TOWARDS A NON-PROPRIETARY MODELING LANGUAGE FOR PROCESSING NETWORK SIMULATION

Gerd Wagner

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

ABSTRACT

Processing networks have been investigated in the mathematical theory of queueing and have been the application focus of most industrial simulation software products, starting with GPSS and SIMAN/Arena. They allow modeling many forms of discrete *processing processes*, and are mainly used in simulation projects for the manufacturing and services industries. However, there is still no proper vendor-neutral language definition for this paradigm, e.g., in the form of a meta-model defining an abstract syntax for specifying the structure and dynamics of processing networks. We reconstruct the core of this paradigm in the form of a UML-based meta-model and show how to map a processing network specified with this meta-model to an *Object Event Simulation* model providing its operational semantics.

Keywords: GPSS, SIMAN, Arena, Queuing Networks, Processing Networks

1 INTRODUCTION

The well-known *Arena* simulation software is the most prominent representative of a *Discrete Event Simulation (DES)* paradigm that has been pioneered by GPSS (Gordon 1961) and SIMAN/Arena (Pegden and Davis 1992) and is often characterized by the narrative of "entities flowing through a system". This narrative refers to a (typically stochastic) *processing network (PN)* where processing objects arrive at an entry node and are then routed to a series of processing nodes where they are subject to processing activities, possibly inducing queues (in input/output buffers), requiring resources, and transforming the types of processing objects, before they depart at an exit node. We call this modeling and simulation paradigm, which is concerned with modeling the behavior of discrete processing systems (such as manufacturing plants, hospitals, restaurants, etc.) in the form of *processing processes*, the *PN paradigm*.

It is remarkable that the PN paradigm has dominated the discrete event simulation market since the 1990's and still flourishes today, mainly in the manufacturing and services industries, often with object-oriented and "agent-based" extensions. Its dominance has led many simulation experts to view it as a synonym of DES, which is a conceptual flaw because the concept of DES, even if not precisely defined, is clearly more general than the PN paradigm, as we argue in Section 3.

The PN paradigm has often been called a "process-oriented" DES approach. But unlike the business process modeling language BPMN, it is not concerned with a general concept of *business process models*, but rather with the special class of *processing process models* for discrete processing systems, as we discuss in Section 3. A processing process includes the simultaneous handling of several "cases" (processing objects) that may compete for resources or have other interdependencies, while a “business process” in BPM has traditionally been considered as a case-based process that is isolated from other cases.

Certain classes of processing networks have been investigated in the mathematical theory of queueing, which aims to understand, analyze, and control congestion in these networks (Harrison 2000, Williams 2016). They have therefore also been called “queuing networks” in the Operations Research literature. The mathematical models defined in this area are good for stochastic analysis, but they do not provide a computational specification that can be directly implemented and executed, e.g., with Object-Oriented Programming, for allowing simulations.

The most important modeling elements of Arena are the “flowchart modules” *Create*, *Process* and *Dispose* for defining a network of entry, processing and exit nodes, supplemented by *Decide* for conditional branching (XOR-splitting) and the “data modules” *Entity* for defining costing parameters per entity type and *Resource* for defining resource pools. There are four additional “flowchart modules”: *Separate* and *Batch* for ungrouping and grouping processing objects (or AND-splitting and merging the corresponding control flows), *Assign* for assigning variables and *Record* for collecting statistics. And there are four additional “data modules” (*Queue*, *Variable*, *Schedule* and *Set*). In this paper, we do not consider these additional modeling elements that do not belong to the core of Arena.

An example of an Arena simulation model using only core (“flowchart”) elements is shown in Figure 1. It models a Department of Motor Vehicles (DMV) with two consecutive service desks: a reception desk and a case handling desk. When a customer arrives at the DMV, she first has to queue up at the reception desk where data for her case is recorded. The customer then goes to the waiting area and waits for being called by the case handling desk where her case will be processed. After completing the case handling, the customer leaves the DMV via the exit.

Customer arrival events are modeled with a *Create* element (with name “DMV Entry”), the two consecutive service desks are modeled with two *Process* elements, and the departure of customers is modeled with a *Dispose* element (with name “DMV Exit”).

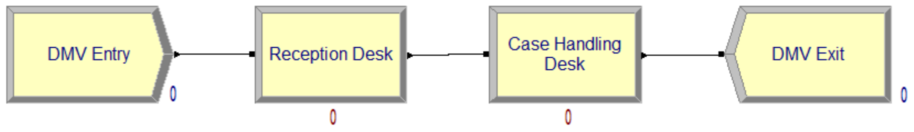


Figure 1: An Arena simulation model of a Department of Motor Vehicles (DMV).

There is no proper vendor-neutral language definition for the PN paradigm, e.g., in the form of a meta-model. The simulation modeling concepts of the PN paradigm have been adopted by many other simulation software products, including Simul8, Simio and AnyLogic. However, each product based on this paradigm uses its own variants of the PN concepts, together with their own proprietary terminology and proprietary diagram language, as illustrated by Table 1.

Table 1: Proprietary terminologies

Arena	*Simul8*	*Simio*	*AnyLogic*
Entity	Work Item	Token	Agent
Create	Start Point	Source	Source
Process	Queue + Activity	Server	Service
Dispose	End Point	Sink	Sink

Notice especially the strange term “agent” used by AnyLogic instead of the Arena term “entity”, which stands, e.g., for manufacturing parts in production systems or for patients in hospitals. It’s confusing to call a manufacturing part, such as a wheel in the production of a car, an “agent”.

As noted by van der Aalst (2014), “the use of proprietary building blocks in tools such as ARENA makes it hard to interchange simulation models”.

Using an approach that resembles the approach of *Inductive Reference Modeling* (Martens et al. 2015), which extracts a common core from individual models, we develop a small core of the PN paradigm in the form of a UML-based meta-model in four steps:

1. In Section 2, we start by identifying the core elements of Arena and describe them with a language model in the form of a UML class diagram, shown in Figure 2. In this step, we already rename certain elements for having self-explaining names and merge certain attributes into functions.
2. Then, in Section 3, we analyze the basic concepts of the Arena language.
3. In Section 4, we rename the core concepts for obtaining a technology-independent and vendor-neutral terminology. meta-model.
4. Finally, still in Section 4, we make further simplifications by dropping non-essential attributes for obtaining a small core of the Arena language.

The resulting meta-model, shown in Figure 3, specifies an abstract syntax of a language that defines a small core subset of the PN paradigm, called *Processing Network Language (PNL) 0.1*. It has to be iteratively extended by adding further PN concepts capturing variation points extracted from other relevant examples of approaches/languages based on the PN paradigm such as *Simio* and *AnyLogic*.

In Section 5, we show how to obtain PNL 0.2 from PNL 0.1 by matching the PNL 0.1 metamodel with a metamodel of a core fragment of *AnyLogic* for identifying further PNL core elements. In a similar way, we are going to define PNL 0.3 by matching PNL 0.2 with a core fragment of *Simio*, and PNL 0.(x+1) by matching PNL 0.x with a core fragment of yet another language.

The resulting *PNL Family* is a sequence of growing language versions that approximates the common core of all variants of processing networks. It can be used as a basis for (a) comparing and evaluating the simulation languages of, and (b) interchanging simulation models between, different simulation technology products based on the PN paradigm, such as Simio, AnyLogic, JaamSim, Simul8, ExtendSim, etc.

In Section 7, we report on a JavaScript-based reference implementation of (a fragment of) PNL 0.1.

2 THE BASIC PN CONCEPTS OF ARENA

We restrict our analysis to the core elements of Arena and summarize them in the meta-model shown in Figure 2 below.

An Arena process simulation model is a directed graph, consisting of one or many *Create* nodes, zero or more *Process* nodes, zero or more *Decide* nodes and one or many *Dispose* nodes. Process, Decide and Dispose nodes can be *successor nodes*.

A *Create* node has no predecessor node and exactly one successor node. It repeatedly creates “entities” (of a certain type) that are routed to a successor node. It simulates a sequence of arrival events where the time of the first arrival is given (either explicitly or by default) and the time in-between two arrivals is computed by a *recurrence* function, which typically implements a random variable.

A *Process* node has an implicit input buffer where “entities” routed to the node are placed. According to the Arena documents, a Process node represents an “activity, usually performed by one or more resources and requiring some time to complete” before the processed “entity” can be routed to a successor node.

A *Decide* node has two or more successor nodes. It allows defining stochastic or conditional branching logic, routing an "entity" to a successor node chosen from a set of potential successor nodes.

A *Dispose* node has no successor node. At a Dispose node, "entities are removed from the simulation".

We reconstruct a core fragment of Arena's PN modeling language in the form of a *meta-model* visually expressed as a UML class diagram in Figure 2.

Such a meta-model defines the abstract syntax of a language. It has to be complemented by a definition of its semantics, which, in the case of a simulation language, must be operational. An execution semantics for our meta-model of the PN paradigm can be implemented (e.g., with Java) independently of how the Arena concepts have been implemented in the Arena software package.

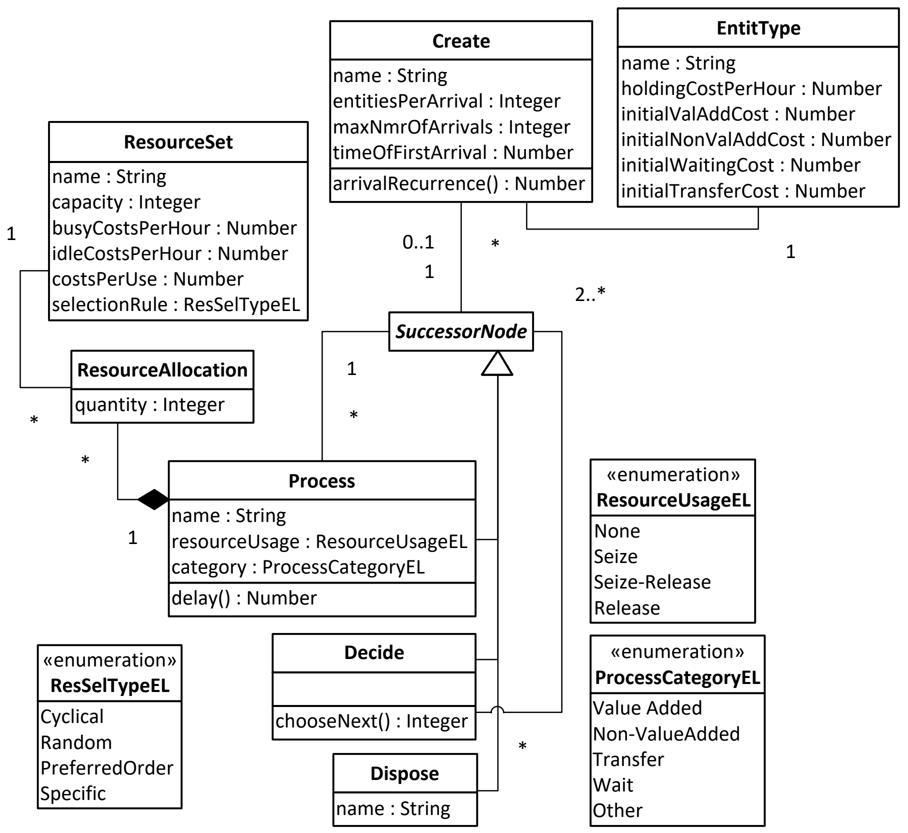


Figure 2: A meta-model of the core elements of Arena.

Notice that in our meta-model of the Arena core,

1. We use an abstract class, *SuccessorNode*, for being able to state control flow constraints requiring that (a) Process, Decide and Dispose nodes *are* successor nodes, and (b) Create, Process, and Decide nodes *have* a successor node.
2. We have renamed several properties of model elements where the original name is not self-explaining:

   (a) instead of "First Creation" in *Create*, we have `timeOfFirstArrival`,

   (b) instead of "Action" in *Process*, we have `resourceUsage`, (None, Seize, Seize-Release, Release),

   (c) instead of "Allocation" in *Process*, we have `category`, (Value Added, etc.).

We have specified three functions: `arrivalRecurrence`(), `delay`() and `chooseNext`(), which are intended to define instance-level functions where the function body is defined per instance, and not per (meta-) class (notice that this approach extends standard UML semantics). For instance, each *Process* element in a model may have its own *delay* function specifying the duration of an activity, typically in the form of a random variable.

The function `arrivalRecurrence`() defined for *Create* elements subsumes the various options (like *random*, *constant* or *expression*) supported by Arena for specifying the recurrence of arrival events.

Likewise, the function `delay`() defined for *Process* elements subsumes the various options (like *constant*, *expression, normal*, *uniform*, etc.) supported by Arena for specifying the delay or duration of a processing activity.

An instance of `ResourceAllocation` essentially is a combination of a resource type and a quantity, which is the minimum information needed for defining a resource allocation in a *Process* element.

The function `chooseNext`() defined for *Decide* elements subsumes the options supported by Arena (of specifying either conditions or probabilities).

3 ANALYSIS

In the following subsections, we express a number of observations about the Arena core concepts described above.

3.1 Arena does not support a general concept of entities or objects

Despite the fact that "entities" are a core element of Arena's PN modeling approach, there is no general concept of *entity types* (or *object types*) in the sense of *Object-Oriented (OO)* modeling and programming in Arena. Arena's concept of entity types only allows defining a name and several costing parameter values for an entity type, but it does not support user-defined attributes or inheritance between entity types.

It is also not possible in Arena to define an entity type, create a pool of instances of this entity type, e.g., each with specific property values, and then iteratively select one of them and introduce it to a processing network simulation run at an entry node.

This shortcoming of Arena has been remedied by object-oriented PN simulation tools, such as *Simio* and *AnyLogic*.

3.2 Arena does not support a general concept of events

Arena does not allow users to define their own types of events like different types of failures of a workstation or critical patient health condition events during an operation. Rather, only certain types of events can be defined implicitly:

When a *Create* element is placed in an Arena model, this implies a hidden definition of an *arrival* event type. When the model is run, the Arena execution engine creates internal arrival events at each *Create* element. Likewise, *Dispose* elements define implicit *departure* event types.

Each *Process* element in a model defines an implicit *activity type*, which is resolved into two implicit event types *start activity* and *end activity*.

An execution semantics for the PN paradigm can be defined on the basis of the discrete event scheduling approach using the four implicit Arena core event type categories *Arrival*, *Start-Activity*, *End-Activity* and *Departure*. This is indicated by Pegden (2010), when he says that all discrete event simulation platforms implement their internal logic using the *event worldview*, regardless of the worldview they present to the user. The event worldview is characterized by viewing a system process as a series of instantaneous events that change the state of the system over time and create follow-up events, such that a model needs to define the types of events in the system and model the state changes that take place, and the follow-up events that occur, when events of those types happen.

3.3 The Arena "flowchart modules" are semantically overloaded

As already explained above, the "flowchart modules" *Create*, *Process* and *Dispose* are semantically overloaded. They define both an object and an associated event type:

1. A *Create* node represents a place or organizational unit where "entities" (of a certain type) arrive, that is, where "entity" arrival events (of a certain type) occur. Thus, it defines (1) an object (the place or organizational unit), (2) an object type (the type of "entities") and (3) an (arrival) event type.
2. A *Process* node represents a non-movable resource (such as a processing machine or service point) where "entities" have to queue up in an input buffer before they are subject to processing (or service) activities of a certain type, possibly involving the use of further resources. An activity is a complex event that at least consists of a pair of a start and an end event. Thus, a *Process* node defines a resource object with an input buffer, and an activity type.
3. A *Dispose* node represents a place or organizational unit where "entity" departure events (of a certain type) occur. Thus, it defines both an object (the place or organizational unit) and an event type.

While the semantic overloading of the "flowchart modules" creates ambiguities that may be confusing for users, it also leads to a concise simulation language that increases the usability of the approach.

3.4 Not all DES models are PN models

The class of DES models is larger than the class of PN models because not all discrete dynamic systems are processing systems. For instance, a public transport system is not a processing system, but it can be captured with some form of DES modeling. Likewise an economy is a discrete dynamic system, but not a processing system. While certain forms of DES allow modeling an economy in a natural way (e.g., an *Object Event Simulation* model of an economy can be run from https://sim4edu.com/sims/20), an economy cannot be modeled as a PN in a natural way.

4 CONSOLIDATING THE ARENA CORE META-MODEL

For obtaining a small core of a general PN modeling language from the Arena core language captured in the meta-model shown in Figure 2 above, we make the following simplifications:

- We do not consider resource usage across several *Process* nodes where one node may seize a resource that is only released later by a successor node. So, we only have two kinds of resource usage: either no resource or a certain quantity of resources from a resource pool is used (seized at the beginning of the processing activity and released at its end). Consequently, we drop the `resourceUsage` attribute and express these two cases by either not having, or having, a resource allocation.
- We do not consider the calculation of cost-based performance indicators and drop the costing parameters of the *Resource* and *Entity* "data modules" as well as the costing-related `category` attribute of *Process* nodes. Since this simplification leaves the *Entity* "data module" with only one attribute, the entity type name, it can be dropped altogether.

For obtaining a coherent and vendor-neutral terminology, we use our own names for the Arena modeling elements listed in Table 2.

Since an important design goal for PNL is using OO modeling as a foundation, both processing objects and resources should be objects that instantiate an object type, which may be part of a type hierarchy in the sense of OO modeling. Consequently, in PNL, (1) a processing object ("entity") is an instance of the output object type of the entry node where the processing object was created or has entered the system, and (2) a resource pool contains resource objects that are instances of a particular object type. This conceptualization allows that the same object (say, a person) may play a resource role (e.g., Surgeon) and be a processing object (a patient), albeit not at the same time, which is not possible to model in Arena.

Table 2: Proprietary terminologies

Arena	*PNL*
Entity	Processing Object
Create	Entry Node
Process	Processing Node
Dispose	Exit Node
Decide	XOR-Gateway
Resource	Resource Pool
entitiesPerArrival	objectsPerArrival

The simplifications, renaming and OO approach described above lead to the meta-model defining our first version of a PN modeling language shown in Figure 3 below.

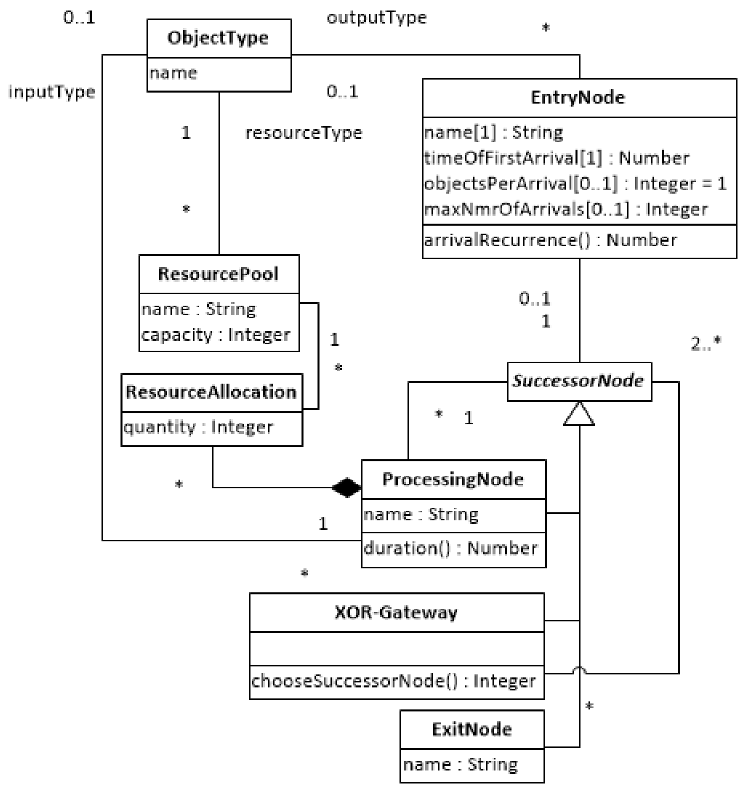


Figure 3: A meta-model for PNL 0.1

In general, in a simulation run, the arrival events occurring at an entry node may introduce processing objects in one of two ways: (1) they are associated with pre-existing objects (e.g., from an object pool), or, otherwise, (2) they trigger the creation of new processing objects, either as instances of the entry node's output type, if there is one, or, otherwise, as instances of a built-in default output type. While in PNL 0.1, as in Arena, only the second method is supported, the first method will be added in PNL 0.2 for matching the corresponding feature of AnyLogic..

A processing node may have any number of resource allocations consisting of a (reference to a) resource pool and a quantity, such as {[resPool:"doctors", quantity:2], [resPool:"nurses", quantity:3]} for a hospital operating room modeled as a processing node.

The Arena model of a DMV shown in Figure 1 above can be expressed as an exemplar model of the PNL 0.1 meta-model. The elements of an exemplar model are instances of elements of the exemplified model. Since in this case the exemplified model is a meta-model defining both meta-classes, such as *ObjectType*, and basic classes, such as *ProcessingNode*, the exemplar model shown in Figure 4 contains both basic classes, such as *Customer*, and instances of basic classes, such as <u>*receptionDesk*</u>.

Notice that a model expressed with PNL 0.1 only defines object types (as the types of resources and processing objects), but no explicit event types, reflecting the situation in Arena, which does not allow defining event types.

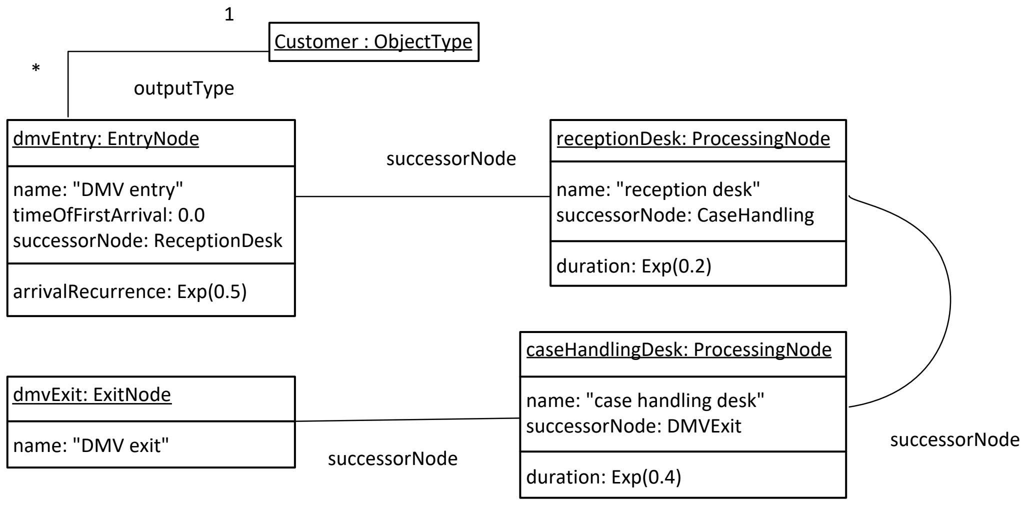


Figure 4: A PNL exemplar model defining the DMV model of Figure 1.

5 MATCHING THE CORE MODELING ELEMENTS OF ANYLOGIC

As can be seen from the AnyLogic core metamodel shown in Figure 5, the overlap between AnyLogic core modeling elements and PNL 0.1, modulo name choices, is pretty large. For instance, a *Source* corresponds to an *EntryNode*, and the attribute *Source::agentsPerArrival* corresponds to *EntryNode::objectsPerArrival*. In addition to these one-to-one correspondences, the AnyLogic core metamodel defines a few elements that have to be added to PNL 0.1:

1. AnyLogic's *Enter* element allows introducing pre-existing objects as processing objects to a PN. In PNL 0.2, this feature is accommodated by adding to the PNL class *EntryNode* (1) an optional property *objectPool* that references an object pool, and (2) an operation *push* for pushing processing objects to the successor node.
2. AnyLogic's *Service::queueCapacity* attribute is added to the PNL class *ProcessingNode*. This requires to add the value *blocked* to the enumeration of processing node status values because it creates the possibility that a processing node gets blocked when its *queueCapacity* has been reached.
3. For accommodating AnyLogic's *Event* element, a new class *EventType* is added to PNL 0.1 with suitable subclasses such as *ExogenousEventType* and *TimeEventType*.

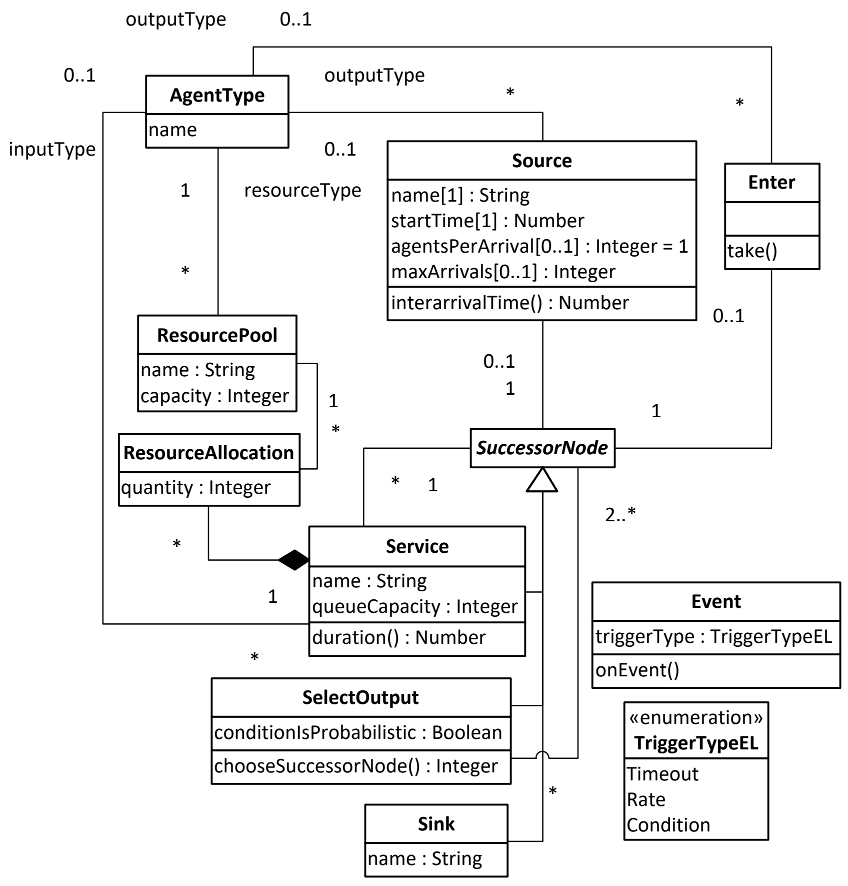


Figure 5: A meta-model for a core fragment of AnyLogic

6 EXECUTION SEMANTICS

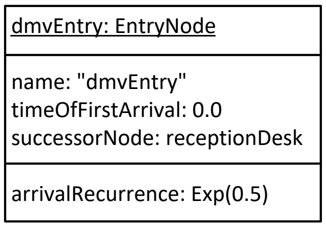
By mapping models expressed with PNL to *Object Event Simulation (OES)* models, the operational semantics defined for OES models in (Wagner 2017a) provides an execution semantics for PN simulation models. The mapping is based on the well-known pattern that an activity corresponds to a pair of coupled events: an activity start event and a succeeding activity end event.

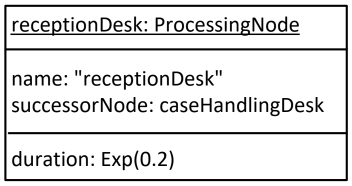
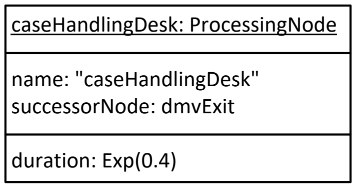
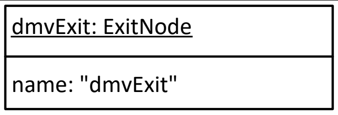
In Table 3, we illustrate the main principles of the mapping of a PNL model to an OES model with our example presented in Figure 4 above.

Object Event Modeling and Simulation (OEM&S) (Wagner 2018) represents a general Discrete Event Simulation approach based on *object-oriented modeling* and *event scheduling*. In OEM&S, object types and event types are modeled as special categories of classes in a UML Class Diagram. *Random variables* are modeled as a special category of class-level operations constrained to comply with a specific probability distribution such that they can be implemented as static methods of a class. *Queues* are not modeled as objects, but rather as ordered association ends, which can be implemented as collection-valued reference properties. Finally, *event rules*, which include *event routines*, are modeled both as BPMN/DPMN process diagrams and in pseudo-code such that they can be implemented in the form of special *onEvent* methods of event classes.

Table 3: Mapping PNL model elements to OES model elements

PNL model element	OES model element(s)

 dmvEntry: EntryNode
name: "dmvEntry"
timeOfFirstArrival: 0.0
successorNode: receptionDesk
arrivalRecurrence: Exp(0.5)
	An entry node is mapped to (1) an object of type *oes.EntryNode*, and (2) an initial event of type *oes.Arrival*:
	Object{ type: oes.EntryNode, name:"dmvEntry", successorNode: receptionDesk}
	Event{ type: oes.Arrival, entryNode: dmvEntry, occTime: 0.0}
	When the simulator processes such an arrival event, it creates an object of type *oes.ProcessingObject*, pushes it to the input buffer of the successor node, schedules an *oes.Processing-ActivityStart* event for the successor node and schedules the next such arrival event using the *arrivalRecurrence* function.

 receptionDesk: ProcessingNode name: "receptionDesk" successorNode: caseHandlingDesk duration: Exp(0.2)	A processing node is mapped to a corresponding object: Object{ type: oes.ProcessingNode, name:"receptionDesk", successorNode: caseHandlingDesk} When the simulator processes an *oes.ProcessingActivityStart* event at a processing node, it schedules an *oes.Processing-ActivityEnd* event with a delay given by the duration function. When the simulator processes an *oes.ProcessingActivityEnd* event at an intermediate processing node, it pushes the processing object to the input buffer of the successor processing node and schedules an *oes.ProcessingActivityStart* event for that node.
 caseHandlingDesk: ProcessingNode name: "caseHandlingDesk" successorNode: dmvExit duration: Exp(0.4)	Object{ type: ProcessingNode, name:"caseHandlingDesk", successorNode: dmvExit} When the simulator processes an *oes.ProcessingActivityEnd* event at the processing node "caseHandlingDesk", it pushes the processing object to the input buffer of the successor node "dmvExit" and schedules an *oes.Departure* event.
	Object{ type: oes.ExitNode, name:"dmvExit"}

7 IMPLEMENTING PNL 0.1 MODELS WITH OESJS

PNL can be implemented as an extension of the JavaScript-based Object Event Simulation framework OESjs, which is presented in (Wagner 2017b) and available online from https://sim4edu.com.

As an illustrative example, we present the code of the DMV model shown in Figure 1, Figure 4 and Table 1 above. The example is available, and can be run, online from https://sim4edu.com/sims/11.

On top of the OESjs framework, the JavaScript code for this example consists of a simulation scenario that defines an initial state consisting of the four PN nodes that have been discussed above:

```
sim.scenario.initialState.objects = {
  "1": {typeName: "eNTRYnODE", name:"dmvEntry",
        successorNode: 2,
        arrivalRecurrence: function () {
            return rand.exponential( 0.5);}
       },
  "2": {typeName: "pROCESSINGnODE", name:"receptionDesk",
        successorNode: 3,
        randomDuration: function () {
            return rand.exponential( 0.2);}
       },
  "3": {typeName: "pROCESSINGnODE", name:"caseHandlingDesk",
        successorNode: 4,
        randomDuration: function () {
            return rand.exponential( 0.4);}
       },
  "4": {typeName: "eXITnODE", name:"dmvExit"}
};
```

Notice how the *arrivalRecurrence()* function is defined as an instance-level function of the entry node object. Likewise, for all the processing node objects, *randomDuration()* functions are defined as instance-level functions.

8 TOWARDS PNL 1.0

We plan to incrementally extend PNL 0.2 for obtaining PNL 1.0 by adding

1. core modeling elements from *Simio* such that PNL gets aligned with it;
2. further, more advanced, modeling elements from Arena, Simio and AnyLogic, such as *resource selection rules*, *task priorities*, *output buffers*, *batching*, *failures/repairs* and *processing activity variants* (all defined in processing nodes), *failures/repairs* of resources (defined in resource object types);
3. modeling elements for accommodating the PN concepts of Queueing Theory.

8.1 Accommodating the PN concepts of Queueing Theory

In the mathematical theory of queueing, which dates back to the 1950's, the term "processing network" has been used only in more recent years, e.g., by Harrison (2000) writing about "open processing networks" or Williams (2016) writing about "stochastic processing networks", as a generalization of the term "queueing network".

Williams (2016) characterizes (stochastic) processing networks by three key components: (a) the buffers ("classes") for storing waiting processing objects ("jobs"), (b) the processing resources ("servers"), and (c) the processing activities, stating that "an activity consumes from certain classes, produces for certain (possibly different) classes, and uses certain servers in the process".

This means that a processing node may have multiple input buffers (with various pull policies) and multiple output buffers such that its processing activity may process multiple input objects of different types, possibly concurrently, allocating multiple resources playing different roles (with various allocation policies), and create multiple output objects of different types.

Williams points out that most PNs cannot be analyzed exactly, but only with the help of approximate models, such as *fluid models* and *diffusion models* (Williams 2016).

9 CONCLUSIONS

We have presented a meta-model obtained from analyzing the Arena simulation language defined by its "flowchart modules" and "data modules". This meta-model defines a *Processing Network Language (PNL)* the goal of which is to capture the Processing Network (PN) simulation paradigm. We have implemented a core fragment of it in the OESjs simulation framework (available from https://sim4edu.com) and used it in teaching the concepts of the PN paradigm.

After completing PNL 1.0, we plan to analyze other simulation software products by determining the degree, with which they match PNL. We expect that this approach will allow to assess if PNL 1.0 can already be used as a basis for evaluating the completeness of simulation software products and as a format that allows interchanging simulation models between different tools based on the PN paradigm, or if it needs to be further extended. The evolving family of PNL versions will be available online at https://sim4edu.com/reading/PNL.

REFERENCES

van der Aalst, W.M.P. 2014. "Business process simulation survival guide". In: J. vom Brocke and M. Rosemann (eds), *Handbook on business process management*, vol 1, 2nd edn. Springer, Heidelberg, pp 337–370.

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

Harrison, J. M. 2000. "Brownian models of open processing networks: canonical representation of workload". *Annals of Applied Probability*, 10(1):75–103.

Martens, A., P. Fettke, and P. Loos. 2015. "Inductive Development of Reference Process Models Based on Factor Analysis". In: Thomas. O. and F. Teuteberg (Eds.): *Proceedings of 12th International Conference Wirtschaftsinformatik (WI 2015)*, Osnabrück, S. 438-452.

Pegden, C.D. and D.A. Davis. 1992. "Arena: a SIMAN/Cinema-based hierarchical modeling system". In *Proceedings of the 24th Winter Simulation Conference (WSC '92)*. ACM, New York, NY, USA, 390–399.

Pegden, C.D. 2010. "Advanced Tutorial: Overview of Simulation World Views." In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson et al, 643−651. Piscataway, NJ: IEEE.

Wagner, G. 2017a. "An Abstract State Machine Semantics for Discrete Event Simulation". In *Proceedings of the 2017 Winter Simulation Conference*. Piscataway, NJ: IEEE.

Wagner, G. 2017b. "Sim4edu.com – Web-Based Simulation for Education". *Proceedings of the 2017 Winter Simulation Conference*. Piscataway, NJ: IEEE.

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

Williams, R.J. 2016. "Stochastic Processing Networks". *Annual Review of Statistics and Its Application* 3:1, 323–345.

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. 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.