simulation Conference*. Piscataway, NJ: IEEE. Available from https://www.informs-sim.org/wsc17papers/includes/files/056.pdf

Wagner, G. 2018a. *Discrete Event Process Modeling Notation (DPMN)*. Language Reference. Available from https://sim4edu.com/reading/DPMN

Wagner, G. 2018b. "Information and Process Modeling for Simulation – Part I: Objects and Events". *Journal of Simulation Engineering*, vol. 1, 2018. Available from https://articles.jsime.org/1/1.

AUTHOR BIOGRAPHY

GERD WAGNER is Professor of Internet Technology at the Dept. of Informatics, Brandenburg University of Technology, Germany, and Adjunct Associate Professor at the Dept. of Modeling, Simulation and Visualization Engineering, Old Dominion University, Norfolk, VA, USA. His research interests include modeling and simulation, foundational ontologies and web engineering. He has more than 150 refereed research publications in these and other areas, with a Google Scholar citation count of more than 5000 and an h-index of 40. In recent years, he has been focusing his research on the development of a general discrete event modeling and simulation framework, called *Object Event Modeling & Simulation (OEM&S)*, which has been implemented by the *OESjs* framework available on https://sim4edu.com. His email address is G.Wagner@b-tu.de.