A Discrete Process Modelling Study of ARGESIM Comparison 'C22 – Non-standard Queuing Policies' with Warteschlangensimulator

Alexander Herzog

Simulation Science Center Clausthal-Göttingen / TU Clausthal, Arnold-Sommerfeld-Straße 6, 38678 Clausthal-Zellerfeld, Germany; *alexander.herzog@tu-clausthal.de*

SNE 34(1), 2024, 29-34, DOI: 10.11128/sne.34.bn22.10674
Received: 2024-02-10; Revised: 2024-02-19
Accepted: 2024-02-22
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. ARGESIM benchmark "C22 – Non-standard Queuing Policies" describes some queueing policies and customer behaviors beyond the default first-in-first-out rule. In this article it is studied how this policies can be implemented and investigated using the discrete-event, stochastic simulation tool Warteschlangensimulator.

Introduction

This article describes the implementation of the queueing policies and customer behaviors described in ARGESIM Comparison "C22 – Non-standard Queuing Policies" using the discrete-event, stochastic simulation tool Warteschlangensimulator. A full description of the considered policies can be found in the SNE model definition [1].

Thinking of a queueing system, first-in-first-out (FIFO) is usually assumed for the queue. But in reality frequently other concepts like choosing the shortest queue or individual priorities are used. Also impatience of the customers is a frequently occurring property not only of human customers but also in industrial context. This article shows how these concepts can be implemented and analyzed using Warteschlangensimulator.

1 Warteschlangensimulator

Warteschlangensimulator (see [2]) is a free and open source, platform independent, Java-based, event-driven, stochastic simulator.

The permissive Apache 2.0 license allows to use the simulator in teaching, research and industrial / commercial context without restrictions.

The simulator allows graphical modelling of queueing systems in form of flow charts. Therefore over 100 station types are available. Inter-arrival times, service times, waiting time tolerances etc. can be modelled using one of the 41 built-in probability distributions (including the option to map measured values as an empirical distribution). An automatic distribution fitter to find a distribution that matches measured values best is also available. For more complex definitions a formula parser is integrated; so for example shifted or truncated probability distribution can be used, too.

Models can be executed in animation mode showing the movement of the entities through the system including the display of queues and in a fast simulation mode without graphical output using multiple CPU cores for faster executing. Since the simulation runtimes are often in the range of a few seconds up to one minute, the effects of changes to input parameters can be investigated in a very interactive way.

During simulation all relevant statistic performance indicators are recorded automatically and are available via the built-in report viewer. Filtering and exporting the results is also possible. The fact that recording the performance indicators does not have to be configured manually in the model keeps the modelling of even large systems clear.

To handle complex control strategies, stations can optionally be extended using Javascript or Java code for branching, holding or changing entities passing through the stations. While Javascript code is interpreted by an internal Javascript engine, Java code is compiled on the fly using the Java runtime environment running the simulator itself and then executed with full machine speed.

The built-in parameter study function and a built-in optimizer allow to easily evaluate a model for different parameter sets and to optimize a model with regard to a target value.

Warteschlangensimulator comes with English and German user interface, documentation and example models. A German text book about modelling and simulation using Warteschlangensimulator is also available, see [3].

2 Models

A full description of the considered models can be found at [1]. All models considered in this article can be downloaded from:

github.com/A-Herzog/Warteschlangensimulator /tree/master/sne-benchmarks

In the model definition priority rules for concurrent events were defined. Since Warteschlangensimulator does not support a specific order for events to be executed at the same time, these priority rules will not be used in the following models.

2.1 Choosing the shortest line

A good heuristic for a customer to get the shortest waiting time in a system consisting of multiple, parallel stations is to choose the station where the fewest customers are located, see Figure 1.

The decide station at Warteschlangensimulator can directly select the following station with the shortest queue (NQ) or with the fewest entities at the station (WIP). Either the immediately following stations or the next following process stations on the path can be considered.

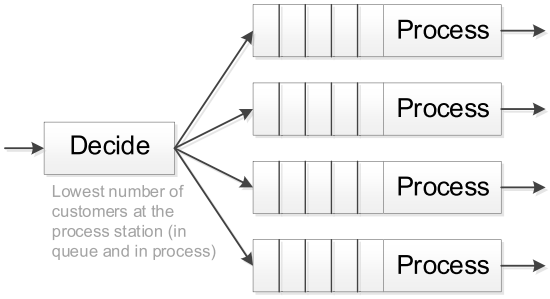

Figure 1: Schematic illustration of customers choosing the shortest queue.

The second option requires that there are no further decide stations on the path to the next process station, i. e. that the relevant process station for each path can be determined unambiguously. In case of tie between two or more stations, the station can be chosen randomly, top down or bottom up. For more complex decide rules (like in the jockeying example) also scripts (Javascript or Java code) can be used to branch the customers. A script code based decide station can also be used if the number of customers at the different stations is to be evaluated using different weights (i. e. customers on the "max. 5 items" express lane with a much lower weight then customers at the extra large shopping carts lane).

2.2 Jockeying

Jockeying means that the last customer in a queue in a system consisting of multiple parallel process stations with individual queues can change his queue if the queue on one of the other stations becomes shorter than the own queue. This behavior is usually combined with initially choosing the shortest queue.

The process station building block template available in Warteschlangensimulator consists of the actual process station and the queue in front of the station. To map jockeying of the last customers of the queue, the queue and the actual service process have to be split into two stations: Customers are served at a regular process station but the queue is organized at a script-based hold station before the process station. The script at these stations is triggered on each system state change (which covers the relevant events of a customer arrival and any state change at any process station). The scripts test two things:

1. If the process station ahead is empty, the first customer from the queue is released. The customer will then be sent to the process station for service.
2. If the jockeying condition is fulfilled, the last customer of the queue will get a "jockeying" flag and will be released. In this case the customer will be redirected to the decide station and will be sent to the shorter queue, see Figure 2.

2.3 Reneging Queues

Impatience of customers is a very common property in call center models and similar customer service systems. Human customers usually only have a limited tolerance to wait in a queue.

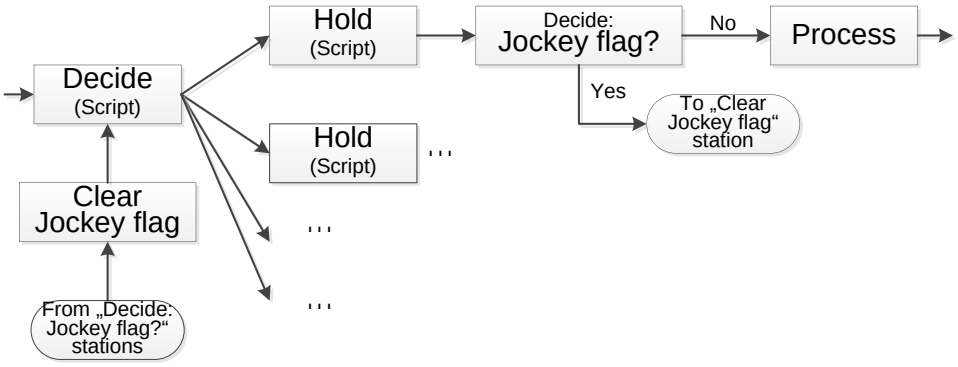

Figure 2: Schematic illustration of a waiting line with jockeying.

If this waiting time tolerance is exceeded, the customer will leave the system without being served. Impatience also occurs in industrial environments: If a workpiece can only be processed in a heated state, it cannot not wait too long after the oven process to be served at the next station.

If there are no special functions for mapping impatience directly in the simulator, handling impatience is quite hard: For each customer arriving at the queue a waiting time tolerance has to be calculated and stored with the customer object. At any time the waiting time tolerance property of each customer has to be checked and customers with an exceeded waiting time tolerance have to be removed on a special path not leading to the process station itself.

Warteschlangensimulator has a built-in impatience option for customers at process stations. This allows to add an impatience property for all customers or also for only some customers types arriving at the station with just one click. If a process station has two outgoing edges, one can be used for successful customers and the other for customers who have given up waiting. The waiting time tolerance (fixed value $t_R = 9$ in the model definition) can be any probability distribution (from which a random number is generated) or any calculation expression. Furthermore the waiting time tolerances formula of the customers can vary from customer type to customer type and can include customer specific data field values and runtime state information.

When simulating a model with impatience for each customer arriving at the process station queue an individual waiting time tolerance is calculated. The customer is added to the queue and a special "Waiting time tolerance exceeded" event is added to the events list in the simulation system.

A reference to this event is stored in the customer object. If the waiting time tolerance of the customer is exceeded, the event is executed and the customer is removed from the queue and forwarded via the "canceled customers" edge. If the customer is selected for service before his waiting time tolerance is exceeded, the corresponding event is searched in the events list and removed from there without being executed. This concept allows to handle impatience in a from the discrete-event point of view very light-weighted and therefore fast to be executed way.

2.4 Reneging Queues — Simulation results

When using stochastic waiting time tolerances for the customers, varying the average waiting time tolerance and measuring the waiting times of the customers who have canceled their waiting process (i. e. for which the individual waiting time tolerance was exceeded and therefore their waiting time tolerance equals their actual waiting time), one can see that the customers who have canceled the waiting process are usually the customers with the rather short waiting time tolerances.

Figure 3 shows the simulation results for a model with an average inter-arrival time of 100 seconds, an average service time of 80 seconds (both exponentially distributed) and one operator. The average waiting time tolerance has been varied from 20 to 500 seconds. The actual waiting time tolerances of the successful customers (red dashed graph) mirrors the global waiting time tolerance settings. But the customers who cancel the waiting process (red solid line) have shorter average waiting time tolerances and therefore are not representative for the entirety of the customers.

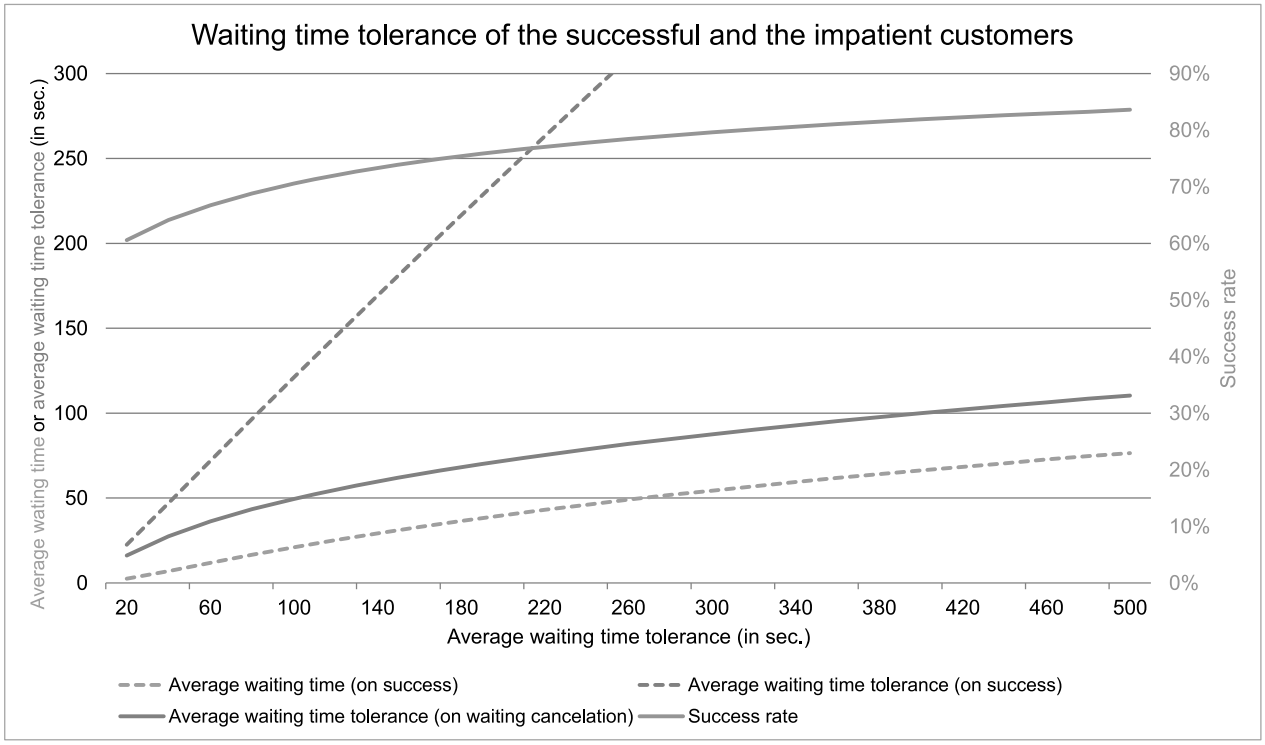

Figure 3: Waiting time tolerances of the successful and the impatient customers.

This is an interesting fact and a big problem when building simulation models for real customer service systems: For the simulation model a formula is needed to calculate the waiting time tolerances for all customers. But as input parameter only the waiting time tolerances of the customers who have canceled their waiting process (their actual waiting times) can be measured. The waiting time tolerances of the successful customers cannot be measured. For the successful customers it is only known that their waiting time tolerances obviously have been longer than their actual waiting times. So the not successful customers cannot be used as representative for all customers. This problem can be handled by using the method of parameter calibration. See [4] for more details about waiting time tolerance calibration in the context of modelling call center systems.

2.5 Classing Queues

In the model definition in [1], customers of a specific type are only to be served during their class is called. So there is no real resorting of the queue but instead there are $n_C = 5$ separate queues.

Depending on the currently active class only customers from one of these queues are sent to the process station. The concept of resorting the (single) queue is much more considered when also allowing the non-called classes to be served but to prioritize the called class (as shown in Figure 2 in [1]).

Since Warteschlangensimulator is not using a limited set of predefined queueing policies (like FIFO, LIFO etc.) but is always using priorities, this can be implemented very easily: Each customer object carries a user data vector which can be used for example to define the class of the current customer. Lets assume v_1 is set to 1 for a customer, if the customer is of class 1, and v_2 is set to 1, if the customer is of class 2 etc. Lets further assume the global variable *clsA* will be set to 1, when class 1 is called etc. Then the service priority of a single waiting customer can be calculated as

$$w + 1000 \cdot v_1 \cdot clsA + 1000 \cdot v_2 \cdot clsB + \ldots ,$$

where w is the current waiting time at the station. If the class of the customer is not called ($v_1 \cdot clsA = v_2 \cdot clsB = \ldots = 0$), the priority is the waiting time (which means FIFO policy between these customers).

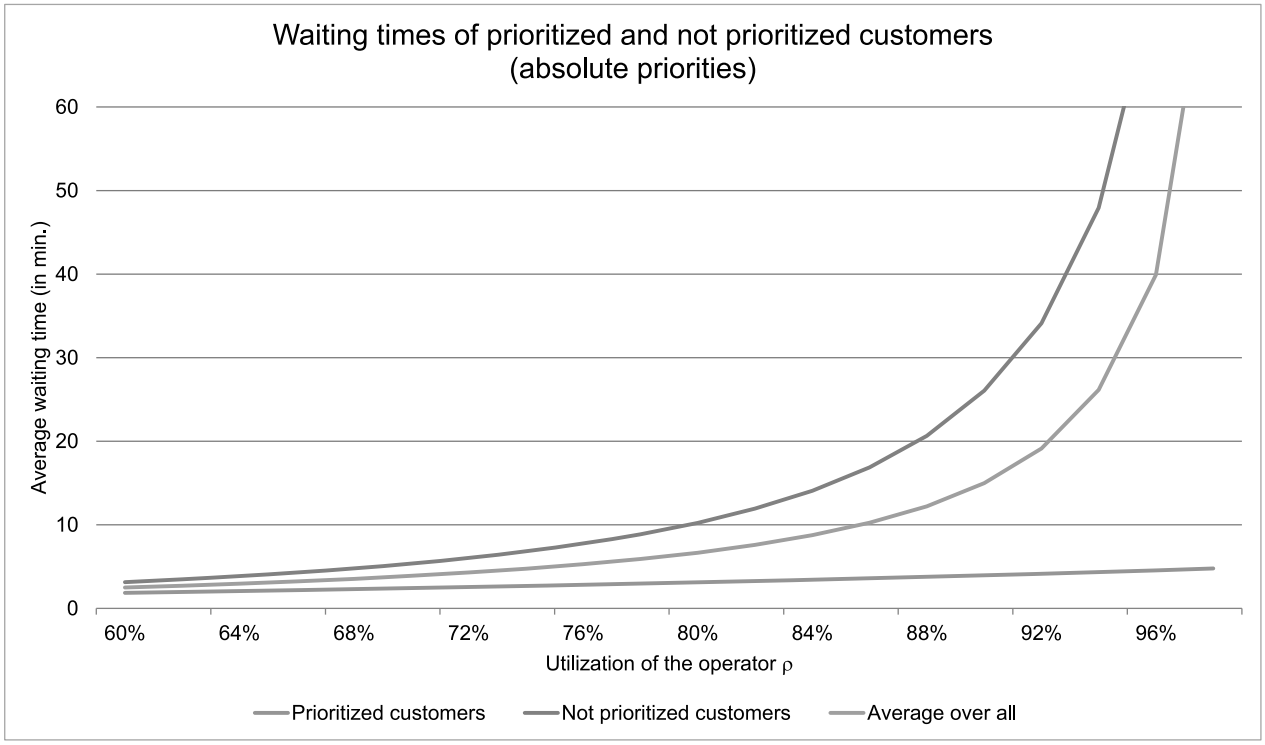

Figure 4: Waiting times of the prioritized and the not prioritized customers — using absolute priorities.

If class 1 is called ($clsA = 1$) and the customer is of this class ($v_1 = 1$) the priority is 1000 plus the waiting time (and the same for all other customers of class 1). This means the customers of class 1 will get 1000 seconds advantage in the queue during their class is called. Within all class 1 customers still FIFO is maintained. If all class 1 customers are served, the other customers will be served (also while maintaining FIFO).

2.6 Classing Queues — Simulation results

To show the effect of a strict prioritization a model with two customer types is considered: A prioritized class and a class of customers who is only served if there are no prioritized customers in the queue. Figure 4 shows the waiting times of the prioritized and the not prioritized customers when increasing the utilization of the system (and therefore increasing the average overall queue length). While the average waiting times of not prioritized customers increase quickly when increasing the utilization of the operator, the average waiting times of the prioritized customers increase only slightly.

So using different priorities allow to offer some customers a better service (shorter waiting times) in a system where customers of different types arrive. When using softer priorities (not only serving non prioritized customers if there are no prioritized customers at all but for example using different weights per waiting second in the priority formula), see Figure 5, the waiting time differences between the different customers types can be adjusted more finely.

References

[1] Model definition: C22 – Non-standard Queuing Policies
www.sne-journal.org/benchmarks/c22

[2] Warteschlangensimulator homepage
a-herzog.github.io/Warteschlangensimulator

[3] Herzog, A. *Simulation mit dem Warteschlangensimulator*. Wiesbaden: Springer Gabler; 2021. 498 p.

[4] Herzog, A. *Callcenter – Analyse und Management*. Wiesbaden: Springer Gabler; 2017. 437 p.

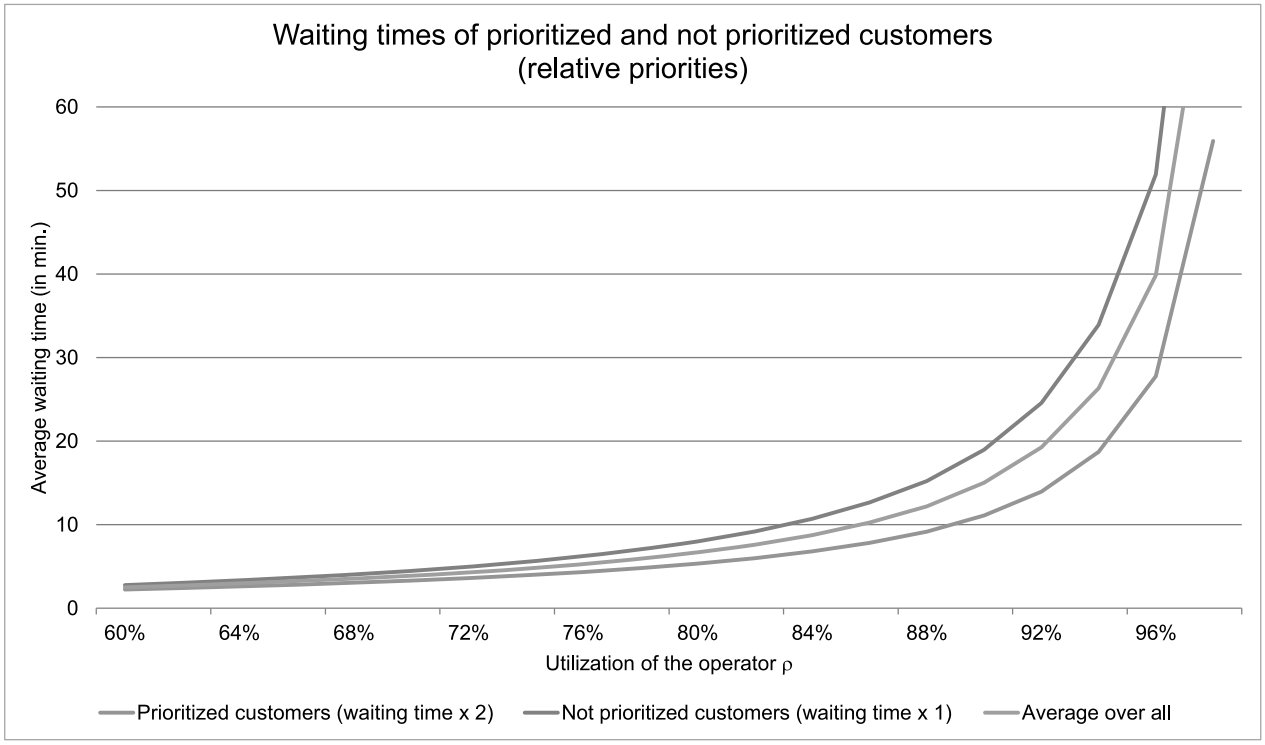

Figure 5: Waiting times of the prioritized and the not prioritized customers — using relative priorities.

Javascript code for mapping the jockeying property

The following Javascript code is used at the Javascript-based hold station in front of the first process station in the model shown in Figure 2.

```
// Which queue are we in?
   let ownRow="A";
   if (Simulation.getWIP(ownRow)==0) {
     // Server is free. Release first waiting customer.
     Clients.release(0);
   } else {
     // Customers at station A
     let wipAQueue=Simulation.getWIP("AQ");
     let wipAProcess=Simulation.getWIP("A");
     let wipA=wipAQueue+wipAProcess;
     // Customers at station B
     let wipBQueue=Simulation.getWIP("BQ");
     let wipBProcess=Simulation.getWIP("B");
     let wipB=wipBQueue+wipBProcess;
     // Customers at station C
     let wipCQueue=Simulation.getWIP("CQ");
     let wipCProcess=Simulation.getWIP("C");
     let wipC=wipCQueue+wipCProcess;
     // Customers at station D
     let wipDQueue=Simulation.getWIP("DQ");
     let wipDProcess=Simulation.getWIP("D");
     let wipD=wipDQueue+wipDProcessM
     // Minimum number of customers at a station
     let wipMin=Math.min(wipA,wipB,wipC,wipD);

     // Customers at the current station
     let wipOwnQueue=Simulation.getWIP(ownRow+"Q");
     let wipOwnProcess=Simulation.getWIP(ownRow);
     let wipOwn=wipOwnQueue+wipOwnProcess;

     // Is there a station with fewer customers?
     if (wipOwn>wipMin+1) {
       // Number of waiting customers
       let waitingCount=Clients.count();
       // Set jockeying flag for last waiting customer
       Clients.clientData(waitingCount-1,1,1);
       // Release last waiting customer.
       Clients.release(waitingCount-1);
     }
   }
```