Coursework 2 - BABIES Fortune Charity

A Simulation & Optimisation Experiment

SIMULATION & OPTIMISATION FOR DECISON SUPPORT

Adam Howlett - 4343476

Introduction

This report documents the process of modelling stores associated with the BABIES Fortune Charity, attempts to optimise its workforce utilisation and explores its dependence on other store parameters. The main result was an optimal utilisation of 80.9% across 5 stores for a single working day. This includes constraints on the model's parameters including stock, cleanliness and customer satisfaction.

The report begins with a discussion of the conceptual model, and decisions regarding model content. These seek their justification in the aim to create a simple yet sufficiently accurate model. An insight into the workings of the model implementation will follow, then a discussion of the experiment and its findings.

The report's conclusions will seek to aid the stores' decision making policy, and ultimately lead to a more efficient day-to-day running for the business. However, the report will terminate with a brief discussion of its shortcomings that hopes to demonstrate the limits on its applicability.

Conceptual Model

Overview

The Conceptual Model begins with an understanding of the problem we are trying to solve. Principally, we will seek to optimise of the efficiency with which 5 stores owned by the charity manage its workforce. The model will take into account a full day of operation for all 5 of the stores, including the time before opening hours and after closing hours. The key objectives of the optimisation experiment will be to maximise staff workforce utilisation pursuant to the following constraints:

- A maximum of 20 employees can be active across all stores
- Maintain a mean customer satisfaction above 9/10 across all stores.
- Maintain a mean store cleanliness of 80% across all stores for the day.
- Maintain a mean stock capacity of 80% across all stores for the day.

To calculate the mean workforce utilization across all stores, the model will collect data on the state of employees at regular intervals, and calculate a utilisation for each store as follows:

$$u_k = \frac{1}{N}\sum_{i=1}^{n} x_{ij} \qquad j = 1, ...N, \; k = 1, ...5, \tag{1}$$

where n is the number of data points, j is the employee number (N total employees), k is the store number, and x_{ij} is 1 if employee j is being utilised at the time data point i is collected, and 0 otherwise. To collect this data into a single objective, we simply calculate the mean of the individual utilisations u_k across all stores (at the end of the simulation):

$$U_{\text{total}} = \frac{1}{5}\sum_{k=1}^{5} u_k. \tag{2}$$

The constraint parameters will be calculated similarly, and parameters associated with employee, customer, and store classes will be defined shortly.

The model will take into account several experimental factors to properly model the system of stores. They will be:

- Number of employees assigned to each store (from a centralized workforce pool).
- Customer arrival rates for each store (which vary throughout the day).
- Stock delivery schedules for each store.

Additionally, the model will generate responses for measuring the objectives/constraints, amongst some additional statistical information:

- Mean workforce utilisation across all stores
- Mean customer satisfaction scores across all stores (ranked out of 10)
- Mean store cleanliness across all stores (as a percentage)
- Mean store stock capacity across all stores (as a percentage)
- Total number of customers that have left with no service

The basic content and scope of the model is captured in the table below. There are additional statistics that will be captured in the implementation itself - these will be covered in the Model Implementation section.

Model Scope	Detail	Decision	Justification
Individual Stores	Stock	Include	Required for responses
	Queuing	Include	Affects customer satisfaction
	Cleanliness	Include	Affects customer satisfaction
Customers		Include	Required for responses
Employees	Serving Customers	Include	Required for responses
	Stocking	Include	Affects customer satisfaction
	Cleaning	Include	Affects customer satisfaction
	Aiding customers outside of serving	Exclude	Unnecessary complexity - low impact on responses

Model Detail	Detail	Decision	Comments
Individual Stores	Stock Capacity	Include	Percentage - Required for responses
	Number of Queues	Include	Different based on store size
	Cleanliness	Include	Percentage - Affected by customer traffic
	Stock Delivery Schedule	Include	One schedule across all stores
	Different Stock Types	Exclude	Effect on outputs not well understood
	Browsing Area	Include	For customer browsing, employee stocking/cleaning
Customers	Arrival Rate	Include	Distribution based on time of day
	Browsing	Include	Contributes to cleanliness
	Leaving Early	Include	Conditional: Store unclean, no stock, waiting too long
	Order Size	Exclude	Customers assumed to buy same amount
	Customer Satisfaction Score	Include	Required for responses (0-10)
Employees	Service Time	Include	Affects outputs
	Staff Roster	Include	Experimental Factor
	Absenteeism	Include	Accounted for in Staff Roster
	Task Prioritisation	Include	Serve customers, then stock, then clean
Queues	Capacity	Exclude	Assumption: Unlimited
	Shortest queue joined	Include	Well understood
	Queue Jumping/Other behaviours	Exclude	Mostly irrelevant to model

Associated with the content are some assumptions & simplifications of the model:

- Assumptions:
  - Customers are identical in behaviour - stereotypes/personalities are not accounted for.
  - Customers all buy the same amount of product (taken as a percentage of the store stock).
  - Each customer reduces the 'Cleanliness' of the store by the same amount (taken as a percentage of the total cleanliness)
  - Customers will only buy product if the store is clean, if stock is available and they do not have to wait longer than 10 minutes to be served.
  - Customers will rate their level of satisfaction with their service as a linear function of the above constraint parameters - any instance of leaving early results in a customer satisfaction of 0/10 for that customer.
  - Employees all work at the same rate, and do not become fatigued.
- Simplifications:
  - Customers do not distinguish between different types of stock.
  - Customers arrive at set rates, varying throughout the day (e.g peak times at noon, lower at other times).
  - Customers will always join the shortest queue, and will not push in/cause other disturbances.
  - Employees will follow strict prioritisation of store tasks - Serving customers first, then stocking, then cleaning.
  - Stock delivery happens at set times across all stores for the day, and contains a certain percentage of the stock.
  - Employees always turn up to work and get paid properly.

Model Classes and Logic

Stores

The stores will come in two sizes - 3 small and 2 large. These will differ from each other in their stock capacities, customer arrival rates, number of queues, and likely number of required employees.

The stores have multiple parameters and statistics that they will keep track of during runtime:

Stores
- Entrance/Exit, Browsing Area, Delivery Area & Queues - Number of Queues - Histogram of Time Spent in Queues - Queue Lengths - Number of Employees - Mean Staff Utilisation - Stock Percentage & Mean Stock Percentage - Cleanliness Percentage & Mean Cleanliness Percentage - Customer Arrival Rates - Histogram of Customer Satisfaction Scores - Number of Customers left with no service

Stores will open at 9:00AM, and close at 5:00PM. 30 minutes before and after the store opens, employees will be available to stock and clean the store. To simulate the 'left-over' tasks from the previous day, the model will begin with the store at 80% 'Cleanliness' and 80% Stock Capacity.

For the two store sizes, customers will arrive on a 'per-hour' rate according to the following schedule:

Time Range	Rate of Arrival (Small)	Rate of Arrival (Large)
08:30 - 09:00	0	0
09:00 - 12:00	50	100
12:00 - 13:00	100	200
13:00 - 17:00	75	150
17:00 - 17:30	0	0

Each customer that leaves the 'Browsing Area' reduces the store cleanliness by 0.25% (Small Store) or 0.125% (Large Store). Similarly, every customer that makes a purchase reduces the store's stock capacity by 0.25% (Small Store) or 0.125% (Large Store). Employees within the store will need to keep either from dropping to 0%, which will make customers leave without making a purchase.

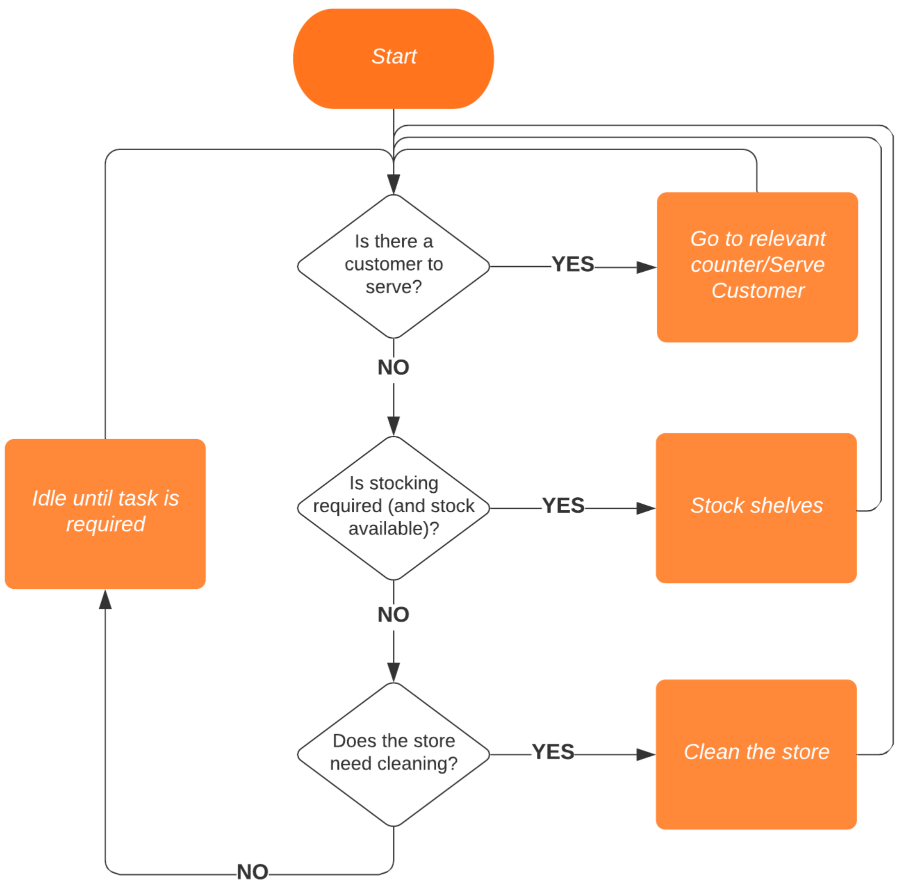
Employees

A number of Employees will be designated for each store as the main experimental factor of the model. They will have associated parameters that determine how they operate during store operation

Employees
- Personal Utilisation - Associated Decision logic - Stocking Rate (same for each) - Cleaning Rate (same for each)

They will have three tasks - Serving Customers, Stocking and Cleaning, and will prioritise them according to the following flowchart:

Employee Decision Logic


Start
Is there a customer to serve?
YES
Go to relevant counter/Serve Customer
NO
Is stocking required (and stock available)?
YES
Stock shelves
NO
Does the store need cleaning?
YES
Clean the store
NO
Idle until task is required

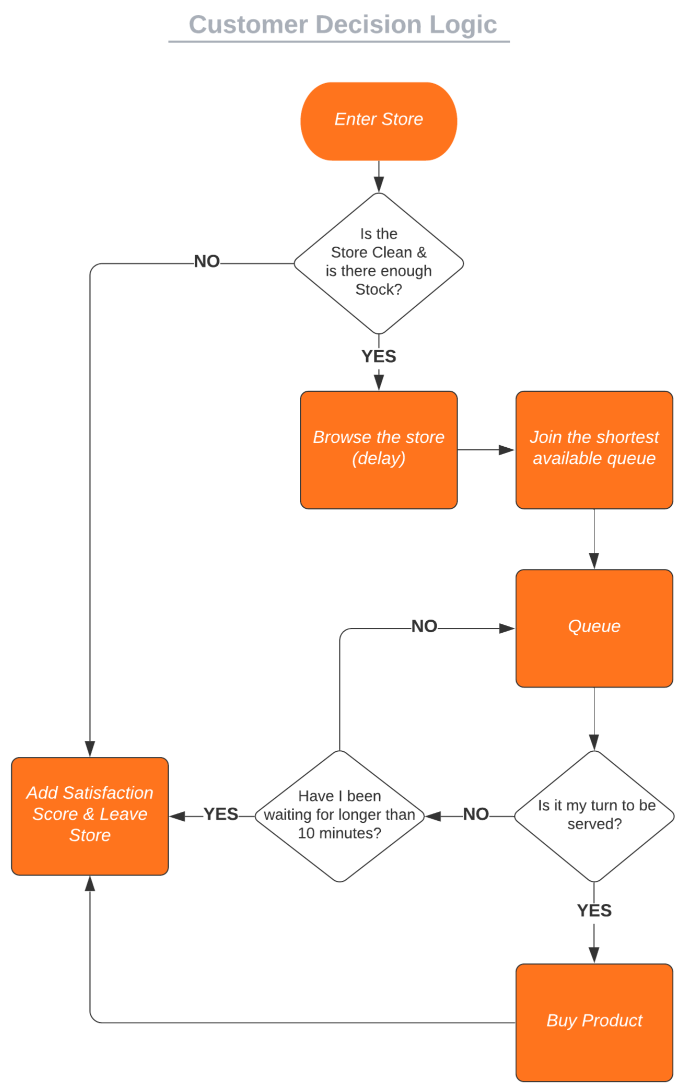
The number of employees assigned to each store will likely have a significant impact on the utilisation associated with each, as well as the other objectives/constraints of the model. Employees are assumed to act identically, following strictly the above logic to make decisions during model runtime.

Customers

Customers coming into the store will affect the stock levels, cleanliness levels, and reorient the workforce to serve them whenever they require. A class description is shown below:

Customers
- Customer Satisfaction Score - Time Spent in Queues - Associated Decision Logic - Time Spent Browsing Store - Service Time when Buying

Customers, like employees, will operate in the store according to a strict decision logic described by the below flowchart:



Customer Decision Logic
Enter Store
Is the Store Clean & is there enough Stock?
NO
YES
Browse the store (delay)
Join the shortest available queue
Queue
NO
Add Satisfaction Score & Leave Store
YES
Have I been waiting for longer than 10 minutes?
NO
Is it my turn to be served?
YES
Buy Product

Customers rate their experience in the store out of 10, taking into account the stock availability, cleanliness and service according to the following formula:

$$s_i = 3.34 - \frac{T}{60 \times 3} + \frac{3.33 \times S}{100} + \frac{3.33 \times C}{100} \tag{3}$$

where s_i is the satisfaction score for customer i, T is the time they spent in the queue (in seconds), S is the stock percentage, and C is the cleanliness percentage. In this way, each third of the score is a linear function of its relevant variable. If a customer leaves the store as a result of cleanliness or stock percentage being 0%, or service taking too long, the s_i for the customer will be set to 0. These scores will be collected and evaluated to a mean across all stores as follows:

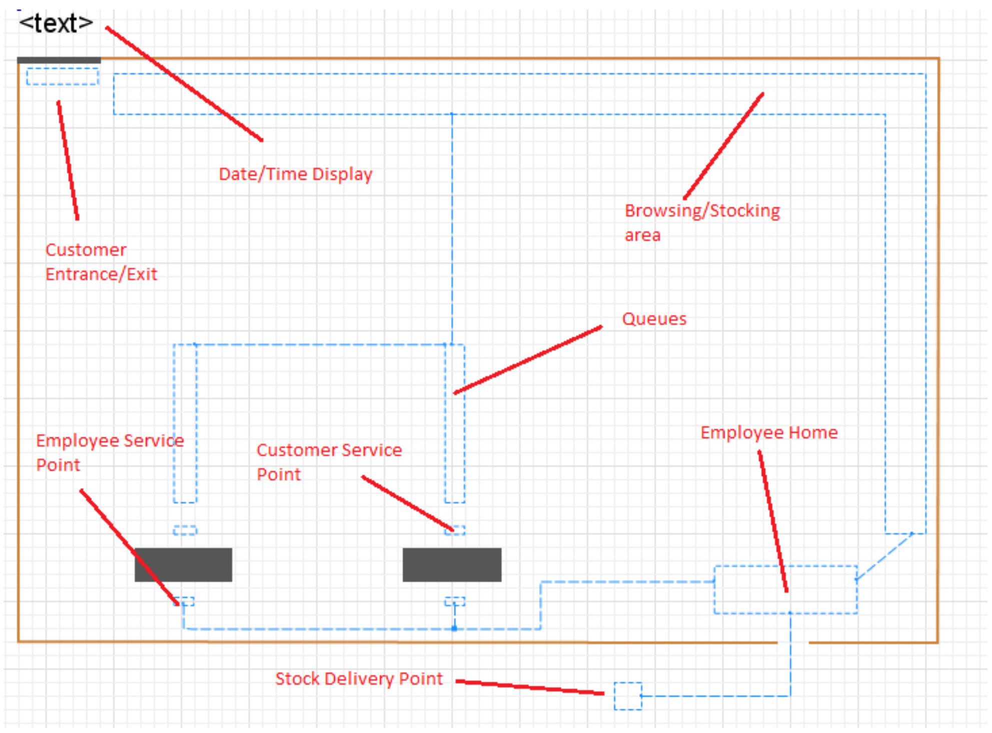
$$s_{\text{mean}} = \frac{1}{N_{\text{total}}} \sum_k \sum_i s_{ik} \tag{4}$$

where s_{mean} is the aggregate average customer satisfaction, N_{total} is the total number of customers that have been processed across all stores, and s_{ik} is the satisfaction score of customer i from store k.

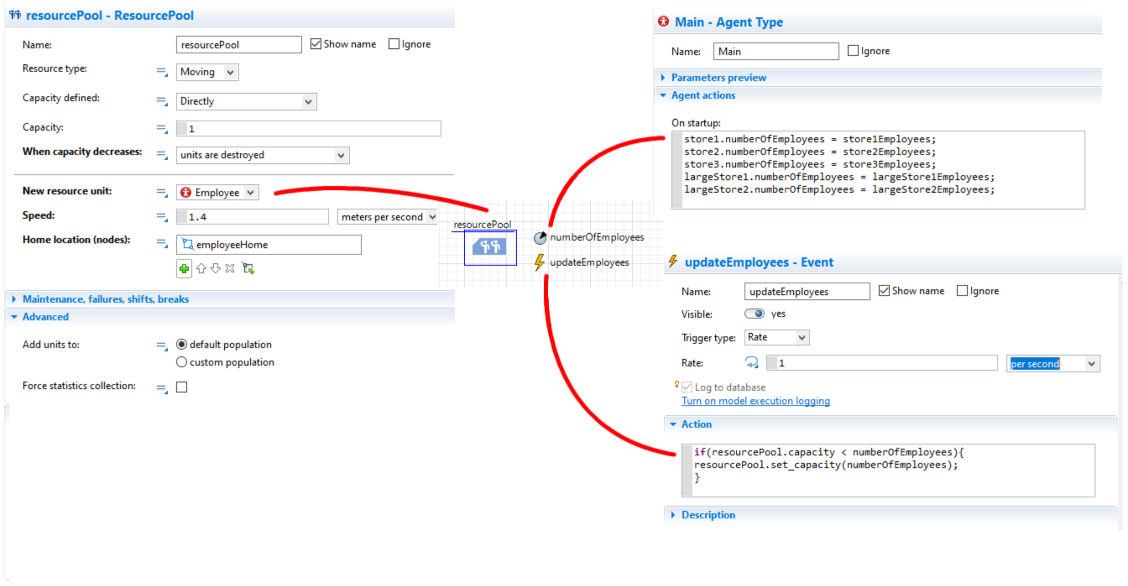
Model Implementation

The First Store & Employee Generation

The model was implemented with Anylogic PLE. Initially, one small store was created to get customer/employee logic working correctly, and features were added sequentially until the full model was realised. The first store layout looks as in the picture below:



Employees for each store were represented with a Resource Pool Block, where the units (employees) are seized by customers or stock deliveries, each with a priority score. The capacity defaults to 1, and is altered by the parameter ”store1Employees” in Main at the beginning of the simulation:



resourcePool - ResourcePool

Name:	resourcePool ✓ Show name ☐ Ignore
Resource type:	Moving
Capacity defined:	Directly
Capacity:	1
When capacity decreases:	units are destroyed
New resource unit:	Employee
Speed:	1.4 meters per second
Home location (nodes):	employeeHome

Maintenance, failures, shifts, breaks

Advanced

Add units to:	[✓ default population] custom population
Force statistics collection:	☐

Main - Agent Type

Name: Main ☐ Ignore

Parameters preview

Agent actions

On startup:

```
store1.numberOfEmployees = store1Employees;
store2.numberOfEmployees = store2Employees;
store3.numberOfEmployees = store3Employees;
largeStore1.numberOfEmployees = largeStore1Employees;
largeStore2.numberOfEmployees = largeStore2Employees;
```

updateEmployees - Event

Name:	updateEmployees ✓ Show name ☐ Ignore
Visible:	yes
Trigger type:	Rate
Rate:	1 per second

✓ Log to database
Turn on model execution logging

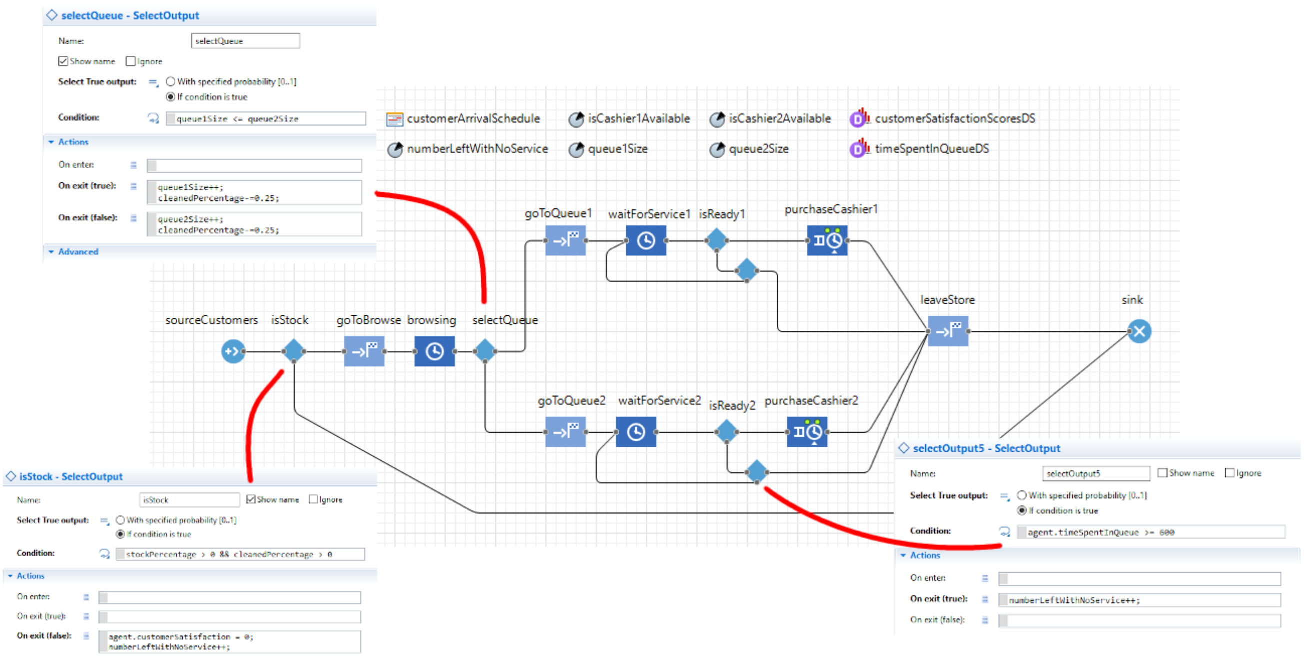
Action

```
if(resourcePool.capacity < numberOfEmployees){
resourcePool.set_capacity(numberOfEmployees);
}
```

Description

Customer Service & Queuing

A customer class was the next step forward. After generating one that housed two parameters - ’timeSpentInQueue’ and ’customerSatisfaction’, the following process flow was designed for the customers’ decision logic:



selectQueue - SelectOutput

Name: selectQueue

[✓ Show name] Ignore

Select True output: With specified probability [0..1] [✓ If condition is true]

Condition:
```
queue1Size <= queue2Size
```

Actions

On enter:

On exit (true):
```
queue1Size++;
cleanedPercentage-=0.25;
```

On exit (false):
```
queue2Size++;
cleanedPercentage-=0.25;
```

Advanced

isStock - SelectOutput

Name: isStock

[✓ Show name] Ignore

Select True output: With specified probability [0..1] [✓ If condition is true]

Condition:
```
stockPercentage > 0 && cleanedPercentage > 0
```

Actions

On enter:

On exit (true):

On exit (false):
```
agent.customerSatisfaction = 0;
numberLeftWithNoService++;
```

selectOutput5 - SelectOutput

Name: selectOutput5

Show name Ignore

Select True output: With specified probability [0..1] [✓ If condition is true]

Condition:
```
agent.timeSpentInQueue >= 600
```

Actions

On enter:

On exit (true):
```
numberLeftWithNoService++;
```

On exit (false):

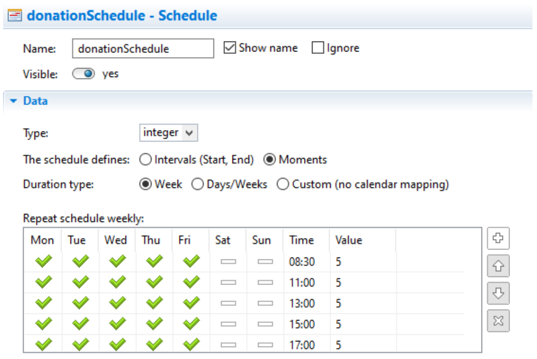
Customers move through the process flow like this:

- Spawn in at rate described in ’customerArrivalSchedule’
- Enter the ’isStock’ check. If the stock capacity or the store cleanliness are at 0%, go directly to the ’sink’ block, set ’customerSatisfaction’ to 0, and increment ’numberLeftWithNoService’ by one. Otherwise, enter the ’goToBrowse’ block.
- Move to the browsing area - delay there for between 2 and 10 minutes (until product to buy is found)
- Check for the shortest queue, and join that queue. On joining the queue, increment the corresponding ’queueSize’ parameter by one.
- At the ’waitForService’ block, which has a delay of one second, increment the agent’s ’timeSpentInQueue’ parameter by one. Leave, and check if the ’isCashierAvailable’ boolean has been set to true. If it is not, perform an additional check to see if the ’timeSpentInQueue’ parameter is greater than 600 (corresponding to a waiting time of over 10 minutes. If this is also untrue, return to the ’waitForService’ block.
- If the customer has been waiting for over 600 seconds, proceed immediately to the ’leaveStore’ block and incremement the ’numberLeftWithNoService’ parameter by one.
- If ’isCashierAvailable’ is found to be true, proceed to the corresponding ’purchaseCashier’ service block and set ’isCashierAvailable’ to false.

- An Employee unit is seized and moves to the 'employeeCashier' node. The customer is delayed at the 'isCashierAvailable' block for between 30 seconds and one minute. The Employee is released upon leaving, and 'isCashierAvailable' is set to true.
- Any agents entering the 'leaveStore' block after being served or otherwise have their 'customerSatisfaction' parameters set. If the waiting time was less than 600, it is calculated according to (3). Otherwise, it is set to 0. Add to the data set 'timeSpentInQueueDS' the agent's 'timeSpentInQueue' parameter.
- All customers entering the sink block have their 'customerSatisfaction' parameter added to the 'customerSatisfactionScoresDS' data set, and are then destroyed.

Stocking & Cleaning

The next implementation was the Stocking and cleaning for the store. Donations come in at a fixed schedule, with each source accounting for 10% of the store's total stock capacity:



donationSchedule - Schedule

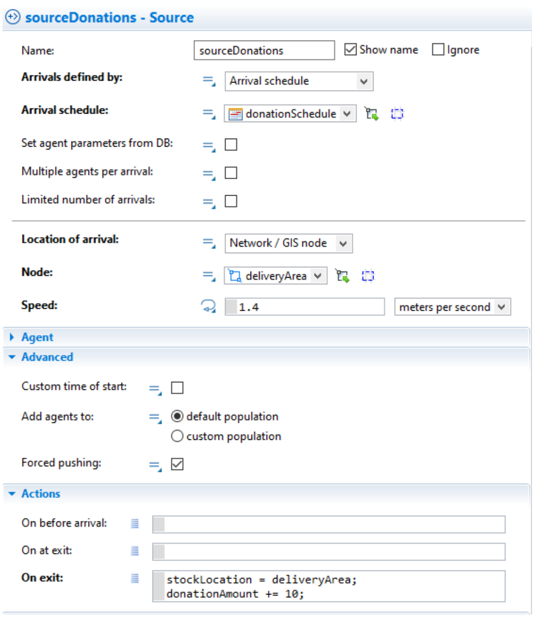
Name:	donationSchedule ☑ Show name ☐ Ignore
Visible:	yes

Data

Type:	integer
The schedule defines:	Intervals (Start, End) [✓ Moments]
Duration type:	[✓ Week] Days/Weeks Custom (no calendar mapping)

Repeat schedule weekly:

Mon	Tue	Wed	Thu	Fri	Sat	Sun	Time	Value
✓	✓	✓	✓	✓	☐	☐	08:30	5
✓	✓	✓	✓	✓	☐	☐	11:00	5
✓	✓	✓	✓	✓	☐	☐	13:00	5
✓	✓	✓	✓	✓	☐	☐	15:00	5
✓	✓	✓	✓	✓	☐	☐	17:00	5



sourceDonations - Source

Name:	sourceDonations ☑ Show name ☐ Ignore
Arrivals defined by:	Arrival schedule
Arrival schedule:	donationSchedule
Set agent parameters from DB:	☐
Multiple agents per arrival:	☐
Limited number of arrivals:	☐
Location of arrival:	Network / GIS node
Node:	deliveryArea
Speed:	1.4 meters per second

Agent

Advanced

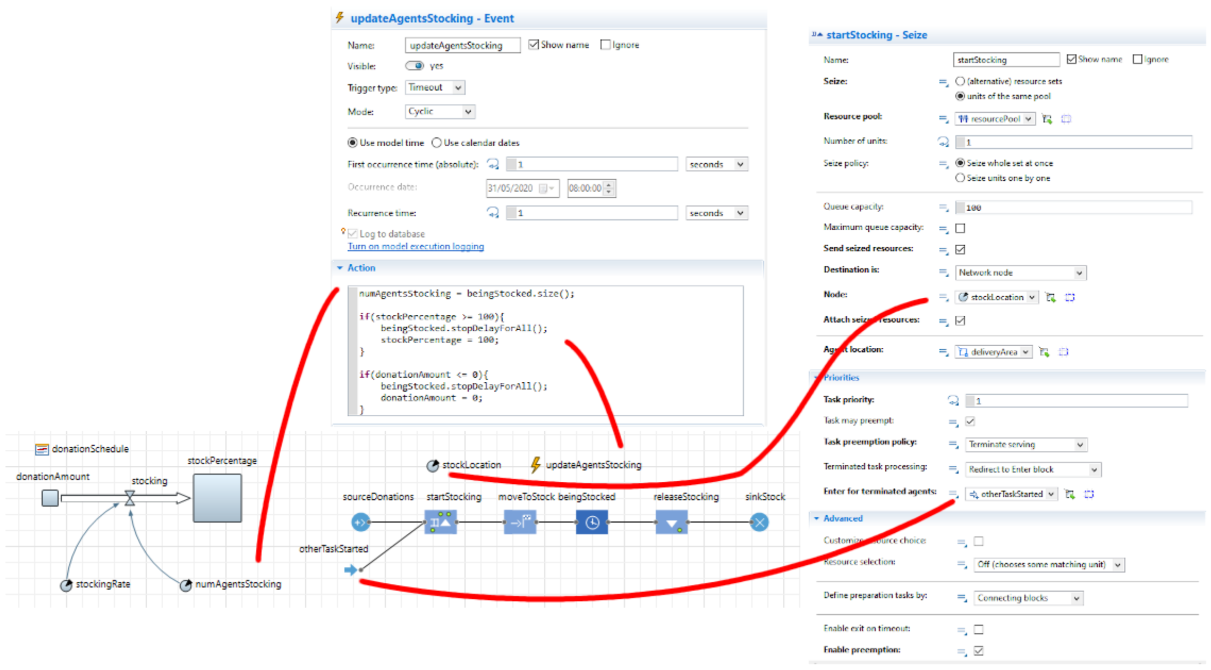
Custom time of start:	☐
Add agents to:	[✓ default population] custom population
Forced pushing:	☑

Actions

On before arrival:	
On at exit:	
On exit:	

```
stockLocation = deliveryArea;
donationAmount += 10;
```

When donations are available, the block "startStocking" seizes an employee unit for the following process:



updateAgentsStocking - Event

Name: updateAgentsStocking [✓ Show name] [ ] Ignore
Visible: yes
Trigger type: Timeout
Mode: Cyclic
[✓ Use model time] Use calendar dates
First occurrence time (absolute): 1 seconds
Occurrence date: 31/05/2020 08:00:00
Recurrence time: 1 seconds
[✓] Log to database
Turn on model execution logging

Action

```
numAgentsStocking = beingStocked.size();

if(stockPercentage >= 100){
    beingStocked.stopDelayForAll();
    stockPercentage = 100;
}

if(donationAmount <= 0){
    beingStocked.stopDelayForAll();
    donationAmount = 0;
}
```

startStocking - Seize

Name: startStocking [✓ Show name] [ ] Ignore
Seize: (alternative) resource sets [✓ units of the same pool]
Resource pool: resourcePool
Number of units: 1
Seize policy: [✓ Seize whole set at once] Seize units one by one
Queue capacity: 100
Maximum queue capacity: [ ]
Send seized resources: [✓]
Destination is: Network node
Node: stockLocation
Attach seized resources: [✓]
Agent location: deliveryArea

Priorities

Task priority: 1
Task may preempt: [✓]
Task preemption policy: Terminate serving
Terminated task processing: Redirect to Enter block
Enter for terminated agents: otherTaskStarted

Advanced

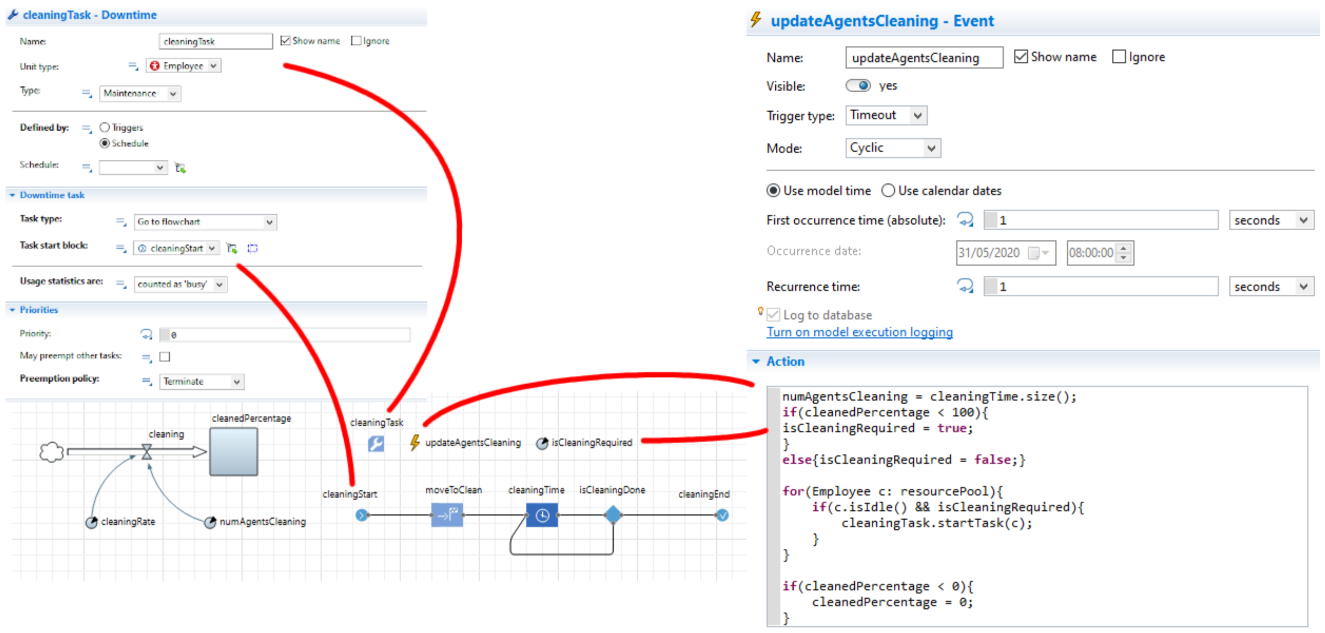
Customize resource choice: [ ]
Resource selection: Off (chooses some matching unit)
Define preparation tasks by: Connecting blocks
Enable exit on timeout: [ ]
Enable preemption: [✓]

The 'Donations' agent moves through the process like this:

- Spawn in the delivery area according to 'donationSchedule' - 'stockLocation' is set to 'deliveryArea' and 'donationAmount' has 10 added to it for each agent.
- Move to 'startStocking' to seize an Employee Unit with priority 1
- If no employees are available, the seize is terminated and the donation agent is sent to the 'otherTaskStarted' block to try again.
- If the seize is successful, the employee moves to the delivery area, picks up the stock and takes it to the browsing area
- While in the 'beingStocked' block, the function 'updateAgentsStocking' sets the 'numAgentsStocking' parameter to the population of the 'beingStocked' block.
- If the process is terminated at this point, the parameter 'stockLocation' is set to 'browsingArea' to ensure the employee goes to the right place when the task is resumed.
- Stocking terminates for all agents when the stock level reaches 100%, or when 'donationAmount' is reduced to 0.

During this process, the flow on the left of the picture updates the current stock percentage as a function of 'numAgentsStocking' multiplied by 'stockingRate' which is set to $1/60$. In this way, one agent stocking will increase the capacity by 1% per minute, until the stock is full.

The cleaning process works in a similar way, although it employs no seize block, as the task is not associated with an additional agent. Instead, it is instigated (with priority 0) by an event firing a ’Downtime’ block. It also has no stock on the left hand side of its flow, because there is no limit to the amount of cleaning that can be done.


cleaningTask - Downtime

Name:	cleaningTask ☑ Show name ☐ Ignore
Unit type:	Employee
Type:	Maintenance
Defined by:	○ Triggers [✓ Schedule]
Schedule:	

Downtime task

Task type:	Go to flowchart
Task start block:	cleaningStart
Usage statistics are:	counted as 'busy'

Priorities

Priority:	0
May preempt other tasks:	☐
Preemption policy:	Terminate

updateAgentsCleaning - Event

Name:	updateAgentsCleaning ☑ Show name ☐ Ignore
Visible:	yes
Trigger type:	Timeout
Mode:	Cyclic
	[✓ Use model time] ○ Use calendar dates
First occurrence time (absolute):	1 seconds
Occurrence date:	31/05/2020 08:00:00
Recurrence time:	1 seconds
	☑ Log to database
	Turn on model execution logging

Action

```
numAgentsCleaning = cleaningTime.size();
if(cleanedPercentage < 100){
isCleaningRequired = true;
}
else{isCleaningRequired = false;}

for(Employee c: resourcePool){
    if(c.isIdle() && isCleaningRequired){
        cleaningTask.startTask(c);
    }
}

if(cleanedPercentage < 0){
    cleanedPercentage = 0;
}
```

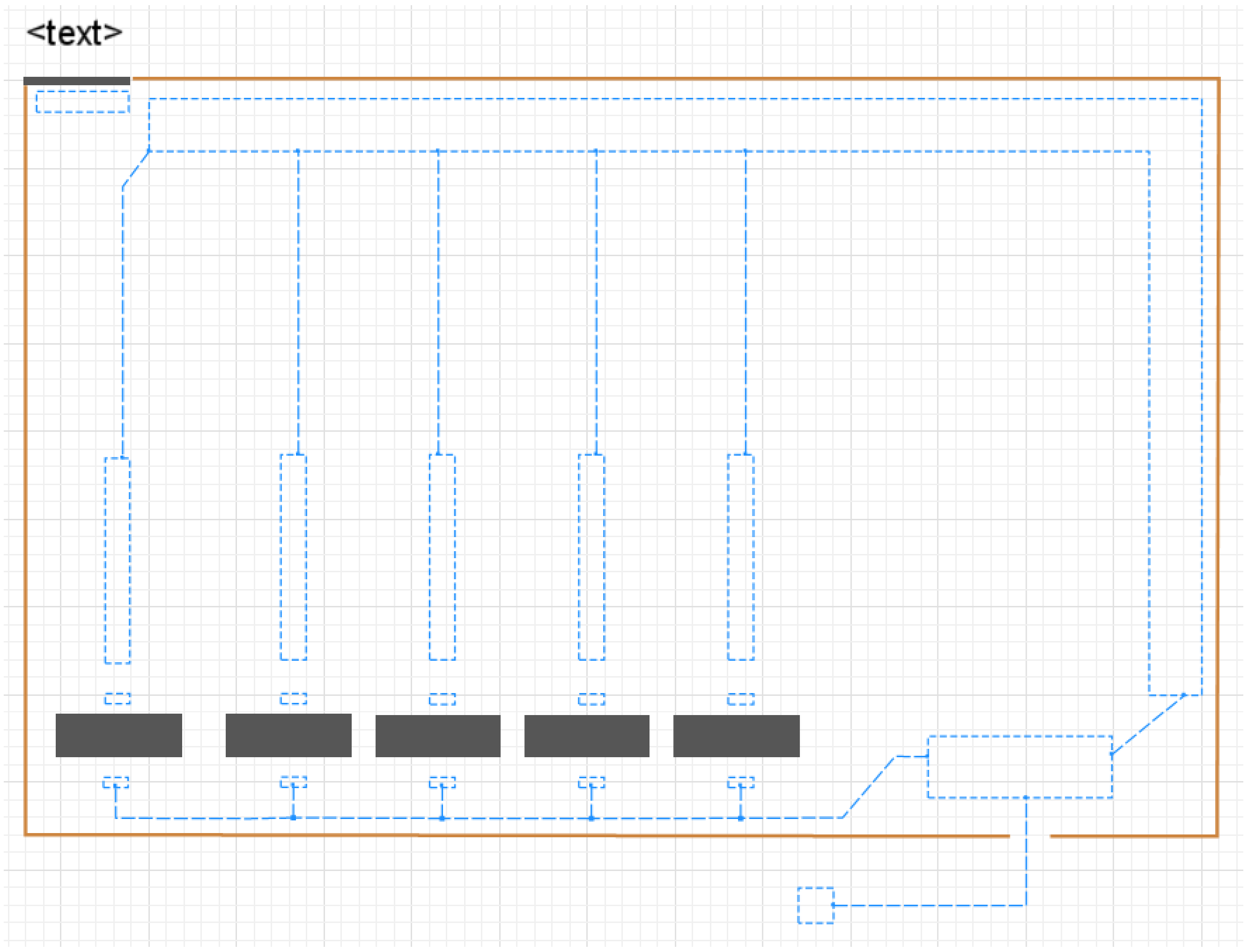
The logic begins with the ’updateAgentsCleaning’ function, which is called every second to check if cleaning is needed and if units are available:

- Check if ’CleanedPercentage’ is 100%. If it is not, assign all currently idle units to the ’cleaningTask’ Downtime block, which points to the ’cleaningStart’ resource task.
- Agents move to the browsing area, and enter the ’cleaningTime’ delay block. The ’updateAgentsCleaning’ function sets the ’numAgentsCleaning’ parameter to the population of this block.
- Each second, the agents check if the ’cleanedPercentage’ has reached 100. If it has, they exit the task via ’cleaningEnd’ and return to their home location. If it hasn’t, they re-enter the ’cleaningTime’ block.

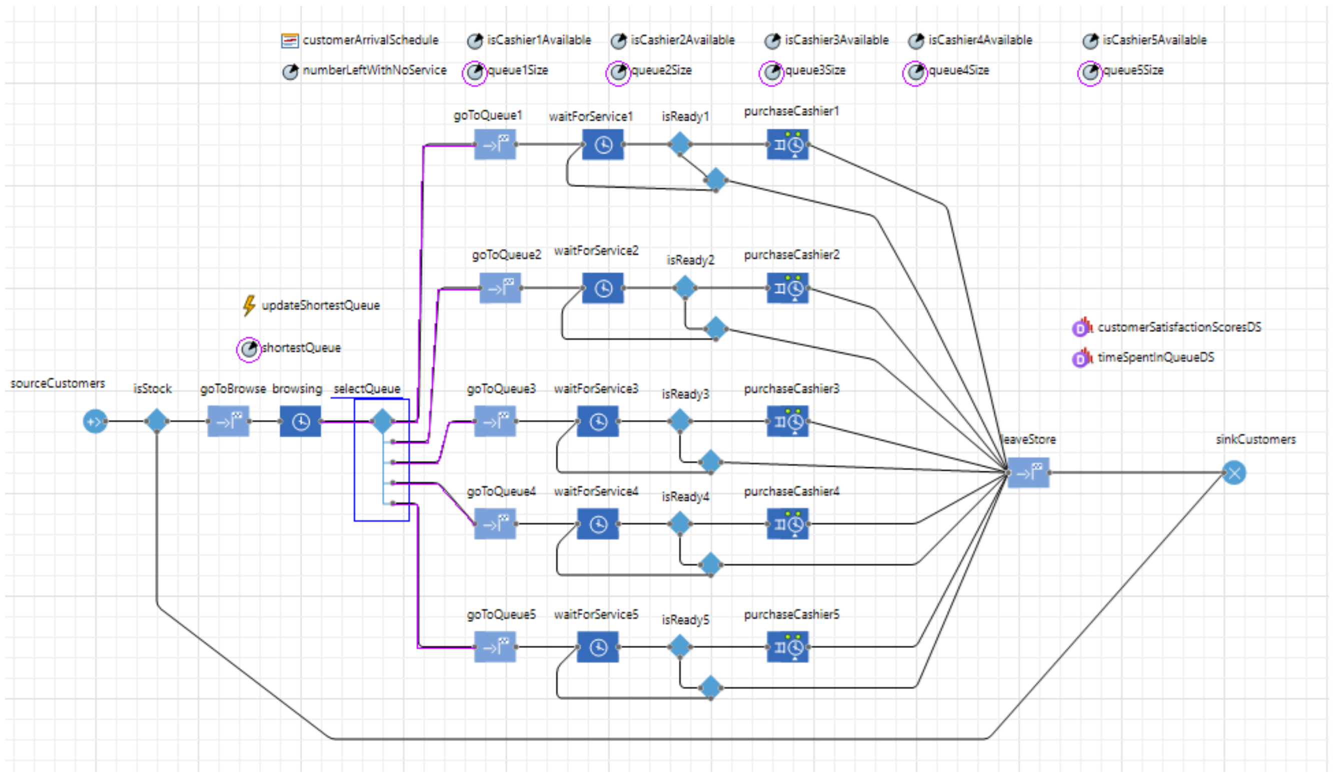
Like for the stocking logic, the ’cleaning’ flow rate is the product of the ’cleaningRate’ parameter (again $1/60$) and the ’numAgentsCleaning’ parameter.

Large Stores

Larger stores were created with identical logic to the small stores, save for the amount of stock customers remove being halved, and the amount of the cleanliness percentage they remove being halved. As the store is bigger, this emulates the fact it will have a higher raw stock capacity, and will be harder to clean.



Another key difference is that the stores have 5 queues instead of 2, as shown above. This means that the logic for customer queuing/service has more branches to accommodate:



To determine the shortest queue size, as the check is no longer the comparison of two parameters, it was necessary to include a new function and associated parameter - 'updateShortestQueue' and 'shortestQueue'. These properly direct output from the 'selectQueue' decision branch. 'updateShortestQueue' finds the shortest queue with the following code:

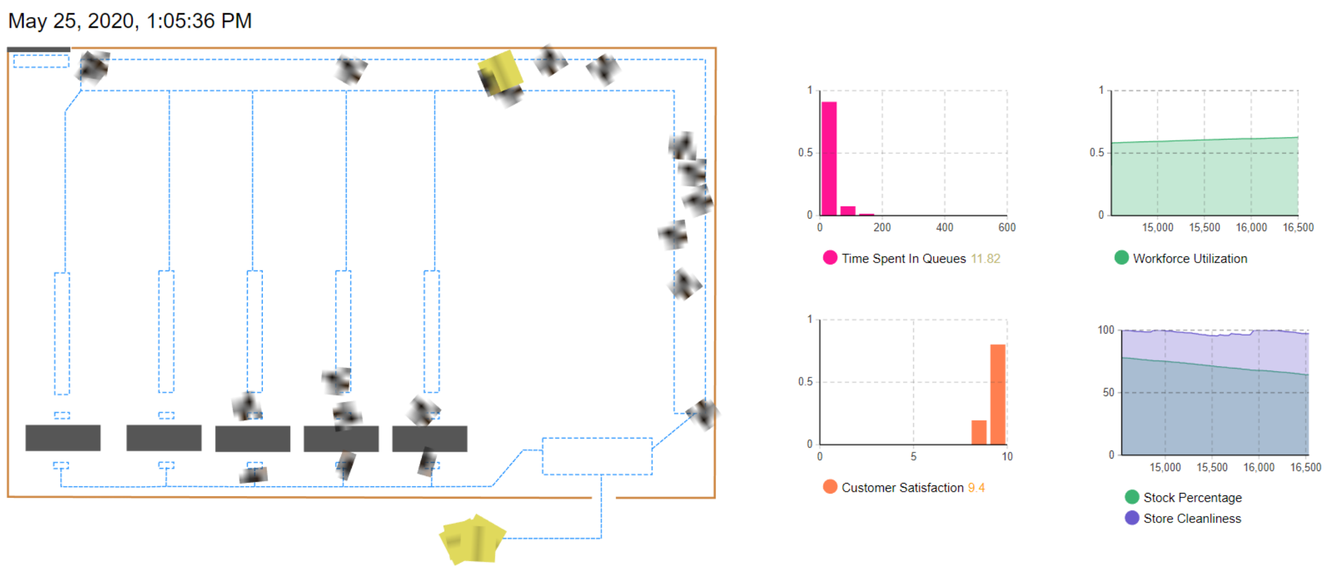
```
int queues[] = {queue1Size, queue2Size, queue3Size, queue4Size, queue5Size};
int a = 0;

for(int i = 1; i <5; i++){
        if(queues[i] <= queues[a]){
                a = i;
        }
}
shortestQueue = a + 1; //Adding 1 to shift the indexing for the output
```

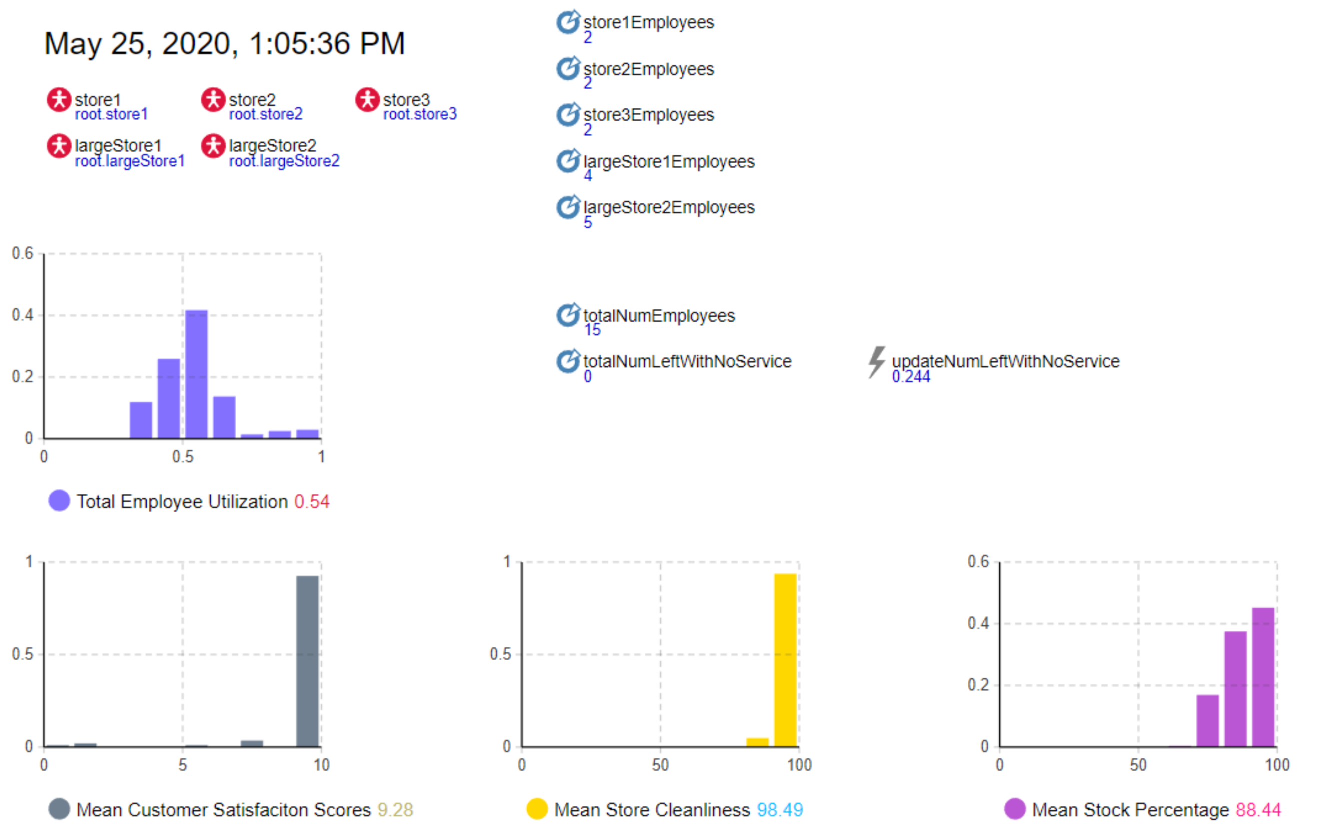
Aside from these factors, the stores function identically, with their decision variables managed by the Main class.

Main & Runtime Statistics

During runtime each store provides its own individual statistics, visualised in histograms and time plots:



The averages across all stores are likewise kept track of in the Main class, which houses the parameters that set the number of Employees:



store1Employees
2

store2Employees
2

store3Employees
2

largeStore1Employees
4

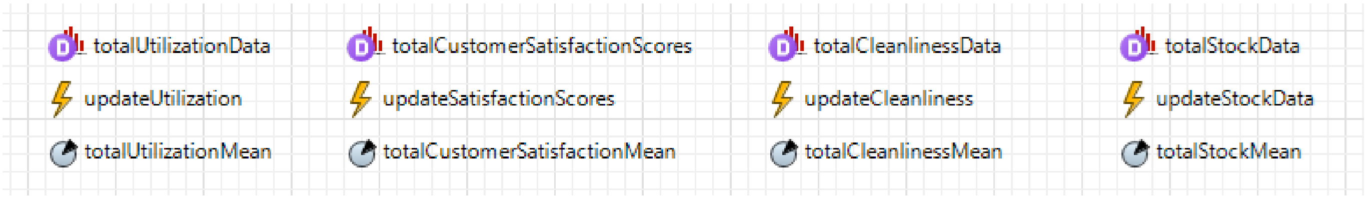
largeStore2Employees
5

totalNumEmployees
15

totalNumLeftWithNoService
0

updateNumLeftWithNoService
0.244

It also houses its own data sets for the four outputs, and stores their means into seperate parameters for use as constraints in the optimisation experiment:



All of these statistics were calculated by summing over all the stores and dividing by 5. However, the individual store utilisations had to be calculated by hand (as expressed in equation (1)). This was due to the fact that Anylogic's built-in resource pool utilisation factor was not recognising the cleaning task as presenting 'busy' units at that time. As a result, I had to calculate it myself by adding parameters 'currentUtilization', 'totalUtilization' and 'numberOfInstances', updated with the following code:

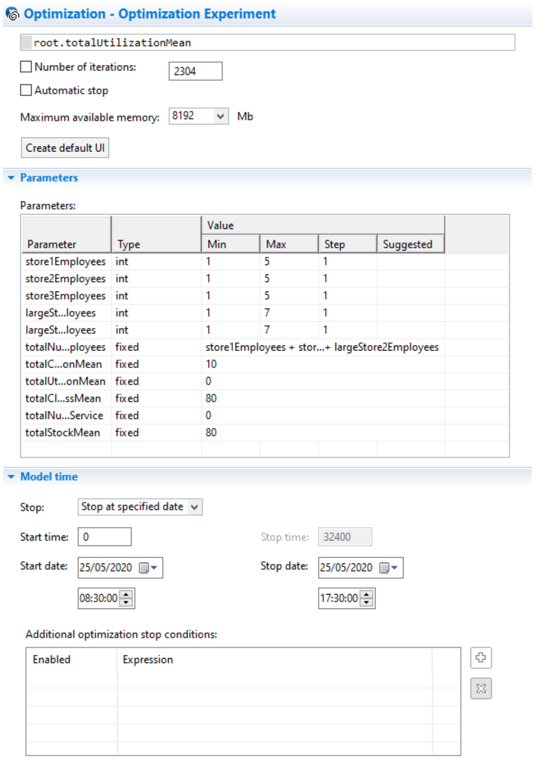
```
    currentUtilization = (double)resourcePool.busy()/(double)resourcePool.size();

    totalUtilization = (totalUtilization*numberOfInstances);
    numberOfInstances++;
    totalUtilization = (totalUtilization + currentUtilization)/numberOfInstances;
```

This updates the mean utilisation since the model has run without storing data points every second. To sum over an increasing number data points (of order 10^5) every second would prove incredibly slow.

Experiments & Output Analysis

To generate an optimisation experiment, I simply included the root (Main) in the built-in optimisation package for Anylogic. The variable parameters were the individual store employee numbers, and the constraints were set:



Optimization - Optimization Experiment

root.totalUtilizationMean

Number of iterations: 2304

Automatic stop

Maximum available memory: 8192 Mb

Create default UI

Parameters

Parameters:

Parameter	Type	Value			
		Min	Max	Step	Suggested
store1Employees	int	1	5	1	
store2Employees	int	1	5	1	
store3Employees	int	1	5	1	
largeSt...loyees	int	1	7	1	
largeSt...loyees	int	1	7	1	
totalNu...ployees	fixed	store1Employees + stor...+ largeStore2Employees			
totalC...onMean	fixed	10			
totalUt...onMean	fixed	0			
totalCl...ssMean	fixed	80			
totalNu...Service	fixed	0			
totalStockMean	fixed	80			

Model time

Stop: Stop at specified date

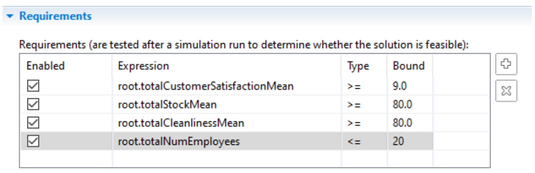
Start time: 0 Stop time: 32400

Start date: 25/05/2020 Stop date: 25/05/2020

08:30:00 17:30:00

Additional optimization stop conditions:

Enabled	Expression

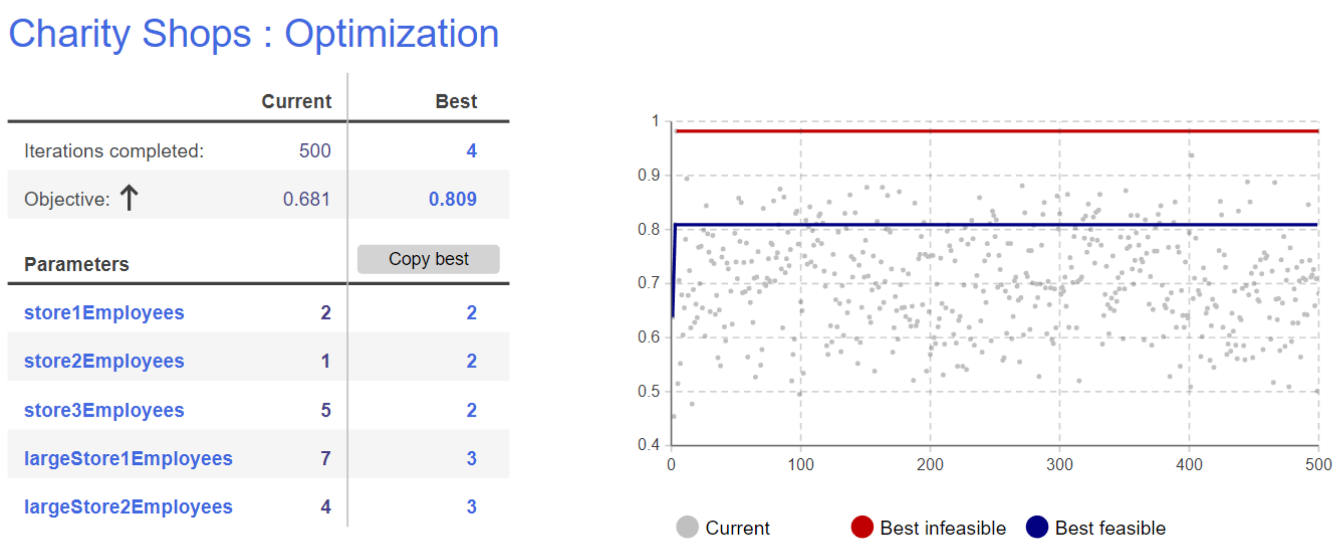


Requirements

Requirements (are tested after a simulation run to determine whether the solution is feasible):

Enabled	Expression	Type	Bound
☑	root.totalCustomerSatisfactionMean	>=	9.0
☑	root.totalStockMean	>=	80.0
☑	root.totalCleanlinessMean	>=	80.0
☑	root.totalNumEmployees	<=	20

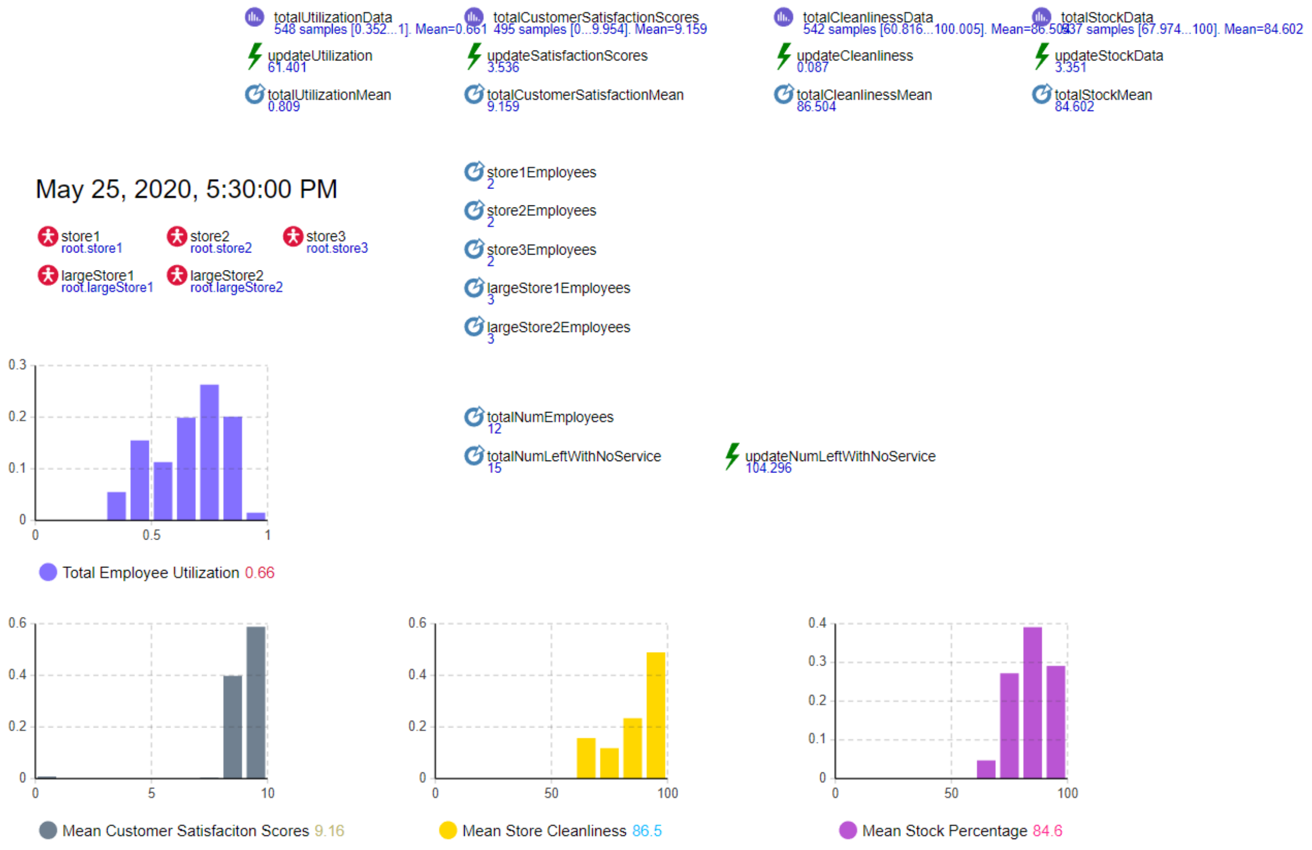
Despite some attempts to optimise the runtime, this experiment ran incredibly slowly, taking over an hour to return the optimal solution:



Charity Shops : Optimization

	Current	Best
Iterations completed:	500	4
Objective: ↑	0.681	0.809
Parameters		Copy best
store1Employees	2	2
store2Employees	1	2
store3Employees	5	2
largeStore1Employees	7	3
largeStore2Employees	4	3

This is the main result; a utilisation of 80.9%, averaged across all stores for the day. We can verify this by running the experiment with the operational parameters, and examine the other parameters to check they fall within constraints:



totalUtilizationData
548 samples [0.352...1]. Mean=0.661
updateUtilization
61.401
totalUtilizationMean
0.809

totalCustomerSatisfactionScores
495 samples [0...9.954]. Mean=9.159
updateSatisfactionScores
3.536
totalCustomerSatisfactionMean
9.159

totalCleanlinessData
542 samples [60.816...100.005]. Mean=86.504
updateCleanliness
0.087
totalCleanlinessMean
86.504

totalStockData
[illegible]37 samples [67.974...100]. Mean=84.602
updateStockData
3.351
totalStockMean
84.602

May 25, 2020, 5:30:00 PM

store1
root.store1
store2
root.store2
store3
root.store3
largeStore1
root.largeStore1
largeStore2
root.largeStore2

store1Employees
2
store2Employees
2
store3Employees
2
largeStore1Employees
3
largeStore2Employees
3

totalNumEmployees
12
totalNumLeftWithNoService
15
updateNumLeftWithNoService
104.296

● Total Employee Utilization 0.66

● Mean Customer Satisfaciton Scores 9.16

● Mean Store Cleanliness 86.5

● Mean Stock Percentage 84.6

We note here that the 'Total Employee Utilization' as calculated by the histogram is actually incorrect. The reason for this is that it attempts to calculate over all data points since the beginning of the simulation, which the individual calculations already account for. The actual utilisation, 'totalUtilizationMean' is calculated, independently of the data set, at 0.809 as expected. This is not an issue for the other parameters, however. We observe that these parameters are within constraints, and verify this result as the optimal solution.

Conclusions & Afterthoughts

The results of the optimisation experiment demonstrate a reasonable allocation of resources to achieve an optimal result. However, it is important to understand what the limitations of the model are. It assumes employees are identical in nature, work rate and attendance to work, and that customers behaviour follows a strict logic. These, amongst other assumptions and simplifications, will likely result in the model lacking a high degree of accuracy. However, it does still provide an applicable insight into the dependence of the operational variables on the stated parameters.

In hindsight, I would have spent more time optimising the model itself to achieve a faster operation, and reducing the 1-2 hour run-time of the optimisation experiments, as well as defining some variables a bit more precisely, and accounting for some extra factors that would also affect the results (e.g employee breaks, redistribution of the workforce during the day, and more variability in customer behaviour).