Solving ARGESIM Benchmark C22 'Non-standard Queuing Policies' with MatlabGPSS

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

SNE 29(4), 2019, 199-205, DOI: 10.11128/sne.29.bn22.10496
Received: November 18, 2019;
Accepted: November 22, 2019
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna,
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. The ARGESIM benchmark C22 'Non-standard Queuing Policies' deals with queueing systems, where entities can leave a queue in a dynamically changed order. It is solved here with MatlabGPSS, a Matlab-based implementation of the well-known GPSS modeling language. Though quite old, it still has its merits, such as precisely defined and flexible methods to deal with concurrent events and the concept of *user chains*, which are more flexible than standard queueing blocks.

Introduction

The ARGESIM benchmark C22 [1] is concerned with special queueing systems, which use a FIFO discipline generally, but where some entities can leave the queue prematurely or in a dynamically changed order. Its implementation can be difficult in modeling environments, which provide only monolithic queueing blocks that don't grant access to the entities waiting in a queue [2].

GPSS [3] 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 It uses the transaction-based paradigm to model discrete event systems, and though being somewhat outdated, it is still a good example of a simple language that uses only a few basic constructions to provide very wide modeling capabilites. Therefore it should be a good tool to study principal questions such as appropriate concepts for modeling of non-standard queues.

Among the few still existing implementations we chose MatlabGPSS [4], because it is easily available [5] and 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 (`generate`/`terminate`), the entering and leaving of queues (`queue`/`depart`) and the reservation and freeing of servers (`seize`/`release`). The `advance` statement delays an entity for a given time and is used to model service times. Each entity can store an arbitrary number of parameters `P(i)`, similar to attributes in other modeling environments. Additional functions `Q(i)`, `F(i)` and `CH(i)` return the number of entities stored in the i-th queue, server (*facility*) or user chain.

Despite its name the `queue` statement does not implement any queueing discipline, it is only used for accounting purposes, e.g. to count the number of entities entering a queue. Usually a simple FIFO discipline within a priority level of entities is defined automatically by the internal scheduling of the GPSS system. For more 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 entitiy 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.

In the following we will implement all tasks of the C22 benchmark, with a special focus on the advantages and shortcomings of the queueing mechanisms in MatlabGPSS and the available methods to specify the ordering of concurrent events. For the exact definition of all systems and parameters refer to the benchmark definition [1]. The models and scripts necessary to reproduce all results presented here are available from [6].

1 Basic Queuing System

1.1 Description of the Basic Model

The basic model can be easily implemented in GPSS using standard methods. The complete source code – omitting most variable initialisations and code for statistical results and plots, and adding line numbers for easier reference – is given by:

```
 1  init
 2    nQ = 4;
 3  model
 4    generate(tA,0,1,nE,0)
 5    [~, nr] = min(Q(1:nQ) + F(1:nQ));
 6    P(3, nr);
 7    queue(P(3),1)
 8    seize(P(3))
 9    depart(P(3),1)
10    advance(tS,0)
11    release(P(3))
12    terminate(1)
```

The only interesting part is in line 5 and 6, where the number of the shortest queue (including server allocation) is computed and stored in an entity parameter.

1.2 Results of Deterministic Model

Entity ids are stored in a parameter, the queue lengths are collected using GPSS function `Q` and plotted with standard Matlab methods. The plots of the ids of the last 20 outgoing entities over their exit time and of the total queue length are shown in Figure 1.

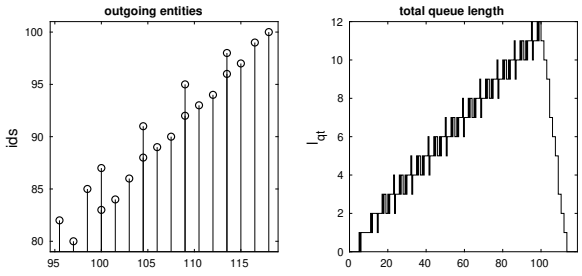

Figure 1: Outgoing ids and total queue length for the basic model.

1.3 Results of Stochastic Model

The stochastic model uses MatlabGPSS functions `expod` and `triangd` to create exponentially or triangularly distributed random numbers. For the computation of waiting times the entry times are stored in an entity parameter. Table 1 displays statistical results for the total queue length and entity waiting times.

l_{qt}		$t_{q,i}$	
avg	**max**	**avg**	**max**
21.131	45.000	24.075	52.696

Table 1: Total queue lengths and queue waiting times for the basic model.

1.4 Variants for Handling Simultaneous Events

If several entities can move at the same time, their order can be defined in GPSS using the command `priority N`, which assigns an integer priority $N \geq 0$ to an entity, where entities with larger priorities take precedence over those with smaller priorities. The concurrency order of entities with the same priority is not defined in general, but depends on the GPSS implementation: While in GPSS/H the order is specified, in GPSS World it is explicitely randomised [7]. Therefore one should not rely on any given order, but use priorities to guarantee the required behaviour.

For the basic model there are two critical situations to consider:

1. An entity E_1 wants to leave a server at the same time that a queued entity E_2 wants to enter a server or a new entity E_3 is generated.

2. A queued entity E_2 wants to enter a server at the same time that a new entity E_3 is generated.

To get the required ordering E_1 needs a higher priority than E_2, which needs a higher priority than E_3. This can easily be achieved by generating events with priority 0 and inserting the statements `priority 1` between lines 7 and 8 of the source code above and `priority 2` between lines 9 and 10. Using knowledge about the internal scheduling algorithm one could reduce this to one priority statement.

To implement the alternative order, where a new entity joins a queue, *before* another one leaves the server, one generates all entities with priority 1 and lowers the priority to 0 before an entity releases its server, e. g. between lines 9 and 10 of the source code above. This works, since the second critical situation is not possible here. The result is shown in Figure 2.

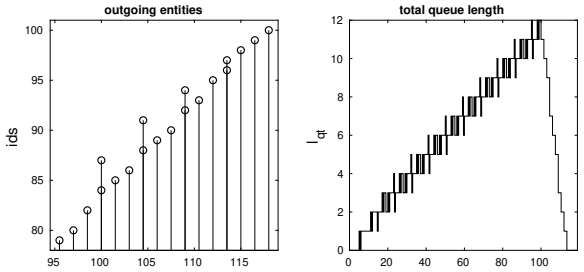

Figure 2: Outgoing ids and total queue length for the concurrency variant.

1.5 Comparison of Standard and Large Version

The implementation of the large model in MatlabGPSS is trivial: One only has to change the parameters, especially $n_Q = 40$, and gets the results shown in Table 2.

l_{qt}		$t_{q,i}$	
avg	**max**	**avg**	**max**
241.598	488.000	27.706	62.494

Table 2: Total queue lengths and queue waiting times for the large model.

2 Jockeying Queues

2.1 Description of the Model

To implement the jockey queue system, 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*. Since such source code is hard to read, it is represented here as a flow chart in Figure 3. In a graphical modeling environment similar models would use routing elements such as gates and switches.

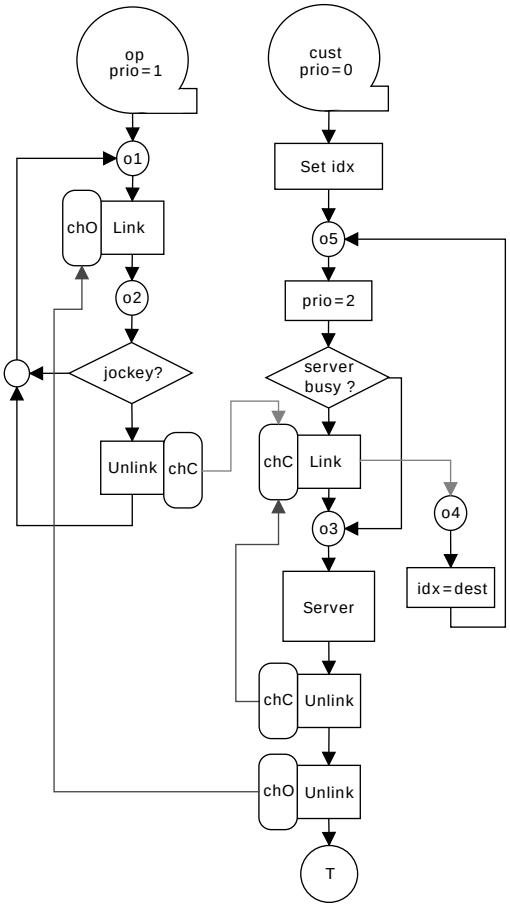

Figure 3: Flowchart of the jockey queues model.

The left side shows the creation of a single entity `op` that acts as an operator: It checks, whether jockeying is possible and moves the corresponding entity from the end of its queue to the end of the shorter one. Since jockeying can only happen when a queue gets shorter after service completion, the operator entity waits in a user chain `chO`, until it is unlinked by a leaving entity. The actual computation of source and destination queues is done with a simple Matlab function.

The right side describes the life of a normal customer entity: After creation it computes the number `idx` of the shortest queue available, then it enters the user chain `chC(idx)`, if the server is busy. It is unlinked either from the head of the queue by a served entity to get to the server next or from the end by the operator to jockey to a different queue.

To make sure that jockeying happens before the generation of a new entity, the operator is generated with priority 1, while the customer entities start with priority 0. Their priority is raised to 2 before they enter a chain or a server, so that servicing and moving up have precedence over jockeying.

2.2 Results of Deterministic Model

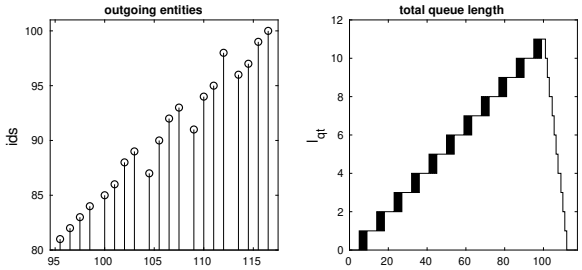

Figure 4: Outgoing ids and total queue length for the jockeying model.

The requested plots are shown in Figure 4 and the first five and last five jockeying events in Table 3.

t	id	src	dest
6.5	6	1	2
7.5	7	1	3
8.5	8	1	4
11.0	10	1	2
12.0	11	1	3
..	..	..	..
89.5	89	2	4
93.0	92	2	3
94.0	93	2	4
98.5	98	3	4
103.0	100	1	4

Table 3: First and last five jockeying events.

2.3 Results of Stochastic Model

To compute total queue waiting times for jockeying entities, the single queue waiting times are accumulated in entity parameters. The statistical results are displayed in Table 4. The number of jockeying events in the corresponding run was 131.

l_{qt}		$t_{q,i}$	
avg	max	avg	max
20.842	45.000	23.618	51.408

Table 4: Total queue lengths and queue waiting times for the jockey model.

3 Reneging Queues

3.1 Description of the Model

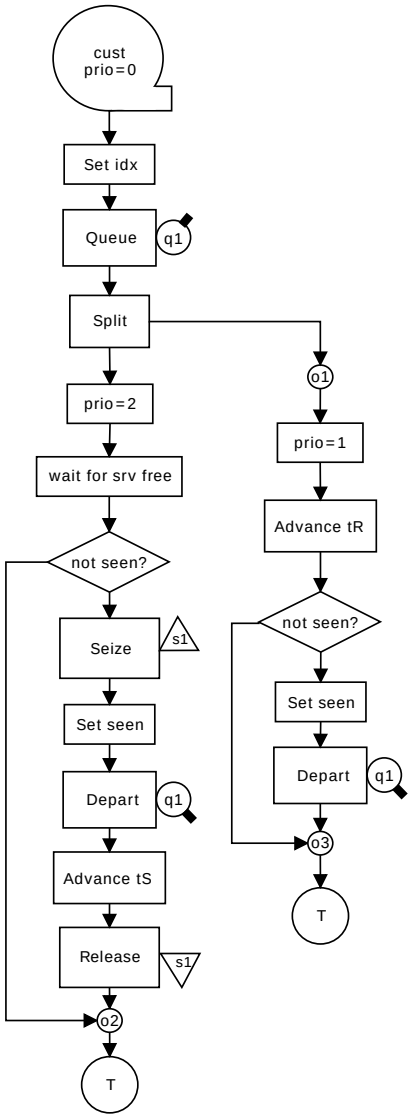

Figure 5: Flowchart of the renege queues model.

The chosen implementation of the reneging queue is similar to the *clone queue* presented in [2]. It uses the GPSS statements `split` to create a copy of an entity and `gate_fnu()`, which halts an entity until a given server is free. The basic idea is shown in Figure 5: After entering a queue an entity is cloned, one copy waits for the total reneging time `tR`, the other one tries to get the server. A bookkeeping variable `seen` 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.

To obtain the required order of concurrent events, entities are generated with priority 0. The priority is raised to 1 for the clones that are going to renege, and to 2 for those that wait for the server.

3.2 Results of Deterministic Model

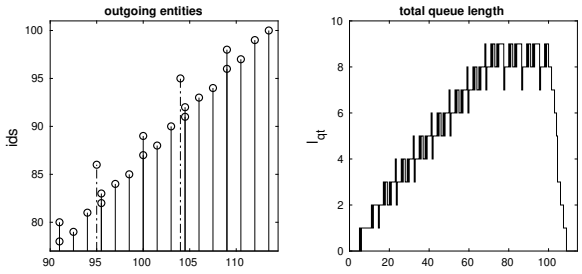

Figure 6: Outgoing ids and total queue length for the reneging model.

Figure 6 shows the plots of the ids of the last outgoing entities including reneging entities and the total queue length. The reneging events are listed in Table 5.

t	id
77.0	68
86.0	77
95.0	86
104.0	95

Table 5: Reneging events.

3.3 Results of Stochastic Model

The statistical results are displayed in Table 6. The number of reneging events in the corresponding run was 53.

l_{qt}		$t_{q,i}$	
avg	max	avg	max
4.615	15.000	4.882	9.000

Table 6: Total queue lengths and queue waiting times for the renege model.

4 Classing Queues

4.1 Description of the Model

The implementation of the classing queue uses a single entity for the operator and two user chains for each classing queue: `c1` for incoming entities and `c2` for the entities, whose class is currently being served (cf. Figure 7). New GPSS constructs used here are logical variables that work like gates in graphical environments, together with the functions `logic_s()`/`logic_r()` to set/reset a variable and `gate_ls()` to wait until a variable is set.

After an initial waiting time t_C the operator entity checks whether there are any entities to be called, if necessary it waits until entities arrive. Then it computes the next active class and unlinks all customer entities from the chain `c1`. After the customer entities of the current class have been served, the operator loops to call the next class.

New customer entities first enter the `c1` chain, choosing the shortest queue as always. When the operator calls for a new class, the matching entities proceed to their `c2` chain and wait to be served, the other ones reenter `c1`. This behaviour is similar to the shuffle queue pattern in [2]. Before leaving the system after service, a customer entity checks, whether there are still entities of its class being served, and wakes up the operator, if not.

Due to the complex "jockeying" structure of the class queue one needs a bit of fine-tuning to obtain the requested order of concurrent events: The operator is created with priority 0, the customer entities with priority 1. This guarantees that the operator call always comes last among concurrent events.

When an entity enters the queue, its priority is risen to 3, so that entering and leaving of the server will preceed the arrival of a new entity. After the entity has left the server and freed the next entity from its chain, its priority is lowered to 2 so that its successor can enter the server, before the server state is checked.

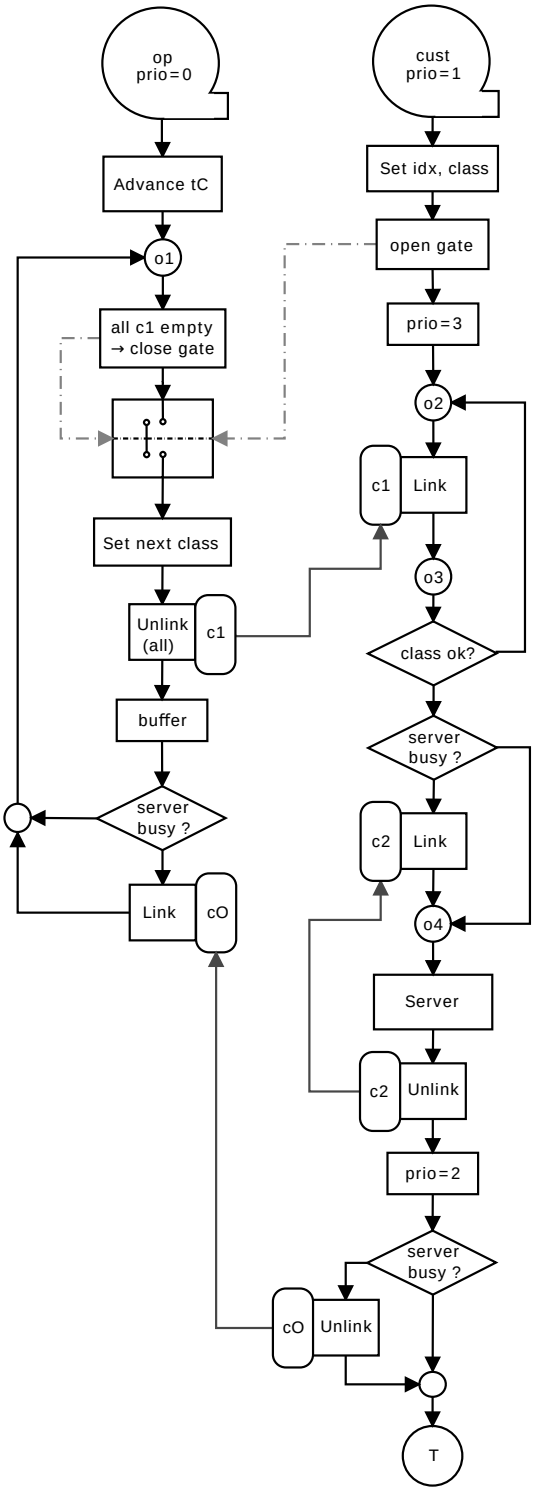

Figure 7: Flowchart of the class queues model.

After the operator has unlinked all entities in the `c1` chains, it has to wait, until any of the freed entities have claimed a server, before checking. This is done with the GPSS `buffer` statement that gives up immediate control and moves the calling entity to the end of the chain of concurrent events.

4.2 Results of Deterministic Model

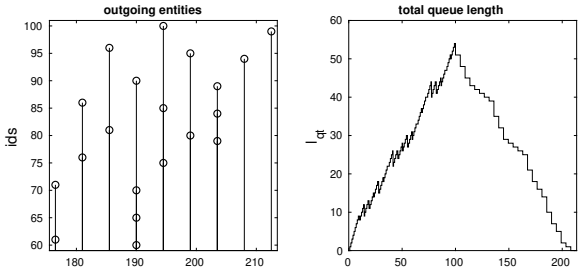

Figure 8: Outgoing ids and total queue length for the classing model.

The plots of the ids of the last outgoing entities and the total queue length are shown in Figure 8, average and maximal queue waiting times per class in Table 7.

class	1	2	3	4	5
avg	66.80	60.85	41.85	50.98	60.77
max	121.50	104.00	77.00	120.00	125.50

Table 7: Queue waiting times per class (deterministic model).

4.3 Results of Stochastic Model

The usual statistical results are displayed in Table 8, average and maximal queue waiting times per class in Table 9.

l_{qt}		$t_{q,i}$	
avg	**max**	**avg**	**max**
82.615	152.000	122.286	279.325

Table 8: Total queue lengths and queue waiting times for the class model.

class	1	2	3	4	5
avg	130.99	118.26	109.18	124.43	126.93
max	279.32	255.62	240.06	273.24	272.97

Table 9: Queue waiting times per class (stochastic model).

5 Conclusions

The combination of basic, but flexible GPSS statements with the versatility of the Matlab environment has made possible a straightforward implementation of all benchmark tasks. Matlab has been especially useful for the overall control structure of the programs, for computations like the determination of source and destination in the jockeying case and for calculating and plotting results.

The immediate advantages of a textual modeling language as compared to a graphical environment have been shown by the trivial implementation of a system with 40 queues. Another bonus is the handling of concurrent events: The behaviour can be flexibly adapted using priorities and the `buffer` statement.

However, the central point of this investigation, namely the flexibility of queue modeling constructs, has shown mixed results: On the one hand the separation of queue storage (`link`) and queue control (`unlink`) with different entities simplified the control structure of the models, and the possibility to extract entities from the front and the back end of a queue allowed for a simple implementation of the jockey queue. On the other hand, for the renege and class queues we had to use clumsy constructs like the clone and shuffle queues, because MatlabGPSS provides no direct access to entities within a queue. Here other GPSS implementations had more options, e. g. the extraction of entities using boolean expressions containing entity parameters [3].

In view of the age of MatlabGPSS an appropriate extension seems to be pointless. Nevertheless the flexibility and preciseness of GPSS are still outstanding features. Whether modern graphical environments can compete, will hopefully be seen by future implementations of the C22 benchmark.

Acknowledgement

The first author (P. J.) likes to thank the team of the CEA research group in Wismar for the warm hospitality extended to him and many helpful discussions.

References

[1] Junglas P, Pawletta Th. 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

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

[4] Pawletta Th, 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.

[5] Pawletta Th., et al. *The MATLAB GPSS Toolbox*. Online: `http://www.cea-wismar.de/tbx/mgpss/` (called 2019-11-14).

[6] Junglas P. Argesim C22 models and scripts. Online: `http://www.peter-junglas.de/fh/simulation/argesimc22.html` (called 2019-11-14).

[7] Minuteman Software. *GPSS World Reference Manual*. Online: `http://minutemansoftware.com/reference/reference_manual.htm` (called 2019-11-14).