**yazı tipi, simge, sembol, logo, amblem içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**REPUBLIC OF TURKEY**

**KADİR HAS UNIVERSITY**

**FACULTY OF ENGINEERING AND NATURAL SCIENCES**

**CMPE412 - Computer Simulation**

**Project 2**

**Manufacturing System**

Esin Can Bozyurt

20191701058

Assoc. Prof. Dr. Doğan Çörüş

2023- 2024 Spring

# ABSTRACT

This paper describes the design and application of a discrete event simulation for increasing the system throughput, for the identification and resolution of constraints in a manufacturing system, and for investigating the effects of operational parameters. Further, the simulation is generalized for the multiple product type environment, reflecting the extra complications and resource management issues. This simulation is implemented in C++ and uses event scheduling, time advancement, and state changes to represent the production process.

# INTRODUCTION

The objective of this project is to develop a discrete-event simulation to optimize the throughput of a manufacturing system. The simulation focuses on:

* Optimizing throughput for a single product line.
* Identifying and mitigating production bottlenecks.
* Analyzing the impacts of operational variables through scenario analysis.
* Extending the simulation to handle multiple product types and analyzing the complexity and resource allocation challenges.

# SYSTEM MODELING

The manufacturing system consists of several stages: raw material handling, machining, assembly, quality control, and packaging .Each stage is represented as a process which has its working time and setting time for different products. The machines may also fail or require servicing and repair at some point of time.

# DATA REQUIREMENTS

* Process and Setup Times: Estimated but real figures for every machine and operation that may be used in the production process.
* Raw Material Inputs: Timing of inputs is determined.
* Failure Rates and Maintenance Times: Mean rates of failure and maintenance time for the machines.
* Shift Patterns and Worker Allocations: The duration of a shift and the number of shifts are inputs that are provided by the user.

# SIMULATION IMPLEMENTATION

## 5.1 Event Class

The Event class is a fundamental building block of the simulation, representing discrete events that occur at specific times. Each event has a time at which it is scheduled to occur and an action to be performed.

* time: This member variable stores the time at which the event is scheduled to occur.
* action: This is a std::function<void()> that stores the action to be performed when the event occurs. This allows for flexible and reusable code, as any callable object (function, lambda, etc.) can be assigned as an action.
* Constructor: The constructor initializes the time and action of the event.
* operator>: This operator is overloaded to allow the Event objects to be compared based on their time, enabling the use of a priority queue for event scheduling.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **class** **Event** {  public:  double time;  std::function<void()> action;  Event(double time, std::function<void()> action) : time(time), action(action) {}  bool operator>(const Event& other) const {  **return** time > other.time;  }  }; |

## 5.2. Simulation Class

The Simulation class manages the progression of time and the execution of events. It maintains a priority queue of events, processes them in the order of their scheduled times, and updates the current simulation time.

* currentTime: This member variable keeps track of the current time within the simulation.
* eventQueue: A priority queue (std::priority\_queue<Event, std::vector<Event>, std::greater<Event>>) that stores events. Events are ordered by their scheduled time, with the earliest event at the top of the queue.
* scheduleEvent: This method schedules a new event by adding it to the event queue.
* run: This method runs the simulation by processing events from the queue. The simulation continues until the event queue is empty.
* getCurrentTime: This method returns the current simulation time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | **class** **Simulation** {  private:  **double** currentTime;  std::priority\_queue<Event, std::vector<Event>, std::greater<Event>> eventQueue;  public:  Simulation() : currentTime(0.0) {}  **void** scheduleEvent(**double** time, std::function<**void**()> action) {  eventQueue.push(Event(time, action));  }  **void** run() {  **while** (!eventQueue.empty()) {  Event nextEvent = eventQueue.top();  eventQueue.pop();  currentTime = nextEvent.time;  nextEvent.action();  }  }  **double** getCurrentTime() **const** {  **return** currentTime;  }  }; |

## 5.3.Machine Class

The Machine class represents a machine in the manufacturing system. Each machine can process different product types with specific processing and setup times. The class also handles machine breakdowns and maintenance.

* name: The name of the machine.
* simulation: A reference to the Simulation instance, allowing the machine to schedule events.
* processTimes: A map that stores the processing times for different product types.
* setupTimes: A map that stores the setup times for different product types.
* onProcessComplete: A callback function that is called when the machine finishes processing a product.
* breakdownProbability: The probability that the machine will break down during processing.
* maintenanceTime: The time required for maintenance if the machine breaks down.
* generator and distribution: Used to generate random numbers for simulating breakdowns.
* isBusy: Indicates whether the machine is currently processing a product.

Methods:

startProcessing: Starts processing a product if the machine is not busy and there is enough time left in the shift. It schedules the next event for either a breakdown or the completion of the processing.

isAvailable: Returns whether the machine is currently busy or not.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | **class** **Machine** {  private:  std::string name;  Simulation& simulation;  std::unordered\_map<std::string, **double**> processTimes;  std::unordered\_map<std::string, **double**> setupTimes;  std::function<**void**(**const** std::string&)> onProcessComplete;  **double** breakdownProbability;  **double** maintenanceTime;  std::default\_random\_engine generator;  std::uniform\_real\_distribution<**double**> distribution;  **bool** isBusy;  public:  Machine(std::string name, Simulation& simulation, std::unordered\_map<std::string, **double**> processTimes, std::unordered\_map<std::string, **double**> setupTimes, std::function<**void**(**const** std::string&)> onProcessComplete,  **double** breakdownProbability = 0.0, **double** maintenanceTime = 0.0)  : name(name), simulation(simulation), processTimes(processTimes), setupTimes(setupTimes), onProcessComplete(onProcessComplete),  breakdownProbability(breakdownProbability), maintenanceTime(maintenanceTime), distribution(0.0, 1.0), isBusy(false) {}  **void** startProcessing(**const** std::string& productType, **double** shiftEndTime) {  **if** (isBusy || simulation.getCurrentTime() + processTimes[productType] > shiftEndTime) **return**;  isBusy = true;  std::cout << *"Machine "* << name << *" started processing "* << productType << *" at time "* << std::fixed << std::setprecision(2) << simulation.getCurrentTime() << std::endl;  **double** totalTime = processTimes[productType] + setupTimes[productType];  **if** (distribution(generator) < breakdownProbability) {  std::cout << *"Machine "* << name << *" broke down! Maintenance required."* << std::endl;  simulation.scheduleEvent(simulation.getCurrentTime() + maintenanceTime, [**this**, productType, shiftEndTime]() { **this**->startProcessing(productType, shiftEndTime); });  }  **else** {  simulation.scheduleEvent(simulation.getCurrentTime() + totalTime, [**this**, productType]() {  isBusy = false;  std::cout << *"Machine "* << name << *" finished processing "* << productType << *" at time "* << std::fixed << std::setprecision(2) << simulation.getCurrentTime() << std::endl;  onProcessComplete(productType);  }); |

## 5.4 Manufacturing System Class

The ManufacturingSystem class coordinates the entire manufacturing process, managing multiple machines and shifts. It handles the scheduling of shifts, the flow of products through different stages, and the completion of products.

Methods:

* startShift: Initiates a new shift, schedules its end, and starts production.
* endShift: Ends the current shift, schedules the next shift if there are more shifts, or prints the results if all shifts are completed.
* startProduction: Begins processing a new product from the queue if the machine is available.
* startMachining, startAssembly, startQualityControl, startPackaging: Methods to move products to the next stage of processing.
* finishProduct: Increments the products completed counter and continues production if the shift is not over.

# 6. Experimentation and Analysis

The simulation is run with different shift durations and number of shifts to establish baseline performance. Key variables are adjusted to observe effects on throughput and identify bottlenecks.

## 6.1. Baseline Scenario

* Shift Duration: 16 hours
* Number of Shifts: 16

Results:

* Total Products Completed: 9
* Total Simulation Time: 256 hours

## 6.2. Scenario Analysis

By varying the shift duration and number of shifts, the impacts on production throughput and efficiency are analyzed. The simulation helps identify the optimal configuration for maximizing throughput and minimizing bottlenecks.

# 7. CONCLUSION

The discrete-event simulation developed in this project successfully models a manufacturing system, optimizing throughput, identifying bottlenecks, and analyzing the impact of operational variables. The extension to handle multiple product types demonstrates the additional complexity and resource allocation challenges. This simulation serves as a valuable tool for decision-making in manufacturing operations.