Preprint, to appear in *Proceedings of the 2023 Winter Simulation Conference* published in early 2024 at https://informs-sim.org/wsc23papers/by_area.html

Event Graphs: Syntax, Semantics and Implementation

Murat M. Gunal
Fenerbahce University, Istanbul, Türkiye

Yahya Ismail Osais
King Fahd University of Petroleum and Minerals, Saudi Arabia

Gerd Wagner
Brandenburg University of Technology, Germany

ABSTRACT

This tutorial aims to introduce Event Graphs (EGs), invented 40 years ago by Lee Schruben to allow event-based modeling of discrete dynamic systems. Their simplicity and naturalness in causality modelling and simulation modelling made EGs popular in research and practice. In a simulation, an event causes state changes in a system as well as other events to happen in the future. EGs provide a parsimonious diagram representation for the Event Scheduling paradigm of Discrete Event Simulation. We first introduce their visual syntax and informal semantics, and then present a recent extension by adding objects to EGs. Our tutorial also includes an introduction to the formal semantics of EGs and a Python implementation for executing EGs.

1 INTRODUCTION

An Event Graph (EG) is a directed graph whose nodes represent events (of various types) and whose directed edges represent the causation, or scheduling, of follow-up events. In an EG diagram, event nodes are rendered as circles, and causation/scheduling edges as solid arrows with a filled arrowhead.

When an EG is intended to describe a discrete system of the real-world (in the sense of a descriptive *conceptual model*), an edge from event A to event B represents the causal regularity that events of type A cause events of type B.

More frequently, however, EGs are used for defining simulation models of real-world systems in the sense of prescriptive *simulation design models*. When an EG is intended to define a simulation design model, an edge from event A to event B represents the computational pattern that when an event of type A occurs, an event of type B is scheduled to happen in the future by adding it to the *Future Events List*. This pattern defines the *Event Scheduling*, or *Event-Based Simulation*, paradigm of *Discrete Event Simulation (DES)*, which is the foundation of many other DES paradigms.

2 SYNTAX AND INFORMAL SEMANTICS

EGs have a visual syntax in the form of a diagram language where *Event* nodes are rendered as circles, and *Event Causation* (or *Event Scheduling*) edges as solid arrows with a filled arrowhead. There are the following basic language elements: Event nodes have *names* and may have a set of *state changes* (in the form of variable value assignments), while Event Scheduling edges may have a *delay* expression and a *condition*. The name of an Event node is, in fact, the name of the *event type* associated with that node. Any event has a type, and an event type specifies a set of attributes and a set of functions, like a class in an *Object-Oriented (OO)* programming language.

When we refer to an EG diagram, we also speak of *Event circles* instead of Event nodes and of *Event Scheduling arrows* instead of Event Scheduling edges. All these basic elements are presented in the next

subsection. In addition, there are three advanced elements in EGs: 1) event attributes, 2) event priorities, and 3) event cancelling, which are discussed in the following subsections.

2.1 Basic Language Elements

Figure 1 shows the basic pattern that composes EGs: an event of type A leads to *state changes* and to a *follow-up event* of type B. According to the execution semantics of EGs, the follow-up event B is scheduled by adding it to the *Future Events List (FEL)* maintained by the execution environment (or *EG simulator*).

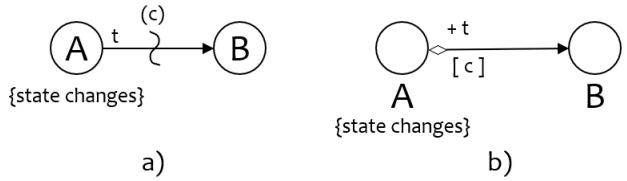

Figure 1: The basic pattern, which composes EGs: a) classical notation, b) modern notation.

The EG in Figure 1 is presented in two notations. In a), the original notation from (Schruben 1983) is used. Our modern notation in b), which is largely borrowed from the business process modeling notation BPMN, improves the visual clarity of EG diagrams. The EG in Figure 1 is read as follows: "when an event of type A occurs, an event of type B is scheduled to occur after t time units, if condition c holds". The time delay expression t and the condition c are optional. This means that when they are not present, follow-up events of type B are scheduled to always happen without any delay. The execution semantics of EGs requires that when an event of type A occurs, first the state changes defined for A are performed, and only then the scheduling condition is evaluated, and if it holds, a follow-up event of type B is scheduled.

The conditional Event Scheduling arrow in the classical EG notation was expressed with a vertical tilde shape in the middle of an edge and a condition in parenthesis. In the modern notation, it is expressed with a "mini-diamond" at the source side of the arrow and a condition *c* in brackets. When this condition is evaluated to true, a follow-up event of type B is scheduled. When the condition is evaluated to false, no follow-up event is scheduled. Let's also note that when an Event Scheduling arrow is unconditional, there is no mini-diamond (or tilde) and no condition. In the rest of this tutorial, we will use the modern EG notation.

The diagrams in Figure 2 and 3 represent the same EG, in both notations. The EG models a fundamental queuing system example (often called *single-server* queuing system) with a first-in-first-out queue of parts that will be processed at a workstation.

Notice that the *Arrival* events of this model are distinct from the events of the other two Event circles, as they are not caused (or scheduled) by other events. Thus, the *Arrival* Event circle represents a *Start Event* node, which means that its events are either *recurrent* (periodically occurring) or initially pre-scheduled by a simulation scenario. In the classical EG of Figure 2, the *recurrence* of *Arrival* events is expressed with the help of a recursive edge (from *Arrival* to *Arrival*) and the variable *tA* attached to it represents the recurrence time. In the modern form of this EG shown in Figure 3, the recursive Event Scheduling arrow from *Arrival* to *Arrival* has been dropped since the recurrence of *Arrival* events is implied by the recurrence (or *nextOccurrence*) function specified for the *Arrival* event node (or event type).

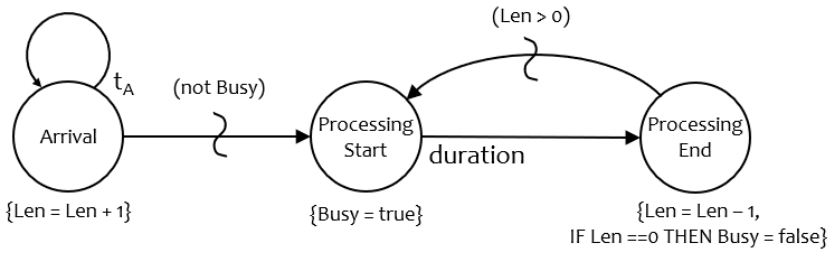

Figure 2: Modeling a simple queuing system as an EG using the classical EG notation.

We assume that the type definition of exogenous start events includes a recurrence (or *nextOccurrence*) function that is invoked by a simulator for automatically scheduling recurrent events such as for creating the next *Arrival* event whenever an *Arrival* event occurs.

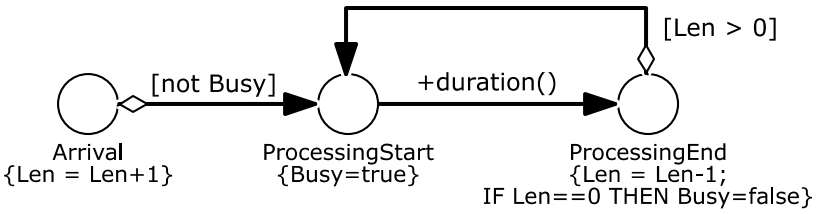

Figure 3: Modeling the simple queuing system of Figure 2 as an EG using the modern notation.

In any case, an *Arrival* event increments the state variable "Len" by one. The variable "Len" represents the length of the workstation's input buffer (which, for simplicity, also includes the part that is currently being processed). If the Boolean state variable "Busy" has the value *false* (that is, if the condition *not Busy* holds), a *ProcessingStart* event is scheduled. When this event occurs, the variable "Busy" is set to *true* and a *ProcessingEnd* event is scheduled with a delay obtained by invoking the *duration()* function. Note that there is no condition for this Event Scheduling arrow since every *ProcessingStart* event causes a corresponding *ProcessingEnd* event.

When the end event occurs, the "Len" variable is decremented and if the value of "Len" has fallen to zero, the workstation is no longer "Busy". In addition, if the value of "Len" is greater than zero (that is, if there are still waiting parts in the workstation's input buffer), a new *ProcessingStart* event is scheduled to occur immediately (without any delay).

EG diagrams, such as the one shown in Figure 3, can be created with BPMN tools such as https://www.drawio.com/ or the *Signavio Process Editor*, which is free for academic use.

2.2 Event Attributes

Event types may define attributes for events. For instance, the event type *Delivery* may define the Integer-valued attribute "deliveryQuantity" for allowing event expressions like *Delivery*(deliveryQuantity: 65) stating that this is a delivery event with a quantity of 65.

Let's assume we are simulating an inventory system's demand and replenishment sections with two events: *DailyDemand* and *Delivery*. We have some daily demand for a product that we manage. When a daily demand event occurs, two variables are updated: the daily sales quantity *S*, and the stock quantity *Q*, as shown underneath the *DailyDemand* Event circle inside curly braces in Figure 4. We use the function "min" for finding the minimum of two numbers and "quantity" for sampling the daily demand random variable. The variable *RP* in the condition represents the reorder point for this inventory system. The condition $Q <= RP \ \& \ Q+S > RP$ captures the situation where the stock quantity *Q* has just fallen below the reorder point.

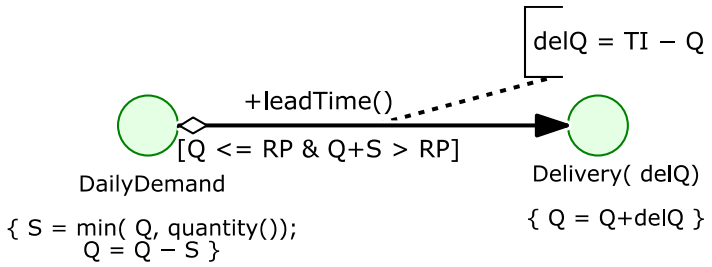

Figure 4: An EG modeling an inventory management system with a continuous review policy.

The "leadTime" function is used for sampling the delivery lead time as the delay of *Delivery* follow-up events (expressed in the form of an annotation "+leadTime()" of the Event Scheduling arrow). The *Delivery* Event Circle's event type name field reads "Delivery(delQ)", which means that *Delivery* events have a "delQ" attribute representing the delivery quantity. Such event attributes have to be assigned when a new event of that type (here: *Delivery*) is scheduled. In the EG diagram of Figure 4, the assignment of the event attribute *delQ = TI - Q* is shown with the help of a left bracket connected to the *Delivery* Event Scheduling arrow with a dashed line (adopting to the visual syntax of BPMN).

2.3 Event Priorities

In Event-Based Simulation (with Event Scheduling), it is important to use event priorities for handling events that happen at the same time and are *non-confluent* (having different effects if processed in a different order). Using event priorities allows to enforce a specific processing order of simultaneous events for avoiding deadlocks or other inconsistencies.

In the model of Figure 3, for example, we need to define a higher priority for *ProcessingStart* events than for *Arrival* events. Otherwise, when we have a situation with a pair of simultaneous *Arrival* and ProcessingStart events, and the workstation is idle, and the *Arrival* event is processed before the *ProcessingStart* event, we will get a second ProcessingStart event, which is incorrect and creates inconsistency. We could prevent this happening by not using the scheduling condition "not Busy", but rather "Len == 1".

Likewise, for the inventory system in Figure 4, *Delivery* events should have a higher priority than *DailyDemand* events. Without this priority setting, *DailyDemand* events may schedule a new *Delivery* event before the state changes of the previously scheduled *Delivery* event have been performed.

2.4 Event Cancelling

In EGs, an event can be cancelled by using event cancelling edges which are visually expressed as dashed lines. For example, consider a server that fails periodically (Osais 2020). When the server fails, the processing activity of the server is interrupted (or *preempted*). This situation can be easily modeled by using an Event Canceling arrow as shown in the model of Figure 5, which is an enhanced version of the model shown in Figure 3, now including failure and repair events. When a *Failure* event occurs, it immediately cancels the most recent pending *ProcessingEnd* event and schedules a *Repair* event to occur with a delay of *repairTime()* time units. After the workstation is repaired, processing is restarted and the next *Failure* event is scheduled.

If we view a start-end pair of events as an activity, an ongoing activity can be interrupted by canceling its end event, which is pending on the FEL. Hence, an Event Canceling arrow removes a pending event from the FEL.

The first *Failure* event, like the first *Arrival* event, is scheduled during the simulation initialization. When the first *Failure* event occurs, the status of the workstation is set to "failed" and then (1) the pending

ProcessingEnd event is canceled (removed from the FEL), and (2) a follow-up *Repair* event is scheduled with a random delay obtained by calling the *repairTime()* function.

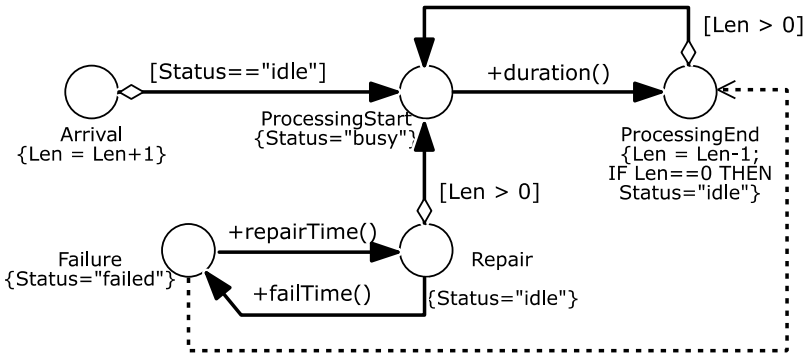

Figure 5: EG for a periodically failing workstation, with an Event Cancelling edge.

When the *Repair* event is due, the workstation's status is set to "idle", meaning that the workstation is again ready to process new parts, and the next *Failure* event is scheduled to happen with a delay obtained by calling the *failTime()* function. A *Repair* event also schedules a *ProcessStart* event if there are still parts in the input buffer.

3 ADDING OBJECTS TO EVENT GRAPHS AND TO EVENT-BASED SIMULATION

Since the basic entities in the real world are *objects* and *events*, and a discrete dynamic system consists of objects that are affected by events, supporting these two ontological categories in a simulation paradigm seems natural and desirable. It also follows the fundamental insights of the simulation language Simula about the value of "objects" as a modeling and programming concept, and it harmonizes EGs with Object-Oriented (OO) modeling and programming languages. However, *Event-Based Simulation (ES)* and EGs, following the classical mathematical modeling approach, abstract away from real-world objects and rather represent their attributes in the form of mathematical variables called *state variables.*

Notice that the term "entities" has been widely used in DES for denoting objects that are being processed (such as parts in manufacturing networks or patients in healthcare service networks). However, in addition to such *processing objects*, discrete dynamic systems also consist of *resource objects* and possibly other types of objects.

Wagner (2017, 2018) has shown that it is straightforward to extend ES and EGs by adding the concept of objects, resulting in *Object Event Simulation (OES)* and *Object Event Graphs (OEGs)*. A similar extension of EGs, called *Object-Oriented Event Graph* modeling, is proposed in (Tiacci 2019).

OES is obtained from ES by extending the definition of a system's state structure *S* consisting of a set of variables by adding to them a set of objects, each with a state defined to be the set of its attribute-value pairs. OEGs are obtained from EGs by allowing to attach Object nodes to Event nodes such that each Object node represents an object variable that can hold objects participating in the events of the Event node. In an OEG diagram, Object nodes are rendered as rectangles attached to Event circles with a dashed line, as shown in Figure 5.

While DES is an umbrella term subsuming many different approaches, ES/EGs form its variable-based foundation, and OES/OEGs provide an OO foundation of DES.

For binding the Object node's object variable (such as the *Workstation* Object rectangle's object variable *ws* in Figure 5) to the corresponding object participating in the event referenced by the Event node's event variable (such as the *Arrival* Event circle's event variable *a*), the first compartment of an

Object rectangle contains an assignment (such as *ws = a.workstation* where *workstation* is the *Arrival* event's attribute for referencing the affected workstation object).

Object rectangles contain object-specific state change statements in their second compartment. For instance, the *Workstation* Object rectangle attached to the *Arrival* Event circle contains the state change statement "INCREMENT inputBufferLength" referring to the *Workstation* attribute *inputBufferLength*.

A further extension of EGs and OEGs is obtained by introducing the concept of activities where an activity corresponds to the pattern provided by a pair of start/end Event nodes such that the start Event circle has an outgoing Scheduling arrow to the end Event circle with a delay expression corresponding to the duration of the implicitly represented activity. This extension of EGs and OEGs has been proposed in (Wagner 2020).

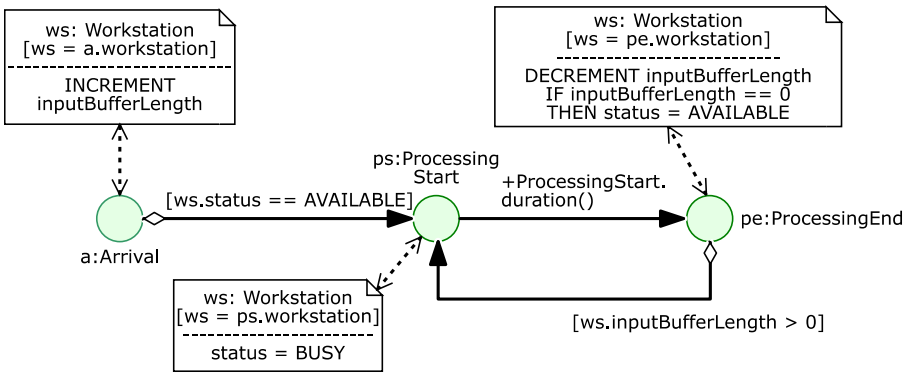

Figure 6: An OEG corresponding to the EG of Figure 2.

4 FORMAL SEMANTICS

In (Yücesan and Schruben 1992), EGs have been defined in a mathematically explicit way under the name of *Simulation Graph Models (SGMs)*. It is shown that an SGM can be "expanded" into an *elementary* SGM where nodes have only atomic state changes and conditional edges have only atomic conditions. Two SGMs are defined to be *structurally equivalent* if they have any elementary SGMs that are isomorphic. The notion of structural equivalence allows to establish the validity of certain EG reduction rules proposed in (Schruben 1983).

While Yücesan and Schruben (1992) define EGs and the structural equivalence of two EGs, as well as a notion of behavioral equivalence of two EGs, in a formally precise way, they do not define a formal execution semantics for EGs. Such a semantics has been proposed by Wagner (2017) for Event-Based Simulation (ES) and for Object Event Simulation (OES) using the concept of an *Abstract State Machine (ASM)*, which is a very expressive kind of state transition system where states are highly structured and correspond to interpretations of a predicate logical language (with relation symbols) similar to the states of a relational database system. Wagner's ASM semantics for ES and OES also applies to EGs and OEGs by decomposing them into sets of *event rules* providing the ASM's transition functions.

4.1 An Abstract State Machine Semantics for Event-Based Simulation

The base concepts of ES are:

1. state variables for describing the state of a system,
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(x)@t where

1. E is the name of an event type,
2. t is a parameter for the occurrence time of events,
3. 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(v)@i, where the event parameter list x and the occurrence time parameter t have been instantiated with a corresponding value list 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. However, for defining the formal semantics of ES, 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 1.

Table 1: Expressing an event routine as a pure function that computes state changes and follow-up events.

Event rule name	ON (event expression)	DO (event routine)
r_{Arr}	Arrival @ t	E' = { Arrival @ (t + recurrence()) } Δ = { Len = Len + 1 } IF not Busy THEN E' = E' ∪ { ProcessingStart @ (t + duration()) } RETURN ⟨ Δ, E' ⟩
r_{PS}	ProcessingStart @ t	Δ = { Busy = *true* } E' = { ProcessingEnd @ (t + duration()) } RETURN ⟨ Δ, E' ⟩
r_{PE}	ProcessingEnd @ t	E' = {} Δ = { Len = Len – 1, IF Len==0 THEN Busy = *false* } IF Len > 0 THEN E' = { ProcessingStart @ t' } RETURN ⟨ Δ, E' ⟩

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

ON E(x)@t DO F(t, x),

where the event expression E(x)@t specifies the type E of events that trigger the rule, and F(t, 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 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.

4.1.1 ES Models

An 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 Figure 3 as an ES model. The set of state variables has two elements:

SV = { Len: NonNegativeInteger, Busy: Boolean }

There are three event types, all of them having an empty signature:

ET = { Arrival(), ProcessingStart(), ProcessingEnd() }

And there are three event rules, one for each type of event:

R = { r_{Arr}, r_{PS}, r_{PE}}

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 removing the current events and 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.

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

S_0 = ⟨ {Len: 0, Busy: *false*}, {Arrival@1} ⟩

S_1 = ⟨ {Len: 1, Busy: *false*}, {ProcessingStart@1.1, Arrival@2} ⟩

Doing one more step, the next transition is given by the next event ProcessingStart@1.1 triggering r_{PS}, which leads to

S_2 = ⟨ {Len: 1, Busy: *true*}, {Arrival@2, ProcessingEnd@2.7} ⟩

In this way, we get a succession of states $\mathbf{S_0} \rightarrow \mathbf{S_1} \rightarrow \mathbf{S_2} \rightarrow$ ... as a history of the transition system defined by the ES model.

4.1.2 Event Rules as Functions

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

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

In general, there may be situations, where we have several simultaneous events, that is, there may be two or more events occurring at the same 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 ⟨ Δ, E' ⟩ 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.

4.1.3 An Event Rule Set as a Simulation State 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' ⟩.

Let Upd denote 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 variable value assignments. Using this operation, we can define the third step of the simulation state transition function with two sub-steps in the following way:

1. $S' = Upd(S, \Delta)$
2. $FEL' = FEL - N \cup E'$

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:

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

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

$\mathbf{S_0} \rightarrow \mathbf{S_1} \rightarrow \mathbf{S_2} \rightarrow ...$

by iteratively applying the transition function R:

$\mathbf{S_{i+1}} = R(\mathbf{S_i})$

4.2 Decomposing an Event Graph into a Set of Event Rules

Any EG can be decomposed into a set event rules, such that one rule is obtained for each Event circle of the EG. This decomposition, together with the ASM semantics for ES models presented above, provides a compositional semantics of EGs.

For instance, the EG shown in Figure 3 above can be decomposed into a set of three event rules, one for each of its Event circles:

1. When an *Arrival* event occurs, the variable *Len* is incremented by 1 and, if not *Busy*, a *ProcessingStart* event is scheduled to happen immediately. In addition (since *Arrival* events are recurrent), a new *Arrival* event is scheduled with a delay obtained by calling the recurrence function specified for the *Arrival* event type.
2. When a *ProcessingStart* event occurs, the variable *Busy* is set to *true*, and a *ProcessingEnd* event is scheduled with a delay provided by invoking the function *duration()*.
3. When a *ProcessingEnd* event occurs, the variable *Len* is decremented by 1 and, if *Len* is equal to 0, the variable *Busy* is set to *false*. In addition, if *Len* is greater than 0, a new *ProcessingStart* event is scheduled to happen immediately.

These three event rules are expressed more formally in Table 1 above.

5 A PYTHON IMPLEMENTATION FOR EXECUTING EVENT GRAPHS

Implementing an Object Event Graph (like the one shown in Figure 6 above) results in an Object Event Simulation (OES) program where the attributes of objects play the role of state variables and follow-up events are scheduled by adding them to the FEL. We provide both a JavaScript and a Python implementation of OES in the Github repository folders https://github.com/gwagner57/oes/tree/master/JavaScript/Core0 and https://github.com/gwagner57/oes/tree/master/Python/Core0.

Python is a multi-paradigm general-purpose programming language (Python 2023). It is very effective in rapid prototyping of concepts. Also, the simple syntax of the language helps making Python programs self-explanatory to some degree. This is one of the reasons why Python is popular in education.

In this section, we show a complete Python simulation program for the Object Event Graph of Figure 6 above. The whole program can be written in one file with around 100 lines of code, which can be downloaded from the Github repo folder provided above.

The simulation program has three sections:

1. The first section includes the foundational classes *Object*, *Event* and *EventList*.
2. The second section consists of the code of the specific simulation model example in the form of a set of object and event classes.
3. The third section consists of three parts: simulation parameter settings, an initial state definition, and the simulation loop.

The following program code listing shows the code for importing the code of a *priority queue* data structure, an *exponential* distribution function and the *mean* function from suitable Python libraries, as well as the code of the utility function *has_method*, which tests if an object has a certain method (and will be used in the simulation loop).

```
from queue import PriorityQueue
from random import expovariate
from statistics import mean
def has_method( object, methodName):
    return callable( getattr( object, methodName, None))
```

In OO programming, we use *base classes* for defining the basic functionality (or *logic*) that other (derived) classes are supposed to inherit and use. For object event simulations, as obtained from Object Event Graphs, we need to define the base classes *Object* and *Event:*

```
class Object:
    def __init__( self, name=None):
        self.name = name
class Event:
    def __init__( self, occTime, priority):
        self.occTime = occTime
        self.priority = priority
    def onEvent( self):
        pass
    def str( self):
        return self.__class__.__name__ + "@" + str( self.occTime)
```

In a simulation program, object types are implemented as classes that extend the base *Object* class. Similarly, event types are implemented as classes that extend the base *Event* class, which defines the general attributes *occurrence time* and *priority*, as well as an abstract method *onEvent*, for which each concrete event class extending the base *Event* class must provide a concrete *onEvent* method that is called whenever an event of that type occurs during a simulation run (this method has also been called *event routine* in the literature). The *Event* class has a special method *__str__* that is used for creating a string representation of

an event including its type and occurrence time. This method is useful for debugging and can be used for creating a simulation log.

The next program listing shows the definition of the *EventList* class, which encapsulates a *PriorityQueue* data structure for representing the *Future Events List (FEL)*. In the *addEvent* method, an event is inserted into a tuple (a form of an array in Python). This event tuple is used for inserting the event into the FEL. The first element in the event tuple is the event's occurrence time (*occTime*), which is the first key that is used for sorting events in the FEL. If two events have the same occurrence time, the next key used for storing an entry in the FEL is the event's priority, which is defined for every event in its event class. Finally, in the *getNextEvent* method, the next event tuple is first fetched from the FEL and then the event itself at index 2 in the event tuple is returned.

```
class EventList:
    def __init__( self):
        self.events = PriorityQueue()
    def addEvent( self, ev):
        self.events.put( (ev.occTime, ev.priority, ev) )
    def getNextEvent( self):
        ev = self.events.get()
        return ev[2]
    def isEmpty( self):
        return self.events.empty()
```

There is only one object in our workstation queuing system simulation model. This object represents the workstation, which includes the queue and the "server" resource. The next program listing shows the definition of the *WorkStation* class with two attributes: *inputBufferLength* and *status* (representing the model's state variables). The *status* attribute represents the state of the server (AVAILABLE or BUSY). Since the *WorkStation* class is derived from the base class *Object*, the attribute *name* defined in the parent class *Object* is assigned using the *super* method.

```
class WorkStation( Object):
    def __init__( self, name, inputBufferLength=0, status="AVAILABLE"):
        super().__init__( name)
        self.inputBufferLength = inputBufferLength
        self.status = status
```

There are three types of events that occur in this single-server queueing system: *Arrival*, *ProcessingStart*, and *ProcessingEnd*. The next program listing shows the definition of the *Arrival* event class. An arrival event is initialized with the event time (*occTime*) and the object participating in the event (*workstation*). The priority for the arrival event is one. Every event class must implement the method *onEvent*. This method is the event handler, which is called when the event is fired in the simulation loop. Inside the event handling method *onEvent*, the state variables expressed as attributes of objects are accessed with the help of OO path expressions of the form *self.objectReference.attribute* by first accessing the object and then its attribute. For abbreviating these path expressions, the workstation object is assigned to a local variable named *ws*. This local variable can be used for accessing the two state variables *inputBufferLength* and *status* as attributes of the *workstation* object. Also, inside the *onEvent* method, a follow-up event of type *ProcessingStart* is created and returned to the main simulation loop for being scheduled.

```
class Arrival( Event):
    def __init__( self, occTime, workstation):
        super().__init__( occTime, 1)  # assign highest priority
        self.ws = workstation
    def onEvent( self):
```

```
        followupEvents = []
        ws = self.workstation
        ws.inputBufferLength += 1
        if ws.status == "AVAILABLE":
            followupEvents.append( ProcessingStart( self.occTime, ws))
        return followupEvents
    def nextOccurrence( self):
        delay = expovariate( rate_nextOccurrence)
        return Arrival( self.occTime + delay, self.workstation)
```

ProcessingStart events are internal (or *caused*) events, which are follow-up events returned by the *onEvent* method. By contrast, *Arrival* events are external (or *exogenous*) events because they are not caused within the single-server queueing system. Typically, external events are *recurrent*, that is, they are created by a special method called *nextOccurrence*. Only classes for external event types should implement this method. Inside the *onEvent* method of the *Arrival* event class, a delay (or *time-to-next-arrival*) is first generated using the imported random variate generator for the exponential distribution called *expovariate*. After that, a new *Arrival* event is created and then returned to the simulation loop.

The definition of the *ProcessingStart* event class is shown in the next program listing. Also events of this type have a *workstation* object as a participant. Since these events are not start events (or internal events), the *nextOccurrence* method is not implemented in their class definition. Again, the attributes of the workstation object are accessed via the path expression self.w*orkstation*. The follow-up event of type *ProcessingEnd* is scheduled to occur after some delay, which represents the service time at the workstation server.

```
class ProcessingStart(Event):
    def __init__( self, occTime, workstation):
        super().__init__( occTime, 2)
        self.workstation = workstation
    def onEvent( self):
        followupEvents = []
        ws = self.workstation
        ws.inputBufferLength -= 1
        ws.status= "BUSY"
        delay = expovariate( rate_serviceTime)
        followupEvents.append( ProcessingEnd( self.occTime + delay, ws))
        return followupEvents
```

The class definition of our last event type, *ProcessingEnd*, is shown in the next program listing. When the processing of a part ends, the workstation (server) becomes available and thus the *status* state variable of the WorkStation object is updated accordingly. Also, a follow-up event of type *ProcessingStart* for processing the next part is only created if the workstation's input buffer is not empty.

```
class ProcessingEnd( Event):
    def __init__( self, occTime, workstation):
        super().__init__( occTime, 3)
        self.workstation = workstation
    def onEvent( self):
        followupEvents = []
        ws = self.workstation
        ws.status = "AVAILABLE"
        if ws.inputBufferLength > 0:
```

```
            followupEvents.append( ProcessingStart( self.occTime, ws))
        return followupEvents
```

Simulation parameters are assigned and the initial simulation state is initialized in the code of the following program listing. The arrival and departure rates are *lambda* and *mu*, respectively. Also, the only object in the simulation program (*ws*) and one initial *Arrival* event are created. Two auxiliary variables (*intEvent* and *extEvent*) are used to track internal and external events generated while the simulation runs. The *queueLength* variable is defined to collect simulation data. Basically, upon every arrival event, the queue length is sampled and the value is recorded.

```
# --- initialize simulator -----
evList = EventList()
simTime = 100000 # Total simulation time
clock = 0
# --- assign model parameters -----
rate_nextOccurrence = 0.5
rate_serviceTime = 0.6
# --- set up the initial state -----
ws = WorkStation("ws")
evList.addEvent( Arrival( 0.0, ws))
# --- initialize simulation statistics -----
queueLength = []
```

The details of the simulation loop are given in the following program listing. The system is simulated for a specific amount of simulated time. The helper function *has_method* is used to detect if the current event is a recurrent event (i.e., it has the *nextOccurrence* method). At the end of the simulation loop, if the current event is an *Arrival* event, the *inputBufferLength* state variable is sampled and the value is decremented by one and then stored in the variable *queueLength*. Finally, the average queue length is computed and printed.

```
while clock <= simTime:
    ev = evList.getNextEvent()
    clock = ev.occTime
    #--- if the event is recurrent, schedule the next one -----
    if has_method( ev, "nextOccurrence"):
        evList.addEvent( ev.nextOccurrence)
    followUpEvents = ev.onEvent()
    for ev in followUpEvents: evList.addEvent(ev)
    #--- collect statistics data -----
    if isinstance( ev, Arrival):
        queueLength.append( ws.inputBufferLength - 1)
#--- print output statistics -----
print("Average Queue Length = ", round( mean( queueLength), 2))
```

The above simulation program can be easily extended to simulate systems with any number of objects and event types. For example, customers flowing through the system can be modeled as an object type with attributes like *ArrivalTime* and *DepartureTime*. A possible statistic that can be computed using this simulation data would be the average system delay.

6 CONCLUSION

Forty years ago, just some decades after the rise of computer simulation, Event Graphs were introduced as a diagram language for visually modelling causality and event-based simulation models. In this tutorial, we have shown that Event Graphs provide an intuitive approach for making simulation models.

ACKNOWLEDGMENTS

Yahya Osais would like to acknowledge KFUPM for financial support.

REFERENCES

Osais, Y. 2020. *Computer Simulation: A Foundational Approach Using Python*. 1st Ed., Routledge.

Savage, E. L., and Schruben, L. W. 1995. "Eliminating Event Cancellation in Discrete Event Simulation". In *Proceedings of the 1995 Winter Simulation Conference*, 744 – 750.

Schruben, L. W. 1983. "Simulation Modeling with Event Graphs". *Communications of the ACM* 26:957–963. https://dl.acm.org/citation.cfm?id=358460.

Schruben, L and E. Yücesan.1993. Modeling Paradigms for Discrete Event Simulation, Operations Research Letters, 13, 265–275.

Tiacci, L. 2019. Object-Oriented Event-Graph Modeling Formalism to Simulate Manufacturing Systems in the Industry 4.0 Era. *Simulation Modelling Practice and Theory* 99.

The Python Programming Language. https://www.python.org, accessed 5th May 2023.

Wagner, G. 2017. "An Abstract State Machine Semantics for Discrete Event Simulation". In W. K. V. Chan et al (Eds.), *Proceedings of the 2017 Winter Simulation Conference*, 762–773, IEEE, URL: https://www.informs-sim.org/wsc17papers/includes/files/056.pdf

Wagner, G. 2018. Information and Process Modeling for Simulation – Part I: Objects and Events, *Journal of Simulation Engineering* 1, 1–25, URL: https://articles.jsime.org/1/1/Modeling-for-Simulation-Part-I.

Wagner, G. 2020. "Business Process Modeling and Simulation with DPMN: Resource-Constrained Activities". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing. 762–773. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Yücesan, E. and L.W. Schruben. 1992. "Structural and Behavioral Equivalence of Simulation Models". *ACM Transactions on Modeling and Computer Simulation* 2:1, 82–103.

AUTHOR BIOGRAPHIES

MURAT M. GUNAL is an Associate Professor in the Department of Industrial Engineering at Fenerbahce University, Istanbul Turkiye. He is also the founder of Simarter Ltd. an analytics company specialised in modelling, simulation and optimisation. His main research area is simulation methodologies, healthcare and industrial applications. He developed many models for decision making. His models are used in real and challanging decision making problems in many sectors. His email address is murat.gunal@fbu.edu.tr.

YAHYA E. OSAIS is an Assistant Professor in the department of computer engineering at King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia. He is also a member of the interdisciplinary research center for intelligent secure systems at KFUPM. His research interests include simulation modeling and reinforcement learning. His email address is yosais@kfupm.edu.sa, and his homepage is https://faculty.kfupm.edu.sa/coe/yosais/.

GERD WAGNER is Professor of Internet Technology in the Dept. of Informatics, Brandenburg University of Technology, Germany. After studying Mathematics, Philosophy and Informatics in Heidelberg, San Francisco and Berlin, he (1) investigated the semantics of negation in knowledge representation formalisms, (2) developed concepts and techniques for agent-oriented modeling and simulation, (3) participated in the development of a foundational ontology for conceptual modeling, the *Unified Foundational Ontology (UFO)*, and (4) created a new Discrete Event Simulation paradigm, *Object Event Modeling and Simulation (OEM&S)*, and a new process modeling language, the *Discrete Event Process Modeling Notation (DPMN)*. Much of his recent work on OEM&S and DPMN is available from sim4edu.com and dpmn.info. His email address is g.wagner@b-tu.de.