Modeling of Non-standard Queuing Policies - An Invitation to ARGESIM Benchmark C22

Peter Junglas[1*], Thorsten Pawletta[2]

[1]Dep. of Engineering "Dr. Jürgen Ulderup", PHWT Vechta/Diepholz, Schlesierstr. 13a, 49356 Diepholz, Germany; [*]*peter@peter-junglas.de*

[2]Wismar Univ. of Applied Sciences, Fac. of Engineering, Research Group CEA, PF 1210, 23952 Wismar, Germany

Abstract. The recently published ARGESIM benchmark C22 'Non-standard Queuing Policies' studies three queuing models, where the queues utilize more complex policies than the standards FIFO, LIFO or priority. *Jockeying queues* allow entities to switch to a shorter queue, in *reneging queues* entities leave a queue after a maximal waiting time, and *classing queues* change the entity order according to an entity attribute ("class") at the call of an external operator.
To encourage the simulation community to publish benchmark solutions this talk will explain the tasks in some detail and comment on the lessons learned from a first implementation.

Introduction

The general transition of modeling from programming or text-based modeling languages to graphical component-based methods has made possible to study highly complex models without explicit programming knowledge on the side of the modeler. On the other hand standard text books like [1] are still using explicit C code for the description of their models and algorithms – and for good reason! In standard modeling cases the set of prebuilt components is often sufficient. But for special situations it can be difficult to work around the limitations of the given building blocks. Furthermore the exact behaviour of the components is generally not defined in every detail. This can lead to all kinds of problems, ranging from awkward workarounds to unexpected behaviour [2].

Different approaches can be used to cope with this problem: One can supplement the components with an interface that allows the easy integration of callback functions or one can provide means to integrate self-programmed components. Both ways have been offered in the redesign of MathWorks' SimEvents library [3]. More in the spirit of the graphical programming paradigm, one could try to identify a set of basic components with a precisely defined behaviour, that allow to model non-standard situations in a clear and easily understandable way.

The ARGESIM benchmark C22 'Non-standard Queuing Policies' [4] addresses this problem for the modeling of queuing systems. To this end it defines three different tasks, where a standard queuing component is not sufficient, either because some entities can leave the queue prematurely or because their queuing order can be changed dynamically. The prospective solutions will help to clarify how one can deal with such situations using common graphical modeling tools. A first implementation using the text-based MatlabGPSS language [5] gives some clues, which features could be helpful and which are still missing (in MatlabGPSS) – and it points out, where graphical tools still have problems, that should be adressed in future modeling environments.

1 Basic Queuing System

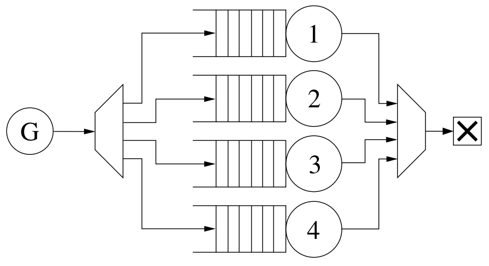

Figure 1: Basic queueing system with four queues.

A simple queueing system consisting of four queues and servers is used as a starting point and reference for all benchmark models (cf. fig. 1). It consists of a generator that creates a given number of entities with fixed or stochastic interarrival times and four FIFO queues and corresponding servers with a capacity of one. Entities choose the shortest line (including the server allocation) and leave the system after being served. The simulation stops after all entities have left.

Two different versions have to be modelled: a small one with fixed interarrival and service times, a larger one with stochastic values. All details, such as the total number of entities and values or distributions of interarrival and service times, are given in the benchmark definition [4], together with the required output values and plots.

To define the models completely, particularly the deterministic version, one has to specify the order of concurrent events. This is done in the following way:

1. an entity leaves a server,
2. a queued entity enters a server,
3. a new entity enters the system and chooses a queue.

It is an interesting question, how a simulation environment allows to fix the order of concurrent events. Therefore the benchmark includes an optional variant of the deterministic model, where this order is changed to

1. a new entity enters the system and chooses a queue,
2. an entity leaves a server,
3. a queued entity enters a server.

Another problem especially for graphical environments is the modeling of large systems. To study this, the benchmark includes an optional variant of the stochastic model with 40 queues and servers.

2 Jockeying Queues

A rather common phenomenon in everyday queueing systems is *jockeying*, i.e. the process that an entity moves from the end of a queue to another shorter queue (cf. fig. 2). This not only happens, when the “entities” are humans (e.g. in supermarkets or on motorways), but it can be used to achieve a better load balancing of servers in computer networks or production lines.

The C22 benchmark asks for variants of the two basic models, where jockeying occurs, whenever a queue (incl. server) is at least shorter by 2 than another one. It defines the behaviour if there are more than one possible source or destination queues and fixes the order of concurrent events. Moreover it requires additional output describing the jockeying events.

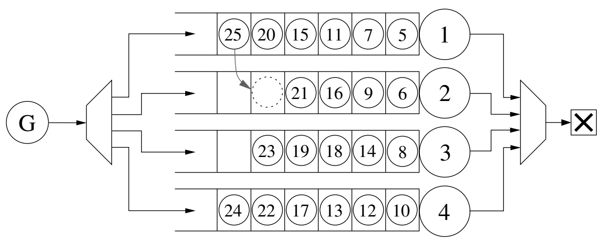

Figure 2: Queueing system with jockeying.

The main problem here is of course, how to detach the last entity of a standard FIFO queue. There are several tricks to achieve this [2], but since they complicate the model and increase the number of events considerably, a simpler solution would be preferred – if the used simulation environment allows for one.

3 Reneging Queues

In some applications an entity leaves a queue before it is served, a behaviour known as *reneging* (cf. fig. 3). This can be a customer, who has lost his patience, food, whose shelf life has been reached, or a workpiece that has to be reheated.

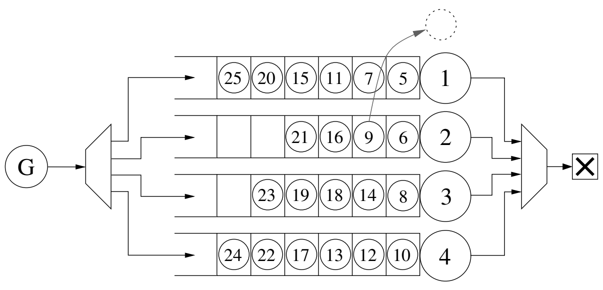

Figure 3: Queueing system with reneging.

The benchmark requests the implementation of model variants where entities renege after a fixed maximal waiting time. This problem seems to be harder than the jockeying case, because now an entity in the middle of the queue has to be released prematurely.

4 Classing Queues

The last benchmark task is inspired by a typical situation during the boarding of a plane: An operator calls “all passengers with seat numbers 15 – 30” to the front of the queue. It assumes that entities have an additional attribute called *class*, which has a positive integer value and is assigned at entity creation in a round-robin way or stochastically.

The standard models are augmented with an operator that at certain times calls for a class number, whereupon all entities with this class procede to the front of their queues. The relative order of the entities within this class remains intact, as does the order of the other entities among themselves (cf. fig. 4).

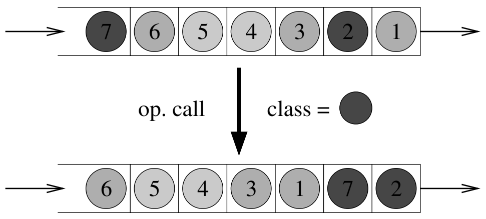

Figure 4: Result of an operator call (class $\hat{=}$ color).

Further practical examples could be the routing of network packets ("now all packets of a given video file") or the loading of a truck ("now all boxes of a given size").

As always the benchmark fixes all details such as the exact behaviour of the operator, the assignment of classes, the order of concurrent events and the output data. It is important to note that there is only one global operator that defines the current class for all queues at once. The case of individual operators for each queue is of practical interest as well, but has not been included in the benchmark, since it doesn't add substantial difficulties to the implementation.

The classing queue model is probably the most challenging of the benchmark, since it requests a dynamical reordering of the entities within the queue. On the other hand it doesn't require an extra queue output like the jockeying and reneging queues, which makes it similar to the standard priority queue, with the essential difference, that the meaning of "high priority" changes at runtime.

5 Benchmark Implementation using MatlabGPSS

A first implementation of the C22 benchmark has been published [5], it is based on MatlabGPSS [6]. Its basic findings may be helpful for further implementors of the benchmark and will be summarised here.

GPSS [7] is one of the oldest existing modeling languages. It uses the transaction-based paradigm to model discrete event systems, and though being somewhat outdated, it is still a good example of a simple language that uses only a few basic constructions to provide very wide modeling capabilites. The freely available implementation MatlabGPSS [8] combines GPSS statements with general Matlab code. This makes the implementation of complex control structures and the compilation of statistical and graphical results much easier than relying on pure GPSS constructs, thereby allowing to concentrate on the basic questions of queue design.

GPSS is text-based and contains statements for the generation and destruction of entities, the entering and leaving of queues, the reservation and freeing of servers and the delaying of an entity for a given (service) time. Each entity can store a set of parameters, auxiliary functions provide the current number of entities stored in a queue or a server.

For complex queueing strategies one can use so called *user chains*, which are more flexible than standard queues. Entities join a user chain with the `link` statement, entering at the front or end or according to a parameter value. The `unlink` statement allows any entity to free one or more entities from a user chain and to route them to arbitrary places. The exact possibilities of `unlink` depend on the specific GPSS implementation; in MatlabGPSS entities can only be extracted from the front or back end of a user chain.

Using these standard GPSS methods the basic model can be implemented easily. Finding the index of the shortest line is done with a Matlab function. Scaling up the model to 40 queues is then just a matter of setting a dimension parameter. And the changing of the order of concurrent events can be done by applying the GPSS `priority` command that allows to change the priority of an entity dynamically.

To implement the jockey queues, one has to move entities between different places in the model. This is done with the GPSS statement `transfer` and labelling of statements, similar to a classical *go-to*. In a graphical modeling environment similar models would use routing elements such as gates and switches. The main problem, namely to extract an entity from the back end of a FIFO queue, is trivial here, since the `unlink` command allows to extract entities from both ends of the queue.

But the slightly harder problem of the reneging queues, where one has to extract an entity from the middle of the queue, can not be handled so easily in MatlabGPSS. Therefore one has to rely to the *clone queue* trick [2]: An entity that enters a queue is cloned, one copy waits for the total reneging time, the other one tries to get the server. A bookkeeping variable is set, whenever one of the pair is ready, and checked before the action of the other one. A clone that comes late, is simply terminated.

For the implementation of the classing queues every queue is cut into two sequential queues with a gate in between, that is opened whenever the operator calls for a new class (cf. fig. 5).

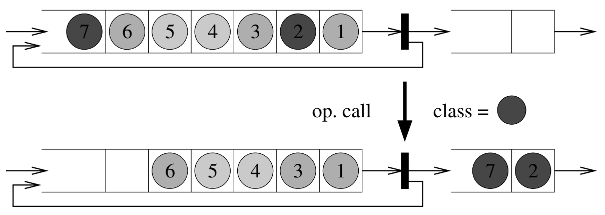

Figure 5: Implementation of the classing queue.

The first queue (at the back end) stores all incoming entities, while the second queue at the front receives all entities with the current class. Since one cannot pick only the matching entities from the first queue, one has to release all entities and route the unwanted ones back into the first queue. This scheme is a variant of the *shuffle queue* from [2].

6 Conclusions

Since all benchmark tasks are rather small and don't need a special mathematical or modeling background, the benchmark is suited for beginners in the field of modeling and simulation. Nevertheless it is probably a challenge for many of the current discrete simulation systems.

As long as perfect solutions to the benchmark problems have not been found generally or in the simulation environment used, second best solutions using special tricks are very welcome. They not only show the state of the art (and its deficiencies), but can help the practitioner to implement non-standard queueing problems in the program at hand.

References

[1] Law AM. *Simulation Modeling and Analysis.* McGraw-Hill, New York, 5. ed. 2014.

[2] Austermann L, Junglas P, Schmidt J, Tiekmann C. Conceptional problems of transaction-based modeling and its implementation in SimEvents 4.4. *Simulation Notes Europe SNE.* 2017; 27(3): 137–142. doi: 10.11128/sne.27.tn.10383

[3] Li W, Mani R, Mosterman P. Extensible discrete-event simulation framework in SimEvents. *Proc. 2016 Winter Simulation Conference*; 2016 Dec; Arlington. New Jersey: IEEE. 943-954.

[4] Junglas P, Pawletta T. Non-standard Queuing Policies: Definition of ARGESIM Benchmark C22. *Simulation Notes Europe SNE.* 2019; 29(3): 111-115. doi: 10.11128/sne.29.bn22.10481

[5] Junglas P, Pawletta T. Solving ARGESIM Benchmark C22 'Non-standard Queuing Policies' with MatlabGPSS. *Simulation Notes Europe SNE.* 2019; 29(4): 199-205. doi: 10.11128/sne.29.bn22.10496

[6] Pawletta T, Drewelow W, Pawletta S. Discrete Event Simulation in Interactive Scientific and Technical Computing Environments. In: *Proc. 12th European Simulation Multiconference on Simulation*; 1998 Jun; Manchester. 529-533. ISBN 1-56555-148-6.

[7] Schriber TJ. *An introduction to simulation using GPSS/H.* New York: John Wiley & Sons, Inc.; 1991. 437 p.

[8] Pawletta T., et al. *The MATLAB GPSS Toolbox.* Online: `http://www.cea-wismar.de/tbx/mgpss/` (called 2020-01-07).