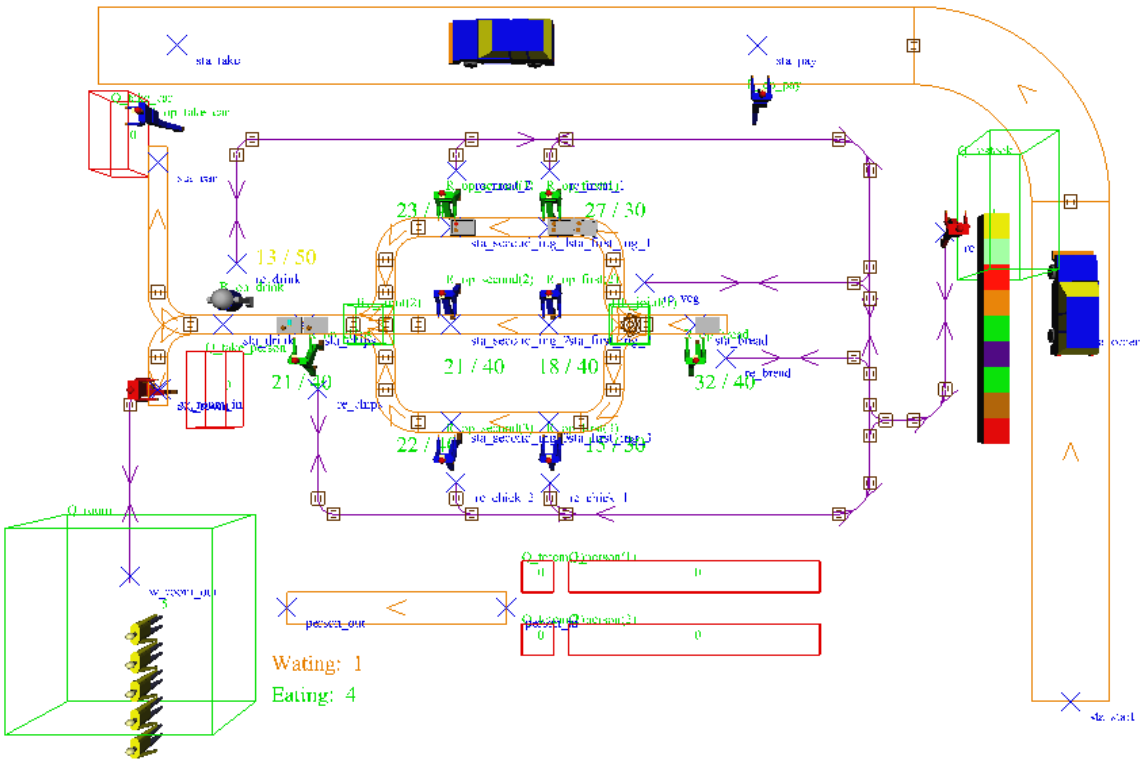
Final Project Group

Modelling and simulation of menu preparation and order management in a fast food restaurant

Group Members			
Periotto	*Riccardo*	*University of Trento*	*230266*
Sartori	*Mattia*	*University of Trento*	*230267*
Mutti	*Giacomo*	*University of Trento*	*233189*

1. Presentation



The project aims at modelling the inner working of a fastfood restaurant, taking into account a correct management of the orders coming from a drive-in and a normal dining room. With Automod, the final aim is to simulate the dynamics of the model to be able to make accurate decisions for a real implementation of the process. Given an average daily arrival rate and given the target of orders to satisfy, the decisions involve the replacement of human operators with machines, the proper setting of some organisational aspects and parameters such as the stations' buffer stocks size and the design of a suitable capacity of the dining room in order to obtain a reasonable preparation time and to analyse the number of people to accommodate in a typical scenario. The decision process is carried out by iterative simulations changing the working conditions and analysing the results with respect to the mentioned simulation objectives.

In the model, the drive-in customers access the restaurant service in two ways: by car through the drive-in lane or walking in to have their meal in place in the dining room. Considering the layout of the model, the first option refers to the cars coming from the far right of the environment and stopping at the order point

to choose the meal. Once the order is issued, they proceed along the road to the payment point. After that they can get to the picking point to retrieve their meal and leave. In correspondence of the payment and picking points, two operators are placed in order to let customers pay and take the meal. The customers that want to dine in the restaurant come from a pathway. They order and pay outside the structure via one of two totems (kiosks). After that, they enter the dining room and wait to be served. As soon as the complete order is ready, a waiter delivers it to the customers. Once they have eaten, they leave the restaurant.

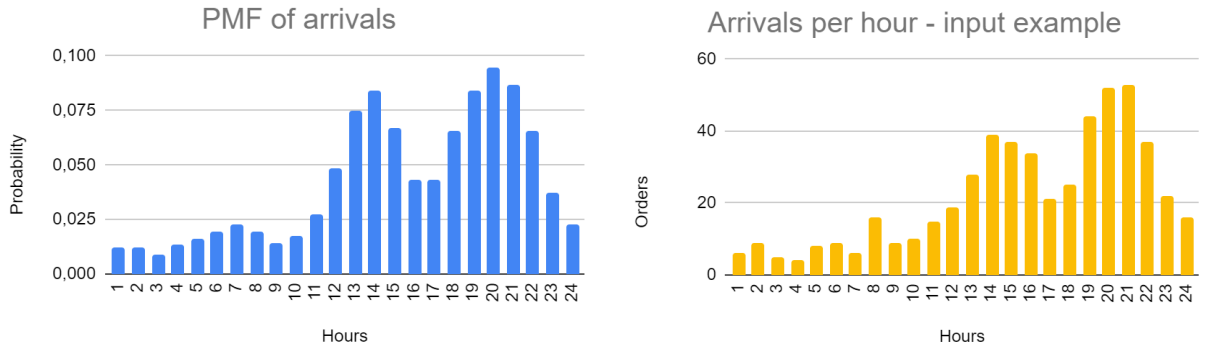
The design of the model just described is based on several real world considerations:

- all the sizes adopted are realistic
- the cooking process does not begin until the customer has completed the order phase
- cars only leave when the correct and complete order has been handed out
- people waiting in the dining room do not start eating until the waiter has completed the trip from the kitchen area to the dining place with the right meal
- timings involved in the different phases of the process are modelled as random variables and resembles real world data
- the average eating time is 20 minutes. This value comes from the consideration of real data available on the Google pages of the McDonald's located in Trento: McDonald's Trento Nord and McDonald's Trento Sud. These report an average stay of 10-40 minutes
- the customers' arrival rate also comes from a study of realistic data.

The last point requested more considerations than the others and deserves further details. To model it, we considered the graphics displayed by Google popular times for the two mentioned McDonald's. Based on a rough estimation of the daily distribution of the average attendance at the two fast foods in the different hours, we developed a Python script which builds an input file scheduling the customers arrival rate.

Here below we report two charts generated in the data sheet file reachable through the link[1]. They contain respectively: the probability mass function of the arrival rate for each hour (left) and the daily arrival schedule used as input in the simulations (right). The data used for the chart on the right are generated given a target cumulative value of 500 orders per day. In the Python script we used the average number of arrivals as the lambda variable of a negative exponential distribution to generate a random scheduling of arrivals satisfying the "real" PMF. The real scheduling is shown below. As it can be seen, the distribution of the arrivals presents two peaks around lunch and dinner time, when the arrivals are more.

Arrivals probabilities and input example



[1]https://docs.google.com/spreadsheets/d/1E2FK2SZrLkDwBWfD2Y4uTdvC0ITWT5Dv-d_478QmidM/edit#gid=1385022804

The cooking process takes place in the middle of the layout in the space designed as the kitchen. We considered a good solution to implement the flow of goods (i.e. the menus) with conveyor belts in order to minimise the motion of operators in the kitchen thus maximising productivity and efficiency.
Every customer order has an assigned ID and can be composed of different menus. A base menu is made of a burger to which chips and/or a drink may be added. The menus share the parent order ID and a tray is assigned to each menu. The trays travel from right to left on top of the conveyor system passing through the work stations where the ingredients are consecutively added to form the menus.
The burgers offered by our model are of three types: meat burger, vegan burger or chicken burger. Each of them is composed of bread, the primary ingredient and some secondary ingredients that differ based on the type (some specific toppings and condiments). The menu composition starts from the station where the bread is heated and added to the tray. After the bread station, the conveyor splits into three lines according to the type of the burger in the menu. Each line has a station for the primary ingredient and then another for the secondary ingredients. After the preparation of the burger, the three lines merge together again in a single one. This line conveys the menus through two additional stations where the fries (chips) and/or a drink may be added.

Once the menu is completed, based on whether the order came from a car or from a person, it is directed towards the queue for the car orders (top left of conveyor system) or towards the queue for the orders to be served in the dining room. The overall cooking process can be seen as an assembly line “building” the menus and sending them to one of the two destinations.

When a menu arrives at the end of the conveyor in the car orders’ side, the ones with the same order ID are put together and delivered to the corresponding car by the human operator at the picking place. Since the car system logic is intrinsically FIFO, a car in the line has to wait for all cars before it even if its order has been completed first. On the other hand, if a menu’s destination is the dining room, it arrives at the bottom-left end of the line and it is grouped together with menus sharing the same order ID. Once the order is complete, a waiter takes the full package with all menus and brings it to the corresponding people/person. In the people system, the FIFO logic is implemented with respect to the orders’ queue and not to the customer. Hence, as soon as an order is ready, regardless of the order ID (and thus of the customers’ arrival time), it will be brought by the waiter to the dining room.

A further real-world aspect that the model contains is the presence of an ingredients’ buffer stock at each station. With this, the system models the dynamics through which the ingredients, that can be present only in a limited amount at the workstations, have to be brought from a common stock to the correct point in the line. The replenishment of the buffer stocks is in charge of an operator that acts as a carrier.
In the base version of the system we “activate” the “restocker” to refill an ingredient when the workstation is completely out of stock. In the following paragraphs we will present an improvement of the system in which we set a threshold at which to start the replenishment for the ingredient that is running out. From an industrial point of view, this last subsystem can be seen as the logistic department supplying the production department with the raw materials needed. The overall aim is to reproduce a simplified real-world stock management system to determine the correct buffer stock size for every station and the number of “logistics” operators in order to minimise downtimes.

The report does not contain data for the improvements on the car lane dimensions since we focused on the internal design of the drive-in with a more productivity-oriented goal towards demand satisfaction. On the contrary, the dining room related statistics are of great interest in order to properly design its real capacity and dimensions. To be able to collect such data, we modelled the dining room as a queue with infinite capacity.

2. Simulation Entities

2.1. Loads

L_dummy (GL:1, Distribution: Constant, Mean: 0): generated one time only when the simulation starts, this load is used to initialise the system through the process *P_init*.
L_car (Generated according to input file): represents a car that uses the fast food drive-in lane. It stops at the first point on the road to place the order, moves on to the next stop to pay and then travels toward the picking point. Once the car gets to the picking point, if the order is ready, it takes the meal and leaves. The instant of creation of every car is specified in the input file.
L_person (Generated according to input file): represents the customers arriving to the system on foot and that eat in the dining room. As for the cars, the creation is specified in the input file. After the creation, every customer follows one of the 2 available lines queuing in the less crowded one. Once the line-related totem is free, the person gets access to it to place the order and pay. Then the customer moves into the dining room, waits for the order to arrive via the waiter, eats, and finally leaves. In order to avoid a too crowded simulation, one load of type person models either 1 person or a group of people ordering and eating together. The number of people related to an order is encoded in the order_dimension field taken from the input file. The considerations about the size of the dining room are made with this last figure.
L_menu (Generated according to input file): represents a single menu for a specific order. It always includes a burger and can eventually contain chips and/or a drink. Every order (not modelled as a load) is made up of one or more (possibly different) menus, each with the same order ID. When this type of load is instantiated into the system, its appearance is an empty tray (described better later in the processes section).
L_order_room (No creation spec): once all the menus of a certain order ID arrive at the end of the conveyor facing the room, they are converted into this single load which is then served by the waiter .
L_stock (No creation spec, generated in P_init_stock): it is the “package” that has to be carried by the restocker operator from the warehouse to the workstation where an ingredient is running out of stock. There are no creation specifications, but, given the 9 stations in which the food is processed, we need 9 loads of type *L_stock*, one for each ingredient. The nine loads are generated in the process *P_init_stock* where also the values of the attributes *V_stock_type* and *V_stock_location* are initialised.
Loads only used for graphics purposes:

- *L_bread*
- *L_meat_ing_1*
- *L_meat_closed*
- *L_meat_chips*
- *L_meat_drink*
- *L_meat_completed*
- *L_veggy_ing_1*
- *L_veggy_closed*
- *L_veggy_chips*
- *L_veggy_drink*
- *L_veggy_completed*
- *L_chicken_ing_1*
- *L_chicken_closed*
- *L_chicken_chips*
- *L_chicken_drink*
- *L_chicken_completed*

These loads are used to graphically display the addition of the different ingredients during the composition of the meal. They are all graphics that are substituted consecutively to the load of type menu in the process *P_menu*. The images and further details about these loads are reported later while describing the process *P_menu_generator.*

2.2. Global Variables

1. *V_car_current_counter (Type: Integer; Dim: 1):* number of cars currently inside the system.
2. V_car_out_counter *(Type: Integer; Dim: 1)*: number of cars that left with the order.
3. V_car_queue_w_time *(Type: Time; Dim: 1)*: time waited on the queue by the customers coming by car before ordering.
4. *V_debug (Type: Integer; Dim: 1)*: it defines if the output has to be verbose or not. In particular, if set to true, it makes the system print messages for verifying the correct implementation of the order lists and to save some specific data on external files.
5. *V_eat_avg (Type: Time; Dim: 1):* average time needed for the people to eat
6. *V_file_car_w_t_name (Type: String; Dim: 1):* name of the output file on which to write the menu preparation time for orders of customers coming by car.
7. *V_file_car_w_t_ptr (Type: FilePtr; Dim: 1):* pointer to the output file on which to write the menu preparation time for orders of customers coming by car.
8. *V_file_downtime_name (Type: String; Dim: 1):* name of the output file on which to write the downtimes encountered by the workstation due to lack of ingredients.
9. *V_file_downtime_ptr (Type: FilePtr; Dim: 1):* pointer to the output file on which to write the downtimes encountered by the workstation due to lack of ingredients.
10. *V_file_input_name (Type: String; Dim: 1):* name of the input file containing all the orders
11. *V_file_input_ptr (Type: FilePtr; Dim: 1):* pointer to the input file containing all the orders
12. *V_file_menu_list_name (Type: String; Dim: 1):* name of the output file on which to write the list of the menu generated by the system during the simulation.
13. *V_file_menu_list_ptr (Type: FilePtr; Dim: 1):* pointer to the output file on which to write the list of the menu generated by the system during the simulation.
14. *V_file_pers_w_t_name (Type: String; Dim: 1):* name of the output file on which to write the menu preparing time for orders of customers coming on foot.
15. *V_file_pers_w_t_ptr (Type: FilePtr; Dim: 1):* pointer to the output file on which to write the menu preparing time for orders of customers coming on foot.
16. *V_headers (Type: String; Dim: 1):* used to read the first row of the input file.
17. *V_menu_car_time (Type: Time; Dim: 1):* time needed to prepare a menu for orders of customers coming by car.
18. *V_menu_counter (Type: Integer; Dim: 1):* total number of menus
19. *V_menu_pers_time (Type: Time; Dim: 1):* time needed to prepare a menu for orders of customers coming on foot.
20. *V_order_counter (Type: Integer; Dim: 1)*: total number of orders.
21. *V_pers_counter (Type: Integer; Dim: 1):* total number of loads of type person that pass through the system. It is used in the process *P_menu* to set the priority of elements in an order list (refer to the process).
22. *V_pers_current_counter (Type: Integer; Dim: 1):* number of loads of type person currently inside the system.
23. *V_pers_eating_counter (Type: Integer; Dim: 1):* number of loads of type person currently in the dining room and eating (used to update the labels).
24. V_*pers*_out_counter *(Type: Integer; Dim: 1)*: number of loads of type person that left with the order.
25. V_pers_queue_w_time *(Type: Time; Dim: 1)*: time waited on the queue by the customers coming on foot before ordering.
26. *V_pers_room_counter (Type: Integer; Dim: 1):* number of physical people currently in the dining room and eating (possibly more than one for a single load of type person).

27. *V_sing_order_car_avg (Type: Time; Dim: 1)*: average time needed to order a single menu for each of the customers coming by car.
28. *V_sing_order_pers_avg (Type: Time; Dim: 1):* average time needed to order a single menu for each of the customers coming on foot (interaction with the totem).
29. *V_sta_oper_time_avg (Type: Time; Dim: 9):* average time needed by the nine workstations to prepare the menu.
30. *V_sta_proc_time_ratio (Type: Real; Dim: 9):* ratio between the actual registered time and the theoretical average time for each workstation to process the menu.
31. *V_stock_downtime (Type: Real; Dim: 1):* total downtime encountered by the workstations due to lack of ingredients.

2.3. Load Attributes

1. *V_order_id (Type: String; Dim: 1)*: ID of the order. One single order can comprehend multiple menus but since they have been ordered by the same customer they will have the same order ID. The order ID is useful also in the implementation of the processes *P_car* and *P_person* in order to make the matching between the customer and corresponding order. In the simulations we used orders starting from ID=1000.
2. *V_arrival_time (Type: Integer; Dim: 1)*: time at which the customer has arrived in the system.
3. *V_order_provenience (Type: String; Dim: 1)*: defines the provenience of the order. The value can be either “car” or “person”. It is used by the processes also when the menu has to be moved toward the exit of the drive-in.
4. *V_order_dim (Type: Integer; Dim: 1)*: number of menus in a single order. A single load of type car or person models one or more people ordering and eating together.
5. *V_drink (Type: Integer; Dim: 1)*: defines whether the menu contains a drink or not (works like a boolean variable).
6. *V_chips (Type: Integer; Dim: 1)*: defines whether the menu contains the chips or not (works like a boolean variable).
7. *V_menu_pointer (Type: ProcessPtr; Dim: 1)*: process pointer used to define the assignment of the load *L_menu* to the correct process *P_menu* (vectorial entity) according to the type of the burger (meat, vegetarian or chicken).
8. *V_stock_type (Type: String; Dim: 1)*: represents the type, i.e. the ingredient, of the stock package in the warehouse. It is used to define the ingredient for the loads of type *L_stock*.
9. *V_stock_location (Type: Location; Dim: 9)*: indicates the control points where the restocker has to bring the load *L_stock* to refill the related station buffer stock.
10. *V_order_ready (Type: Integer; Dim: 1)*: it tells if all the menus of a specific order have arrived in the queue and therefore if that order is ready to be delivered. It works as a binary variable and it is used in the processes *P_car* and *P_person* to check when all the related menus are ready and can be removed from the order list
11. *V_order_level (Type: Integer; Dim: 1)*: this variable is used to check the number of menus of an order that have been completed and are ready to be delivered. It is used in the processes *P_car* and *P_person* to control if all menus of an order are ready (*V_order_level=V_order_dim*) and if this is the case *V_order_ready* is set to 1. For a better understanding of this procedure refer to *P_car* and *P_person*.
12. *V_load_ptr (Type: LoadPtr; Dim: 1)*: this load pointer is used in the processes *P_person* and *P_car* to check if the order is ready to be delivered
13. *V_load_type_ptrs (Type: LoadTypePtr; Dim: 5)*: given the menu composition, it is used to store the graphics that the menu has to take in the different steps. In particular, the dimension is 5 as the five

phases of the meal composition: bread, first ingredient, complete burger, chips, drink (refer also to the loads defined for graphic purposes).

14. *V_menu_value (Type: Real; Dim: 1)*: states the total price of the menu which depends on the type of the meal and on whether the chips and or drink have been ordered. The components of the menu have the the following prices assigned:
    Meat burger = 7
    Vegan burger = 5
    Chicken burger = 6
    Chips = 3
    Drink = 2.5
15. *V_menu_processing_time (Type: Time; Dim: 1)*: used to calculate how long it takes for a menu to move from its generation to the exit from the drink station (completion).
16. *V_station_proc_time (Type: Time; Dim: 1):* used to track the processing time of each station for the current load of type menu. The processing time starts when the previous task is finished and ends when the current task is finished, so it comprises both the waiting time on the conveyor and the actual processing.
17. *V_order_time (Type: Time; Dim: 1):* load attribute used to measure the time between the arrival of the customer and the end of the ordering phase.
18. *V_menu_type (Type: String; Dim: 1)*: it specifies the type of the menu. This information is used when the details of the menus are printed on the external file.

2.4. Labels

LB_eating_pers (Dim: 1): displays the number of customers eating in the dining room.
LB_stock (Dim: 9): display the current amount of ingredients in the workstations' buffer stock.
LB_waiting_pers (Dim: 1): displays the number of customers waiting in the dining room for their order to arrive

2.5. Resources

R_op_bread (Dim: 1): operator devoted to bread preparation: heat the buns and put them in the trays.
R_op_first (Dim: 3): operators devoted to the preparation and addition of the main ingredients. Setting them as vectorial entities allows for efficient access exploiting the same procindex of the process *P_menu* (vectorial entities of dimension 3 as the number of burger types).
R_op_second (Dim: 3): operators devoted to the preparation and addition of the secondary ingredients and toppings.
R_op_chips (Dim: 1): operator devoted to chips preparation: put the chips in the fryer, preparation of chips portions, addition to the tray.
R_op_drink (Dim: 1): resource devoted to glasses filling.
R_op_pay (Dim: 1): this operator manages the car payments.
R_op_take_car (Dim: 1): this operator takes the correct menus from the pile in *Q_take_car* once a full order is ready and hands it out to the car with corresponding order ID.

2.6. Queues

Q_person (Dim: 2; Capacity: Infinite): places where people line up waiting to use the totems. As soon as a totem is available, one customer in the respective line leaves this queue to access *Q_totem* in a FIFO fashion.

Q_totem (Dim: 2; Capacity: 1): represent the 2 totems (the kiosks) where one customer at a time (one for each totem, capacity=1) can order and pay for the meal. Once this process has finished, the person leaves this queue and moves towards the dining room. The totem becomes available for the following customer.
Q_room (Dim: 1; Capacity: Infinite): models the dining room where the customers will wait for the correct order to arrive and then eat. The eating time is modelled with a gaussian distribution and when it "expires" the customer leaves the system. The capacity is infinite because in our design process we want to track the overall number of customers present at every time in the dining room to properly choose the dimensions.
Q_restock (Dim: 1; Capacity: Infinite): this queue aims at modelling the warehouse where all the ingredients are stocked before being brought to the buffer stocks in the kitchen. Every time a workstation's buffer stock falls under a certain threshold, the "restocker" picks up the respective *L_stock* from this queue and then goes to refill the station.
Q_take_person (Dim: 1; Capacity: Infinite): once the menu is completed, if it is destined to go to the dining room it moves in this queue where it will wait for the entire order to be ready. At that point, the waiter will take the menus corresponding to the same order and bring them to the room.
Q_take_car (Dim: 1; Capacity: Infinite): this queue is analog to the previous one, but it holds the menus ordered by the cars.

2.7 Order Lists

- *OL_take_car (Dim: 1, Sort by: Load Attribute V_order_id)*: used to manage the menus in *Q_take_car*. Once all the menus of an order are in the order list, they can be handed out to the car with the corresponding order ID. It is the *L_car* that makes the order action for the menus in the list and takes a number *V_order_dim* of them. It is not important to check for correspondence in the order ID since the menus are sorted by it in the list. Menus are taken (ordered) by sending them to die.
- *OL_take_person (Dim: 1, Sort by: Lowest priority)*: used to manage the menus in *Q_take_person*. The logic is similar to the one explained in the previous order list. The difference here is that the sorting is made by priority. The priority of the menus is changed when the complete order is ready and it is set to a number depending on the order ID. In this way we "highlight" the menus that can be taken from the list. Moreover, we take first the ready orders based on the ID order. To make an example suppose to have menus belonging to two different orders 1 and 2 with ID_1 < ID_2:
  - if the order 2 is ready while the order 1 is not, the waiter can take out the order 2
  - if the orders are both ready, the waiter takes out order 1 (the one with lower order ID and so lower priority value)
- *OL_waiting_person (Dim: 1, Sort by: Lowest entry time)*: contains all the people waiting in the room for which the order is ready to be served. Once a customer is placed in this list, it enters a waiting condition and the process is paused. The process is reactivated only when the waiter brings the meal to the corresponding person which exits the order list and starts eating. In fewer words, we use the *OL_waiting_person* to simulate the idleness of the *P_person* to avoid that the customer starts eating before receiving the food from the waiter.

2.8 Blocks

- *BL_joint (Dim: 2)*: it avoids the physical overlapping of different menus from different conveyor lines and is used also to guarantee the FIFO passage of the menus through the conveyor joint (the default policy used by Automod would let the menus pass in "batches" depending on the line of origin).

Here FIFO is related to the claims that the objects (the menus) make in order to enter the block: the first incoming menu would claim the block first and would pass through it first.

2.9 Counters

- *C_stock (Dim: 9):* these counters keep track of the current value of the buffer stock for the different workstations. Depending on the restocking policy, when a counter reaches a specific value the restocker is called to operate a refill.
  In this specific case we used a counter instead of a variable because it allows us to retrieve statistics from it. In particular we are interested in setting and reading the counter attributes “capacity” and “current”.

2.10 Tables

- *T_sta_proc_time_ratio (Dim: 9):* it is used to store the values of the variable *V_sta_proc_time_ratio*. We used this table to store all the values of the variable for each station and for each menu passing through the stations. This is useful since the plot of the punctual values assigned to the variable was unreadable. The table allows us to create a Business Graphic of the average value of the variable throughout the shifts. This results in a meaningful Business Graphic that we used to find and test specific improvements on the system (refer to point 3. Simulation results).

2.11 Systems

In this section we list all the systems present in the *drive_in* model giving a brief description of their usage.

- *conv (Type: Conveyor):* main conveyor over which the trays of the menus flow from the generation of the menu to its delivery. We have 11 stations, 9 of which correspond to the points in which ingredients are added to the meal and the other 2 stations are for the two possible menu destinations, namely the car exit and the dining room exit.
  Watching real conveyors data, we assumed a velocity of 1 m/s to be appropriate in our system.
  Since for the nine stations we have a lot of similar entities modelled as vectorial entities of dimension 9 and we often refer to them using the index, we report here the list of the ingredients and the corresponding station number. (Note: the stations have a name, not a number, but to use the procindex we refer to the common entities of the stations with a number).

Bread	1		Chicken	6
Meat	2		Chicken toppings	7
Meat toppings	3		Chips	8
Veggy	4		Drink	9
Veggy toppings	5			

- *conv_car (Type: Conveyor):* conveyor modelling the car lane
- *conv_person (Type: Conveyor):* conveyor modelling the people sidewalk from the totems to the dining room
- *restock (Type: Path Mover):* it is the system that allows to model the replenishment dynamics in the application. Everytime a station finishes its local stock (or falls under a certain threshold) it calls the restock-employee that, starting from the warehouse (modelled as the parking area of the path mover), brings the goods up to the correct station to refill it. The employee can refill only one station at a time and its speed changes whether it is empty or carrying some ingredients. For

simplicity, the pickup time from the warehouse and the place time at the corresponding station are equal for all the ingredients.

- *waiter (Type: Path Mover):* path mover that models a waiter moving the orders from the end of the conveyor to the dining room. If there isn't any order ready to be picked up it will wait at its parking area, located in front of the end of the conveyor *conv*.

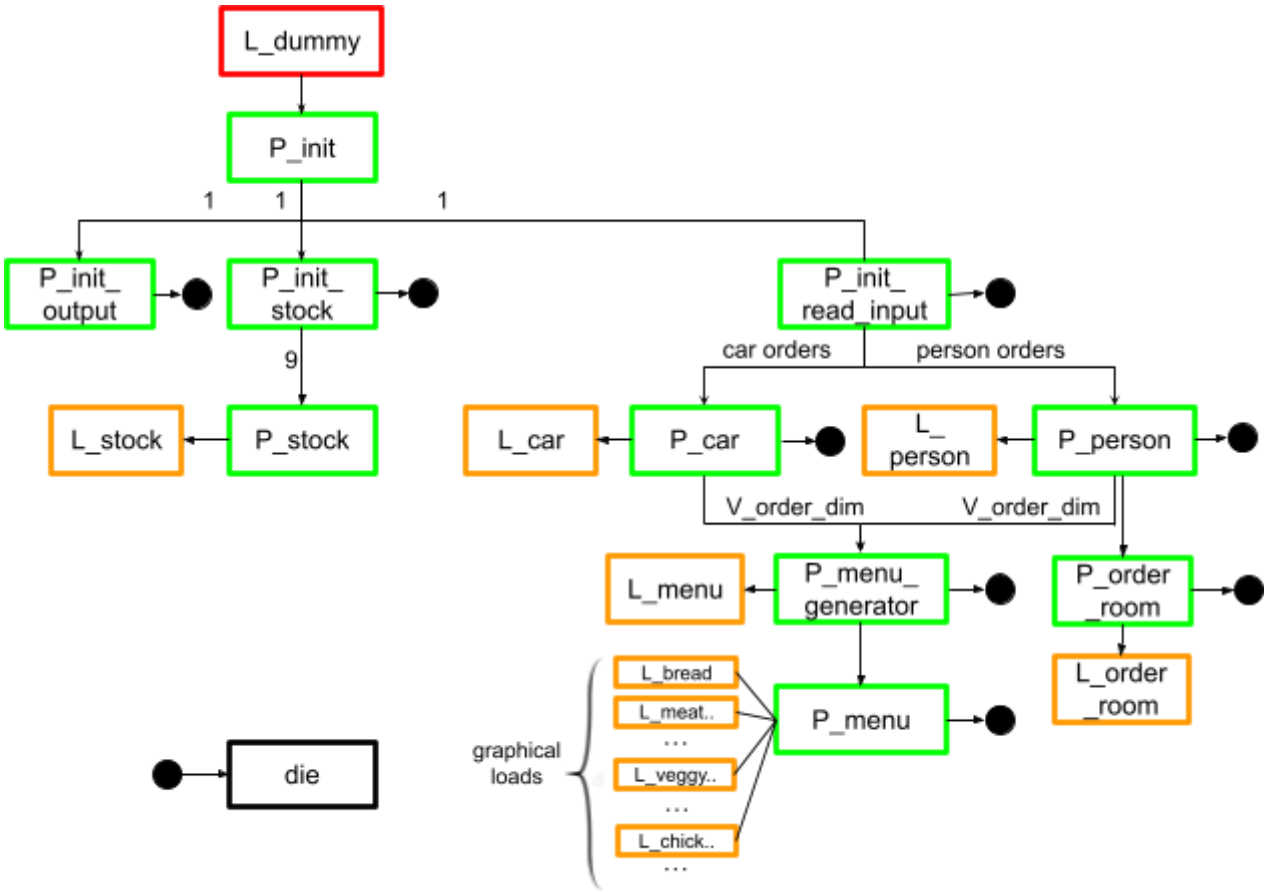
2.12 Processes

Here are reported all the processes developed for the application. We report the code for reference, but for better readability and understanding we omitted the print instructions of some messages that in the real code can be displayed by setting the variable *V_debug* to 1. The messages are kept in the original code, ready to be displayed and exploited in case of changes to the scripts to check the correct execution.

The following is a diagram representing the relations between processes and loads. The first load created when simulating the system is the one called *L_dummy* that is instantiated only once. It runs the process *P_init* that, through other calls, initialises and runs the coded part of the simulation. In the diagram, all the processes are bordered in green while the loads in orange. The execution of a generic process involves

- the setting of a load type,
- the calling of other processes and
- the execution of the *die* process.

Overall processes and loads hierarchy



P_init (Dim:1;Default Traffic Limit:1)

This process starts when the system generates the load *L_dummy*, at the start of the simulation. As the name says, the process is devoted to the initialisation of the system. This includes:

- the definition of the average time for processing, ordering and eating
- the definition of buffer stock's size and the choice of the colours for the labels
- the definition of the input and output files names
- the call of the other processes responsible for proceeding with the simulation.

```
begin
	// SET THE PARAMETERS OF ALL THE NORMAL DISTRIBUTIONS
	set V_sing_order_car_avg = 15
	set V_sing_order_pers_avg = 30
	set V_eat_avg = 20*60
	set V_sta_oper_time_avg(1) = 12
	set V_sta_oper_time_avg(2) = 35 // meat menu: time to prepare the meat
	set V_sta_oper_time_avg(3) = 35 // veggy menu: time to put the vegan burger
	set V_sta_oper_time_avg(4) = 35 // chicken menu: time to prepare the chicken
	set V_sta_oper_time_avg(5) = 30 // meat menu: time to complete the burger
	set V_sta_oper_time_avg(6) = 25 // veggy menu: time to complete the burger
	set V_sta_oper_time_avg(7) = 30 // chicken menu: time to complete the burger
	set V_sta_oper_time_avg(8) = 15 // chips: time to put the chips (second improvement)
	set V_sta_oper_time_avg(9) = 8  // drink: time to put the drink (second improvement)

	// SET THE DEFAULT COLORS OF THE LABELS
	set LB_waiting_pers color to orange
	set LB_eating_pers color to green

	// SET THE BUFFER STOCK SIZE
	set C_stock(1) capacity to 40 // bread
	set C_stock(2) capacity to 30 // meat menu: ingredient 1
	set C_stock(3) capacity to 40 // meat menu: ingredient 2
	set C_stock(4) capacity to 40 // veggy menu: ingredient 1
	set C_stock(5) capacity to 40 // veggy menu: ingredient 2
	set C_stock(6) capacity to 30 // chicken menu: ingredient 1
	set C_stock(7) capacity to 40 // chicken menu: ingredient 2
	set C_stock(8) capacity to 40 // chips
	set C_stock(9) capacity to 50 // drink

	// SET THE NAME OF THE FILES INVOLVED IN THE MODEL, AutomoD only accepts .csv or .txt
	set V_file_input_name = "arc/input_3.csv"
	set V_file_car_w_t_name = "arc/car_waiting_time.txt"
	set V_file_downtime_name = "arc/downtime.txt"
	set V_file_menu_list_name = "arc/menu_list.txt"
	set V_file_pers_w_t_name = "arc/pers_waiting_time.txt"

	// EXECUTE THE OTHER FUNCTIONALITIES
	clone 1 load to P_init_output
	clone 1 load to P_init_read_input
	clone 1 load to P_init_stock
end
```

P_init_output (Dim:1; Default Traffic Limit: 1)

This process associates a file pointer to each output file and initialises them.

```
begin
	// WRITE HEADER FOR CAR ORDER WAITING TIME FILE
	open V_file_car_w_t_name for writing save result as V_file_car_w_t_ptr
		print "Time elapsed [min],Timestamp [s]" to V_file_car_w_t_ptr
	close V_file_downtime_ptr

	// WRITE HEADER FOR DOWNTIME FILE
	open V_file_downtime_name for writing save result as V_file_downtime_ptr
		print "Station [number],Downtime [min],Timestamp [s]" to V_file_downtime_ptr
	close V_file_downtime_ptr

	// WRITE HEADER FOR MENU LIST FILE
	open V_file_menu_list_name for writing save result as V_file_menu_list_ptr
		print "OrderId [number],Type [string],Chips [bool],Drink [bool],Value [€],
			Timestamp [s]" to V_file_menu_list_ptr
	close V_file_menu_list_ptr

	// WRITE HEADER FOR PERSON ORDER WAITING TIME FILE
	open V_file_pers_w_t_name for writing save result as V_file_pers_w_t_ptr
		print "Time elapsed [min],Timestamp [s]" to V_file_pers_w_t_ptr
	close V_file_pers_w_t_ptr
end
```

P_init_stock (Dim:1; Default Traffic Limit: 1)

This process is devoted to the initialisation of the restocking dynamics. The process is divided into nine equivalent parts, one for each workstation involved in the menu composition. For each workstation, the process sets the load attributes *V_stock_type* and *V_stock_location* depending on the considered ingredient. Assuming that a working day can be a direct continuation of a previous one, we set the initial value of each of the *C_stock* counters to half of their capacity. Finally, the system initialises the labels visible on the layout and clones one load to the process *P_stock* for each workstation.

```
begin
	set V_stock_type = "bread"
	set V_stock_location(1) = restock:re_bread
	set C_stock(1) current = C_stock(1) capacity / 2
	set LB_stock(1) color to green
	clone 1 load to P_stock(1)

	set V_stock_type = "meat1"
	set V_stock_location(2) = restock:re_meat_1
	set C_stock(2) current = C_stock(2) capacity / 2
	set LB_stock(2) color to green
	clone 1 load to P_stock(2)

	set V_stock_type = "meat2"
	set V_stock_location(3) = restock:re_meat_2
	set C_stock(3) current =  C_stock(3) capacity / 2
```

```
        set LB_stock(3) color to raw umber
        clone 1 load to P_stock(3)

        // there is only a point where to load veggy ingredients
        set V_stock_type = "veggy"
        set V_stock_location(4) = restock:re_veg
        set C_stock(4) current =  C_stock(4) capacity / 2
        set LB_stock(4) color to green
        clone 1 load to P_stock(4)

        set V_stock_location(5) = restock:re_veg
        set C_stock(5) current =  C_stock(5) capacity / 2
        set LB_stock(5) color to green
        clone 1 load to P_stock(5)

        set V_stock_type = "chicken1"
        set V_stock_location(6) = restock:re_chick_1
        set C_stock(6) current =  C_stock(6) capacity / 2
        set LB_stock(6) color to green
        clone 1 load to P_stock(6)

        set V_stock_type = "chicken2"
        set V_stock_location(7) = restock:re_chick_2
        set C_stock(7) current = C_stock(7) capacity / 2
        set LB_stock(7) color to green
        clone 1 load to P_stock(7)

        set V_stock_type = "chips"
        set V_stock_location(8) = restock:re_chips
        set C_stock(8) current = C_stock(8) capacity / 2
        set LB_stock(8) color to green
        clone 1 load to P_stock(8)

        set V_stock_type = "drink"
        set V_stock_location(9) = restock:re_drink
        set C_stock(9) current = C_stock(9) capacity / 2
        set LB_stock(9) color to green
        clone 1 load to P_stock(9)

        print "*** STOCK FOR EACH OPERATOR REFILLED ***" to message
end
```

P_stock (Dim:9; Default Traffic Limit: 9)

This process(es) is responsible for controlling the restocking dynamics. As it can be seen, it has dimension nine, as there is one running instance of this process for each of the different workstations. The core functionality of the process is inside the while loop in which it continues to verify the stock level of the related ingredient. Depending on the implemented restocking policy, when the level goes under a specific threshold, it triggers the restocker operator to bring a box of ingredients from the warehouse to the workstation. In the reported code, the policy implemented is the 20%-policy, but it is possible to change it by swapping the commented instruction.

Once the restocker reaches a workstation, the refill is simulated setting the placing time of the vehicle to 15 seconds. After the refill, the label showing the current buffer level is updated and the process restarts from the beginning of the while loop. Thus, the loop is infinite and the process does not need to send the load to die. When the loop restarts, the load is moved ("teleported") back again in the warehouse. This is just to simulate the restocking action with just one load per ingredient without having to cancel it and generate it again.

```
begin
    set this load type = L_stock
    if procindex = 1 then set this color to gold
    else if procindex = 2 then set this load color to red
    else if procindex = 3 then set this color to green
    else if procindex = 4 then set this color to ltgreen
    else if procindex = 5 then set this color to orange
    else if procindex = 6 then set this color to red orange
    else if procindex = 7 then set this color to yellow
    else if procindex = 8 then set this color to green

    while 1 = 1 do
    begin
        // WAIT TO STOCK TO FINISH AND CALL RESTOCKER USING THE SPECIFIED POLICY
        move into Q_restock

        // IMPROVED RESTOCK POLICY
        while C_stock(procindex) current > C_stock(procindex) capacity / 5  do begin
        // DEFAULT RESTOCK POLICY
        //while C_stock(procindex) current > 0   do begin
            wait for 5 sec
        end
        move into restock.re_stock
        rotate this current container vehicle absolute to z 0
        travel to V_stock_location(procindex)

        // REFILL THE STOCK
        set C_stock(procindex) current = C_stock(procindex) capacity

        // -- change the label
        print C_stock(procindex) current, "/", C_stock(procindex) capacity LB_stock(procindex)
        set LB_stock(procindex) color to green
        // -- label changed

        print "*** STOCK OF INDEX ", procindex, " REFILLED. CURRENT QUANTITY: ",
               C_stock(procindex) current, " ***" to message
        rotate this current container vehicle absolute to z 180
    end
end
```

P_init_read_input (Dim:1; Default Traffic Limit: 1)

This process is the one responsible for reading the .csv input file with the scheduling of the orders. It is created at time zero and it runs till the end of the simulation. In the code we open the file, read and discard

the first row of inputs and then start reading all the rows of the input file.
Each row of the file has four columns containing the information about:

1. the order id, which is stored in the load variable *V_order_id*
2. the release time, which is stored in the load variable *V_arrival_time*
3. the order provenience, which is stored in the load variable *V_order_provenience*
4. the order dimension, which is stored in the load variable *V_order_dim*

For each row order read, this process clones a load either to the process *P_car* or *P_person* based on the value of the provenience field of the order.

```
begin
	// SET THE BEHAVIOUR REGARDING THE DEBUG MESSAGES
	set V_debug = 1

	// READING ORDERS FROM FILE
	print "*** FILE READING STARTED ***" to message
	open V_file_input_name for reading save result as V_file_input_ptr
	read V_headers from V_file_input_ptr with delimiter "\n"
	// print V_headers to message

	while V_file_input_ptr eof = false do
	begin
		read V_order_id, V_arrival_time, V_order_provenience, V_order_dim from
		     V_file_input_ptr with delimiter ";" at end
		print "    Order read: Id: ", V_order_id, ", time: ", V_arrival_time,
		       ", provenience: ", V_order_provenience, ", dim: ", V_order_dim  to message
		wait for (V_arrival_time - ac) sec   // need to wait the time of creation

		if V_order_provenience = "car" then
			begin
				clone 1 load to P_car
			end
		else
			begin
				clone 1 load to P_person
			end
	end
	print "*** FILE READING COMPLETED ***" to message
	send to die
end
```

P_car (Dim:1; Default Traffic Limit: Infinite)

This process aims at modelling the interactions between a customer coming by car through the drive-in lane and the drive-in itself. The road around the restaurant is modelled as a conveyor on which the cars are generated at the bottom right of the display and move only in the forward direction. When a car gets to the first control point, it places the order and the number of menus requested are created in the process *P_menu_generator* (that models the decisions of the customers). After the order, the car moves to the second control point where the customers can pay and finally it travels to the last control point to receive the meal.

To check whether the entire order is ready, the process cycles the order list of prepared menu *OL_take_car* counting the number of menus that are ready and have the matching order id (the local value of the load variable *V_order_id*). If this number equals the order dimension, then the process is allowed to continue taking the order from the operator and exiting from the system.

```
begin
	// ADD THE CAR INTO THE SYSTEM
	print "** CAR WITH ORDER ", V_order_id, " ARRIVED AT TIME ", ac, "**" to message
	set this load type = L_car
	inc V_car_current_counter by 1
	move into conv_car.sta_start
	travel to conv_car.sta_order
	wait for normal 20+V_order_dim*V_sing_order_car_avg,7 sec

	// SEND THE ORDERS OF THE CAR TO THE MENU GENERATOR
	clone V_order_dim load to P_menu_generator
	set V_order_time to ac
	set V_car_queue_w_time to (V_order_time - V_arrival_time)/60

	// GO TO THE NEXT STATION TO PAY
	travel to conv_car.sta_pay
	use R_op_pay for normal 30,5 sec

	// GO TO THE LAST STATION AND WAIT FOR THE ORDER
	travel to conv_car.sta_take

	// check the list as long as the whole order (all the menus) are not ready
	while V_order_ready <> 1 do begin
		if (OL_take_car current loads > 0) then begin
			wait for 1 sec
			for each V_load_ptr in OL_take_car load list do begin
				if V_load_ptr V_order_id = V_order_id then begin
					inc V_order_level by 1
				end
				if V_order_level = V_order_dim then begin
					set V_order_ready to 1
					break
				end
			end
		end
		wait for 5 sec
		set V_order_level to 0
	end

	// all the menus are ready, the operator will give them to the car
	use R_op_take_car for normal 15,3 sec
	order V_order_dim loads from OL_take_car to die
	dec V_car_current_counter by 1
	inc V_car_out_counter by 1
	set V_menu_car_time to (ac-V_order_time)/60
	open V_file_car_w_t_name for appending save result as V_file_car_w_t_ptr
		print (ac-V_order_time)/60, ",", ac to V_file_car_w_t_ptr
```

```
        close V_file_car_w_t_ptr
        print "*** CAR ORDER ", V_order_id, " COMPLETED AT TIME ", ac, " ***" to message
        send to die
end
```

P_person (Dim:1; Default Traffic Limit: Infinite)

This process is devoted to the simulation of the dynamics that involve the load (customer) of type *L_person*, starting from the arrival, to when the person finishes eating and leaves the system. One load of type person represents either a single person or a group of *V_order_dim* people eating together. We will refer to the load as "the person" or "the customer" even if it models a group.
The person is created directly in one of the two queues at the entrance and, in particular, it is placed in the shortest queue. The ordering point, i.e. the totem, is modelled as a queue with capacity 1 and the customers access it from the waiting queues in a FIFO logic as soon as the respective totem is available. The order time for a single person is a gaussian random variable with average set in *P_init* and has to be multiplied by the number of orders placed by the load *L_person*. Once the customer has ordered, we modelled its approach to the dining room by placing it on a conveyor and making it travel from the totems' side to the dining room's side.
When the customer is inside the queue, we perform a control over the completed menus' order list *OL_take_person* to search for the menus with the corresponding order ID. If all the menus ordered by the customer are ready (the variable *V_order_level* is equal *to* the value contained in *V_order_dim*), then we order them to die and clone a load to *P_order_room*. This new load models the full order and it will be delivered to the dining room by the waiter.
To select the correct menus we tried to use a built-in function that allows us to order loads from an order list based on a load attribute. The code should have been:

order V_order_dim loads satisfying V_order_id = this load V_order_id from OL_take_person to die

Unfortunately, we reported some issues with this use of the order lists and so we proceeded by setting a priority to the loads that have to be ordered and selecting the ones with lower priority as shown in the code below. The priority depends on the order ID so that if two orders are ready at the same time, the one with the lower ID is delivered first. After the control, the *L_person* is put in the order list *OL_waiting_person* waiting for the meal to arrive. Once the order (modelled as a load of type *L_order_room*) is brought in the dining room by the waiter, the customer is "unblocked" from the order list and we simulate the eating time. After that, the customer leaves (*L_person* is sent to die).

```
begin
        // ADD THE PERSON INTO THE SYSTEM
        print "** PERSON WITH ORDER ", V_order_id, " ARRIVED AT TIME ",ac, "**" to message
        set this load type = L_person
        inc V_pers_counter by 1
        inc V_pers_current_counter by 1

        // select the shortest queue
        if Q_person(1) current <= Q_person(2) current then
                begin
                        move into Q_person(1)
                        move into Q_totem(1)
                end
                else
                begin
                        move into Q_person(2)
```

```
                move into Q_totem(2)
        end

    wait for normal V_order_dim*V_sing_order_pers_avg,V_order_dim*5 sec

    // SEND THE ORDERS OF THE PERSON TO THE MENU GENERATOR
    clone V_order_dim load to P_menu_generator
    set V_order_time to ac
    set V_pers_queue_w_time to (V_order_time - V_arrival_time)/60

    // GO TOWARDS THE EATING ROOM
    move into conv_person.person_in
    travel to conv_person.person_out
    move into Q_room
    inc V_pers_room_counter by V_order_dim
    print "Wating: ", Q_room current - V_pers_eating_counter to  LB_waiting_pers

    // check the list to see if the whole order (all the menus) are ready
    while V_order_ready <> 1 do begin
        if (OL_take_person current loads > 0) then begin
                wait for 1 sec
                for each V_load_ptr in OL_take_person load list do begin
                        if V_load_ptr V_order_id = V_order_id then begin
                                inc V_order_level by 1
                        end
                        if V_order_level = V_order_dim then
                                begin
                                        set V_order_ready to 1
                                break
                        end
                end
        end
        wait for 5 sec
        set V_order_level to 0
    end

    // all the menus are ready:
    // - set their priority to the menus of the order to 0 (set okay for the waiter)
    // - wait for them to be delivered by the waiter
    for each V_load_ptr in OL_take_person load list do begin
                if V_load_ptr V_order_id = V_order_id then begin
                        set V_load_ptr priority = V_load_ptr priority - 100
                end
        end

    order V_order_dim loads from OL_take_person to die
    clone 1 load to P_order_room

    wait to be ordered on OL_waiting_person

    // the order (all the menus) arrived!
    inc V_pers_eating_counter by 1
```

```
        print "Wating: ",  (Q_room current - V_pers_eating_counter) to LB_waiting_pers
        print "Eating: ", V_pers_eating_counter to  LB_eating_pers
        set V_menu_pers_time to (ac-V_order_time)/60
        open V_file_pers_w_t_name for appending save result as V_file_pers_w_t_ptr
               print (ac-V_order_time)/60, ",", ac to V_file_pers_w_t_ptr
        close V_file_pers_w_t_ptr
        wait for normal V_eat_avg,V_eat_avg/5 sec

        // order has been consumed, the person can go away
        dec V_pers_eating_counter by 1
        dec V_pers_room_counter by V_order_dim
        dec V_pers_current_counter by 1
        dec V_pers_out_counter by 1

        print "Eating: ", V_pers_eating_counter to  LB_eating_pers
        print "*** PERSON WITH ORDER ", V_order_id, " WENT AWAY ", ac, " ***" to message
        send to die
end
```

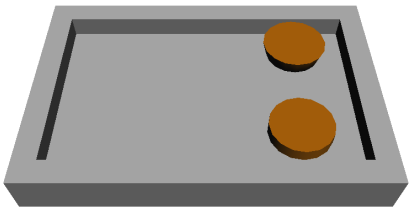
P_menu_generator (Dim:1; Default Traffic Limit: Infinite)

This process is called by a customer reaching his/her order point. It aims at modelling the customer order decisions: a menu always includes a burger that has the same probability to contain either meat, a vegetarian burger or chicken. Drinks and chips instead are optional and have respectively 75% and 90% of probability to be ordered by every menu. The graphical aspect of the menu changes as long as it goes through the kitchen chain. Before reporting the code of this process, we report and describe here briefly the different graphics used in the model.

At the beginning, the load is modelled as an empty tray. When the first operator successfully places the bread on the tray, the graphic is updated.

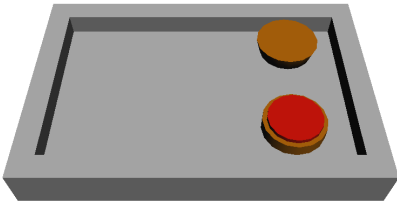
Empty tray graphic

First station graphic: bread added

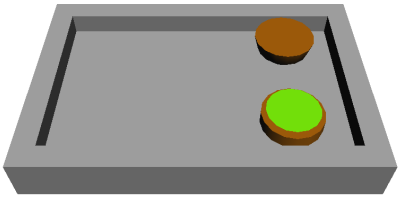


The load is then sent to the next station and, depending on the type of burger ordered by the customer, one of the following graphics will be displayed:

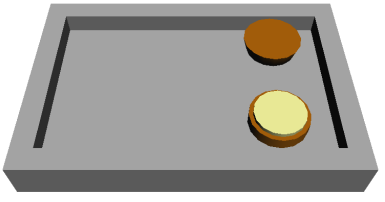
First ingredient meat burger



First ingredient veggy burger

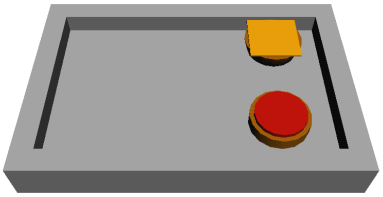


First ingredient chicken burger

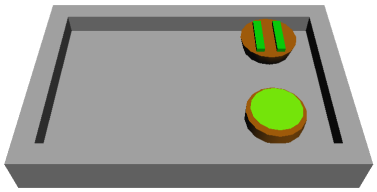


After that, the burger is filled with other ingredients that depend again on the type of the menu. The updated graphics are the following:

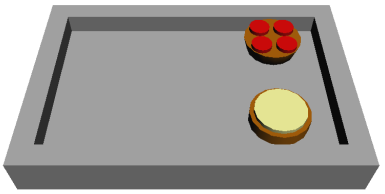
Second ingredient meat burger



Second ingredient veggy burger

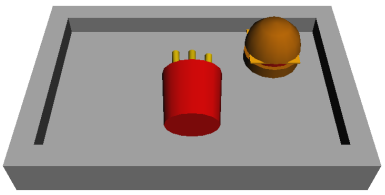


Second ingredient chicken burger

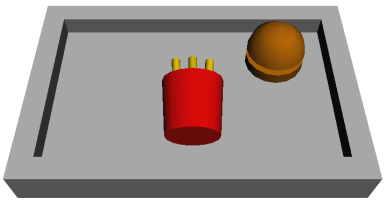


As anticipated, all the menus may or may not include chips and a drink. Depending on the presence or not of these two parts, the graphic can be set to be one of the following:

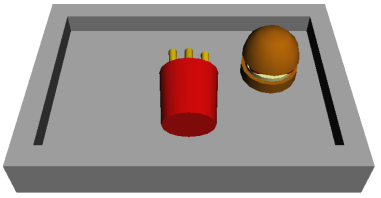
Meat burger with chips



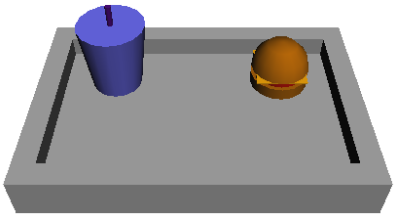
Veggy burger with chips



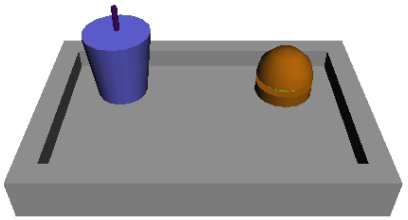
Chicken burger with chips



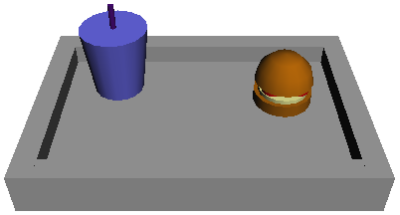
Meat burger with drink



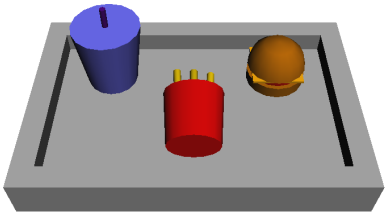
Veggy burger with drink



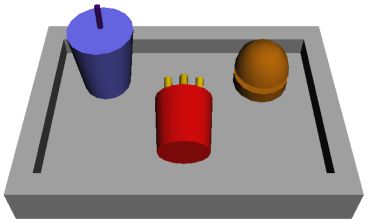
Chicken burger with drink



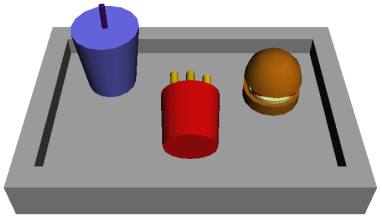
Complete meat menu



Complete veggy menu



Complete chicken menu



```
begin
        set this load type = L_menu

        // GENERATE THE ORDERED MENU
        set V_menu_pointer = oneof(33:P_menu(1),33:P_menu(2),33:P_menu(3))
        set V_chips = oneof(25:0,75:1)
        set V_drink = oneof(10:0,90:1)

        set V_load_type_ptrs(1) = L_bread

        // SET THE GRAPHIC PARAMETERS
        if V_menu_pointer = P_menu(1) then
                begin
                        set V_menu_type = "meat"
```

```
                set V_load_type_ptrs(2) = L_meat_ing_1
                set V_load_type_ptrs(3) = L_meat_closed

                if V_chips = 0 and V_drink=0 then
                        begin
                                set V_load_type_ptrs(4) = L_meat_closed
                                set V_load_type_ptrs(5) = L_meat_closed
                        end
                if V_chips = 1 and V_drink=0 then
                        begin
                                set V_load_type_ptrs(4) = L_meat_chips
                                set V_load_type_ptrs(5) = L_meat_chips
                        end
                else if V_chips = 0 and V_drink=1 then
                        begin
                                set V_load_type_ptrs(4) = L_meat_closed
                                set V_load_type_ptrs(5) = L_meat_drink
                        end
                else
                        begin
                                set V_load_type_ptrs(4) = L_meat_chips
                                set V_load_type_ptrs(5) = L_meat_complete
                        end
        end
else if V_menu_pointer = P_menu(2) then
        begin
                set V_menu_type = "veggy"
                set V_load_type_ptrs(2) = L_veggy_ing_1
                set V_load_type_ptrs(3) = L_veggy_closed
                if V_chips = 0 and V_drink=0 then
                        begin
                                set V_load_type_ptrs(4) = L_veggy_closed
                                set V_load_type_ptrs(5) = L_veggy_closed
                        end
                else if V_chips = 1 and V_drink=0 then
                        begin
                                set V_load_type_ptrs(4) = L_veggy_chips
                                set V_load_type_ptrs(5) = L_veggy_chips
                        end
                else if V_chips = 0 and V_drink=1 then
                        begin
                                set V_load_type_ptrs(4) = L_veggy_closed
                                set V_load_type_ptrs(5) = L_veggy_drink
                        end
                else
                        begin
                                set V_load_type_ptrs(4) = L_veggy_chips
                                set V_load_type_ptrs(5) = L_veggy_complete
                        end
        end
else
        begin
```

```
                    set V_menu_type = "chicken"
                    set V_load_type_ptrs(2) = L_chicken_ing_1
                    set V_load_type_ptrs(3) = L_chicken_closed
                    if V_chips = 0 and V_drink=0 then
                            begin
                                    set V_load_type_ptrs(4) = L_chicken_closed
                                    set V_load_type_ptrs(5) = L_chicken_closed
                            end
                    else if V_chips = 1 and V_drink=0 then
                            begin
                                    set V_load_type_ptrs(4) = L_chicken_chips
                                    set V_load_type_ptrs(5) = L_chicken_chips
                            end
                    else if V_chips = 0 and V_drink=1 then
                            begin
                                    set V_load_type_ptrs(4) = L_chicken_closed
                                    set V_load_type_ptrs(5) = L_chicken_drink
                            end
                    else
                            begin
                                    set V_load_type_ptrs(4) = L_chicken_chips
                                    set V_load_type_ptrs(5) = L_chicken_complete
                                    end
            end

        // CALCULATE THE MENU PRICE
        if V_menu_pointer = P_menu(1) then set V_menu_value = 7
        if V_menu_pointer = P_menu(2) then set V_menu_value = 5
        if V_menu_pointer = P_menu(3) then set V_menu_value = 6

        if V_chips = 1 then set V_menu_value=V_menu_value + 3
        if V_drink = 1 then set V_menu_value=V_menu_value + 2.5

        open V_file_menu_list_name for appending save result as V_file_menu_list_ptr
                print V_order_id, ",", V_menu_type, ",", V_chips, ",", V_drink, ",",
                      V_menu_value, ",", ac to V_file_menu_list_ptr
        close V_file_menu_list_ptr

        send to V_menu_pointer
end
```

P_menu (Dim:3; Default Traffic Limit: Infinite)

The following process models the central part of the production, i.e. the menu preparation, starting from the bread up to the delivery points. Based on the meal preparation phases the code can be divided as follows:

1) Station 1: bread
   The load is moved into the first workstation of the conveyor, *sta_bread,* where the counter of bread-stock and its respective label are decremented by 1. This happens only if there is still bread available at the station, otherwise, a loop would stop the process until the stock is refilled.
   Successively, the operator places the bread on the tray in a time modelled with a gaussian distribution. As the last step, the graphic of the load is updated.

2) Station 2, 3 and 4: main ingredient
   The load is moved to the next station. This is the one responsible for the main ingredient of the burger. *P_menu* has dimension 3, therefore in this section of code (and also in the next one) it is possible to exploit *procindex* to avoid code duplication:
   2(procindex=1)* refers to the first ingredient of the meat burger
   1+2(procindex=2)* refers to the first ingredient of the vegetarian burger
   1+2(procindex=3)* refers to the first ingredient of the chicken burger
   The steps are similar to all the other stations: once the supply quantity has been checked, the operator executes its activity (in a time modelled with a normal distribution), the stock quantity and its respective label are updated, the graphics are changed and the load moves on to the next station
3) Station 5, 6 and 7: secondary ingredients
   This section is analogous to the previous one: *procindex* is exploited to obtain a cleaner code and the sequence of steps is the same. Once the remaining ingredient quantity, the label and the graphics are updated the trays coming from 3 different conveyor segments merge into one section.
4) Station 8: chips
   If the processed menu contains chips (*V_chips == 1*), after a check on the local stock, they are added by the operator. Counter, label and graphics are updated and the load moves forward
5) Station 9: drink
   Similarly to the previous station, if the menu contains a drink (*V_drink == 1*) the operator adds it. Once the counter, label and graphics are changed, the load is ready for the last phase.
6) Delivery
   The menu is sent toward the correct location: if the load variable *V_order_provenience* has value *"car"*, then the load moves towards the conveyor station *sta_car,* otherwise it will move towards *sta_room*. Depending on the destination, they are also placed inside 2 different order lists (*OL_take car* or *OL_take_person*) which are used in the processes *P_car* and *P_person* to group the correct menus and deliver them.

In addition to the passages described above, the process is also in charge of saving information about the status of the menus and the performance of the various workstations.

```
begin
   set V_menu_counter = V_menu_counter + 1
   set V_menu_processing_time = ac

   // ADD BREAD
   move into conv.sta_bread
   set V_station_proc_time = ac
   if C_stock(1) current = 0  then
      begin
         while C_stock(1) current = 0 do
            begin
               wait for 2 sec
            end
            set V_stock_downtime = V_stock_downtime +(ac-V_station_proc_time)/60
         end
      dec C_stock(1) by 1

      // -- LB_bread_qty: change of  label --
      print C_stock(1) current, "/", C_stock(1) capacity to LB_stock(1)
      if C_stock(1) current < C_stock(1) capacity / 10 then
```

```
      begin
         set LB_stock(1) color to red
      end
   else if C_stock(1) current < C_stock(1) capacity * 3 / 10 then
      begin
         set LB_stock(1) color to yellow
      end
   // -- LB_bread_qty: label changed --

use R_op_bread for normal V_sta_oper_time_avg(1),V_sta_oper_time_avg(1)/5 sec
set V_station_proc_time = ac - V_station_proc_time
set V_sta_proc_time_ratio(1) = V_station_proc_time / V_sta_oper_time_avg(1)
tabulate V_sta_proc_time_ratio(1) in T_sta_proc_time_ratio(1)

set this load type = V_load_type_ptrs(1)

// ADD FIRST INGREDIENT
set V_station_proc_time = ac
travel to conv.sta_first_ing_(procindex)
if C_stock(2*procindex) current = 0  then
   begin
      while C_stock(2*procindex) current = 0 do
         begin
            wait for 2 sec
         end
      set V_stock_downtime = V_stock_downtime +(ac-V_station_proc_time)/60
   end
dec C_stock(2*procindex) by 1

// -- LB_stock(2*procindex): change of  label --
print C_stock(2*procindex) current, "/",C_stock(2*procindex) capacity to
      LB_stock(2*procindex)
if C_stock(2*procindex) current < C_stock(2*procindex) capacity / 10 then
   begin
      set LB_stock(2*procindex) color to red
   end
else if C_stock(2*procindex) current < C_stock(2*procindex) capacity * 3 / 10 then
   begin
      set LB_stock(2*procindex) color to yellow
   end
// -- LB_first_ing_qty(2*procindex): label changed --

use R_op_first(procindex) for normal
      V_sta_oper_time_avg(2*procindex),V_sta_oper_time_avg(2*procindex)/5 sec
set V_station_proc_time = ac - V_station_proc_time
set V_sta_proc_time_ratio(2*procindex)=V_station_proc_time/V_sta_oper_time_avg(2*procindex)
tabulate V_sta_proc_time_ratio(2*procindex) in T_sta_proc_time_ratio(2*procindex)

set this load type = V_load_type_ptrs(2)


// ADD SECOND INGREDIENT
```

```
set V_station_proc_time = ac
travel to conv.sta_second_ing_(procindex)
if C_stock(1+2*procindex) current = 0 then
   begin
      while C_stock(1+2*procindex) current = 0 do
         begin
            wait for 2 sec
         end

         set V_stock_downtime = V_stock_downtime +(ac-V_station_proc_time)/60
   end
dec C_stock(1+2*procindex) by 1

// -- LB_stock(1+2*procindex): change of  label --
print C_stock(1+2*procindex) current, "/", C_stock(1+2*procindex) capacity to
      LB_stock(1+2*procindex)
if C_stock(1+2*procindex) current < C_stock(1+2*procindex) capacity / 10 then
   begin
      set LB_stock(1+2*procindex) color to red
   end
else if C_stock(1+2*procindex) current < C_stock(1+2*procindex) capacity * 3 / 10 then
   begin
      set LB_stock(1+2*procindex) color to yellow
   end
// -- LB_stock(1+2*procindex): label changed --

use R_op_second(procindex) for normal
      V_sta_oper_time_avg(1+2*procindex),V_sta_oper_time_avg(1+2*procindex)/5 sec
set V_station_proc_time = ac - V_station_proc_time
set V_sta_proc_time_ratio(1+2*procindex)=V_station_proc_time/V_sta_oper_time_avg(1+2*procindex)
tabulate V_sta_proc_time_ratio(1+2*procindex) in T_sta_proc_time_ratio(1+2*procindex)

set this load type = V_load_type_ptrs(3)

// ADD CHIPS
if V_chips = 1 then
   begin
      set V_station_proc_time = ac
         travel to conv.sta_chips
         if C_stock(8) current = 0 then
            begin
               while C_stock(8) current = 0 do
                  begin
                     wait for 2 sec
                  end

                  set V_stock_downtime = V_stock_downtime +(ac-V_station_proc_time)/60
               end
               dec C_stock(8) by 1

               // -- LB_stock(8): change of  label --
               print C_stock(8) current, "/", C_stock(8) capacity to LB_stock(8)
```

```
                if C_stock(8) current < C_stock(8) capacity / 10 then
                   begin
                      set LB_stock(8) color to  red
                   end
                else if C_stock(8) current < C_stock(8) capacity * 3 / 10 then
                   begin
                      set LB_stock(8) color to yellow
                   end
                // -- LB_stock(8): label changed --

                use R_op_chips for normal V_sta_oper_time_avg(8),V_sta_oper_time_avg(8)/5 sec

                set V_station_proc_time = ac - V_station_proc_time
                set V_sta_proc_time_ratio(8) = V_station_proc_time / V_sta_oper_time_avg(8)
                tabulate V_sta_proc_time_ratio(8) in T_sta_proc_time_ratio(8)

                set this load type = V_load_type_ptrs(4)
   end

// ADD DRINKS
if V_drink=1 then
   begin
      set V_station_proc_time = ac
      travel to conv.sta_drink
      if C_stock(9) current = 0 then
         begin
            while C_stock(9) current = 0 do
               begin
                  wait for 2 sec
               end

               set V_stock_downtime = V_stock_downtime +(ac-V_station_proc_time)/60
         end
      dec C_stock(9) by 1

      // -- LB_stock(9): change of  label --
      print C_stock(9) current, "/", C_stock(9) capacity to LB_stock(9)
      if C_stock(9) current < C_stock(9) capacity / 10 then
         begin
            set LB_stock(9) color to red
         end
            else if C_stock(9) current < C_stock(9) capacity * 3 / 10 then
         begin
            set LB_stock(9) color to yellow
         end
      // -- LB_stock(9): label changed --

      use R_op_drink for normal V_sta_oper_time_avg(9),V_sta_oper_time_avg(9)/5 sec

      set V_station_proc_time = ac - V_station_proc_time
      set V_sta_proc_time_ratio(9) = V_station_proc_time / V_sta_oper_time_avg(9)
      tabulate V_sta_proc_time_ratio(9) in T_sta_proc_time_ratio(9)
```

```
        set this load type =  V_load_type_ptrs(5)
      end

   // MENU COMPLETED, GET PROCESSING TIME
   set V_menu_processing_time = (ac - V_menu_processing_time)/60

   // SEND TOWARDS THE CORRECT DIRECTION
   if V_order_provenience = "car" then
      begin
         // print "travelling to car"
         travel to conv.sta_car
         move into Q_take_car

         // final setting to hand in the order
         // take out the order from the conveyor and put it in the "bag" for the delivery
         use R_op_take_car for 5 sec
         wait to be ordered on OL_take_car
      end
   else
      begin
         // print "travelling to room"
         travel to conv.sta_room

         // take out the order from the conveyor
         // final setting to hand in the order will be performed by the waiter
         // the time to prepare the final order is modelled with the pick-up time
         move into Q_take_person
         set priority to V_pers_counter
         wait to be ordered on OL_take_person
      end
end
```

P_order_room (Dim:1; Default Traffic Limit: Infinite)

This process receives one load from the process *P_person* and converts it in *L_order_room* that would represent the entire order which is picked up by the waiter and brought to the dining room. As the order reaches the room, the corresponding customer (the one with the same order ID) is removed from the waiting order list and the *L_order_room* is sent to die. At this point the customer will continue to follow the instructions specified in *P_person.*

```
begin
       set this load type = L_order_room
       move into waiter.w_room_in
       rotate this current container vehicle absolute to z 0
       travel to waiter.w_room_out
       order 1 loads satisfying V_order_id=this load V_order_id from OL_waiting_person to continue
       rotate this current container vehicle absolute to z 180
       send to die
end
```

3. Simulation Results

After the first simulation and results analysis, we discovered that some assumptions and aspects that we fixed were causing inefficiency in the menu preparation and delivery. In particular, the time spent by waiting for the meals was too high for a fast food restaurant. Thus, we decided to improve the model and iterate the simulations.

In the following section we list the detected inefficiencies and explain the improvements, also comparing the situation before and after the upgrades. Finally, we present the results obtained with the final version of the model and some possible further improvements.

3. 1 Applied improvements

1) Restocking management

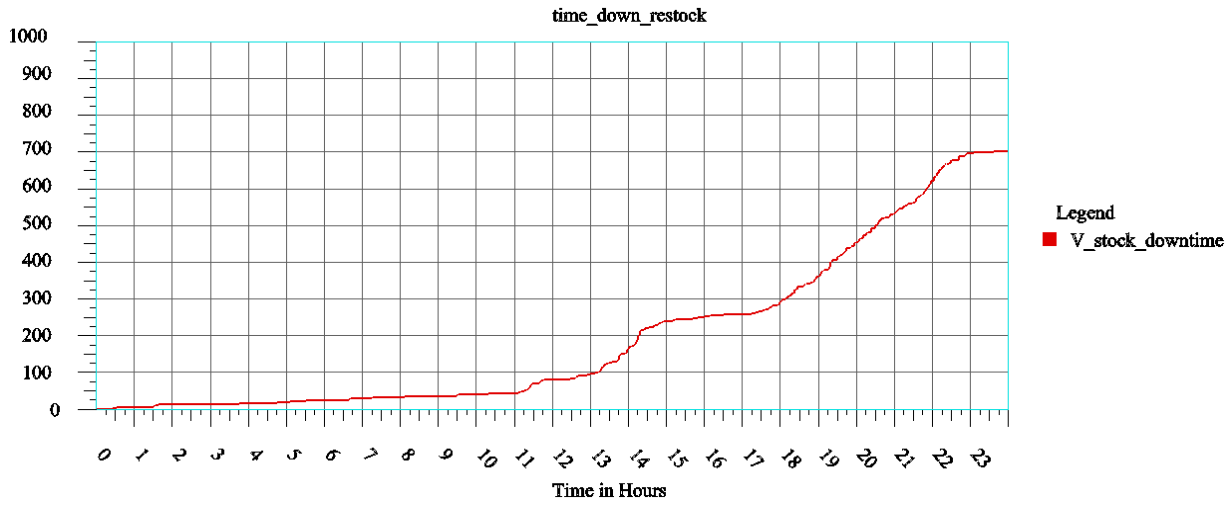
   a) Restocking policy

The first improvement we applied concerns the ingredients' buffer replenishment management. In the first model we developed, we started by considering relatively low buffer sizes and we defined a restock policy we called "0%-policy". With this policy, the restocker is activated for a specific ingredient only when the buffer of the related station is completely empty.

Studying the running simulation, we immediately realised that the menus were waiting too long in the conveyor due to the lack of ingredients in each station. We decided to measure this time and to plot it on a Business Graphic. We called this time "restock downtime", as in practice it is a workers blocking time caused by a bad management of the restocking procedure. Note that this time does not correspond to the time in which the workers are idle, but it is the time in which the menus are blocked on the conveyor without being elaborated because of the lack of ingredients. It is a wasted time that infers both on the quality of the final product (the more the menu waits in each station, the colder it gets) and on the average waiting time of the customers (as shown later).

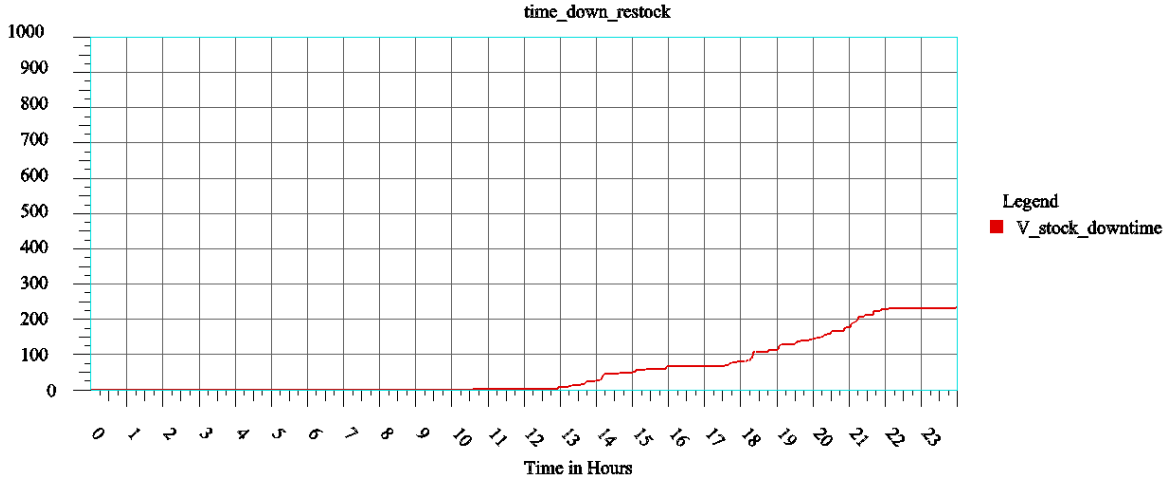
What follows is the Business Graphic we got by running the model: the total amount of downtime at the end of the day is about 704 minutes (i.e. more than 11 hours).

Restock downtime with little buffers and 0%-policy



To enhance the situation, the more logical improvement was to change the 0%-policy with a 20%-policy, in which the restocker is called for a specific ingredient once its buffer reaches 20% of its capacity. Running the simulation, such a policy immediately more than doubled the performances, reducing the downtime to 232 minutes.

Restock downtime with little buffers and 10%-policy



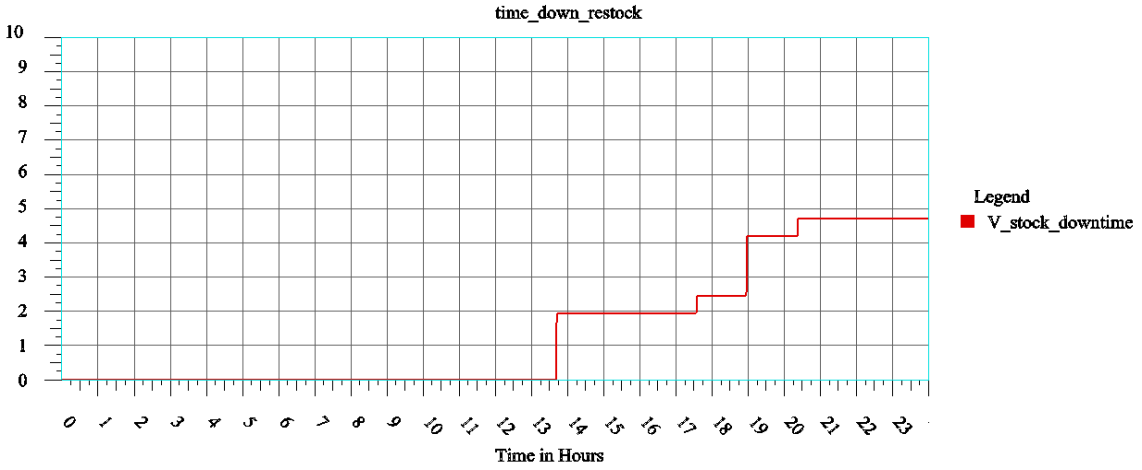
b) Ingredients buffers' size

Even if the improvement was meaningful, there were still about 4 hours of downtime. We thought and confirmed that a further improvement to reduce the same problem was to double the buffer sizes of the stations. The following table shows the old and the new sizes adopted:

conditions\station	1	2	3	4	5	6	7	8	9
previous	20	15	20	20	20	15	20	20	25
improved	40	30	40	40	40	30	40	40	50

To model this choice as realistically as possible, we also slowed down the restocker by a 20% factor (i.e. we changed the "Default Vehicle Specification Velocity" from 0.8 to 0.64 metres per second). The increase of the quantity of ingredients in fact is possible by using a trolley to help in the transport, but this slows down the movement of the operator. The result of the simulation can be seen in the following Business Graphic:

Restock downtime with big buffers, 20%-policy and slowed restocker



The improvement is self-evident: the total downtime is now lower than 10 minutes and, indeed, it was possible to reduce the magnitude of the y-axis by two orders. All the data contained in the previous Business Graphic can be consulted in a data sheet available via the link[2].

c) System behaviour with both improvements

Since we were studying the consequences of the restocking policy, we found it appropriate to carry out further analysis concerning other aspects of the system, in order to see its behaviour in relation to our choices. In particular, we evaluated the impact of the improvements on the workload of the restocker and on the time waited by the customers.

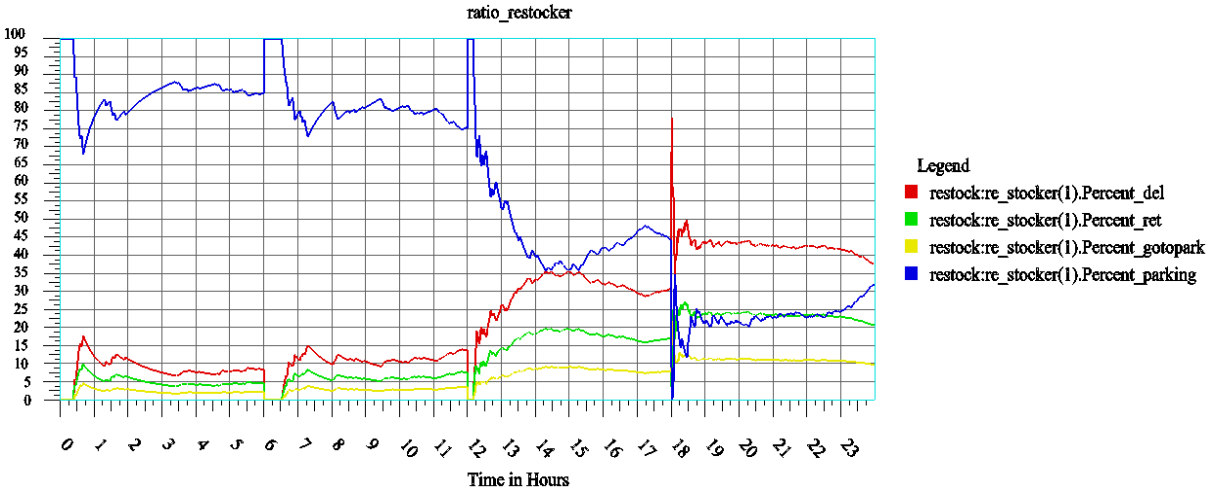
To measure the restoker working time, we considered the time in which he/she is outside the warehouse delivering ingredients (i.e. for Automod, the time the restocker is not in the parking area). It is worth noticing that, thinking about a real situation, even the moments in which the restocker is inside the warehouse can actually be working time and not idle time. For the sake of simplicity, we assumed that the warehouse has infinite stock and is always in order, therefore the working time corresponds to the time spent outside the parking area. In a real scenario, the workers of the first two shifts may not be those with the main task of bringing the ingredients to the kitchen, but they may be in charge of tidying up the warehouse and replenishing the ingredients from an external supplier.

From the results of the simulations (see next page), it can be seen that the restocker is never fully loaded: the most critical situation happens when running the model with small buffers and the adoption of the 20%-policy, in which the workload in the fourth shift (the one with the greatest number of customers) exceeds 80% for most of the time. We believed that this value could be critical considering the other activities that the restocker should carry out in a real environment, but we obtained that this problem is solved in the final version of the model, where the workload in the fourth shift is reduced to about 50%. This last value also justifies our choice of utilising just one restocker per shift, without the need of employing others.

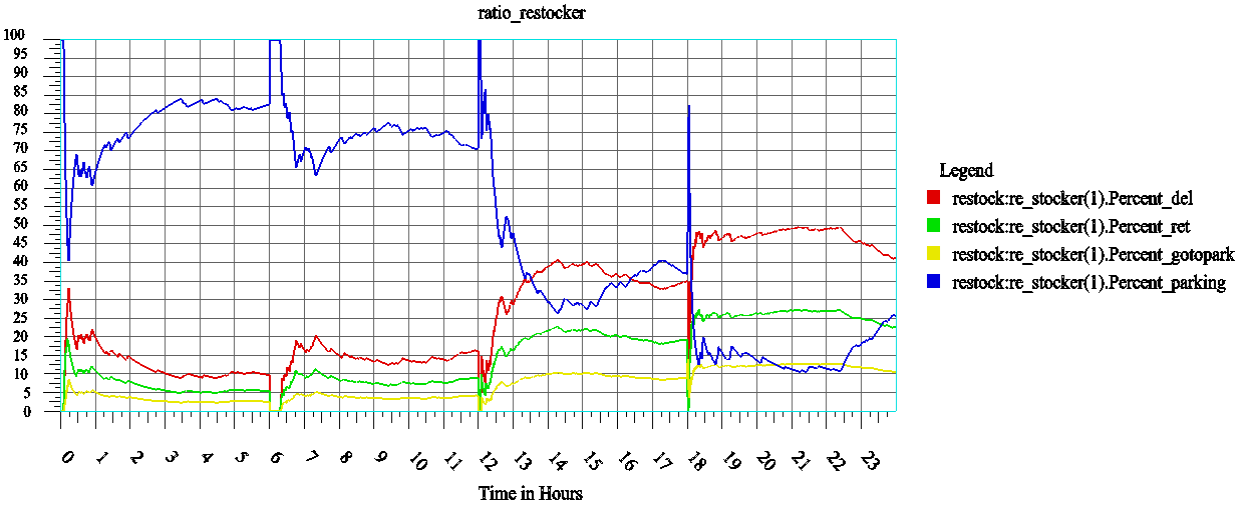
The data taken into account for these considerations are represented in the charts reported in the following page.

[2]https://docs.google.com/spreadsheets/d/e/2PACX-1vS3ketCFxOd1IDw64JyAjjTUzYR_YNIHddQOlXLyKa9Yuotudm8cA_nmM_KmfjuUj5T3zpyk9yoYR-l/pubhtml?gid=0&single=true

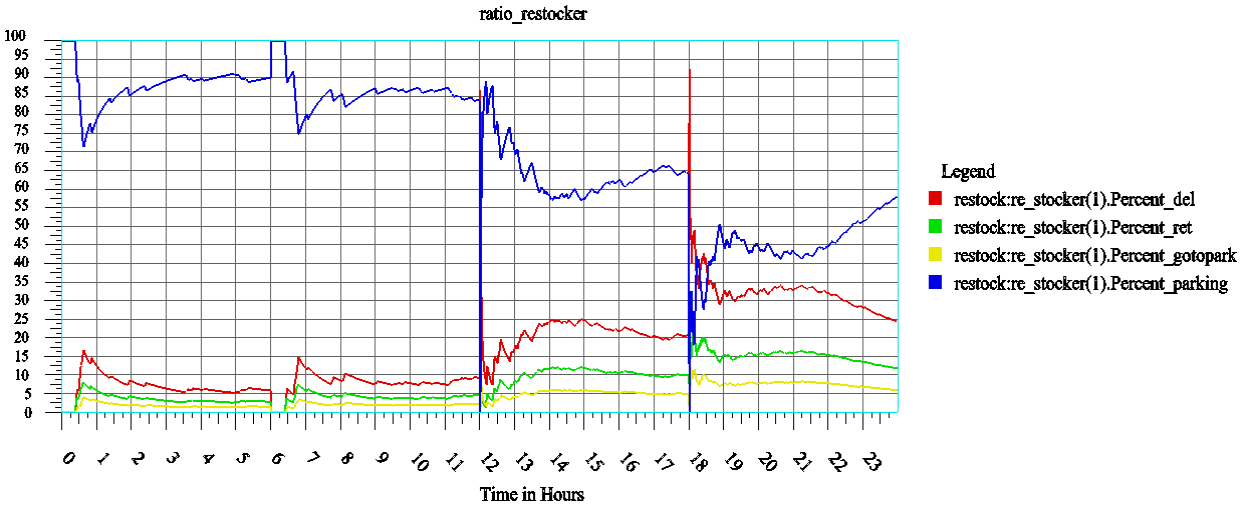
Restocker tasks percentage divided in shifts in the case of little buffers and 0%-policy



Restocker tasks percentage divided in shifts in the case of little buffers and 20%-policy



Restocker tasks percentage divided in shifts in the case of big buffers, 20%-policy and slowed resocker

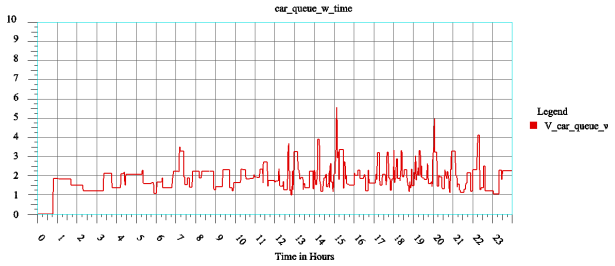


The time waited by the customers to get the meal is one of the main KPI for a fast food restaurant. Its value is influenced by different aspects of the system that can be grouped in: the ordering policy, the meals' preparation procedure and the delivery system.

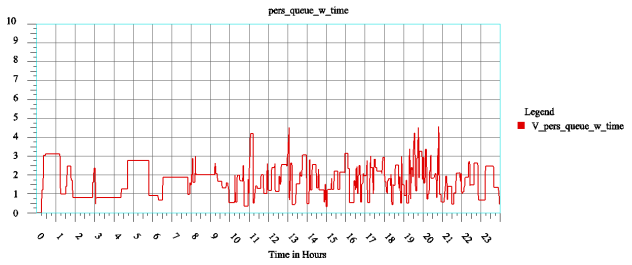
For our model, we could state that the part influencing the most the time waited by the customers was the menu preparation. Indeed, based on how the system is organised, it is easy to understand that the delivery phase is quick: the cars' orders are immediately handed over when they are complete and, similarly, the dining room's orders are quickly delivered by the waiter as soon as they are ready. Moreover, we registered the time waited by the customers from their arrival to the placement of the order and the results (two charts below) show that most of the clients spent only a time between 1 and 5 minutes queuing to make their order. We consider this behaviour totally acceptable, also because the trend does not show any peak in the hours of greater affluence.

Time waited once entered in the system before making the order

for customers coming by car



for customers coming on foot



These considerations allowed us to proceed with the study of the waiting time by focusing only on the time taken by the menus to be prepared. This time is thus meaningful for both the KPI under study and of the kitchen performance in terms of the capacity to accomplish the task quickly.

For completeness, we should mention that the time needed for the preparation of each single menu does not correspond exactly to the one waited by the customers, since they have to wait for the whole order to be completed. Nonetheless, the two variables are strictly correlated and the fact that all the menus of an order are created in sequence provides more evidence to the choice of considering the menu preparation time only, without considering the whole dimension of the order.

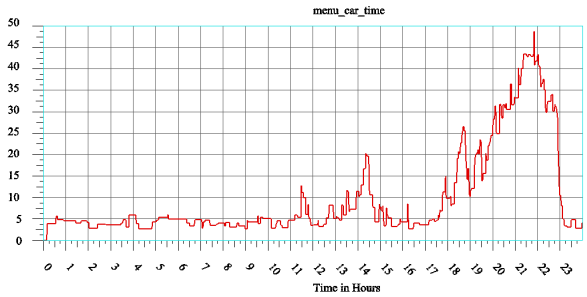
In the next page, we report the plots of the preparation times for the menus in the different scenarios described up to now (with or without the improvements about restocking). It can be seen that, before the upgrades, some menus had to wait up to 45 minutes to be completed. This time was obviously unacceptable for the type of service we wanted to achieve. The most critical moment was, as expected, the dinner time, in which the system is subject to the higher arrival rate. We also expected higher peaks during the lunch time, but for this case we can say that the system was able to absorb the lunch demand quite well (compared to the dinner time), probably because of a smoother change in the number of arrivals (refer to the graph of the arrivals schedule).

The important aspects to notice in the charts are the time trends and value of the peaks. For better visualisation and more in-depth study the same data used are available at the following links: link

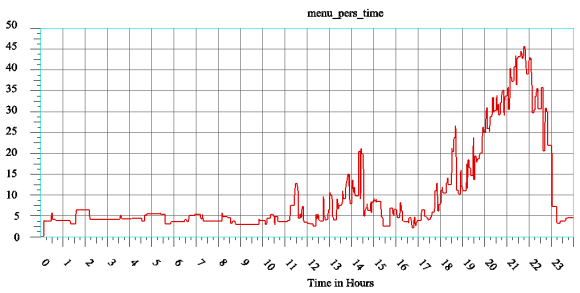
for car menus data[3], link for walking menus data[4].

Time to deliver the menus with little buffers and 0%-policy

for customers coming by car

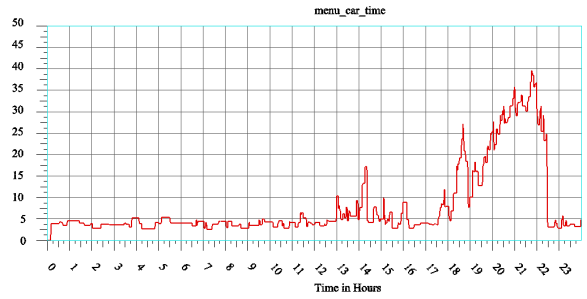


for customers coming on foot

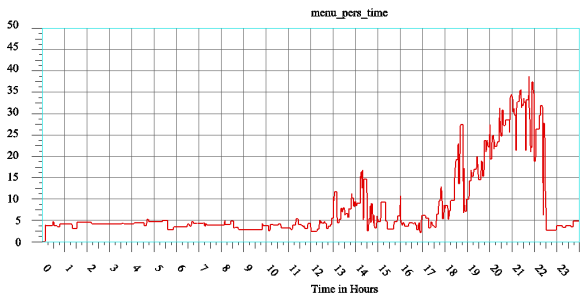


Time to deliver the menus with little buffers and 20%-policy

for customers coming by car

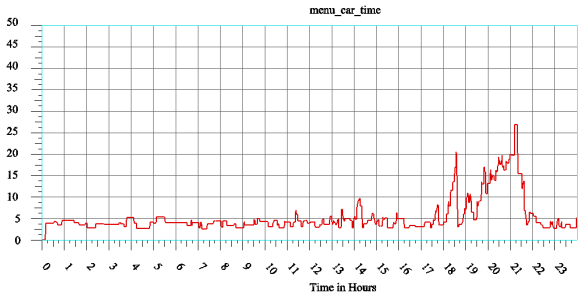


for customers coming on foot

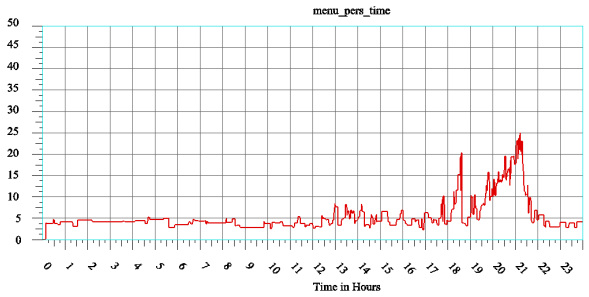


Time to deliver the menus with little buffers and 20%-policy and slowed restocker

For customers coming by car



for customers coming on foot



Despite the significant progress, the preparation time in the final scenario still exceeds 25 minutes for some menus. This time is again too long to resemble the one of a drive-in, both in terms of menu preparation and time waited by the customers to get their meal. To better tackle this problem, we upgraded the model with a second main improvement described hereafter.

[3]https://docs.google.com/spreadsheets/d/1E2FK2SZrLkDwBWfD2Y4uTdvC0ITWT5Dv-d_478QmidM/edit#gid=1907011131

[4]https://docs.google.com/spreadsheets/d/1E2FK2SZrLkDwBWfD2Y4uTdvC0ITWT5Dv-d_478QmidM/edit#gid=457031480

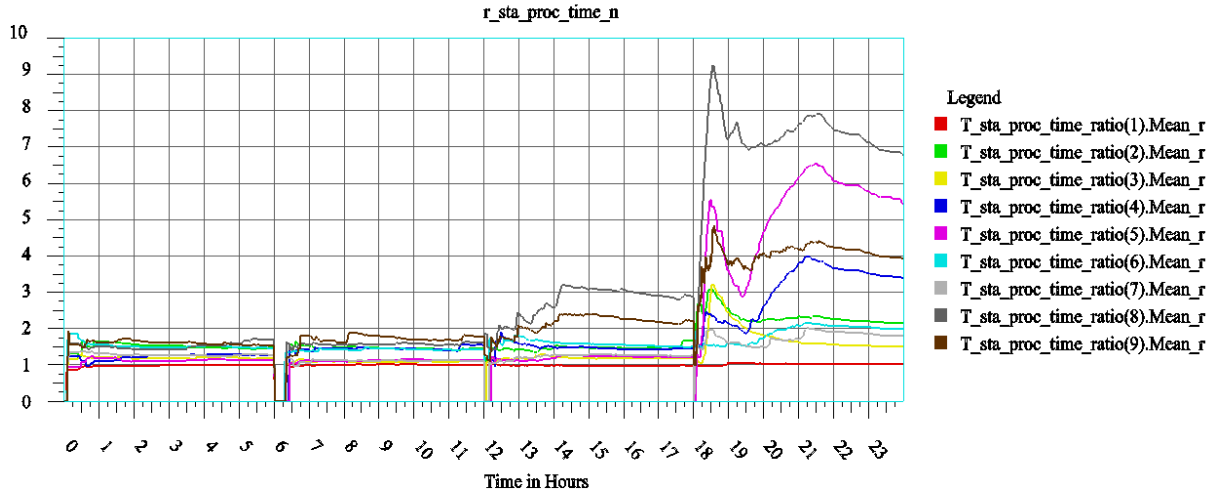
2) Improvements station 8 and station 9 (chips and drink stations)

a) Station 8, fryer upgrade

As anticipated at the end of the previous section, in some cases the preparation time for the menus reaches peaks that are too high for a real world drive-in as the one we wanted to model.

To identify the cause of this behaviour, we analysed an index we called “relative production rate” of every station (*T_sta_proc_time_ratio* variable in the code). First, we computed the processing time at each station for every menu as the time elapsed between the end of the previous task (previous preparation step) and the end of the current task. We then defined the relative production rate index as the ratio between the measured processing time and the theoretical processing time (the mean of the process time, chosen in the *P_init* process). In practice, it corresponds to an efficiency measure that in an ideal production process has to be approximately one, so as to ensure a constant flow of products. In the following chart it is possible to see the plot of the index for the different stations averaged throughout the four shifts. During our studies, we also plotted the chart containing the value of the index itself without averaging throughout the shifts, but it turned out to be unreadable and the punctual samples were not providing a clear understanding of the situation.

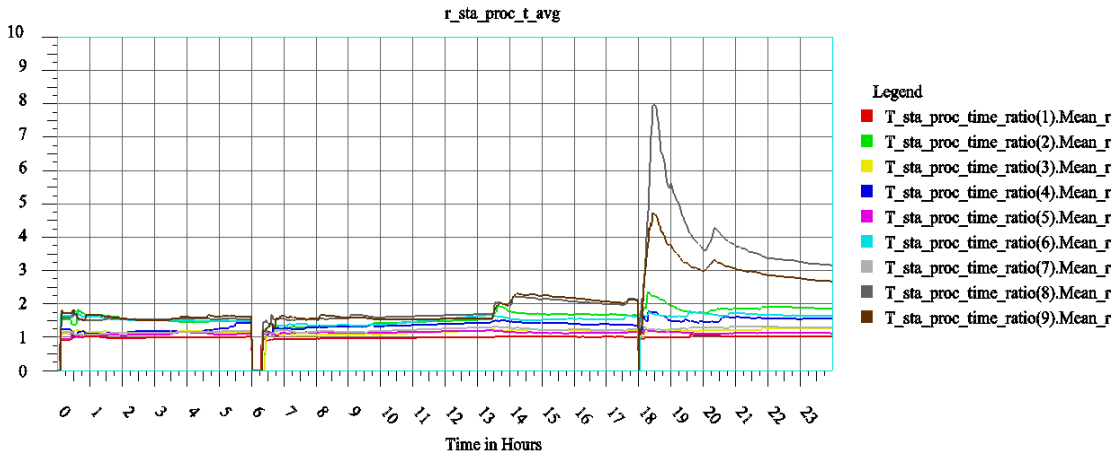
Relative production rate of the different stations before improvement



As expected, before midday, the situation is almost stable: all the rates are below the value 2. The situation worsens from midday onwards and the absolute worst performances are those of station 8 (the one responsible for the chips) during the fourth shift, followed by station 5 and station 9. Just for clarification, the ordinate axis represents the times the theoretical processing time has to be multiplied to obtain the measured one (represented in the plot). So we see that for station 8 the processing time has a peak of about 9,15 times more than the theoretical value.

The solution we thought was to increase the performance of station 8. In a real environment, this can be done with a new, higher-performance fryer or by hiring a new employee to work in parallel with the one already present. In accordance with this assumption, we halved the average time requested by the station to complete the task, reducing it to 15 seconds. Here is reported the relative production rate index chart in the new case:

Relative production rate of the different stations after upgrading station 8



From the chart, it can be said that the improvement has alleviated the problem, but it has not completely solved it. Station 8 and station 9 still generate critical values for the index. Between 6 p.m. and 7 p.m. station 8 takes up to eight times longer than it would if it were working ideally.

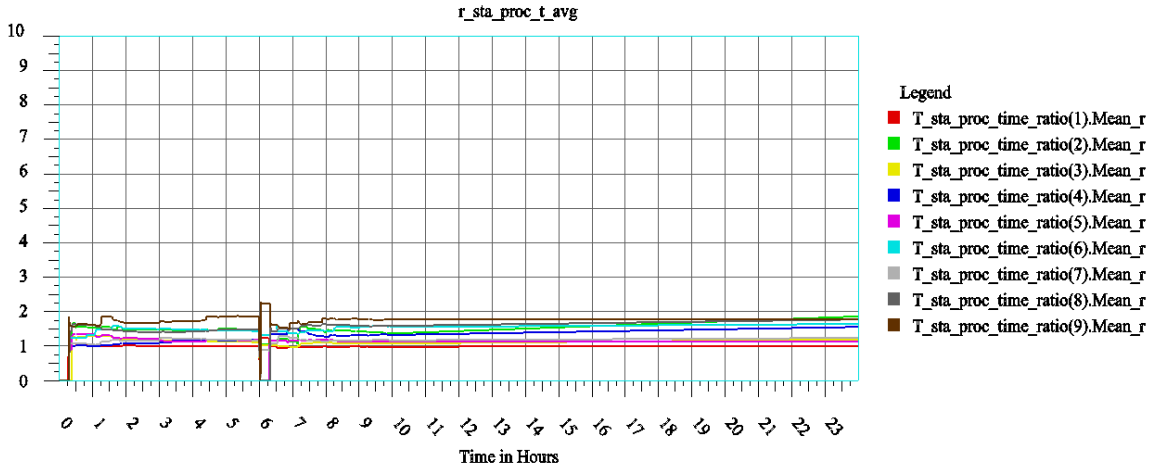
Looking at a running simulation it was clear that the problem was due to low performances of station 9 causing the blocking of station 8 and thus the longer processing time at that station. By definition, a station is blocked when the subsequent buffer is full. In this case, there are no buffers within the system, but the space available on the conveyor plays the same role: if the conveyor after a station is full, the station can not work.

b) Station 9, automatic drink dispenser

To unlock station 8 and enhance the situation, we modelled another improvement similar to the previous one, but applied to station 9. We reduced its average time from 20 to 8 seconds. This improvement can be achieved in reality with an automatic, computerised drink dispenser. As shown in the Business Graphic on the following page, the result is marvellous as all ratios assume an acceptable value.

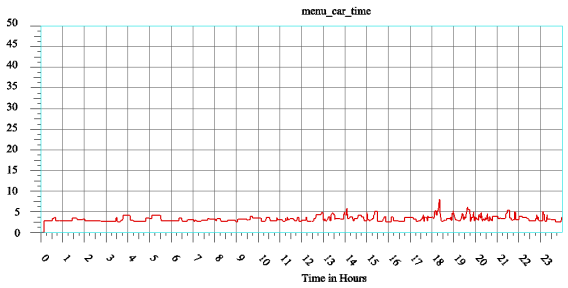
The reduction of inefficiency and blocking time gained with the new machines also improved the time needed by the chain to prepare the menus. We finished the previous chapter saying that there were still peaks that were too high for a drive-in. With the new improvements, also these times reach acceptable values. Again, the data considered to say this are summarised in the two Business Charts on the following page and are collected in the same data sheet linked before.

Relative production rate of the different stations with the upgrades of station 8 and station 9

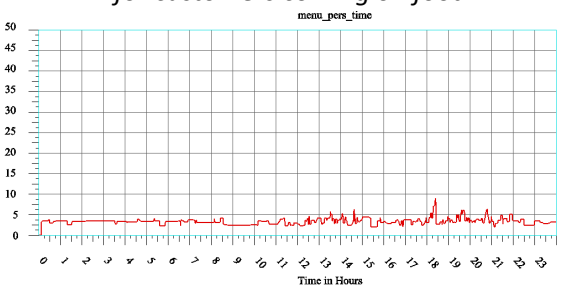


Time to deliver the menus with the upgraded stations

For customers coming by car



for customers coming on foot



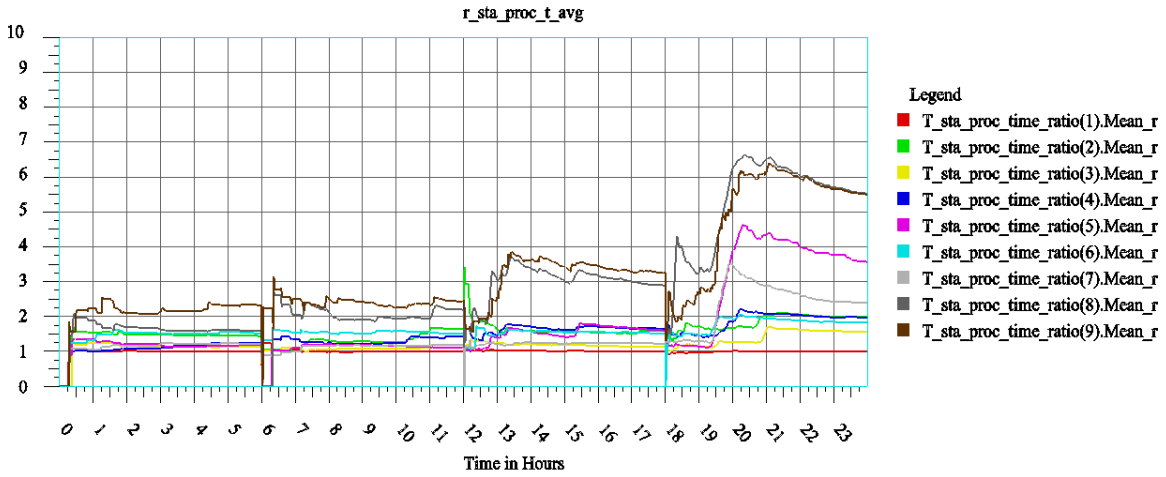
We verified that, as expected, this second upgrade did not affect neither the downtime nor the working time of the restocker discussed in the first round of improvements. To avoid being repetitive, the business graphs are not reported.

We also wanted to ensure that the path taken up to this point was the best: as seen before, we first decreased the processing time of station 8 and only then we improved station 9. To verify that the only chance to reach such performances was to improve both stations, we also studied the case in which only the processing time of the station dedicated to drinks (station 9) is decreased. The situation is reported in the chart on the next page.

Also looking at the real time simulation it can be seen that without the upgrade on station 8, the ninth is affected by starvation: in the most critical moments of the day (particularly at dinner time) the drink dispenser remains idle for the majority of the time, waiting for the previous station to process its item. The result is a mass of trays queued along the conveyor. This fenomena is so intense that it causes the blocking of stations 5, 6 and 7 which have finished their activity, but can't move to the next tray because the stretch of conveyor that leads to station 8 is jammed. This can be seen at around 20:00 when the relative production rate of these stations increases dramatically.

This proves that stations 8 and 9 represent a bottleneck for the system and the improvement of both is indeed needed.

Relative production rate of the different stations after upgrading station 9



3. 2 Results

In this section we report the results obtained with the final version of the model. As stated in the introduction, the objective of the project was to create a model of the restaurant to help in making accurate decisions for a real implementation of the process. We aimed at investigating, through simulations, the proper setting of some organisational aspects and parameters such as the stations' buffer stocks size, the restocking dynamics, the need of the replacement of human operators with machines and the design of a suitable capacity of the dining room in order to obtain a reasonable preparation time and to analyse the number of people to accommodate in a typical scenario.

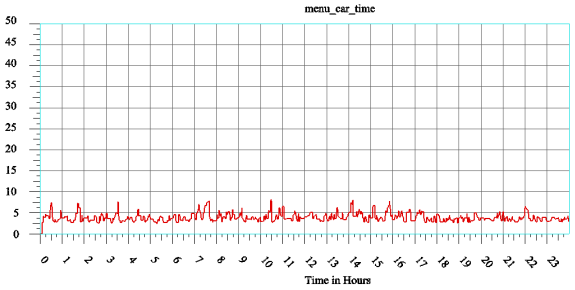
With the improvements described in the previous sections we were able to set the buffers' size, decide for the replacement of the drink operator with an automatic dispenser and the upgrade of the chips-making process and finally we defined a new restocking policy. All these upgrades made us succeed in the reduction of the preparation time and, therefore, of the time waiting by the customers, to an acceptable value.

Of course, the data used for the discussion in the previous chapter represents only an instance of a possible scenario. Indeed, running again the Python script, the input file would change slightly. Nonetheless, the overall distribution is the same for all instantiations and it represents a typical arrival scheduling of a real fast food.

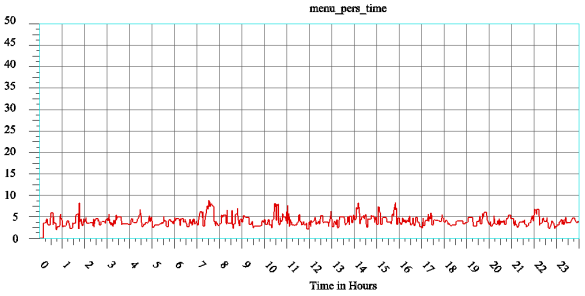
Thanks to the simulation, we were able to demonstrate that the system can manage the critical peaks of demand at dinner time. With the considered distribution of the arrivals and the hypothesis of 500 orders, the highest arrival rate obtained was 47 orders per hour. We won't go through all the results obtained in the reduction of the preparation time as all the material was already reported in previous sections, but to furtherly challenge the capabilities of our system, we ran a stress test in which we kept the highest arrival rate (47) constant during the day. The following Business Graph shows that the time waited by the customer is always under 10 minutes. This proves the capability of our model to manage high volume affluence for prolonged periods of time.

Time to deliver the menus for the final model with constant and high arrival rate

For customers coming by car



for customers coming on foot



To gain more insights about the potential of the model, we collected and analysed data about the single menus requested to the system during a day with three different instances of the input, all characterised by the typical arrival rate. This data are available at the usual link[5]. We obtained that the average number of single menus requested is about 1500 and that the average income per day is 15760 euros, calculated with the prices of the products defined in the process *P_init*. Together with a cost analysis, this data can be used to draw up a balance sheet and a financial statement for the activity. Obviously, this is outside the scope of the project and won't be furtherly discussed here.

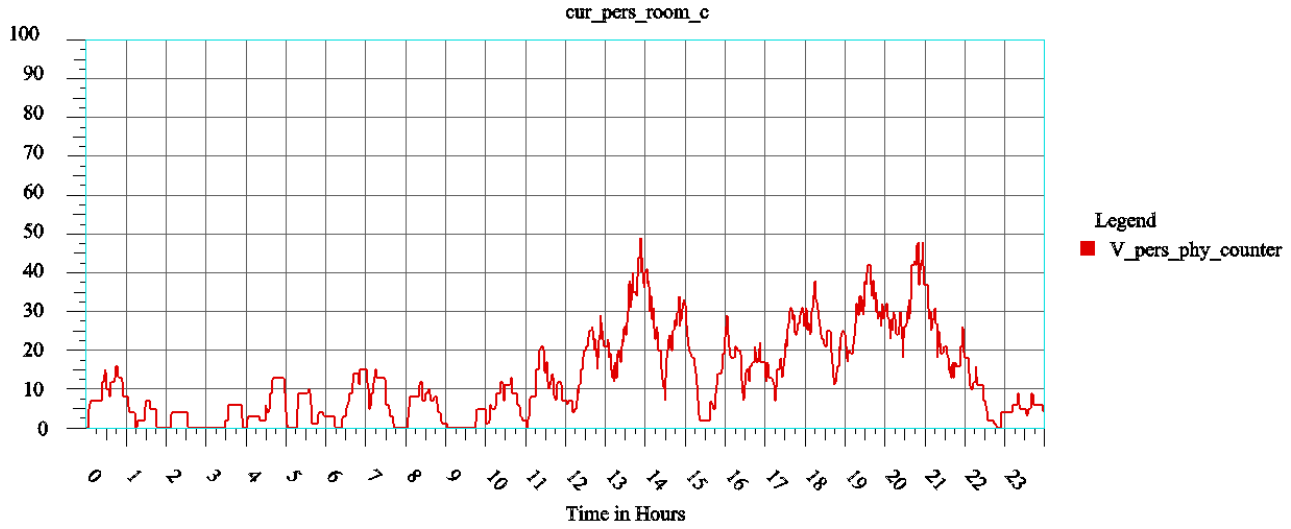
Under the design assumptions we have adopted, customers coming on foot will stop to eat in the dining room for an average time of 20 minutes (as defined in *P_init*). In the simulation, the room is modelled as a queue of infinite size, so as to not limit the affluence of customers and track the number of people eating inside the restaurant. This data is helpful in designing the real capacity of the dining room either in the case of renting or building it to avoid making people wait too long for a place to seat.

The determination of the most suitable capacity of the room is a trade-off problem between the size of the room itself in terms of physical space (therefore building/renting and running costs) and maximum number of clients served (therefore revenues). Missing the knowledge on this kind of problem in which the management of real restaurants is highly involved, we decided to only give insights that can be crucial for more educated decisions.

That said, here we propose the simple result in which we can accomodate all people coming on foot considering our simulation outcomes. From the data collected, it appears that, in order to avoid generating waits for access to the room, it would be appropriate to buy or rent a room where at least 50 customers can eat. In real life this result may be appropriately scaled with risk and utilisation factors to obtain a more robust design.

[5]https://docs.google.com/spreadsheets/d/1E2FK2SZrLkDwBWfD2Y4uTdvC0ITWT5Dv-d_478QmidM/edit#gid=715949125

Number of physical people in the dining room at the given time



Since, on average, a bar must have about 1.2 square metres of space for each person and a restaurant about 1.5, we believe that the value of 1.4 square metres could be a good compromise. With this hypothesis, for the realisation of the project, it would be necessary to find a room whose size is at least 75 square metres for eating seats. In the same building there should of course also the space for the kitchen, for the corridors and other functional areas.

Again, those just reported are simple ideas that would need further development to be completed, but we still wanted to mention them to show that with the created model and the simulation conducted it is possible to go really in depth with the considerations.

3. 3 Further improvements

In this section we report some of the potential further improvements and considerations that could be added to the model, but were not fully developed due to time constraints, software limitations or because they are outside the scope of the course.

The brainstormed ideas are the following:

- New layout

  The whole project aimed at studying and optimising performances of a restaurant, maintaining a fixed layout. A valuable alternative could be to study different station dispositions and analyse how customer waiting time and other crucial factors change depending on the chosen layout.

  The main problem with the current disposition lies in the fact that the hot portions of the menu (burgers and fries) have to wait for the cold drinks to be prepared before being delivered to the client. It would make more sense to move the drink station in front of the production line, before the bread, in order to serve freshly made burgers. The downside would be tray handling: moving a tray with a beverage on top is a much more delicate action. If properly managed this would add value to the product and ultimately influence positively the client.

- More options for the client

  At the moment the model supports only 3 types of burger while some fast food chains such as McDonald's propose up to 145 different items. A crucial improvement would be to further develop

each station giving more flexibility to the operator in order to produce multiple types of burger and therefore making the overall simulation more truthful and accurate. This could be enclosed together with many more slight modifications such as the option for the client to take away the order, or a specific queue for delivery riders (which typically have priority over the normal customer) and more.

- Basic Economic analysis and optimization

  As we have proposed in this project, given the choice of the customer and the affluence throughout the day, it is possible to have an idea of the expected revenue. The model could be further exploited to analyse and reduce costs related to production and maintenance of the restaurant. An idea could be to monitor the utilisation rate and idle time of every operator in order to manage the tradeoff between number of workers and wages (therefore between production and costs). Stock management, rent and utilities could be other factors to consider to have a more complete economic report.

4. Conclusions

In our scenario, we were able to model a simplified version of an actual drive-in restaurant. Despite our simplifications, we could design a layout that showed the interactions between a lot of entities. The simulation process allowed us to spot the problems of our first basic assumptions and to draw improvements that led to a better functioning of the overall system. Indeed, as shown in this report, we were able to refine the management of the process in order to obtain the desired behaviour considering a realistic input. The obtained improved model still shows simplifications, nonetheless it offers crucial insights and a solid starting point to further develop more realistic mechanics.
Thanks to the project we could appreciate the power of simulation used as a tool for helping in design and decision making.