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

AN ABSTRACT STATE MACHINE SEMANTICS FOR DISCRETE EVENT SIMULATION

Gerd Wagner

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

ABSTRACT

We define an operational (transition system) semantics for the two most basic forms of Discrete Event Simulation (DES): event-based simulation (without objects) and object-event simulation. We show that under our operational semantics, DES models correspond to a certain form of *abstract state machines (ASMs)* such that the *Future Event List (FEL)* is part of the transition system state and the transition function is based on *event routines*. Unlike other formalisms proposed for DES (such as Petri Nets or DEVS), our ASM semantics takes all basic DES concepts (like event types and the FEL) into consideration and allows for expressive transition system states representing the objects, properties, relations and functions of the evolving possible worlds of a simulation run. As a direct formal semantics of DES, it provides a basis for comparing, and explaining design choices in, different DES approaches.

1 INTRODUCTION

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 a discrete dynamic system by modeling its state as being composed of *state variables*, and modeling its dynamics by modeling the events that are responsible for its state changes. There is, however, no generally accepted definition of DES. Simulation textbooks, like (Banks et al. 2005), and tutorials, like (Ingals 2008), avoid defining the term "DES" in a precise way.

Pegden (2010) characterizes the various forms of DES in terms of three paradigms ("worldviews"), which are discussed in Section 2. In this paper, we propose an operational semantics, based on the concept of transition systems, for two of these paradigms: the event worldview (without objects) and the object worldview (in the form of the *object-event worldview*). Because Pegden's third paradigm (originally called "process worldview", but better called *processing network worldview*, since it is not about a general concept of processes, but about processing, or queueing, networks) can be viewed as a conservative extension of the object-event worldview, it is possible to extend our formal semantics, such that it provides a direct formal semantics of processing network simulations.

According to our semantics, a DES model, together with an initial state, defines a (typically nondeterministic) transition system corresponding to an *abstract state machine (ASM)* in the sense of Gurevich (1985). Despite their name, ASMs allow non-abstract transition systems with a rich state structure defined by a predicate logic language with predicates, function symbols and constant symbols, as explained by Reisig (2008). This implies that they also support state structures defined by object-oriented (OO) languages where object types and datatypes (classes) define unary predicates/relations and properties define binary relations and functions.

Unlike classical formalisms proposed for DES, like Petri Nets (Jensen et al. 2007) or DEVS (Zeigler et al. 2000), which do not include events and an FEL as first-class citizens and do not consider states composed of objects, our ASM semantics of DES accounts for the basic DES concepts (such as events and an FEL) and allows for "natural" transition system states representing the events, objects, properties, relations and functions of possible worlds (being ∑-structures) as interpretations of a predicate logical language (based on a signature ∑) in the sense of Tarski's mathematical theory of truth (Tarski 1954).

We also briefly argue that our ASM semantics of DES provides the basis for improving/extending the formalism and diagram language of Event Graphs by adding the concepts of (1) conditional branching through event rules and (2) objects as state components.

2 DISCRETE EVENT SIMULATION PARADIGMS

Pegden (2010) explains that the 50 year history of DES has been shaped by three fundamental paradigms: Markowitz, Hausner, and Karr (1962) pioneered the *event worldview* with *SIMSCRIPT*, Gordon (1961) pioneered the *processing network worldview* with *GPSS*, and Dahl and Nygaard (1967) pioneered the *object worldview* with *Simula*. Notice that we have changed Pegden's original name "process worldview" to "processing network worldview" because this paradigm is not based on a general concept of processes, but rather on a special concept of *processing processes* where entities are subject to processing steps performed at the nodes of a (queueing) network.

We illustrate the proposed formal semantics of DES with the help of an example. We model a system of one or more service desks, each of them having its own queue, as a discrete event system characterized by the following narrative:

1. Customers arrive at a service desk at random times.
2. If there is no other customer in front of them, and the service desk is available, they are served immediately, otherwise they have to queue up in a waiting line.
3. The duration of services varies, depending on the individual case.
4. When a service is completed, the customer departs and the next customer is served, if there is still any customer in the queue.

2.1 The Event Worldview

According to Pegden (2010), the most fundamental DES paradigm is the *event worldview*, where the system under investigation is viewed as a series of instantaneous events that change its state over time. The modeler "defines the events in the system and models the state changes that take place when those events occur". More precisely, the modeler defines the *types* of events that cause *state changes* and/or *follow-up events*.

Pegden also explains that in the event worldview,

1. a simulation creates events that are supposed to occur in the future (called *future events*)
2. future events are scheduled (using an *event scheduling* mechanism),
3. time advances to the time of the next event (*next-event time progression*)
4. the series of events corresponds to a sequence of state transitions of a *transition system* where the "transition logic" of each event type is specified in the form of a procedure definition.

The "transition logic" procedures defined for all event types of an event-based simulation model are often called *event routines*. They can be expressed at an abstract level, e.g., using pseudo code as in (Pegden 2010), or in a simulation or programming language. In an object-oriented approach, it is natural to define an event routine as a method of the class defining the event type.

Pegden does not make any attempt to clarify the philosophical nature of (types of) events and their "transition logic". We have argued in (Guizzardi and Wagner 2013) that, philosophically, (1) all events have *participants* (objects that participate in them); (2) the combination of an event type and its "transition logic" procedure (or *event routine*) amounts to an ***event rule*** of the form ON *event* DO *procedure*, which captures the type of dispositions inherent in the participants of events of that type.

Event rules model the causal regularities of a discrete dynamic system by defining what happens when an event (of a certain type) occurs, or, more specifically, which state changes and which follow-up events are caused by an event of that type. Event rules can be expressed at an abstract level, e.g., using pseudo-code or process models as shown in (Wagner et al. 2016), or in a simulation or programming language. For instance, Table 1 shows the two event rules defining the transition logic of a simple model of the service desk system, expressed in pseudo-code.

Table 1: Event rule examples.

Event rule name	**ON (event type)**	**DO (event routine)**

r_{Arr}	Arrival @ t	SCHEDULE Arrival @ (t + recurrence())
		INCREMENT queueLength
		IF queueLength = 1
		THEN SCHEDULE Departure @ (t + serviceDuration())

r_{Dep}	Departure @ t	DECREMENT queueLength
		IF queueLength > 0
		THEN SCHEDULE Departure @ (t + serviceDuration())

The model is based on two types of events: *Arrival* and *Departure* (for simplicity, it abstracts away from the fact that both have a service desk as their participant). It has only one state variable: *queueLength*. Notice that both event routines invoke the function *serviceDuration*(), which implements a random variable modeling the variations in the duration of service activities.

2.1 The Object-Event Worldview

The event worldview is ontologically incomplete because the real world essentially consists of objects and events, as argued philosophically by Casati and Varzi (2015), and as implied by the object worldview pioneered by Dahl and Nygaard (1967) with their influential simulation programming language *Simula*. The main concepts of Simula are classes (with subtyping), objects and procedures, while events are only implicitly available in the form of procedure calls.

Combining the event and the object worldviews results in the *object-event worldview*, in which both objects and events are first-class citizens. We define a DES formalism for the object-event worldview in Section 4.

2.2 The Processing Network Worldview

In this DES paradigm, the system under investigation is described as a processing network where "passive 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 *activities*, possibly requiring *resources* and inducing *queues* of work objects waiting for the availability of resources. This approach, mainly pioneered by *GPSS*, *SIMAN* and *Arena*, is still widely used today.

The concepts of the processing network worldview can be defined on the basis of objects and events. It can therefore be considered as a conservative extension of the object-event worldview.

3 A TRANSITION SYSTEM SEMANTICS FOR THE EVENT WORLDVIEW

3.1 Basic Concepts

The base concepts of the event worldview are:

1. state variables,

2. event types,
3. event expressions,
4. event routines,
5. future events lists (FEL).

A ***state variable*** is declared with a name and a range, which is a datatype defining its possible values.

An ***event type*** is defined in the form of a class: with a name, a set of property declarations and a set of method definitions, which together define the *signature* of the event type.

An ***event expression*** is a term $E(\underline{x})@t$ where

1. E is an event type,
2. *t* is a parameter for the occurrence time of events,
3. $\underline{x}$ is a (possibly empty) list of event parameters $x_1, x_2, \ldots, x_n$ according to the signature of the event type E

For instance, Arrival@*t* is an event expression for describing Arrival events where the signature of the event type Arrival is empty, so there are no event parameters, and the parameter *t* denotes the arrival time (more precisely, the occurrence time of the Arrival event). An individual event of type E is a *ground* event expression, $e = E(\underline{v})@i$, where the event parameter list $\underline{x}$ and the occurrence time parameter *t* have been instantiated with a corresponding value list $\underline{v}$ and a specific time instant *i*. For instance, Arrival@1 is a ground event expression representing an individual Arrival event.

An ***event routine*** is a procedure that essentially computes state changes and follow-up events, possibly based on conditions on the current state. In practice, state changes are often directly performed by immediately updating the state variables concerned, and follow-up events are immediately scheduled by adding them to the FEL, as in the rules described in Table 1. For our formal semantics, we assume that an event routine is a pure function that computes state changes and follow-up events, but does not apply them, as in the rules described in Table 2.

Table 2: Expressing event routines as pure functions that compute state changes and follow-up events.

Event rule name	ON (event expression)	DO (event routine)

r_{Arr}	Arrival @ t	E' := { Arrival @ (t + recurrence())}
		Δ := { INCREMENT queueLength}
		IF queueLength = 0
		THEN E' := E' ∪ { Departure @ (t + serviceDuration())}
		RETURN $\langle \Delta, E' \rangle$

r_{Dep}	Departure @ t	E' := {}
		Δ := { DECREMENT queueLength}
		IF queueLength > 1
		THEN E' := E' ∪ { Departure @ (t + serviceDuration())}
		RETURN $\langle \Delta, E' \rangle$

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

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

where the event expression $E(\underline{x})@t$ 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* and a set of *follow-up events*, 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 state variables.

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, as in the example { Departure@4, Arrival@4}.

A ***simulation state*** is a pair ⟨ *S*, *FEL* ⟩ consisting of a system state *S* and a future events list *FEL*.

3.2 Exogenous Events

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 simple DES approaches, and also in the pseudo-code of the Arrival event routine presented in (Pegden 2010) and in our event rule tables, the event routine has to take care of creating the next event of an exogenous event type, but it is preferable if 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.

3.3 Event-Based Simulation

An ***event-based simulation (ES)*** model is a triple ⟨ SV, ET, R ⟩ where

1. SV is a set of state variable declarations defining the structure of possible system states;
2. ET is a set of event type definitions;
3. R is a set of event rules expressed in terms of SV and ET.

We show how to express the example model of a simple service desk system as an ES model. The set of state variables is a singleton:

SV = { queueLength: NonNegativeInteger}

There are two event types, both having an empty signature:

ET = { Arrival(), Departure()}

And there are two event rules:

R = { r_{Arr}, r_{Dep}}

which are defined as in Table 1 above.

Such a model, together with an initial state (specifying initial values for state variables and initial events), defines an ES system, which is a transition system where

1. system states are defined by value assignments for the state variables,
2. transitions are provided by event occurrences triggering event rules that change the simulation state through changing the system state (by changing the values of affected state variables) and the FEL (by adding follow-up events).

Whenever the transitions of an ES system involve computations based on random numbers (if the simulation model contains random variables), the transition system defined is non-deterministic.

For instance, assuming that the initial system state is S_0 = {queueLength: 0}, and there is an initial event {Arrival@1}, then, as a consequence of applying r_{Arr}, there is a system state change {queueLength := 1} and, assuming a random service time of 2 time units (as a sample from the underlying probability distribution function), a follow-up event Departure@3, which has to be scheduled along with the next Arrival event, say Arrival@3 (with a random inter-arrival time of 2), because Arrival is an exogenous event type (with a random recurrence). Consequently, the next system state is S_1 = {queueLength: 1}.

We need to distinguish between the *system state*, like S_0 = {queueLength: 0}, which is the state of the simulated system, and the *simulation state*, which adds the FEL to the system state, like

$\boldsymbol{S}_0 = \langle$ { queueLength: 0}, { Arrival@1} $\rangle$

$\boldsymbol{S}_1 = \langle$ { queueLength: 1}, { Arrival@2, Departure@3} $\rangle$

Doing one more step, the next transition is given by the next event Arrival@2 again triggering r_{Arr}, which leads to

$\boldsymbol{S}_2 = \langle$ { queueLength: 2}, { Departure@3, Arrival@4} $\rangle$

In this way, we get a succession of states $\boldsymbol{S}_0 \rightarrow \boldsymbol{S}_1 \rightarrow \boldsymbol{S}_2 \rightarrow \ldots$ as a history of the transition system defined by the ES model.

3.4 Event Rules as Functions

An event rule r = **ON** E$(\underline{x})@t$ **DO** $F(t, \underline{x})$ can be considered as a 2-step function that, in the first step, maps an event e = E$(\underline{v})@i$ to a parameter-free state change function $r_e = F(i, \underline{v})$, which maps a system state to a pair $\langle \Delta, E' \rangle$ of system state changes Δ and follow-up events E'. When the parameters t and $\underline{x}$ of $F(t, \underline{x})$ are replaced by the values i and $\underline{v}$ provided by a ground event expression E$(\underline{v})@i$, we also simply write $F_{i,\underline{v}}$ instead of $F(i, \underline{v})$ for the resulting parameter-free state change function.

We say that an event rule r is *triggered* by an event e when the event's type is the same as the rule's event type. When r is triggered by e, we can form the state change function $r_e = F_{i,\underline{v}}$ and apply it to a system state S by mapping it to a set of system state changes Δ and a set of follow-up events E':

$$r_e(S) = F_{i,\underline{v}}(S) = \langle \Delta, E' \rangle$$

We can illustrate this with the help of our running example. Consider the rule r_{Arr} defined in Table 1 above triggered by the event Arrival@1 in state S_0 = {queueLength: 0}. The resulting state change function F_1 defined by the corresponding event routine from Table 1 maps S_0 to the set of state changes Δ = { INCREMENT queueLength} and the set of follow-up events E' = {Departure@3}. We show how the pair $\langle \Delta, E' \rangle$ amounts to a transition of the simulation state in the next section.

In ES, a system state change is an update of one or more state variables. Such an update is specified in the form of an assignment where the right-hand side is an expression that may involve state variables. For instance, the state change `INCREMENT queueLength` is equivalent to the assignment `queueLength := queueLength + 1`.

In general, there may be situations, where we have several concurrent events, that is, there may be two or more events occurring at the same (next-event) time. Therefore, we need to explain how to apply a set of rules R_E triggered by a set of events E, even if both sets are singletons in many cases.

The rule set R of an ES model can also be considered as a 2-step function that, in the first step, maps a set of events E to a state change function R_E, which maps a system state to a pair $\langle \Delta, E' \rangle$ of state changes Δ and follow-up events E'.

For a given set of events E and a rule set R, we can form the set of state change functions obtained from rules triggered by events from E:

$$R_E = \{ r_e : r \in R \;\&\; e \in E \;\&\; e \text{ triggers } r\}$$

Notice that the elements C of R_E are parameter-free state change functions, which can be applied as a block, in parallel, to a system state S:

$$R_E(S) = \langle \Delta, E' \rangle$$

with

$$\Delta = \bigcup \{ \Delta_C : C \in R_E \;\&\; C(S) = \langle \Delta_C, E'_C \rangle \}$$
$$E' = \bigcup \{ E'_C : C \in R_E \;\&\; C(S) = \langle \Delta_C, E'_C \rangle \}$$

Notice that when forming the union of all state changes brought about by applying rules from R_E, and likewise when forming the union of all follow-up events created by applying rules from R_E, the order of rule applications does not matter because they do not affect the applicability of each other, so any selection function for choosing rules from R_E and applying them sequentially will do, and they could also be applied simultaneously if such a parallel computation is supported.

However, computing a set of state changes Δ raises the question if this set is, in some sense, consistent. A simple, but too restrictive, notion of consistent state changes would require that if Δ contains two or more updates of the same state variable, all of them must be equivalent (effectively assigning the same value). A more liberal notion just requires that if Δ contains two or more updates of the same state variable, their collective application must result in the same value for it, no matter in which order they are applied.

If Δ contains inconsistent updates for a state variable, this may be a bug or a feature of the simulation model. If it is not a bug, a conflict resolution policy is needed. The simplest policy is ignoring, or discarding, all inconsistent updates. Another common conflict resolution policy is based on assigning priorities to event rules.

Consider again our running example with a system state *S* = {queueLength: 1} and the set of next events *N* = {Arrival@4, Departure@4}. Then, R_N consists of the two parameter-free change functions:

1. F_1: function () {Δ := { INCREMENT queueLength}; IF queueLength = 0 THEN *E′* := { Departure @ (4 + serviceDuration())}; RETURN ⟨ Δ, *E′* ⟩ }
2. F_2: function () {Δ := { DECREMENT queueLength}; IF queueLength > 1 THEN *E′* := { Departure @ (4 + serviceDuration())}; RETURN ⟨ Δ, *E′* ⟩}

No matter in which order we apply F_1 and F_2, forming the union of their state changes always results in Δ = {}, because the incrementation and decrementation of the variable *queueLength* neutralize each other, and forming the union of their follow-up events always results in *E′* = { Departure@(4+d)} where d is the random value returned by the *serviceDuration* function.

3.5 A Set of Event Rules as a Transition Function

We show that the event rule set R of an ES model ⟨ SV, ET, R ⟩ defines a transition function that maps a simulation state ⟨ *S*, *FEL* ⟩ to a successor state ⟨ *S′*, *FEL′* ⟩ in 3 steps:

1. R maps the set of next events *N* extracted from the *FEL* to a set R_N of state change functions of rules triggered by one of the next events from *N*.
2. R_N maps the current system state *S* to a set of state changes Δ and a set of follow-up events *E′*.
3. The pair ⟨ Δ, *E′* ⟩ amounts to a transition of the current simulation state ⟨ *S*, *FEL* ⟩ by applying the updates from Δ to *S* yielding *S’* and by removing *N* from *FEL* and adding *E′*.

We have already explained how to obtain R_N from R and how to apply R_N to *S* for getting ⟨ Δ, *E′* ⟩ in the previous subsection, so we only need to provide more explanation for the last step: processing ⟨ Δ, *E′* ⟩ for obtaining the next simulation state ⟨ *S′*, *FEL′* ⟩.

We use the symbol Upd for denoting an update operation that takes a system state *S* and a set of state changes Δ, and returns an updated system state Upd(*S,* Δ). When the system state consists of state variables, the update operation simply performs assignments. Using this operation, we can define the third step of the simulation state transition function with two sub-steps in the following way:

a) $S' = \text{Upd}(S, \Delta)$
b) $FEL' = FEL - N \cup E'$

This completes our definition of how the event rule set R of an ES model works as a transition function that computes the successor state of a simulation state:

$$\mathrm{R}(\langle S, FEL \rangle) = \langle S', FEL' \rangle$$

such that for a given initial simulation state $\boldsymbol{S}_0 = \langle S_0, FEL_0 \rangle$, we obtain a succession of states

$$\boldsymbol{S}_0 \rightarrow \boldsymbol{S}_1 \rightarrow \boldsymbol{S}_2 \rightarrow \ldots$$

by iteratively applying R:

$$\boldsymbol{S}_{\mathrm{i+1}} = \mathrm{R}(\boldsymbol{S}_{\mathrm{i}})$$

Consider again our running example. In simple cases we do not have more than one next event, so R_N is a singleton and we do not have to apply more than one rule at a time. For instance, when

$\boldsymbol{S}_1 = \langle\{$ queueLength: 1$\}$, $\{$ Arrival@2, Departure@3$\}\rangle$

There is only one next event: Arrival@2, so we do not have to form a set of applicable rules, but can immediately apply the rule triggered by Arrival@2 for obtaining a set of system state changes and a set of follow-up events:

$\mathrm{r}_{Arr}(\boldsymbol{S}_1) = \langle\{$ queueLength := 2$\}$, $\{$ Arrival@4$\}\rangle$

Now consider a simulation state where we have more than one next event, like the following one:

$$\boldsymbol{S}_3 = \langle\{\text{ queueLength: 1}\}, \{\text{ Arrival@4, Departure@4}\}\rangle$$

We obtain

$$\mathrm{R}(\boldsymbol{S}_3) = \langle\{\text{ queueLength: 1}\}, \{\text{ Arrival@5, Departure@6}\}\rangle$$

assuming a random inter-arrival time sample of 1 and a random service duration sample of 2.

4 A TRANSITION SYSTEM SEMANTICS FOR THE OBJECT-EVENT WORLDVIEW

The object-event worldview is obtained from the event worldview by replacing the definition of system states given by state variables with system states given by a set of objects of certain types. It is a conservative extension of the event worldview since the attributes of objects can be considered as state variables.

4.1 Basic Concepts

The base concepts of the object-event worldview are:

1. object types,
2. event types,
3. event expressions,
4. event routines,
5. future events lists (FEL).

Both *object types* and *event types* are defined in the form of classes: with a name, a set of properties and a set of methods, which together define their *signature*.

A *property declaration* defines the property's name and its *range*, which is either a datatype or an object type. When the range of a property is a datatype, we call it an *attribute*; otherwise, when its range is an object type, we call it a *reference property*. We make the simplifying assumption that all properties are functional (single-valued), so they assign instances of their range to instances of their *domain* (the object type for which they are defined).

Event expressions, event routines, event rules and the FEL are defined in the same way as in the event worldview. However, event routines now have to deal with a more complex system state structure defined by the set of object types of the object-event simulation model.

4.2 Object-Event Simulation

Object-Event Simulation (OES) is a DES paradigm based on the object-event worldview. An OES model 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.

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: NonNegativeInteger) }

The set of event types contains, as before, two event types, which now 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), Departure(sd: ServiceDesk)}

And, as before, there are two event rules:

R = { r_{Arr}, r_{Dep}}

which are defined as in Table 3 below.

Table 3: Event routines using the classes *ServiceDesk*, *Arrival* and *Departure*.

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

r_{Arr}	Arrival(sd) @ t	*E′* := { Arrival(sd) @ (t + Arrival.recurrence())}
		Δ := { INCREMENT sd.queueLength}
		IF sd.queueLength = 0
		THEN *E′* := *E′* ∪ { Departure @ (t + ServiceDesk.serviceDuration())}
		RETURN ⟨ Δ, *E′* ⟩

r_{Dep}	Departure(sd) @ t	*E′* := {}
		Δ := { DECREMENT sd.queueLength}
		IF sd.queueLength > 1
		THEN *E′* := *E′* ∪ { Departure @ (t + ServiceDesk.serviceDuration())}
		RETURN ⟨ Δ, *E′* ⟩

Compared to the event rules of the ES model defined in Table 2, there are three differences:

1. The event expressions *Arrival(sd)@t* and *Departure(sd)@t* do not only have a parameter *t* for the occurrence time, but, in addition, a parameter *sd* for referencing the service desk object where the event occurs. This facilitates extensions of the simple model by adding further service desks.
2. While the state variable *queueLength* is defined as a global variable in the ES model, it is defined as a an attribute (a variable associated with an object) in the OES model. This facilitates extensions of the simple model by adding further service desks, each with its own queue.
3. The functions *recurrence()* and *serviceDuration()* are now expressed as class-level ("static") methods of the classes to which they logically belong, *Arrival* and *ServiceDesk*.

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

1. a system state is given by the union of all property-value slots of all objects;
2. transitions are provided by event occurrences triggering event rules that change the simulation state through changing the system state (by changing the states of affected objects) and the FEL (by adding follow-up events).

For instance, assuming that the initial state is S_0 = {queueLength: 0}, and there is an initial event {Arrival@1}, then, as a consequence of applying r_{Arr}, there is a state change {queueLength := 1} and, assuming a random service time sample of 2 time units, a follow-up event Departure@3, which has to be scheduled along with the next Arrival event, say Arrival@3 (with a random inter-arrival time sample of 2), because Arrival is an exogenous event type (with a random recurrence). Consequently, the next state is S_1 = {queueLength: 1}.

5 ES AND OES SYSTEMS AS ABSTRACT STATE MACHINES

We show in a sketchy manner that the simulation systems defined by ES models and OES models are special types of *abstract state machines (ASMs)* in the sense of Gurevich (1985). An elaborated formal treatment would require too much space and would not necessarily be more accessible to those readers that are not familiar with Tarski's model theory (Tarski 1954).

ASMs are transition systems with ∑-structures ("models") as states and sets of update rules as transitions. According to Gurevich, they provide "a computation model that is more powerful and more universal than standard computation models".

ASMs have been applied in computer science for formal specification in various domains such as hardware and software architectures, protocols, and programming languages (Börger and Stärk 2003). A good introduction to ASMs is provided by Reisig (2008).

An ES model defines a signature ∑ in the following way:

1. Its set of state variables defines a set of constant symbols.
2. Its set of event types *E* in ET defines a set of unary predicates, and each property defined for an event type *E* defines a function from the population of *E* to the range of the property.

An ES state consists of a value assignment for the set of state variables corresponding to an interpretation of a set of constants by a ∑-structure, and a set of ground event expressions over ET as the value of the FEL corresponding to an interpretation of ET as a set of event types where each event type *E* in ET corresponds to a unary predicate that is interpreted by the set E^I collecting all events *e* in FEL of type *E*, and each attribute *a* of *E* is interpreted as a function from E^I to the range of *a* defined by the pairs ⟨*e*, *e.a*⟩ where *e.a* denotes the value of *a* for *e* as defined by the ground event expression *e*.

An OES model defines a signature ∑ in the following way:

1. Its set of object types *O* in OT defines a set of unary predicates, and each property defined for an object type *O* defines a function from the population of *O* to the range of the property.
2. Its set of event types *E* in ET defines a set of unary predicates, and each property defined for an event type *E* defines a function from the population of *E* to the range of the property.

An OES state consists of populations for the object types in OT and a set of ground event expressions over ET as the value of the FEL, corresponding to populations for the event types in ET. As explained for event types above, each population of an object or event type *T* corresponds to an interpretation of T as a set of objects or events, and of *T's* properties as functions.

The set R of event rules of an (O)ES model corresponds to a set of update rules that can be applied simultaneously to a simulation state resulting in an ASM transition.

6 RELATED WORK

There are three established formalisms for DES. Schruben (1983) has proposed to use *Event Graphs* as a diagram language (with an operational semantics) for specifying ES models. Zeigler et al. (2000) have proposed a number of formalisms for capturing various aspects of system simulation using the acronym *DEVS*. Jensen et al. (2007) define the graph formalism of *Colored Petri Nets* for simulating distributed and concurrent processes based on the concepts of places and transitions.

As can be seen in Table 4, which summarizes the basic terms/concepts of these formalisms and of our (O)ES ASM semantics, neither Petri Nets nor DEVS support objects (with properties, relations and functions) and events (with an FEL and an event scheduling mechanism) as first class citizens. Rather, the basic DES concepts have to be mapped to (or interpreted as) suitable elements of these formalisms. For instance, event types are typically mapped to transitions in Petri nets and to input ports in DEVS. Thus, these computational formalisms only provide an indirect semantics for DES.

Only Event Graphs and our ES ASMs provide a *direct* semantics of basic DES without objects, and only our OES ASMs provide a *direct* semantics of basic DES with objects.

As opposed to our ES ASM semantics, Event Graphs do not support expressing conditional state changes, but only conditional event scheduling. In general, however, event routines (within event rules) may specify conditional state changes, which would require a modification of Event Graphs. Our ES ASMs provide a formal basis for such an improvement of Event Graphs by adding conditional branching (e.g., in the visual form of BPMN gateways) and by moving state change annotations from event circles to (the end of) edges/arrows pointing to follow-up events. This has to be elaborated in future work.

A further possible generalization of Event Graphs is based on our OES ASMs: adding the concept of objects such that conditions and state change expressions may to refer to the state of specific objects. This would harmonize the Event Graph formalism with object-oriented programming.

Table 4: Comparing the basic terms/concepts of DES formalisms.

Formalism	Basic terms/concepts	States
(Colored) Petri Nets	places, transitions, arcs	markings with (colored) tokens
DEVS	states, time advance, internal transition function, outputs and output ports, output function, inputs and input ports, external transition function	named black boxes, no state structure is considered (in examples, state variables are used)
Event Graphs	FEL, event type nodes with state change annotations, (conditional) event scheduling edges	sets of state variables
(O)ES ASMs	event types, event expressions, FEL, event routines/rules with state changes and event scheduling	sets of state variables (or objects with property-value slots)

7 CONCLUSIONS

We have proposed a new formalization of discrete event simulation on the basis of abstract state machines, which are transition systems with highly expressive states in the form of ∑-structures. The proposed formalization allows precise definitions of the two most basic forms of DES: (1) ES, which is based on the event worldview, and (2) OES, which is based on the object-event worldview. The terminology and precisely defined concepts of our formalization provide a basis for comparing, and explaining design choices in, different DES approaches.

A web-based implementation of OES is presented in (Wagner 2017a) and an introduction to information and process modeling for OES is provided in (Wagner 2017b).

In future work, we will show that our formalization allows defining abstract discrete event simulators, which are reference algorithms that define the captured form of DES in a computationally precise way.

REFERENCES

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

Börger, E., and R. Stärk. 2003. *Abstract State Machines - A Method for High-Level System Design and Analysis*. Springer-Verlag.

Casati, R., and A. Varzi. 2015. Events, In: Zalta, E.N. (ed.) *The Stanford Encyclopedia of Philosophy*, http://plato.stanford.edu/archives/win2015/entries/events/

Dahl, O.-J., and K. Nygaard. 1967. Simula 67. IFIP TC 2 Working Conference on Simulation Languages, Oslo.

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. 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: IEEE.

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

Ingalls, R.G. 2008. "Introduction to Simulation". In *Proceedings of the 2008 Winter Simulation Conference*, edited by S.J. Mason, R.R Hill, L. Monch, O. Rose, T. Jefferson and J. W. Fowler, 17–26. Piscataway, New Jersey: IEEE.

Jensen, K., L.M. Kristensen, and L. Wells. 2007. "Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems." *International Journal on Software Tools for Technology Transfer (STTT)* 9(3/4): 213–254.

Markowitz, H., B. Hausner, and H. Karr. 1962. *SIMSCRIPT: A Simulation Programming Language*. Englewood Cliffs, N. J.: Prentice Hall.

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.

Reisig, W. 2008. "Abstract State Machines for the Classroom." In D. Bjorner and M.C. Henson (Eds.), *Logics of Specification Languages*, Springer-Verlag, 15–46.

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

Tarski, A. 1954. "Contributions to the Theory of Models I". *Indagationes Mathematicae* 16:572–581.

Wagner, G. 2017a. "Sim4edu.com – Web-Based Simulation For Education." In *Proceedings of the 2017 Winter Simulation Conference*, edited by W.K.V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer and E. Page, to appear. Piscataway, New Jersey: IEEE.

Wagner, G. 2017b. "Introduction to Information and Process Modeling for Simulation" In *Proceedings of the 2017 Winter Simulation Conference*, edited by W K.V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer and E. Page, to appear. Piscataway, New Jersey: IEEE.

Zeigler, B.P., T.G. Kim and H. Praehofer. 2000. *Theory of Modeling and Simulation*. Academic Press, London.

AUTHOR BIOGRAPHIES

GERD WAGNER is Professor of Internet Technology in the Dept. of Informatics, Brandenburg University of Technology, Cottbus, Germany, and Adjunct Associate Professor in the Dep. of Modeling, Simulation and Visualization Engineering, Old Dominion University, Norfolk, VA, USA. His research interests include modeling and simulation, foundational ontologies, knowledge representation and web engineering. His email address is G.Wagner@b-tu.de.