Simulation Group Project

SYSC 4005 - Discrete Simulation / Modeling
Deliverable 1

Authors
Connor Judd, 101109256
Marko Majkic, 101109409
Millan Wang, 101114457

Due : 23h00 February 10th 2022

1 - Problem Formulation

The problem to be addressed by this simulation describes a manufacturing company attempting to find performance metrics for a product manufacturing system with hopes to find data that will inspire methodologies that maximize production rates and minimize asset downtime. The manufacturing process creates 3 products which shall be referred to as P1, P2, and P3. The products are created on corresponding workstations referred to as W1, W2, W3 with components referred to as C1, C2, and C2.

The product composition is as follows:

- P1 is composed of one C1
- P2 is composed of one C1 and one C2
- P3 is composed of one C1 and one C3

Creating products involves two inspectors, referred to as I1 and I2, that inspect, clean, and repair components before they are sent to workstation buffers. The supply of components is infinite, obtaining a component to inspect happens instantaneously, however, it takes a period of time to complete an inspection, and this time period will follow a specified distribution.

Inspector I1 is responsible for inspecting component C1, and inspector I2 is responsible for inspecting components C2 and C3. For now, inspector I2 will randomly choose to select either C2 or C3 for inspection without any consideration for other production conditions.

Once an inspection is complete, the components are sent to one of the three workstations W1, W2, and W3, which each assemble products P1, P2, P3 respectively. Each workstation has a buffer with a capacity of two components for each of the components it requires to assemble its respective product and removes its needed components from the buffers to begin assembly. A workstation can only assemble one product at a time, and assembly may begin as soon as all of its component buffers have a component to take. Each workstation will have assembly times that will follow a specified workstation-specific distribution.

When an inspector completes an inspection, the inspector will try to place the component on the workstation buffer with the freest space for that component type. If all buffers have an equal amount of free space, the buffer corresponding to the lowest workstation number will be chosen.

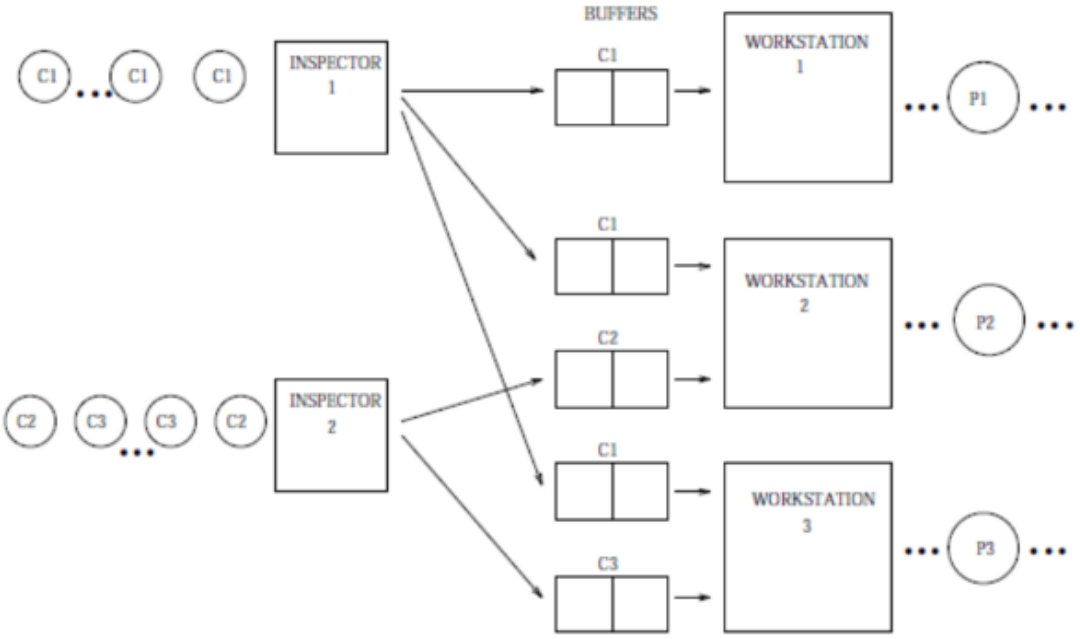

Figure 1: Schematic illustration of the manufacturing facility.

Inspectors start inspections without considering the buffers, and if an inspection completes when all buffers are full, the inspector will be blocked and wait until the first available buffer opening appears. The inspector is required to inspect components in the order that they appear in the queue, and cannot alter the order.

The company is interested in determining the throughput of products created over time, the proportion of time that each workstation is assembling compared to time spent waiting for buffers to populate, the average buffer occupancy for each buffer, and the time each inspector spends waiting for buffers to create space. The company would also like to evaluate the priority policy for inspector 1 when routing C1 to the three workstations, to maximize throughput, minimize the idle time of the inspector, and to evaluate incorporating a system for inspector I2 to determine which component to inspect next.

2 - Setting of Objectives

The main question that this simulation will address is to determine what would be the optimal way to arrange the inspectors to maximize the rate of product creation and minimize the rate that the inspectors are blocked. A simulation is the best way to address this question as it would be very difficult to find an analytical solution to this problem and it would be easier, and thus less resource-intensive, to approximate a solution with simulation.

The alternative systems to explore during the simulation would be to assess the impact of changing the methodology in which the inspector I2 selects which component to inspect next as well as which workstation buffers their inspected products will be sent to.

The completion of this project will involve the collaboration of 3 group members Connor Judd, Marko Majkic, and Millan Wang, and is expected to require a time budget of around 2 hours per week per member. No financial resources are currently expected to be needed to complete this project. The first deliverable deadline is on February 10th, and it will be assumed that the subsequent three deliverable deadlines will be approximately biweekly across the remaining 6.5 weeks of the 2nd half of the term (February 28th - April 13th). These deliverable deadlines are subject to change at the professor's discretion. Below is the plan on the deliverables for each deliverable deadline, and more detailed plans for deliverables 2-4 will be updated when they become a work in progress.

Deliverable 1 - February 10th 2022
- -Report Iteration 1
  - -Project Formulation
  - -Settings of Objectives & Overall Project Plan
  - -Model Conceptualization
  - -Model Translation
  - -UML Class and sequence diagrams for code implementation
- -Code iteration 1
  - -Skeleton code that models the simulation.
  - -Instructions on how to run and read outputted reports

Deliverable 2 - Tentatively March 10th 2022
- -Data Collection & Input modeling
- -Detailed description of generating inputs based on identified model

Deliverable 3 - Tentatively March 24th 2022
- -Model verification & validation
- -Production runs and statistical analysis

Deliverable 4 - Tentatively April 7th 2022
- -Discussion of alternative operating policies
- -Conclusion
- -Final report
- -Complete simulation source code

3 - Model Conceptualization

This product manufacturing process will be modeled using simulation software. Inspectors, components, buffers, workstations, and products will not be real and will instead be represented in a python simulation. The simulation will use object-oriented programming to model these different entities.

To simulate the process of component inspection and product assembly, delay generators will be used and the generated delay values will emerge from process-specific time distributions based on historical data. Each component and workstation has a corresponding .dat file containing historical timing data, and these files will be used to create the distributions needed to generate the delay values.

For inspector I1, only C1 components will be inspected, and upon completion, a process for selecting the workstation buffer to send the component to will be run. This process will try to select the buffer with the most empty space. If multiple buffers are tied for having the most empty space, then the workstation with the lowest workstation number will be selected. If all buffers are full, I1 will wait and then send the component to the first buffer that opens a buffer spot. After sending an inspected component to a buffer, I1 will start to inspect another C1 component.

For inspector I2, the random selection of components C2 & C3 for inspection will be modeled using random number generation. Since I2 does not currently have any choice on which component to inspect, I2 will also have no choice on which workstation to attempt to send the inspected component to. If the buffer corresponding to the component that I2 just inspected is full, I2 will wait until that buffer opens a spot before refilling the buffer with the held component and then randomly selecting a C2||C3 to inspect again.

Workstations will begin assembly when all of the workstation buffers have at least one component. Starting assembly will begin by removing components from the buffers and then generating an assembly time for completion. Once the completion time is reached, the assembly process will wait until all the buffers are populated again before starting another assembly. This wait time may be zero if the buffers are already populated.

All of these processes will be initiated in a main Simulation class that will track the timings of events, translate these timings into metrics of interest, and then process statistics from these metrics accordingly.

4 - Model Translation

Python was chosen as a language of simulation as all members of the group are familiar with it and it was determined that the time investment required to become fluent with dedicated simulation languages would not justify the potential increases in performance given the current time budgets. It was also determined that increasing the group members' fluency with Python's data analysis libraries would be more professionally marketable than learning dedicated simulation languages.

SimulationLogger.py

The **`SimulationLogger`** class will serve to allow progress messages to be logged at different stages of the simulation. The simulation logger includes three Boolean flag variables that are set to either true or false and allow the object to know if logging is enabled for each respective aspect of the simulation. The first flag is the

__enable_inspector_logging flag that indicates if logging for the inspector actions is enabled. The second flag is the **__enable_workstation_**logging flag that indicates if logging for the workstation actions is enabled. Finally, there is the **__create_log_files** flag that indicates whether the simulation updates should be logged to a file (the use for this will be further explained below).

The **SimulationLogger** includes 4 functions and a constructor. The constructor takes as parameters the Boolean values for **enable_inspector_logging**, **enable_workstation_logging**, and **create_log_files** and initializes the class's flags with those values. The first function **log_inspector_component_selection** takes a component and time as parameters, and logs at which time this component has started selection. The second function, **log_inspector_buffered_component**, takes a component, workstation, and time as parameters and logs at which time the component has been put in the workstation's buffer. The third function, **log_workstation_unbuffer**, takes a workstation,and time as parameters, and logs at which time the workstation starts to build it's assigned product. Finally, the **log_product_created** function takes a product and time as parameters and logs at which time the product has been produced.

Each function first checks that its respective enable flag is not set to false, and then constructs the string and prints it. If the flag is set to false, it will do nothing.
In the future, the logging will be implemented using the python Logging library which will allow messages to be easily logged to a file in an organized manner, instead of to the console. The **__create_log_files** flag, while currently unused, will serve as a flag that will toggle the file logging feature on and off.

SimulationEnums.py

The **SimulationEnums** file contains three classes that implement enumerations. The first class is the **Product** enumeration which contains the three possible products P1, P2, P3, and their respective ID's. The second class is the **Component** enumeration which contains the three possible components C1, C2, and C3, and their respective ID's. The third class is the **Event_Types** enumeration which includes all the four possible events that occur in the simulation which are **Inspection_Complete** which is when an inspector completes their inspection, **Add_to_Buffer** which is when the inspector adds a component to a buffer, **Unbufffer_Start_Assembly** which is when a component is removed from a buffer and placed in the workstation for assembly, and **Assembly_Complete** which is when a product has been fully assembled.

Buffers.py

The **Buffer.py** file contains two classes – the **Component_Buffer** class and the **Component_Buffer_Manager** class. The **Component_Buffer** implements the buffer object itself where components are stored and waiting to be pushed into the workstation. The **Component_Buffer** has an **attempt_to_add_buffer** function which attempts to add a component to the buffer object, and returns true if successful, and false otherwise. It implements this by simply incrementing the **component_count** variable of the buffer object. The **Component_Buffer** also includes a function called **remove_from_buffer** which removes components from the buffer and throws an error if the buffer is empty. It also implements this by decrementing the **component_count** variable.

The **Component_Buffer_Manager** class acts as a controller for all the buffers. It routes components to the appropriate buffer and pushes components out of the buffer into the workstations. The **Component_Buffer_Manager** has two functions and a constructor. The constructor initializes all the **Component_Buffer** objects needed with their associated component product pairs. The **attempt_to_add_to_buffer** function takes in a Component as a parameter and attempts to find a buffer to send the component to. It returns True and the Product if successful, and False and None otherwise. It implements this by checking the type of component that was passed to it and then checking if it can add that component to one of the buffers that is associated with that component. The **attempt_to_assemble_product** method takes a product as a parameter and takes the needed components off the workstation buffer to assemble the product. It returns true if the needed components were provided from the buffers, false otherwise. It implements this by checking the product type and then checks if a buffer associated to that product can provide one of the components.

Inspectors.py

The **Inspector** File contains two classes, one for each inspector. Each inspector class will be responsible for calculating the delay time based on the imputed data file for the corresponding component. In **Inspector1** there is an **__init__** method to initialize the class with the mean of component 1. After this is the **__build_distribution_from_dat** method that will find the mean from the imputed data set.The code will locate the data files then work through it sumating the contents. It uses the sum to calculate and return the mean. The last method of **Inspector1** is **generate_inspect_time** and this uses the mean to generate a random delay within a distribution of the mean. This uses numpy.random.exponential with the mean as an input. **Inspector2** is very similar to **Inspector1. Inspector2** also has the **__init__** file to initialize but in **Inspector2** it sets the mean for C2 and C3. The following method **__generate_comp2or3** is used to randomly select between C2 and C3 for the next part to inspect. This uses random.getbits to create the random input. **__build_distribution_from_dat** is the next method and will find the mean from the imputed data set and component. The final method is **generate_inspect_time** and this will output the time delay depending on what component is sent in.

MainSim.py

The **MainSim.py** file is the main file that contains the **Simulation** class and the main script that facilitates running the simulation. The simulation is run using a single future event list and will continue to address and schedule events until the product creation limit is reached. The constructor method **__init__** for the **Simulation** class takes a logger object and sets up the needed properties to run the simulation. The **get_next_chronological_event** method identifies the closest chronological event in the future event list and returns & removes it to continue the simulation. This is implemented by iterating through the future event list and then comparing each event start time with the previously tracked minimum. The **schedule_add_to_buffer** method is given an inspector number, and adds an event to the future event list for when the inspection of the current component will complete. The calculation of this inspection time is done with the corresponding inspectors inspection time generation mechanism. The **process_add_to_buffer** method processes an attempt to add

an inspected component to a buffer. If no buffers are available to receive this component nothing happens. If a buffer can accept the component, then the construction of a product will be immediately scheduled by adding a corresponding event to the future event list. The **`process_workstation_unbuffer`** method deals with the previously mentioned workstation construction initiation events and will try to retrieve the needed components from the buffer. If successful, a new event will be added to the future event list for the completion of the product assembly. The **`process_product_made`** method deals with the arrival of the completed assembly events and increases the simulation's product counters accordingly. This then adds an event to the future event list to make the workstation indicating that it is to start constructing a new product.

Workstation.py

The **`Workstation.py`** file contains a single class **`Workstation`** whose purpose is to receive components and calculate a delay based on the imputed distribution of data. The Class is set up with the **`__init__`** file to initialize the mean of the production times as well as the product. The Next method is **`__build_distrubution_from_dat`** which will see what product is being worked on and select the corresponding datafile and calculate the mean to be used for random number generation. The final method is **`get_unbuffer_time`** and this method is used to generate a product assembly delay time from the mean.