STROBOSCOPE Models for ARGESIM Benchmark C22 'Non-standard Queuing Policies'

Photios G. Ioannou[1*], Veerasak Likhitruangsilp[2]

[1]Dept. of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI, USA, *photios@umich.edu
[2]Center of Digital Asset Management for Sustainable Development (CDAM), Dept of Civil Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand

SNE 35(4), 2025, 171-177, DOI: 10.11128/sne.35.bn22.10751
Submitted: 2025-09-23
Received Improved: 2025-11-13; Accepted: 2025-11-15
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. The STROBOSCOPE simulation system is used to model a queueing system with four servers, each with its own dynamically changing queue, where entities are initially assigned to each of four queues based on minimum queue length. In addition to the *base* case, alternatives examined include *jockeying* from one queue to a shorter queue, *reneging* (leaving the system) if the waiting time exceeds a limit, and serving entities based on *classes*.

Introduction

ARGESIM Benchmark C22 [1] investigates how to model FIFO queues that include additional dynamic behaviour. In the base system, arriving entities can be served by one of four servers, each with its own queue, where entities choose the shortest queue to join upon arrival. In a jockeying system, queued entities observe the dynamic size of all queues and switch to a shorter queue when possible. In a reneging system, queued entities grow impatient if their waiting time exceeds a set limit and depart from the system without being served. In a classing system, arriving entities are assigned to different classes and are served in groups according to their class.

These four queueing systems were modelled in STROBOSCOPE [2] (an acronym for State and Resource-Based Simulation of Construction Processes). STROBOSCOPE is a free-to-use general-purpose discrete-event simulation language and system co-developed by the first author. Its simulation models use graphical networks (similar to *activity cycle diagrams*) to communicate easily to others the main modelling elements (queues, activities, links, etc.).

The complete models are defined in text files with statements written in the STROBOSCOPE language. These statements define the resources and their properties, the attributes and behaviour of the network modelling elements, the logic by which resources and network elements interact, the definition of auxiliary objects (such as for the collection of custom statistics), the initialization of queues with resources, the overall control of the simulation, etc.

The design of STROBOSCOPE is based on three-phase activity scanning that can easily model the complex resource interactions that characterize *cyclic* operations. In STROBOSCOPE, there is no distinction between resources that serve (servers or scarce resources) and those served (customers or moving entities).

Resources can be

(a) generic resources (i.e., without attributes),
(b) characterized resources that are objects that belong to types (general classes) and to specific subtypes (subclasses with specific properties), and
(c) compound resources that have types (general classes) and which can be *assembled* by combining any number of other resources.

For example, the resources "bus" and its "passengers" can be assembled into one compound resource object called a "loaded bus" that can flow through queues and activities as one object. Each characterized and compound resource can have its own dynamic properties.

During simulation, resources spend time in queues or in activity instances. An arriving entity, for example, enters a queue where it waits conditionally until it can be served. When the entity reaches the front of the queue, and when the associated server becomes free, a new instance of the service activity can be created and draw both the entity and the server from their respective queues. The duration of the service activity instance is the required service time.

The relative order in which activities create instances and draw resources from preceding queues (if they could start at the *same* simulation time) is controlled by their *priorities* which are evaluated dynamically while the simulation runs. Similarly, the order in which resources are arranged in a queue is controlled by a user-defined *discipline* that is evaluated dynamically each time a new resource enters a queue. The *order*, *filter*, and *number* of resources drawn from a queue by a link are also dynamic.

The following models make heavy use of STROBOSCOPE *filters*. Filters are powerful objects that can be attached to queues or activities and that define the criteria for creating *dynamic subsets* of the resources that are currently in a queue or an activity instance. Moreover, two filters can call each other and create complex constructs that filter two queues at the same time.

For example, two coupled filters working together can create a subset of all *entities* in a single queue (irrespective of which actual physical queue they reside in) that can be served by the subset of all currently idle *servers*.

The entities for all the models presented below wait in a single queue node called *Q* (instead of four separate queue nodes, *Q*1, *Q*2, *Q*3, *Q*4). Similarly, when idle, the four servers wait in a single queue node called SQ (instead of four separate queue nodes, SQ1, SQ2, SQ3, and *SQ*4). All models below utilize coupled filters to create dynamic subsets and match waiting entities with their corresponding servers.

There are several possible models for the C22 benchmark. The models below use a network with the fewest nodes and links, and as a result, require the most complex filters. These models are available from the authors.

1 Basic Queuing System

The STROBOSCOPE network for all simulation models (except for reneging) is shown in Fig. 1. This network includes only one queue *Q* for all waiting entities, one queue *SQ* where all idle servers reside, and one *Service* activity (instead of including four of each). The model uses coupled *filters* to match entities to their servers.

In the deterministic model of the base system, entities arrive every t_A=1, select the shortest queue from among four FIFO server queues, and store it in a *SaveProp*.

Each queue is served by a single server with service time t_S=4.5. Simulation stops after n_E=100 entities arrive.

The model defines two compound resource types:

```
COMPTYPE Entity;    COMPTYPE Server;
SAVEPROPS Entity ServerID;
```

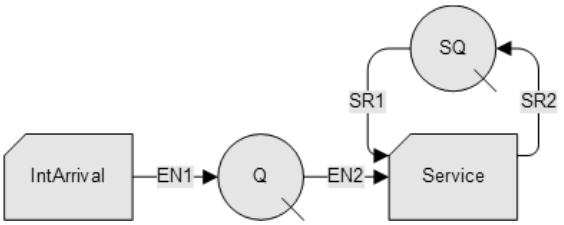

Figure 1: STRBOSCOPE Simulation Model Network.

The *SaveProp ServerID* of the resource *Entity* is assigned at runtime and stores the *ResNum* property (i.e., 1,2,3,4) of the *Entity's* assigned *Server*.

The relative order of *events* is controlled by the priorities of activities. Combi activity *Service* has a higher priority to start and create an instance than *IntArrival*.

Thus, the order of events is as follows (important when ending and starting activity events occur at the same time).

1. *Service* ends: The idle *Server* is returned to *SQ*, and the *Entity* that was in *Service* is terminated.
2. *IntArrival* ends: An *Entity* is generated and is assigned a *Server* when it flows through link *EN*1 to queue *Q*.
3. *Service* starts: If any *Entities* in *Q* can be *matched* with an idle *Server* in *SQ*, then a new *Service* instance starts and draws the first matched Entity and its *Server*.
4. *IntArrival* starts: A new *IntArrival* instance is created.

The assignment of a *Server* to an arriving *Entity* in (2) considers *both* the number of *Entities* in *Q* already assigned to each *Server*, as well as the *Entity* being served by that *Server*. The counting of the *Entities* in *Q* that have already been assigned to each *Server* is done with filters.

The *matching* of *Entities* in *Q* with their *Servers* in (3) is done by the following two coupled filters as follows.

```
FILTER MatchedServers Server 1; /forward def
FILTER MatchedEntities Entity
   SQ.MatchedServers.Count;
VARIABLE ChoiceOfCursoredEntity
  'MatchedEntities.HasCursor?
   MatchedEntities.ServerID : EN2.ServerID';
FILTEREXP MatchedServers
   'ResNum==ChoiceOfCursoredEntity';
```

An instance of *Service* can start when the *ENOUGH* attributes of its incoming links *SR*1 and *EN*2 return the value *true* (i.e., any number greater than zero). Link *SR*2 has the default *ENOUGH*, which returns *true* when the preceding queue *SQ* is not empty.

The more important *ENOUGH* is that of link *EN*2 (shown below) that applies filter *MatchedEntities* to queue *Q* to create the subset of *Entities* that have a free *Server* in *SQ*. If that subset is not empty, then a new instance of *Service* is created.

```
ENOUGH EN2 'Q.MatchedEntities.Count';
```

The new instance of *Service* then draws resources through its incoming links. First, it draws through link *EN*2 the first *Entity* for which its *Server* is currently idle in queue *SQ*. And then it draws through link *SR*1 the matching *Server* (i.e., the one whose *ResNum* is the same as the *SaveProp ServerID* of the *Entity* already drawn).

```
DRAWWHERE EN2 'SQ.MatchedServers.Count';
DRAWWHERE SR1
    'ResNum==Service.Entity.ServerID';
```

1.1 Deterministic Basic Model Results

STROBOSCOPE can dynamically write data about the state of the simulation to files. Figure 2 shows an example Excel graph of the data in such a file with the *ID*s (i.e., *ResNum*) of the last 20 *Entities* vs. simulation time. STROBOSCOPE also has an *add-on* that can create Excel graphs of the contents of selected queues vs. simulation time. Such a graph for queue *Q* is shown in Figure 2.

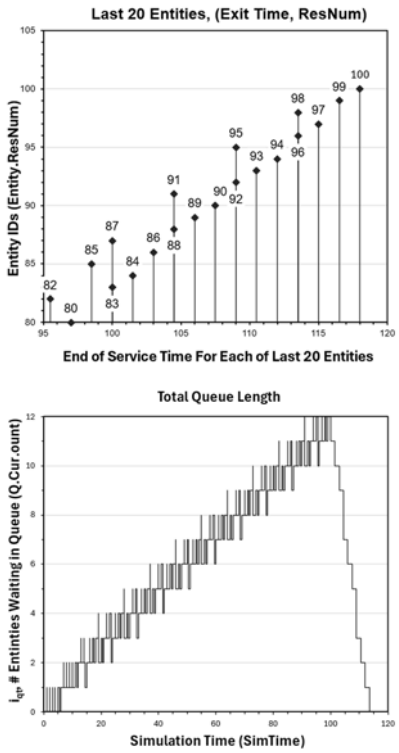

Figure 2: Outgoing *Entity ID*s (*ResNum*) and total length of queue *Q* for the base model.

Table 1 shows statistics about the length l_{qt} and the waiting time $t_{q,i}$ in *Q*, for the basic deterministic model.

	l_{qt}		$t_{q,i}$	
SimTime	**Avg**	**Max**	**Avg**	**Max**
118	5.68	13.00	6.70	13.50

Table 1: Basic Deterministic Model Statistics.

1.2 Stochastic Basic Model Results

For the stochastic version of the base model, the number of arriving *Entities* before simulation ends, and the durations of activities were as follows:

```
VARIABLE NE 500; /max arriving Entities
DURATION IntArrival Exponential[1]; /TA
DURATION Service Triangular[2.5,4.5,6.5];/TS
```

Table 2 shows the resulting statistics about the length l_{qt} and the waiting time $t_{q,i}$ in queue *Q*, from three runs.

		l_{qt}		$t_{q,i}$	
Run	**SimTime**	**Avg**	**Max**	**Avg**	**Max**
1	577.578	27.34	64	31.58	78.19
2	581.334	26.71	46	31.06	54.64
3	572.973	33.97	69	38.93	79.32

Table 2: Basic Stochastic Model Statistics.

1.3 Variant Order of Concurrent Events

In the deterministic model, several concurrent events can occur at the same time, and their relative order produces different simulation results.

For example, the following order of concurrent events occurs when activity *IntArrival* has a higher priority than activity *Service*:

1. *IntArrival* ends: An *Entity* is generated and is assigned a *Server* when it flows through link *EN*1 to queue *Q*.
2. *Service* ends: The idle *Server* is returned to queue *SQ*, and the *Entity* that was in *Service* is terminated.
3. *IntArrival* starts: A new instance is created.
4. *Service* starts: If any *Entities* in *Q* can be *matched* with an idle *Server* in *SQ*, then a new instance of *Service* starts and draws the first matched *Entity* and its *Server*.

In this variant order of concurrent events, an arriving *Entity* is assigned a *Server before* the departing *Entity* in the terminating instance of *Service* is released.

Thus, the departing *Entity* is counted as still being in *Service* at that *Server* (i.e., incorrectly) and changes the assignment of a *Server* to the arriving *Entity*. The results are shown in Figure 3. The statistics are identical to those in Table 2.

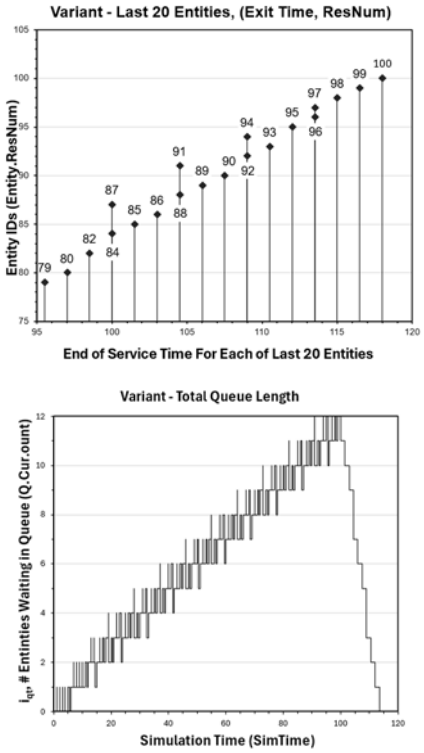

Figure 3: Outgoing Entity IDs (*ResNum*) and total queue *Q* length for the variant basic model.

1.4 Large Version with 40 Queues

STROBOSCOPE has preprocessing statement capabilities that enable scaling simulation models to any number of queues, activities, links, resources, etc.

Here, the *stochastic* model was scaled to a larger model as follows:

$n_Q = 40$, t_A = Exponential[0.1], $n_E = 5000$

The simulation results are shown in Table 3.

	l_{qt}		$t_{q,i}$	
SimTime	**Avg**	**Max**	**Avg**	**Max**
572.13	267.71	548	30.63	69.29

Table 3: Large Basic Stochastic Model Statistics

2 Jockeying Queues

2.1 Model Description

In a system with multiple queues, jockeying occurs when an entity leaves its current queue to join a shorter queue.

In these simulation models, all *Entities* wait in queue *Q*. Thus, jockeying simply requires changing their *SavePropServerID* from the currently assigned *Server* (with the longer queue) to another *Server* (with a shorter queue).

This is accomplished as follows, using *filters* that create the appropriate *logical subsets* of *Entities* in *Q* to make decisions and act (without the need to remove and reinsert any *Entity* objects in *Q*).

1. Jockeying may occur at the end of an instance of activity *Service* when a *Server* returns to queue *SQ*. This reduces the *Entities* assigned to that *Server* by one.
2. At that point, the last *Entity* assigned to each of the four *Servers* evaluates the minimum queue length for the other three *Servers*, and if it is shorter than its own queue by two *Entities*, then it changes its *SavePropServerID* to the target *Server* with the shorter queue.
3. In case of ties, the chosen target *Server* is the one with the smallest *Server ID* (i.e., 1,2,3,4).
4. The comparisons and assignments in step (2) are done in reverse order of *Server ID*s, starting with the logical queue (subset) for *Server* 4 and finishing with the logical queue (subset) for *Server* 1.

2.2 Deterministic Jockeying Model Results

The result graphs for the jockeying model are shown in Figure 4.

The first five and last five jockeying events are shown in Table 4. They are identical to the results shown in [3].

2.3 Stochastic Jockeying Model Results

The corresponding stochastic jockeying model was obtained by changing the duration of activities *IntArrival* and *Service*, like in the base model.

Table 5 shows results from three runs that were obtained automatically as part of the standard STROBOSCOPE reports.

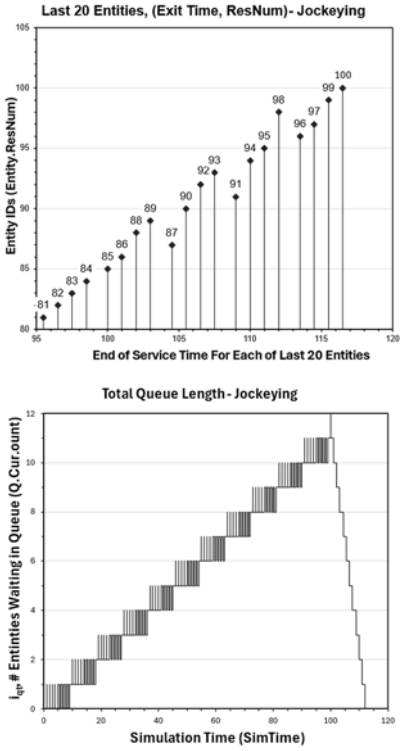

Figure 4: Outgoing ids and total queue length for the jockeying model.

SimTime, t	Entity ID	Source Q	Destin. Q
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 4: First five and last five jockeying events.

		l_{qt}		$t_{q,i}$	
Run	SimTime	Avg	Max	Avg	Max
1	565.915	22.03	46	24.93	54.98
2	576.728	11.42	33	13.17	40.87
3	579.266	19.13	63	22.16	74.73

Table 5: Jockeying Stochastic Model Statistics.

3 Reneging Queues

3.1 Model Description

Reneging occurs when the waiting time for an *Entity* in *Q* exceeds its waiting time *tolerance* (e.g., $t_R = 9$), and it chooses to leave the system without being served.

The STROBOSCOPE network for the reneging model is shown in Figure 5. In this model, each terminating instance of *IntArrival* generates two resources, an *Entity* and a matching *Twin* with the same *ResNum*.

Twin is a new compound resource type that makes reneging easy to model. *Twin* is released to an instance of activity *MaxWaitTime*, with duration $t_R = 9$, and then to queue *TQ*. Combi activity *Reneg* can then *always* start and draw the *Twin* from *TQ*. *Reneg* also attempts to draw the matching *Entity* with the same *ResNum* that might renege from *Q*. Drawing the matching *Entity* can only occur if that *Entity* has not been served yet and is still in *Q*. Otherwise, no *Entity* is drawn.

For this reason, combi activity *Reneg* is assigned less priority than activity *Service*. Only *Twins* for the reneged *Entities* are collected in queue *RenTn*.

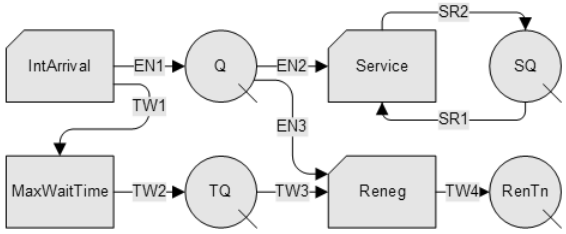

Figure 5: STRBOSCOPE Simulation Model Network.

3.2 Deterministic Reneging Model Results

Figure 6 shows a graph of the *ID*s (*ResNum*) of the last 40 outgoing *Entities* and the total length of *Q* vs *Simtime*. There were four reneging *Entities*, and they are shown by orange circles in Figure 6. They are also listed in Table 6.

SimTime, t	Entity ID	Server ID
77	68	1
86	77	2
89	80	1
104	95	1

Table 6: All four reneging events for deterministic model.

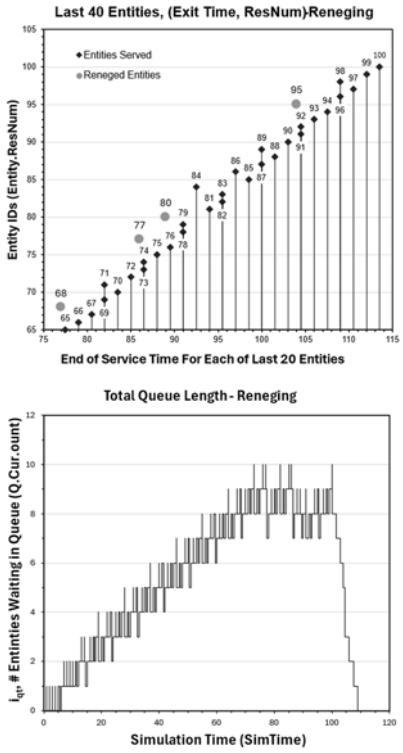

Figure 6: Outgoing *ID*s and total length of queue *Q* for the reneging model.

SimTime, *t*	Entity ID	Server ID
77	68	1
86	77	2
89	80	1
104	95	1

Table 6: All four reneging events for deterministic model.

3.3 Stochastic Reneging Model Results

The statistics results from three runs of the stochastic reneging model are shown in Table 7.

		l_{qt}		$t_{q,i}$	
Run	**SimTime**	**Avg**	**Max**	**Avg**	**Max**
1	544.171	4.26	14	4.64	9
2	482.019	5.83	16	5.62	9
3	517.067	4.94	15	5.10	9

Table 7: Reneging Stochastic Model Statistics.

4 Classing Queues

4.1 Model Description

In the deterministic classing queue model, an *Entity* is generated at the end of an instance of activity *IntArrival* and flows through link *EN*1 where it chooses the shortest *Server* queue (which it stores in its *SaveProp ServerID*), and an assigned *class* (i.e., a number from 1 to 5, which it stores in a new *SaveProp* named *Class*). Thus, the first *Entity* is assigned to *Class* 1, the second *Entity* to *Class* 2, etc. This assignment order repeats with *Entity* 6, which is again assigned to *Class* 1, etc.

Arriving *Entities* enter queue *Q* but are not served right away. Instead, service at all *Servers* starts when *SimTime* reaches 10. The order in which *Entities* are served is controlled by dynamic filters and the *SaveValue CurClass*, which is the number of the class called by an *operator*. Only the queued *Entities* whose *Class* equals *CurClass* can pass the filters and be served in FIFO order.

The value of *CurClass* starts at 5 and is decremented by 1 whenever there are no more queued *Entities* whose *Class* equals *CurClass*. Eventually, the value *CurClass* decreases to 1, and after that, the process is repeated by setting *CurClass* back to 5, etc.

Activity *Service* can start and create an instance whenever there are any *matched Entities* in queue *Q*. *Matched Entities* are those whose *Class* equals *CurClass* and which belong to a *logical queue* (i.e., a *filtered subset* of *Q*) that currently has a free *Server*. Each new instance of *Service* draws first the *matched Entity* from *Q* that is currently at the front of its logical queue and then draws the matching idle *Server* from queue *SQ*.

Each time a *matched Entity* is drawn from *Q* to a new instance of *Service*, the model counts the current number of *Entities* still in *Q* whose *Class* equals *CurClass*. When that number becomes zero, the value of *CurClass* is decremented by 1. When *CurClass* reaches the value 1, and there are no more *Entities* in queue *Q* with *Class* equal to 1, *CurClass* starts again at 5.

STROBOSCOPE supports user-defined *collectors* that can collect any calculated data and produce statistics. Five collectors were defined in this model to produce waiting time statistics for each of the five classes. Each time a *matched Entity* was drawn from *Q* to a new instance of *Service*, the model calculated its waiting time in *Q* and sent it to the statistics collector for that *Class*.

4.2 Deterministic Classing Model Results

Figure 7 shows a graph of the *ID*s (*ResNum*) of the last 20 outgoing *Entities* and the total length of *Q* vs *SimTime*.

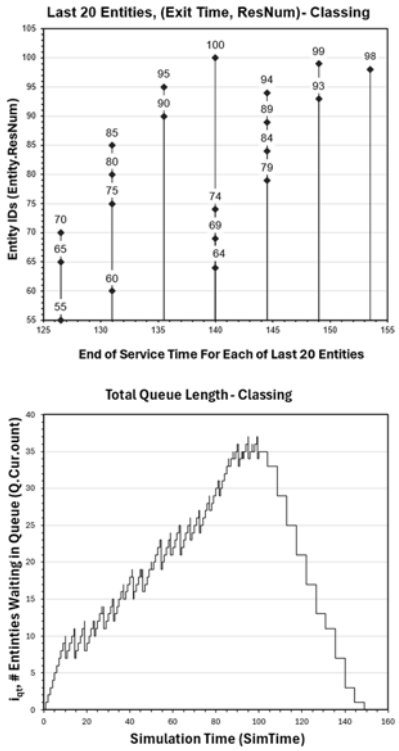

Figure 7: Outgoing ids and total queue length for the classing model.

Table 8 shows the average and maximum queue waiting times for the *Entities* in each of the five *Classes* that were compiled by the user-defined statistics collectors.

Class:	1	2	3	4	5
Avg:	30.05	26.45	20.85	32.55	30.52
Max:	62.50	60.50	51.50	71.50	67.00

Table 8: Queue waiting times per class (deterministic classing model).

4.3 Stochastic Classing Model Results

As in the base model, the stochastic classing model was produced from the deterministic model by changing the durations of the activities *IntArrival* and *Service* and increasing the number of generated *Entities* to 500. For the stochastic version, the *Class* of each arriving *Entity* was chosen with equal probability from 1 to 5.

Table 9 shows statistics from three runs about the length and the waiting times in queue *Q*.

		l_{qt}		$t_{q,i}$	
Run	**SimTime**	**Avg**	**Max**	**Avg**	**Max**
1	733.365	84.49	179.00	123.92	350.00
2	706.638	60.05	133.00	84.87	259.83
3	702.241	91.04	184.00	127.86	322.06

Table 9: Stochastic Classing Model Statistics.

Table 10 shows the average and maximum queue waiting times in *Q* for the *Entities* in each of the five *Classes* from three runs of the stochastic classing model.

Run	Class:	1	2	3	4	5
1	**Avg:**	146.2	151.3	124.2	58.1	131.2
	Max:	350.0	326.9	281.9	278.8	342.9
2	**Avg:**	64.1	78.1	80.4	112.6	91.8
	Max:	214.5	227.4	246.4	259.8	249.7
3	**Avg:**	142.9	150.4	103.5	96.0	142.3
	Max:	322.1	299.0	238.4	260.7	312.8

Table 10: Queue waiting times per class (stochastic classing model).

5 Conclusion

The STROBOSCOPE graphical and statistical results are very close to those produced by GPSS in [3]. The versatility of STROBOSCOPE queues, activities, and especially filters, makes it straightforward to model the non-standard queueing systems in ARGESIM Benchmark C22 by using compact simulation networks that are suitable for education.

References

[1] 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

[2] STROBOSCOPE Simulation System Software. Retrieved from www.stroboscope.org. Sept. 20, 2025.

[3] Junglas P, Pawletta Th. 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