

Simulation of Tandem and Re-Entrant Manufacturing Lines

by

Christina C. Royce

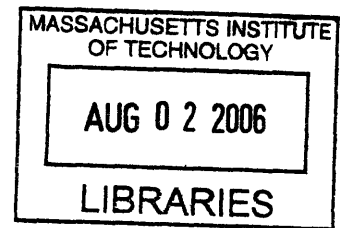
SUBMITTED TO THE DEPARTMENT OF MECHANICAL
ENGINEERING IN PARTIAL COMPLETION OF THE
REQUIREMENTS FOR THE DEGREE OF

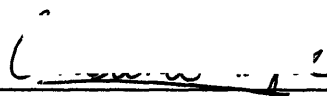
BACHELOR OF SCIENCE IN MECHANICAL ENGINEERING
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

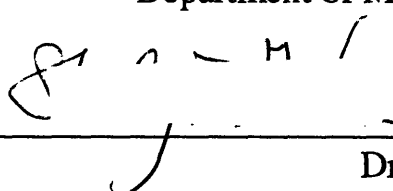
JUNE 2006

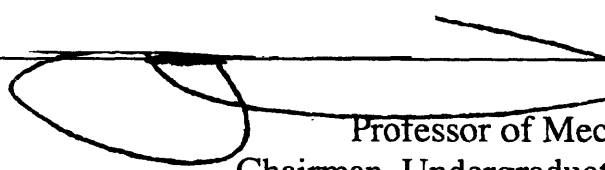
©2006 Christina C. Royce. All rights reserved.

The author hereby grants to MIT the ability to reproduce
and to distribute publicly in paper or electronic
copies of this thesis document in whole or in part
in any medium now known or hereafter created.



Signature of Author: 
Department of Mechanical Engineering
May 12, 2006

Certified by: 
Dr. Stanley B. Gershwin
Senior Research Scientist
Department of Mechanical Engineering
Thesis Supervisor

Accepted by: 
John H. Lienhard V
Professor of Mechanical Engineering
Chairman, Undergraduate Thesis Committee

ARCHIVES

Simulation of Tandem and Re-Entrant Manufacturing Lines

by

Christina C. Royce

Submitted to the Department of Mechanical Engineering
on May 12, 2006 in Partial Fulfillment of the
Requirements for the Degree of Bachelor of Science in
Mechanical Engineering

ABSTRACT

Modeling manufacturing systems is a necessary tool in the process of finding a way to analyze and improve design. Increasingly complex systems are now being modeled, and two such systems are the focus of this report. The Tandem and Re-Entrant systems allow for multiple part types to be sent through a single line of processing machines. The parts have different priorities which determine the order in which they are produced. The Re-Entrant system is unique because it produces a single part that is processed through the same machine line multiple times. As the part travels through the processing line, it loops back to the beginning at the end of every run as a higher priority part. These simulations were tested for their validity by running with different input parameters to see how the system reacted. These programs can be used in the future with more complex systems and the knowledge gained from the results of these simulations can be applied to improving these systems and maximizing their efficiency.

Thesis Supervisor: Dr. Stanley B. Gershwin

Title: Senior Research Scientist, Department of Mechanical Engineering

CONTENTS

| | |
|----------------------------------|----|
| 1. Introduction | 4 |
| 2. Background | 5 |
| 3. Algorithm | 7 |
| 4. Program Verification | 10 |
| 5. Conclusions | 12 |
| Appendices (code) | |
| A. User Guide | 13 |
| B. Programmer's Manual | 20 |
| C. Tandem Program Code | 24 |
| D. Re-Entrant Program Code | 36 |
| E. Input and Output Files | 49 |
| References | 52 |

1. Introduction

A manufacturing system is a set of machines, transportation elements, computers, storage buffers and other items that are used together for manufacturing (Gershwin, 1994). They are complex dynamic systems that we rely on every day to process and produce all types of goods ranging from toys to automobiles and beyond.

Surprisingly, as central as manufacturing systems are, they are still not completely understood. A new and growing field is Manufacturing System Analysis which studies a way in which to model these systems to build and analyze them effectively.

Due to the complexity of manufacturing systems, to precisely calculate their performance over time takes too long in some cases, and is impossible in others. So instead, these systems are represented by approximations. In order to build approximations that accurately reflect the manufacturing systems, the equations used are equally intricate and require verification via simulation. The mathematical approximation is applied to a range of initial parameters of the system, and the simulation runs using these same parameters. The degree to which the approximation can accurately predict the behavior of the simulation is the ultimate measurement of its success.

The work in this thesis focuses on a particular type of manufacturing system in which the machine line allows for part re-entry. This means that once a part has gone from one end of the machine line to the other, it comes through again. Therefore multiple part types, where type is defined by the number of times the part has gone through the system, are all run through the same set of machines. This situation is often seen in the production of silicon wafers where a part has to go through the machine line several

times before it is finished. My work has to do with building the simulation and using it to test for interesting phenomenon when different input values are varied.

2. Background

The study of manufacturing systems is gaining a depth in literature that defines the building blocks of the field. Studying these systems has begun with looking at the simplest possible types of machine lines and building up to being able to map more complex systems.

The basic background in this field begins with the definition of some of the major terms used to describe manufacturing systems. In Figure 1 below is a simplified depiction of the system which I will be working with for my thesis. Every box that has an 'M' inside represents a machine in the line, and the arrows show the direction in which parts are moved from one part of the system to the next. Since there are often periods of time that a part waits between machines, this is shown here as a buffer, which holds a limited amount of parts as they wait between machines. The buffers are the circles with a 'B' inside that are situated between machines.

Each machine has a rate at which it produces parts and each buffer has a maximum capacity. In addition there is a probability attached to every machine representing how often it will break down, and a probability of repair once the machine is down. The way we represent a machine's status is either as 'up' when it can work on parts or 'down' when it is broken and needs repair.

The basic elements of a machine and buffer have been modeled successfully, so building off of these, the Tandem and Re-Entrant systems are more complex for several reasons. Both systems allow for multiple part types that the machine line processes where each part type has a different priority level. This priority determines when it is processed by the machines, because a higher priority part when possible will always be chosen over a lower priority part. In that way the machine must know not only whether it can process a part, but must know which part it should process when given the choice. A depiction of this system is below in Figure 1.

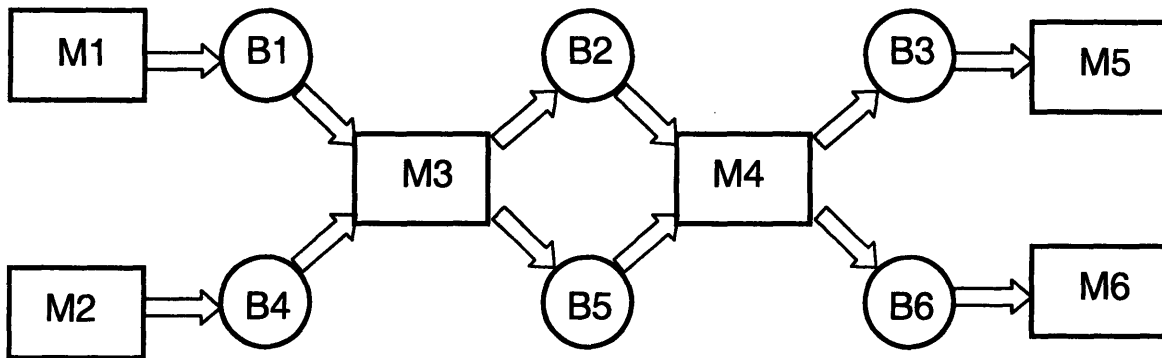


Figure 1: Multi-Part Tandem Machine Line

The Re-Entrant system has the added complexity of re-entry into the system. Looking at Figure 2, Machine 3, which is in the lower right hand of the diagram represented by a box labeled 'M₃' produces a part which does not leave the system, but instead returns to be machined again by Machine 2, labeled 'M₂' at the center of the diagram. This is the definition of re-entry because a part having gone through the system once is looped back to go through it again.

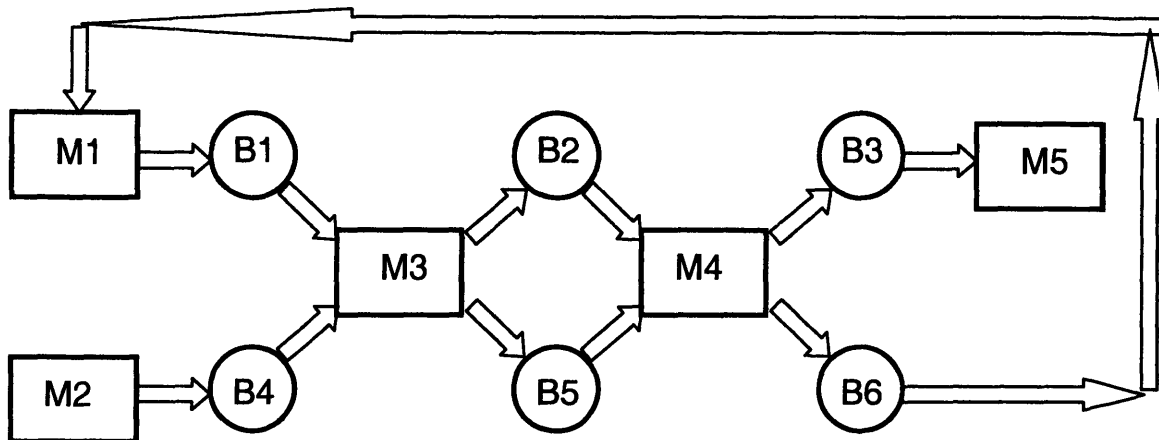


Figure 2: Two-Part Re-Entry Machine Line

The multi-part re-entry machine line is building on the work done already to study multi-part lines as with the Tandem system. Combining these features into a significantly more complex system has drawn from the methods used to define the simpler systems, and the simulation builds from those of the simple systems.

3. Algorithm

The Tandem and Re-entrant systems described above are simply designed, yet simulating their process of part production is a challenging task. Of the two models presented here, the Tandem model was developed first and then modified to allow for re-entrance. This section will describe the step by step process or basic algorithm for these programs. The basic program structure consists of taking in data, initializing core variables, running a certain length of simulation steps over which machines' status and buffer sizes are constantly calculated, and finally finding average values for machine productivity and buffer size.

The simulation is built as a Java program that is aimed at simulating the two-part machine line with re-entry. The duration of the simulation is defined by the transient

period and steady state period length declared as global variables at the beginning of the program. The transient period is the number of steps taken by the simulation before it is considered to be stable, and the steady state period is how many steps are taken while in this steady state condition that will be used to evaluate the system's behavior.

The program begins by taking in basic data from the user about the setup and characteristics of the machines and buffers. Information on the buffer size, machine repair and failure probabilities, as well as the number of processing machines and part types are all entered by the user through a text file. This text file must be organized in a specific structure which is described at length in Appendix A: User Guide. If the text follows the right format all the required information will be taken in by the simulation.

Once the program has taken in all of the facts about the machine line it needs, the next step is to begin the simulation. There are certain base line facts that the simulation assumes and runs on. First, all machines in the line begin 'up' or operational and second, all buffers in the system start with one part each. Once these initial values are set, the simulation is ready to run.

Only the steady state period is counted toward production rates and average buffer sizes, and the transient period is excluded to make sure that any transient properties of the system start-up do not impact the final results. In each time step several sub steps are taken by the simulation. The status for all machines is checked, beginning with supply machines, then going on to processing machines and finally demand machines.

Each machine is checked to see whether it changes its status as either up or down, using the probabilities of failure and repair. Then based on this information, if a machine is operating, the buffers before and after it are examined to see whether the machine can process a part. If the buffer after a machine is full, the machine will not start processing a part because it has nowhere to place it when complete. This condition is known as the machine being blocked. If the buffer before the machine is empty, the machine is starved because it has no part to work on. If the machine is neither blocked nor starved, and is currently up, it can process the part.

The Tandem and Re-Entrant machine lines allow for several part types to be processed by one machine. In order for this to occur there must be a method of determining which part the machine should work on next, if there are several options. The way in which the next part is chosen is based on its priority in the line. The machine will attempt to process parts of higher priority first unless it is blocked or starved for that part. The processing machine will only move on to lower priority parts if it cannot process the higher priority part.

A part moving through the machine is tracked in two ways. A matrix of part production is updated to keep count of when each machine is working on each part. Also, when a part goes through the machine, buffers on either side for that part are incremented or decremented as necessary.

Finally, once the simulation has run through the total set of steps, two main pieces of information are calculated. Based on the buffer sizes throughout the steady state period, the average size of each buffer is reported. Secondly, the rate at which each

machine is producing each part type is also calculated, by finding the probability that at any given time the machine is working on that part.

The two simulations in general take the same approach, but there is a major area of divergence, and that comes in the re-entrance of parts. The Re-Entrant program cycles parts back through the system, by linking the last buffer for each part type to the supply machine of the next higher part type. Examples of both the input and output code can be found in Appendix A: User's Guide.

4. Program Verification

In order to test the validity of the new simulations, the results of the Tandem program were compared against those of a similar multi-part simulation with a different algorithm. A set of 300 input files were run through both simulations to compare the outputs over a range of input parameters.

This particular input file was for a two-part system with five processing machines. A depiction of the system is below in Figure 3.

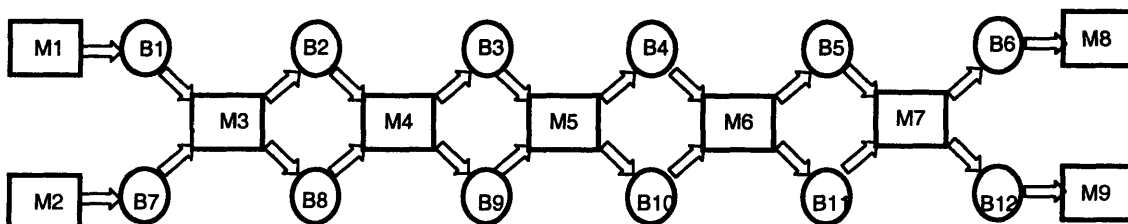


Figure 3: Example Tandem System Tested

The machine repair and failure rates and the buffer sizes are listed in the input file in Appendix E. When run through the system this input gave the following results:

| Output Value | Tandem Program | Comparison Simulation |
|----------------------|----------------|-----------------------|
| Buffer 1 Avg Level | 8.665 | 8.127 |
| Buffer 2 Avg Level | 24.847 | 27.481 |
| Buffer 3 Avg Level | 25.410 | 26.731 |
| Buffer 4 Avg Level | 25.916 | 27.204 |
| Buffer 5 Avg Level | 26.620 | 27.920 |
| Buffer 6 Avg Level | 26.498 | 27.312 |
| Buffer 7 Avg Level | 10.596 | 10.322 |
| Buffer 8 Avg Level | 35.935 | 39.146 |
| Buffer 9 Avg Level | 17.558 | 15.970 |
| Buffer 10 Avg Level | 15.647 | 14.179 |
| Buffer 11 Avg Level | 14.450 | 15.458 |
| Buffer 12 Avg Level | 2.674 | 3.356 |
| Part 1 Avg Prod Rate | 0.562 | 0.560 |
| Part 2 Avg Prod Rate | 0.229 | 0.249 |

Table 1: Program Verification Results

To examine how the Tandem system works a basic input parameter set was run to verify the simulation's results. First, a system in which all the machines are perfectly reliable was run with five machines in one line with buffers of size 10 between them. The results as given by the Tandem Program were buffers of average level 1.0 and production rates of 1.0. For a more detailed look at the input and output files, please see Appendix E. These results are exactly what is expected because if the machines never break down, at every step one part moves into and out of every buffer. That process leaves the buffers at the equilibrium level of one part, and every machine is always working on a part and therefore has a production rate of 1.0.

5. Conclusions

Simulating manufacturing systems is a way in which scientists can analyze and optimize the complex web of interactions that determine a machine line. In this work two such simulations were created to work with multi-part lines on one hand and even more complicated re-entrant lines on the other. Verifying this work by comparing it to previously accepted simulations of multi-part systems enables the programs to be applied over a range of input parameters. The results of using these different sets of inputs will uncover information about the way these multi-part systems function.

Further work beyond this thesis should focus on using the new dimensions of flexibility that the programs provide. Adjusting the relative rates of production for the machines and using a variable number of parts and processing machines, broadens the spectrum of machines lines that can be simulated. These more complex systems can now be studied in order to find how their most efficient and effective compositions.

APPENDIX A: User Guide

Preface

This User Guide is intended to provide users of the Tandem and Re-Entrant programs the information necessary to successfully use these programs to their fullest capacity. Any questions not addressed in this User's Guide, may be answered in the Programmer's Manual.

Contents

Section 1: Purpose of Program

Section 2: Program Capabilities

Section 3: Tandem Program

3.1: Description

3.2: How to create an input file

3.3: Program prompts

Section 4: Re-Entrant Program

4.1: Description

4.2: How to create an input file

4.3: Program prompts

Section 5: Input/Output File Examples

5.1: Diagrams and Input File Examples

Section 5.1.1: Tandem Program

Section 5.1.2: Re-Entrant Program

5.2: Output File and Interpretation

Section 6: Trouble Shooting

6.1: File Reading Errors

6.1.1: Does File exist?

6.1.2: Is input in correct format?

6.1.3: Are input values right?

6.2: File Writing Errors

6.2.1: Does output file already exist in read-only form?

Section 1: Purpose of Program

The Tandem and Re-Entrant programs were written to enable users to run simulations of these two types of systems with several levels of flexibility. These programs take in and export data through text files and depending on the settings used can adjust to the user's specific goals.

Section 2: Tandem Program

Section 2.1: Description

The Tandem Program simulates a simple system in which parts flow from the Supply Machine through Processing Machines to the Demand Machines. There is a Supply Machine for each part type as well as a Demand Machine for each part type. The Processing Machines are used for all part types and prioritize the parts in the order in which they are numbered.

Section 2.2: How to create an input file

The input file should be in the following format:

```
< Number of Part Types >
< Number of Processing Machines >
< Supply Machine Repair Rate > < Supply Machine Failure Rate > < Supply Buffer >
< Supply Machine Repair Rate > < Supply Machine Failure Rate > < Supply Buffer >
...
< Processing Machine Repair Rate > < Processing Machine Failure Rate >
    < Buffer Part 1 > < Buffer Part 2 > < . . . >
< Processing Machine Repair Rate > < Processing Machine Failure Rate >
    < Buffer Part 1 > < Buffer Part 2 > < . . . >
...
< Demand Machine Repair Rate > < Demand Machine Failure Rate >
< Demand Machine Repair Rate > < Demand Machine Failure Rate >
...
```

For an example of this please see Section 5.1.1

Section 2.3: Program prompts

The program has two prompts regarding reading files and writing to files. Once the programming welcomes you, it will ask you for an input file name. This file name can be entered as either just the name when the file is in the same directory as the program, or as the full directory address.

The second prompt once the simulation has been completed is for a file in which to store the results. This file can either be a new file name, which will then be created, or

can be an old file that will be rewritten. If the file entered is a read-only file you will receive an error because the information cannot be written on that file.

Section 3: Re-Entrant Program

Section 3.1: Description

The Re-Entrant Program simulates a system in which parts begin at the lowest priority and flow through the Processing Machines, and then return to the beginning of the system now as a part type one level higher in priority. Once the part has reached the highest priority it is processed for the last time and exits through the single Demand Machine. There is a Supply Machine for each part type and only one Demand Machine. The Processing Machines are used for all part types and prioritize the parts in the order in which they are numbered which coincides with the length of time they have been in the system.

Section 3.2: How to create an input file

The input file should be in the following format:

```
< Number of Part Types >  
< Number of Processing Machines >  
< Supply Machine Repair Rate > < Supply Machine Failure Rate > < Supply Buffer >  
< Supply Machine Repair Rate > < Supply Machine Failure Rate > < Supply Buffer >  
...  
< Processing Machine Repair Rate > < Processing Machine Failure Rate >  
    < Buffer Part 1 > < Buffer Part 2 > < . . . >  
...  
< Demand Machine Repair Rate > < Demand Machine Failure Rate >
```

For an example of this please see Section 5.1.2

Section 3.3: Program prompts

The program has two prompts regarding reading files and writing to files. Once the programming welcomes you, it will ask you for an input file name. This file name can be entered as either just the name when the file is in the same directory as the program, or as the full directory address.

The second prompt once the simulation has been completed is for a file in which to store the results. This file can either be a new file name, which will then be created, or can be an old file that will be rewritten. If the file entered is a read-only file you will receive an error because the information cannot be written on that file.

Section 4: Program Capabilities

The Tandem and Re-Entrant programs allow users to simulate machine lines with several different dimensions of variability. The first two dimensions are in the number of processing machines and the number of part types. The user specifies both of these variables through the input file. The third dimension is the relative speed at which the three different groups of machines (demand, processing and supply) produce parts. The user can specify any integer ratio of these processing times. In order to adjust these times, see the Programmer's Manual under 'How to Modify Program.'

Section 5: Input/Output File Examples

Section 5.1: Diagrams and Input File Examples

Section 5.1.1: Tandem Program

In the diagram below is a simple Tandem Machine Line. This particular line has two Processing Machines (M3 and M4) and two types of parts. An example of input code to represent this setup is after the diagram.

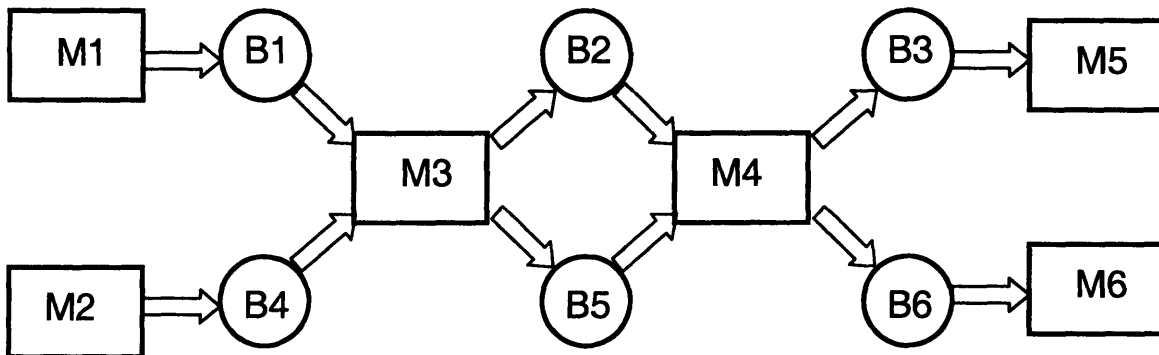


Figure 1: Tandem Machine Line

Assuming the above system has the following parameters, where r is the repair rate and p is the probability of failure:

| | |
|-------------------------|--------|
| M1: $r = 0.1, p = 0.01$ | B1: 10 |
| M2: $r = 0.2, p = 0.02$ | B2: 20 |
| M3: $r = 0.3, p = 0.03$ | B3: 30 |
| M4: $r = 0.4, p = 0.04$ | B4: 40 |
| M5: $r = 0.5, p = 0.05$ | B5: 50 |
| M6: $r = 0.6, p = 0.06$ | B6: 60 |

InputFile.txt

```

2
2
0.1 0.01 10
0.2 0.02 40
  
```


| | | | |
|-----|------|----|----|
| 0.3 | 0.03 | 20 | 50 |
| 0.4 | 0.04 | 30 | 60 |
| 0.5 | 0.05 | | |
| 0.6 | 0.06 | | |

Section 5.1.2: Re-Entrant Program

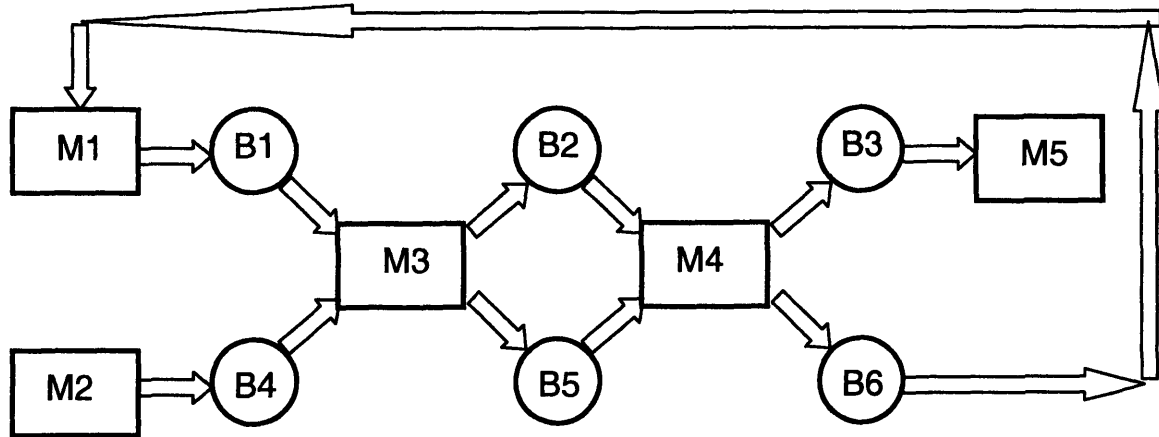


Figure 2: Re-Entrant Machine Line

Assuming the above system has the following parameters, where r is the repair rate and p is the probability of failure:

| | |
|-------------------------|--------|
| M1: $r = 0.1, p = 0.01$ | B1: 10 |
| M2: $r = 0.2, p = 0.02$ | B2: 20 |
| M3: $r = 0.3, p = 0.03$ | B3: 30 |
| M4: $r = 0.4, p = 0.04$ | B4: 40 |
| M5: $r = 0.5, p = 0.05$ | B5: 50 |
| M6: $r = 0.6, p = 0.06$ | B6: 60 |

Example: InputFile.txt

```

2
2
0.1 0.01 10
0.2 0.02 40
0.3 0.03 20 50
0.4 0.04 30 60
0.5 0.05

```

Section 5.2: Output File and Interpretation

Example OutputFile.txt

```
Buffer 1 has average size: 8.214408396946565
Buffer 2 has average size: 23.998320610687024
Buffer 3 has average size: 24.4024427480916
Buffer 4 has average size: 24.910152671755725
Buffer 5 has average size: 25.607480916030536
Buffer 6 has average size: 25.06973282442748
Buffer 7 has average size: 10.093702290076337
Buffer 8 has average size: 34.32675572519084
Buffer 9 has average size: 15.847003816793894
Buffer 10 has average size: 13.808473282442748
Buffer 11 has average size: 14.663091603053434
Buffer 12 has average size: 2.8237213740458014
The probability that M1 is working on Part 1 is: 0.5278625954198474
The probability that M1 is working on Part 2 is: 0.0
The probability that M2 is working on Part 1 is: 0.0
The probability that M2 is working on Part 2 is: 0.22853053435114504
The probability that M3 is working on Part 1 is: 0.527881679389313
The probability that M3 is working on Part 2 is: 0.22853053435114504
The probability that M4 is working on Part 1 is: 0.5283396946564886
The probability that M4 is working on Part 2 is: 0.2285496183206107
The probability that M5 is working on Part 1 is: 0.5281488549618321
The probability that M5 is working on Part 2 is: 0.22824427480916032
The probability that M6 is working on Part 1 is: 0.5281106870229008
The probability that M6 is working on Part 2 is: 0.22790076335877862
The probability that M7 is working on Part 1 is: 0.5281297709923665
The probability that M7 is working on Part 2 is: 0.22786259541984732
The probability that M8 is working on Part 1 is: 0.5281488549618321
The probability that M8 is working on Part 2 is: 0.0
The probability that M9 is working on Part 1 is: 0.0
The probability that M9 is working on Part 2 is: 0.22790076335877862
```

The output file above provides two types of information. First it reports the average load for all the buffers in the system. Second, it calculates the rate at which each machine produces each part type. Keep in mind that not every machine works on every part type. Specifically the demand and supply machines only work on one part type each. Please refer to Figure 1 above for the pattern in which machines and buffers are numbered.

Section 6: Trouble Shooting

The most likely error the user could run into while using these programs is related to the use of files in the input and output.

Section 6.1: File Reading Errors

Section 6.1.1: Does File exist?

Make sure that the file referred to in the program does exist and that its name was correctly entered. The file can be located either in the same directory as the program code and referred to by just its name, or it could be in another directory as long as its fully address is used.

Section 6.1.2: Is input in correct format?

The program may return an error if the input file does not provide complete information or the information is not correctly formatted. Please refer to Sections 2.2 and 5.1 for how to create the input file.

Section 6.1.3: Are input values right?

The input values for the Tandem and Re-Entrant programs must meet certain expectations for a machine line. These expectations are as follows:

Machine Repair Rate (r): $0 \leq r \leq 1$

Machine Failure Rate (p): $0 \leq p \leq 1$

Buffer Size ≥ 1

Number of Part Types ≥ 1

Number of Processing Machines ≥ 1

Section 6.2: File Writing Errors

Section 6.2.1: Does File already exist in read-only form?

If an error occurs when writing to a file it is most likely because the file already exists and is in read-only form. The program can not write to this type of file and will return an error.

APPENDIX B: Programmer's Manual

Preface

This Programmer's Manual is intended for those users who want to understand the basic algorithm used in the Tandem and Re-Entrant Programs and to make more in depth modifications to the program. Any questions not addressed here may be answered in the general User's Guide found in Appendix A.

Contents

Section 1: Purpose of Program

Section 2: Basic Algorithm

Section 3: Program Capabilities

Section 3.1: What can I change using the input file?

Section 3.2: What can I change within the code?

Section 4: How to Modify Program

Section 4.1: Increasing Processing Machine Speed

Section 4.2: Running Multiple Files

Section 1: Purpose of Program

The Tandem and Re-Entrant programs were written to enable users to run simulations of these two types of systems with several levels of flexibility. These programs take in and export data through text files and depending on the settings used can adjust to the user's specific goals.

Section 2: Basic Algorithm

The basic structure of the Tandem and Re-Entrant programs consists of taking in data, initializing core variables, running a certain length of simulation steps over which machines' status and buffer sizes are repeatedly calculated, and finally finding average values for machine productivity and buffer size.

The simulation is built as a Java program that is aimed at simulating the two-part machine line with re-entry. The duration of the simulation is defined by the transient period and steady state period length declared as global variables at the beginning of the program. The transient period is the number of steps taken by the simulation before it is considered to be stable, and the steady state period is how many steps are taken while in this steady state condition that will be used to define the system's behavior.

The program prompts the user for an input file, from which it reads in basic data about the setup and characteristics of the machines and buffers. Specifically this information is the buffer size, machine repair and failure rates, as well as the number of processing machines and part types. The input text file must be organized in a specific structure which is described in Appendix A: User Guide. If the text follows the right format all the required information will be taken in by the simulation.

Once the program has taken in all of the facts about the machine line it needs, the next step is to begin the simulation. The program initializes several values using the following assumptions: all machines in the line begin 'up' or operational and all buffers in the system start with one part each. Once these initial values are set, the simulation begins to run through steps

Only the steady state period is counted toward production rates and average buffer sizes, and the transient period is excluded to make sure that any transient properties of the system start-up do not impact the final results. In each time step several sub steps are taken by the simulation. The status for all machines is checked, beginning with supply machines, then going on to processing machines and finally demand machines.

Each machine is checked to see whether it changes its status as either up or down, using the probabilities of failure and repair. Then based on this information, if a machine is operating, the buffers before and after it are examined to see whether the machine can process a part. If the buffer after a machine is full, the machine will not start processing a part because it has nowhere to place it when complete. This condition is known as the machine being blocked. If the buffer before the machine is empty, the machine is starved

because it has no part of work on. If the machine is neither blocked nor starved, and is currently up, it can process the part.

Each time step goes through first the Supply Machines, then the Processing Machines and finally the Demand Machines. The status of the machine is adjusted along with the neighboring buffer sizes for each machine of that type before moving on to the next type.

A part moving through the machine is tracked in two ways. A matrix of part production is updated to keep count of when each machine is working on each part. Also, when a part goes through the machine, buffers on either side for that part are incremented or decremented as necessary.

Finally, once the simulation has run through the total set of steps, two main pieces of information are calculated. Based on the buffer sizes throughout the steady state period, the average size of each buffer is reported. Secondly, the rate at which each machine is producing each part type is also calculated, by finding the probability that at any given time the machine is working on that part.

The two simulations in general take the same approach, but there is a major area of divergence, and that comes in the re-entrance of parts. The Re-Entrant program cycles parts back through the system, by linking the last buffer for each part type to the supply machine of the next higher part type. Examples of both the input and output code can be found in Appendix A: User's Guide.

Section 3: Program Capabilities

There are several degrees of freedom allowed in this program and split into two categories. The first set you can adjust by manipulating the input file, and the second require adjusting the code.

Section 3.1: What can I change using the input file?

The input file adjusts the major machine parameters of number of parts, number of processing machines, repair and failure rates of all these machines and buffer sizes. In order to create an input file, please see the User's Manual Section 5: Input/Output File Examples.

Section 3.2: What can I change within the code?

The code has separate blocks for the operation of the Supply, Processing and Demand Machines. This separation was done intentionally to allow users to run these machines at different speeds. The user can create loops around the blocks for each type of machine separately to allow them to run a different number of times each step. For example a loop around the Processing Machines can specify that these machines run five times each step, so that the simulation represents a system where the Processing Machine

is five times faster than the Supply and Demand machines. For more details see Section 4.1 below.

Section 4: How to Modify Program

Section 4.1: Increasing Processing Machine Speed

As mentioned in the sections above, the code allows for the separate functioning of Supply, Processing and Demand Machines during each time step. To increase the Processing Machine speed there is one simple loop that needs to be added to the code. Around the block of code that updates the Processing Machine Status, place a loop that will repeat x times, where x is the number of times faster the Processing Machine is.

Section 4.2: Running Multiple Files

In order to run a string of input files quickly the program can automate the reading and creation of text files. The user can create a loop around the entire block of code inside the Tandem or Re-Entrant class to run through however many files as desired. Then the user must adjust the name of the file to be read in such a way that all intended input files are looped through.

For example if there are 400 input files the user wants to process that are named systematically “input1” through “input400”, then create a loop that goes through the program 400 times. Allow the counter variable to be called fileNum, and replace this code:

```
Scanner console = new Scanner (System.in);  
System.out.print("Input file: ");  
String inputFileName = console.next();
```

with this code:

```
String inputFileName = "input" + fileNum;
```

This will read in the data files of all 400 cases. To create the outputs with the file name “output1” through “output400” for instance, replace this code:

```
System.out.print("Output file: ");  
String outputFileName = console.next();
```

with this code:

```
String outputFileName = "output" + fileNum;
```

This should allow the user to more quickly and easily run many sets of input files and use the code for experiments and analysis.

APPENDIX C: Tandem Program Code

```
import java.util.*;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class LINEAR
{

    public static void main(String[] args)
    {

        // Initialize Variables

        // steadyStatePeriod is the number of time steps in the actual simulation
        int steadyStatePeriod = 5000;

        // transientPeriod is the number of time steps to warm up the system
        // These time steps will not be counted when finding average buffer size and
        // productivity
        int transientPeriod = 2400;

        // periodLength is the total number of time steps simulated
        // periodLength is simply the sum of the warm-up and simulation time
        int periodLength = transientPeriod + steadyStatePeriod;

        System.out.println("WELCOME TO A SIMPLE LINEAR MANUFACTURING
        SYSTEM SIMULATION!");

        // A Scanner is created to communicate with the Console
        Scanner console = new Scanner (System.in);

        // The user is prompted to enter the file name in which the simulation details are
        // given
        System.out.print("Input file: ");
        String inputFileName = console.next();

        // S = number of supply machines
        int S = 0;

        // K = number of processing machines
        int K = 0;

        try
```



```

{
    // A FileReader is created to take in information from the input file
    FileReader reader = new FileReader (inputFileName);

    // Scanner 'in' is created to read the information taken in from the input file
    Scanner in = new Scanner(reader);

    // Assign number of parts (or number of supply machines)
    S = in.nextInt();

    // Assign number of processing machines
    K = in.nextInt();

    // Close the input file
    in.close();
}

// In case an exception is thrown, let user know there is a problem with the file
catch (IOException exception)
{
    System.out.println("Error processing file: " + exception);
}

// Create array of machine repair rates
double[] r = new double[2*S + K + 1];

// Create array of machine failure rates
double[] p = new double[2*S + K + 1];

// Create array of buffer sizes
int[] N = new int[S*(K+1) + 1];

// Once the number of machines and part types are known, can take in machine
information

try
{
    // A FileReader is created to take in machine information from the input file
    FileReader reader = new FileReader (inputFileName);

    // Scanner 'in' is created to read the machine information taken in from the input
file
    Scanner in = new Scanner(reader);

    // Skip past first two integers representing number of part types and processing
machines

```

```

int junk = in.nextInt();
junk = in.nextInt();

// Get information for Supply Machines
for (int z = 1; z <= S; z++)
{
    // Read in repair rate
    r[z] = in.nextDouble();

    // Read in failure rate
    p[z] = in.nextDouble();

    // Read in next buffer size
    N[(z-1)*(K+1)+1] = in.nextInt();
}

// Get information for Process Machines
for (int z = 1; z <= K; z++)
{
    // Read in repair rate
    r[S+z] = in.nextDouble();

    // Read in failure rate
    p[S+z] = in.nextDouble();

    // Read in buffer sizes for all buffers after machine
    for (int y = 1; y <= S; y++)
    {
        N[(y-1)*(K+1) + z + 1] = in.nextInt();
    }
}

// Get information for Demand Machines
for (int z = 1; z <= S; z++)
{
    // Read in repair rate
    r[S+K+z] = in.nextDouble();

    // Read in failure rate
    p[S+K+z] = in.nextDouble();
}

// Close the input file
in.close();
}

```

```

// If an exception is thrown, let the user know there was an error with the file
catch (IOException exception)
{
    System.out.println("Error processing file: " + exception);
}

// Create running tally variables

// Initialize numBuffers and numMachines
int numMachines = 2*S + K;
int numBuffers = S*(K+1);

// Initialize counter variables
int i = 0;
int y = 0;

//All machines begin 'up', alpha = 1
int[][] alpha = new int[periodLength][numMachines+1];
for (i=1; i<=numMachines; i++)
    alpha[0][i] = 1;

//All buffers begin with one part in them
int[][] B = new int[periodLength][numBuffers+1];
for (i=1; i<=numBuffers; i++)
    B[0][i] = 1;

//Matrix of number of parts being worked on during the simulation
int[][][] prod = new int[periodLength][numMachines+1][S+1];
for (i=0; i<periodLength; i++)
    for(int j=0; j<=numMachines; j++)
        for(int k=0; k<=S; k++)
            prod[i][j][k] = 0;

// Create a random number generator
Random generator = new Random();

// Run through simulation for 'periodLength' steps
for (i=1; i<periodLength; i++)
{
    // Initially set all buffers in time i to what they were in time i-1
    for(int count = 1; count<=numBuffers; count++)
    {
        B[i][count] = B[i-1][count];
    }
}

```

```

// FIND STATUS FOR ALL MACHINES

// Find Supply Machines' Status

// For all supply machines 1 through S
for(y=1; y<=S; y++)
{
  // if (Supply Buffer is full -> Supply Machine is blocked)
  if (B[i-1][(y-1)*(K+1)+1] == N[(y-1)*(K+1)+1])
  {
    // Supply Machine is blocked -> not operating -> can't break down
    alpha[i][y] = 1;
  }
  else
  {
    // if Supply Machine is 'up' in last step
    if(alpha[i-1][y] == 1)
    {
      // Check to see if Supply Machine fails now

      // Generate random number between 0 and 1
      double x = generator.nextDouble();

      // if random number is less than probability of failure
      if (x<p[y])
      {
        // Supply Machine fails
        alpha[i][y] = 0;
      }
      else
      {
        // Supply Machine is still running
        alpha[i][y] = 1;

        // Process part type y through machine
        // Increase buffer after Supply Machine
        B[i][(y-1)*(K+1)+1] = B[i][(y-1)*(K+1)+1] + 1;

        // Fill production matrix, Machine y is working on Part y
        prod[i][y][y] = 1;
      }
    }
  }

  // if Supply Machine was 'down' in last step
  else
  {

```

```

// Check to see if Supply Machine is repaired now

// Generate random number between 0 and 1
double x = generator.nextDouble();

// if random number is less than probability of repair
if (x<r[y])
{
    // Supply Machine is fixed
    alpha[i][y] = 1;

    // Process part type y through machine
    // Increase buffer after Supply Machine
    B[i][(y-1)*(K+1)+1] = B[i][(y-1)*(K+1)+1] + 1;

    // Fill production matrix, Machine y is working on Part y
    prod[i][y][y] = 1;
}
else
{
    // Supply Machine is still down
    alpha[i][y] = 0;
}
}
}

// Find Processing Machines' Status

// For all processing machines from 1 through K
for(int x = 1; x <= K; x++)
{
    // Initialize variable 'yfit' to hold the Part Type that will be processed
    // 'yfit' will remain 0 until a Part Type to be processed has been chosen
    int yfit = 0;

    // Initialize Part Type counter
    y=1;

    // While Part Type counter has not reached last part type
    // and Part Type to be processed has not been selected
    while(y<=S && yfit == 0)
    {
        // if the buffer for Part Type y before Machine x is empty
        if(B[i-1][(y-1)*(K+1)+x] <= 0)
        {

```

```

    // Machine x is starved for Part Type y
    y++;
}

// else if the buffer for Part Type y after Machine x is full
else if(B[i-1][(y-1)*(K+1) + (x+1)] >= N[(y-1)*(K+1)+(x+1)])
{
    // Machine x is starved for Part Type y
    y++;
}

// else the Part Type to be processed has been found
else
{
    // set yfit to the Part Type to be processed
    yfit = y;
}
}

// if yfit does not equal 0, a Part Type to be processed has been found
// otherwise the Machine is starved and/or blocked for every Part Type
if(yfit!=0)
{
    // Set y to be the Part Type to be processed
    y = yfit;

    // if Machine S+x (Processing Machine x) was 'up' last step

    if(alpha[i-1][S+x] == 1)
    {
        // Check to see if Processing Machine fails now

        // Generate random number between 0 and 1
        double m = generator.nextDouble();

        // if random number is below probability of failure
        if (m < p[S+x])
        {
            // Processing Machine now fails
            alpha[i][S+x] = 0;
        }

        else
        {
            // Processing Machine is still running
            alpha[i][S+x] = 1;
        }
    }
}

```

```

// Part y goes through Processing Machine
// Decrease buffer for Part y before Processing Machine
B[i][(y-1)*(K+1)+x] = B[i][(y-1)*(K+1)+x] - 1;
// Increase buffer for Part y after Processing Machine
B[i][(y-1)*(K+1)+x+1] = B[i][(y-1)*(K+1)+x+1] + 1;

// Fill production matrix, Machine S+x working on Part y
prod[i][S+x][y] = 1;
}
}

// if Machine S+x (Processing Machine x) was 'down' last step
else
{
// Check to see if Supply Machine is repaired now
// Generate random number between 0 and 1
double m = generator.nextDouble();

// if random number is below probability of repair
if (m < r[S+x])
{
// Processing Machine is fixed
alpha[i][S+x] = 1;

// Part y goes through Processing Machine
// Decrease buffer for Part y before Processing Machine
B[i][(y-1)*(K+1)+x] = B[i][(y-1)*(K+1)+x] - 1;
// Increase buffer for Part y after Processing Machine
B[i][(y-1)*(K+1)+x+1] = B[i][(y-1)*(K+1)+x+1] + 1;

// Fill production matrix, Machine S+x on Part y
prod[i][S+x][y] = 1;
}
else
{
// Processing Machine is still down
alpha[i][S+x] = 0;
}
}
}

// Find Demand Machines' Status

// For Demand Machines for Parts 1 through S

```

```

for (y=1; y<=S; y++)
{
  // if (Demand Buffer is empty -> Demand Machine is starved)
  // Demand Machine can not fail if it is blocked or starved
  if (B[i-1][y*(K+1)] == 0)
  {
    // Demand Machine is starved -> not operating -> can't break down
    alpha[i][S + K + y] = 1;
  }
  else
  {
    // Demand Machine was 'up' last time step
    if(alpha[i-1][S + K + y] == 1)
    {
      // Check to see if Demand Machine fails now
      // Generate random number between 0 and 1
      double m = generator.nextDouble(); // Generate random number between
0 and 1

      // if random number is below probability of failure
      if (m<p[S + K + y])
      {
        // Demand Machine now fails
        alpha[i][S + K + y] = 0;
      }

      else
      {
        // Demand Machine is still working
        alpha[i][S + K + y] = 1;

        // Process Part Type y through Demand Machine
        // Decrease buffer for Part y before Demand Machine
        B[i][y*(K+1)] = B[i][y*(K+1)] - 1;

        // Fill Production Matrix, Demand Machine working on Part Type y
        prod[i][S+K+y][y] = 1;
      }
    }
  }

  // Demand Machine was 'down' last time step
  else
  {
    // Check to see if Demand Machine is repaired now
    // Generate random number between 0 and 1

```



```

double m = generator.nextDouble();

// if random number is below probability of repair
if (m < r[S + K + y])
{
    // Demand Machine is fixed
    alpha[i][S + K + y] = 1;

    // Process Part Type y through Demand Machine
    // Buffer for Part Type y before Demand Machine decreased
    B[i][y*(K+1)] = B[i][y*(K+1)] - 1;

    // Fill Production Matrix, Demand Machine working on Part Type y
    prod[i][S+K+y][y] = 1;
}
else
{
    // Demand Machine is still down
    alpha[i][S + K + y] = 0;
}
}
}
}

// Create array of counters to sum and average buffer size
double[] bufferSizes = new double[numBuffers+1];

// Loop through all buffers
for (int z = 1; z <= numBuffers; z++)
{
    // Initialize all buffers to be 0
    bufferSizes[z] = 0;

    // Loop through all the time steps in steadyStatePeriod
    for(i=transientPeriod; i<periodLength; i++)
    {
        // Add the buffer size in period i for buffer z to sum
        bufferSizes[z] += B[i][z];
    }

    // Calculate average buffer size by dividing sum by number of steadyStatePeriod
time steps
    bufferSizes[z] = bufferSizes[z]/steadyStatePeriod;
}

```

```

// Create array of counters to sum and average production rate
double[][] work = new double[numMachines+1][S+1];

// Initialize all counters to be zero
// Loop through all the machines
for(int step = 1; step<=numMachines; step++)
{
    // Loop through all the Part Types
    for(int count = 1; count<=S; count++)
    {
        // Initialize all counters to 0
        work[step][count] = 0;
    }
}

// Create output file to store results
System.out.print("Output file: ");
String outputFileName = console.next();

try
{
    // Create PrintWriter 'out' to send data to output file
    PrintWriter out = new PrintWriter(outputFileName);

    // Loop through all buffers
    for(int z=1; z<=numBuffers; z++)
    {
        // Print out average buffer size to output file
        out.println("Buffer " + z + " has average size: " + bufferSizes[z]);
    }

    // Loop through all machines
    for(int k=1; k<=numMachines; k++)
    {
        // Loop through all parts
        for(int count=1; count<=S; count++)
        {
            // Loop through the steadyStatePeriod
            for (int step = transientPeriod; step<periodLength; step++)
            {
                // Sum the number of times Machine 'k' worked on part 'count'
                work[k][count] = work[k][count] + prod[step][k][count];
            }

            // Print out production rate of Part Type 'count' through Machine 'k'
            out.println("The probability that M" + k + " is working on Part " + count + "

```

```
is: " + (work[k][count]/periodLength));
    }
}
// Close output file
out.close ();
}

// if an exception is thrown, let user know there is an error with the file

catch(IOException exception)
{
    System.out.println("Error Processing File");
}

// M1 -> B1      B2 -> M4
//      -> M3 ->
// M2 -> B3      B4 -> M5

}
}
```

APPENDIX D: Re-Entrant Program Code

```
import java.util.*;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class REENTRANT
{

    public static void main(String[] args)
    {

        // Initialize Variables

        // steadyStatePeriod is the number of time steps in the actual simulation
        int steadyStatePeriod = 5000;

        // transientPeriod is the number of time steps to warm up the system
        // These time steps will not be counted when finding average buffer size and
        // productivity
        int transientPeriod = 2400;

        // periodLength is the total number of time steps simulated
        // periodLength is simply the sum of the warm-up and simulation time
        int periodLength = transientPeriod + steadyStatePeriod;

        System.out.println("WELCOME TO A RE-ENTRANT MANUFACTURING
        SYSTEM SIMULATION!");

        // A Scanner is created to communicate with the Console
        Scanner console = new Scanner (System.in);

        // The user is prompted to enter the file name in which the simulation details are
        // given
        System.out.print("Input file: ");
        String inputFileNames = console.next();

        // S = number of supply machines
        int S = 0;

        // K = number of processing machines
        int K = 0;

        try
```

```

{
    // A FileReader is created to take in information from the input file
    FileReader reader = new FileReader (inputFileName);

    // Scanner 'in' is created to read the information taken in from the input file
    Scanner in = new Scanner(reader);

    // Assign number of parts (or number of supply machines)
    S = in.nextInt();

    // Assign number of processing machines
    K = in.nextInt();

    // Close the input file
    in.close();
}

// In case an exception is thrown, let user know there is a problem with the file
catch (IOException exception)
{
    System.out.println("Error processing file: " + exception);
}

// Create array of machine repair rates
double[] r = new double[S + K + 1];

// Create array of machine failure rates
double[] p = new double[S + K + 1];

// Create array of buffer sizes
int[] N = new int[S*(K+1) + 1];

// Once the number of machines and part types are known, can take in machine
information

try
{
    // A FileReader is created to take in machine information from the input file
    FileReader reader = new FileReader (inputFileName);

    // Scanner 'in' is created to read the machine information taken in from the input
file
    Scanner in = new Scanner(reader);

    // Skip past first two integers representing number of part types and processing
machines

```

```

int junk = in.nextInt();
junk = in.nextInt();

// Get information for Supply Machines
for (int z = 1; z<=S; z++)
{
    // Read in repair rate
    r[z] = in.nextDouble();

    // Read in failure rate
    p[z] = in.nextDouble();

    // Read in next buffer size
    N[(z-1)*(K+1)+1] = in.nextInt();
}

// Get information for Process Machines
for (int z = 1; z <= K; z++)
{
    // Read in repair rate
    r[S+z] = in.nextDouble();

    // Read in failure rate
    p[S+z] = in.nextDouble();

    // Read in buffer sizes for all buffers after machine
    for (int y = 1; y <= S; y++)
    {
        N[(y-1)*(K+1) + z + 1] = in.nextInt();
    }
}

// Get information for Demand Machine

// Read in repair rate
r[2*S + K] = in.nextDouble();

// Read in failure rate
p[2*S + K] = in.nextDouble();

// Close the input file
in.close();
}

// If an exception is thrown, let the user know there was an error with the file

```

```

catch (IOException exception)
{
    System.out.println("Error processing file: " + exception);
}

// Create running tally variables

// Initialize numBuffers and numMachines
int numMachines = 2*S + K;
int numBuffers = S*(K+1);

// Initialize counter variables
int i = 0;
int y = 0;

//All machines begin 'up', alpha = 1
int[][] alpha = new int[periodLength][numMachines+1];
for (i=1; i<=numMachines; i++)
    alpha[0][i] = 1;

//All buffers begin with one part in them
int[][] B = new int[periodLength][numBuffers+1];
for (i=1; i<=numBuffers; i++)
    B[0][i] = 1;

//Matrix of number of parts being worked on during the simulation
int[][][] prod = new int[periodLength][numMachines + 1][S+1];
for (i=0; i<periodLength; i++)
    for(int j=0; j<=numMachines; j++)
        for(int k=0; k<=S; k++)
            prod[i][j][k] = 0;

// Create a random number generator
Random generator = new Random();

// Run through simulation for 'periodLength' steps
for (i=1; i<periodLength; i++)
{
    // Initially set all buffers in time i to what they were in time i-1
    for(int count = 1; count<=numBuffers; count++)
    {
        B[i][count] = B[i-1][count];
    }

    // FIND STATUS FOR ALL MACHINES

```

```

// Find Supply Machines' Status

// For all supply machines 1 through S-1
for(y=1; y<S; y++)
{
    // if Supply Buffer is full -> Supply Machine is blocked
    // if Buffer at end of last Part Type is empty -> Supply Machine is starved
    if ((B[i-1][(y-1)*(K+1)+1] == N[(y-1)*(K+1)+1]) || B[i-1][(y+1)*(K+1)] == 0)
    {
        // Supply Machine is blocked or starved -> not operating -> can't break down
        alpha[i][y] = 1;
    }
    else
    {
        // if Supply Machine is 'up' in last step
        if(alpha[i-1][y] == 1)
        {
            // Check to see if Supply Machine fails now

            // Generate random number between 0 and 1
            double x = generator.nextDouble();

            // if random number is less than probability of failure
            if (x<p[y])
            {
                // Supply Machine fails
                alpha[i][y] = 0;
            }
            else
            {
                // Supply Machine is still running
                alpha[i][y] = 1;

                // Process part type y through machine
                // Increase buffer after Supply Machine
                B[i][(y-1)*(K+1)+1] = B[i][(y-1)*(K+1)+1] + 1;

                // Fill production matrix, Machine y is working on Part y
                prod[i][y][y] = 1;
            }
        }
    }

    // if Supply Machine was 'down' in last step
    else
    {

```



```

// Check to see if Supply Machine is repaired now

// Generate random number between 0 and 1
double x = generator.nextDouble();

// if random number is less than probability of repair
if (x<r[y])
{
// Supply Machine is fixed
alpha[i][y] = 1;

// Process part type y through machine
// Increase buffer after Supply Machine
B[i][(y-1)*(K+1)+1] = B[i][(y-1)*(K+1)+1] + 1;

// Fill production matrix, Machine y is working on Part y
prod[i][y][y] = 1;
}
else
{
// Supply Machine is still down
alpha[i][y] = 0;
}
}
}

// Lowest Priority Supply Machine S
// if Supply Buffer is full -> Supply Machine is blocked
if ((B[i-1][(S-1)*(K+1)+1] == N[(S-1)*(K+1)+1]))
{
// Supply Machine is blocked or starved -> not operating -> can't break down
alpha[i][S] = 1;
}
else
{
// if Supply Machine is 'up' in last step
if(alpha[i-1][S] == 1)
{
// Check to see if Supply Machine fails now

// Generate random number between 0 and 1
double x = generator.nextDouble();

// if random number is less than probability of failure
if (x<p[S])

```

```

    {
        // Supply Machine fails
        alpha[i][S] = 0;
    }
    else
    {
        // Supply Machine is still running
        alpha[i][S] = 1;

        // Process part type S through machine
        // Increase buffer after Supply Machine
        B[i][(S-1)*(K+1)+1] = B[i][(S-1)*(K+1)+1] + 1;

        // Fill production matrix, Machine S is working on Part S
        prod[i][S][S] = 1;
    }
}

// if Supply Machine was 'down' in last step
else
{
    // Check to see if Supply Machine is repaired now

    // Generate random number between 0 and 1
    double x = generator.nextDouble();

    // if random number is less than probability of repair
    if (x < r[S])
    {
        // Supply Machine is fixed
        alpha[i][S] = 1;

        // Process Part Type S through machine
        // Increase buffer after Supply Machine
        B[i][(S-1)*(K+1)+1] = B[i][(S-1)*(K+1)+1] + 1;

        // Fill production matrix, Machine S is working on Part S
        prod[i][S][S] = 1;
    }
    else
    {
        // Supply Machine is still down
        alpha[i][S] = 0;
    }
}
}

```

```

// Find Processing Machines' Status

// For all processing machines from 1 through K
for(int x = 1; x <= K; x++)
{
    // Initialize variable 'yfit' to hold the Part Type that will be processed
    // 'yfit' will remain 0 until a Part Type to be processed has been chosen
    int yfit = 0;

    // Initialize Part Type counter
    y=1;

    // While Part Type counter has not reached last part type
    // and Part Type to be processed has not been selected
    while(y<=S && yfit == 0)
    {
        // if the buffer for Part Type y before Machine x is empty
        if(B[i-1][(y-1)*(K+1)+x] <= 0)
        {
            // Machine x is starved for Part Type y
            y++;
        }

        // else if the buffer for Part Type y after Machine x is full
        else if(B[i-1][(y-1)*(K+1) + (x+1)] >= N[(y-1)*(K+1)+(x+1)])
        {
            // Machine x is starved for Part Type y
            y++;
        }

        // else the Part Type to be processed has been found
        else
        {
            // set yfit to the Part Type to be processed
            yfit = y;
        }
    }

    // if yfit does not equal 0, a Part Type to be processed has been found
    // otherwise the Machine is starved and/or blocked for every Part Type
    if(yfit!=0)
    {
        // Set y to be the Part Type to be processed
    }
}

```

```

y = yfit;

// if Machine S+x (Processing Machine x) was 'up' last step

if(alpha[i-1][S+x] == 1)
{
    // Check to see if Processing Machine fails now

    // Generate random number between 0 and 1
    double m = generator.nextDouble();

    // if random number is below probability of failure
    if (m < p[S+x])
    {
        // Processing Machine now fails
        alpha[i][S+x] = 0;
    }

    else
    {
        // Processing Machine is still running
        alpha[i][S+x] = 1;

        // Part y goes through Processing Machine
        // Decrease buffer for Part y before Processing Machine
        B[i][y-1][S+x] = B[i][y-1][S+x] - 1;
        // Increase buffer for Part y after Processing Machine
        B[i][y][S+x+1] = B[i][y][S+x+1] + 1;

        // Fill production matrix, Machine S+x working on Part y
        prod[i][S+x][y] = 1;
    }
}

// if Machine S+x (Processing Machine x) was 'down' last step
else
{
    // Check to see if Supply Machine is repaired now
    // Generate random number between 0 and 1
    double m = generator.nextDouble();

    // if random number is below probability of repair
    if (m < r[S+x])
    {
        // Processing Machine is fixed
        alpha[i][S+x] = 1;
    }
}

```

```

// Part y goes through Processing Machine
// Decrease buffer for Part y before Processing Machine
B[i][(y-1)*(K+1)+x] = B[i][(y-1)*(K+1)+x] - 1;
// Increase buffer for Part y after Processing Machine
B[i][(y-1)*(K+1)+x+1] = B[i][(y-1)*(K+1)+x+1] + 1;

// Fill production matrix, Machine S+x on Part y
prod[i][S+x][y] = 1;
}
else
{
// Processing Machine is still down
alpha[i][S+x] = 0;
}
}
}
}

// Find Demand Machine's Status
// if (Demand Buffer is empty -> Demand Machine is starved)
// Demand Machine can not fail if it is blocked or starved
if (B[i-1][K+1] == 0)
{
// Demand Machine is starved -> not operating -> can't break down
alpha[i][S + K + 1] = 1;
}
else
{
// Demand Machine was 'up' last time step
if(alpha[i-1][S + K + 1] == 1)
{
// Check to see if Demand Machine fails now
// Generate random number between 0 and 1
double m = generator.nextDouble(); // Generate random number between 0
and 1

// if random number is below probability of failure
if (m < p[S + K + 1])
{
// Demand Machine now fails
alpha[i][S + K + 1] = 0;
}

else
{

```

```

// Demand Machine is still working
alpha[i][S + K + 1] = 1;

// Process Part Type 1 through Demand Machine
// Decrease buffer for Part 1 before Demand Machine
B[i][K+1] = B[i][K+1] - 1;

// Fill Production Matrix, Demand Machine working on Part Type 1
prod[i][S+K+1][1] = 1;
}
}

// Demand Machine was 'down' last time step
else
{
// Check to see if Demand Machine is repaired now
// Generate random number between 0 and 1
double m = generator.nextDouble();

// if random number is below probability of repair
if (m < r[S + K + 1])
{
// Demand Machine is fixed
alpha[i][S + K + 1] = 1;

// Process Part Type y through Demand Machine
// Buffer for Part Type y before Demand Machine decreased
B[i][K+1] = B[i][K+1] - 1;

// Fill Production Matrix, Demand Machine working on Part Type y
prod[i][S+K+1][1] = 1;
}
else
{
// Demand Machine is still down
alpha[i][S + K + 1] = 0;
}
}
}

// Create array of counters to sum and average buffer size
double[] bufferSizes = new double[numBuffers+1];

// Loop through all buffers

```

```

for (int z = 1; z<= numBuffers; z++)
{
    // Initialize all buffers to be 0
    bufferSizes[z] = 0;

    // Loop through all the time steps in steadyStatePeriod
    for(i=transientPeriod; i<periodLength; i++)
    {
        // Add the buffer size in period i for buffer z to sum
        bufferSizes[z] += B[i][z];
    }

    // Calculate average buffer size by dividing sum by number of steadyStatePeriod
time steps
    bufferSizes[z] = bufferSizes[z]/steadyStatePeriod;
}

// Create array of counters to sum and average production rate
double[][] work = new double[numMachines+1][S+1];

// Initialize all counters to be zero
// Loop through all the machines
for(int step = 1; step<=numMachines; step++)
{
    // Loop through all the Part Types
    for(int count = 1; count<=S; count++)
    {
        // Initialize all counters to 0
        work[step][count] = 0;
    }
}

// Create output file to store results
System.out.print("Output file: ");
String outputFileName = console.next();

try
{
    // Create PrintWriter 'out' to send data to output file
    PrintWriter out = new PrintWriter(outputFileName);

    // Loop through all buffers
    for(int z=1; z<=numBuffers; z++)
    {
        // Print out average buffer size to output file
        out.println("Buffer " + z + " has average size: " + bufferSizes[z]);
    }
}

```

```

    }

    // Loop through all machines
    for(int k=1; k<=numMachines; k++)
    {
        // Loop through all parts
        for(int count=1; count<=S; count++)
        {
            // Loop through the steadyStatePeriod
            for (int step = transientPeriod; step<periodLength; step++)
            {
                // Sum the number of times Machine 'k' worked on part 'count'
                work[k][count] = work[k][count] + prod[step][k][count];
            }

            // Print out production rate of Part Type 'count' through Machine 'k'
            out.println("The probability that M" + k + " is working on Part " + count + "
is: " + (work[k][count]/periodLength));
        }
    }
    // Close output file
    out.close();
}

// if an exception is thrown, let user know there is an error with the file

catch(IOException exception)
{
    System.out.println("Error Processing File");
}

// M1 -> B1    B2 -> M4
//      -> M3 ->
// M2 -> B3    B4 -> M5
}
}

```


APPENDIX E: Input and Output Files

Program Verification Input and Output Files

Linear System Input File:

```
2
5
0.09269 0.01488 11
0.09970 0.00706 11
0.08071 0.00657 32 48
0.09546 0.01493 32 32
0.09837 0.01043 31 31
0.10036 0.00869 31 31
0.09972 0.01448 31 31
0.10049 0.07814
0.09771 0.07856
```

Linear Program Output File:

```
Buffer 1 has average size: 8.66486
Buffer 2 has average size: 24.84678
Buffer 3 has average size: 25.41018
Buffer 4 has average size: 25.91634
Buffer 5 has average size: 26.61992
Buffer 6 has average size: 26.49766
Buffer 7 has average size: 10.5957
Buffer 8 has average size: 35.93458
Buffer 9 has average size: 17.55844
Buffer 10 has average size: 15.6468
Buffer 11 has average size: 14.44988
Buffer 12 has average size: 2.67404
The probability that M1 is working on Part 1 is: 0.56176
The probability that M1 is working on Part 2 is: 0.0
The probability that M2 is working on Part 1 is: 0.0
The probability that M2 is working on Part 2 is: 0.22862
The probability that M3 is working on Part 1 is: 0.56182
The probability that M3 is working on Part 2 is: 0.22862
The probability that M4 is working on Part 1 is: 0.5616
The probability that M4 is working on Part 2 is: 0.22864
The probability that M5 is working on Part 1 is: 0.56166
The probability that M5 is working on Part 2 is: 0.22892
The probability that M6 is working on Part 1 is: 0.56164
The probability that M6 is working on Part 2 is: 0.22882
The probability that M7 is working on Part 1 is: 0.56162
The probability that M7 is working on Part 2 is: 0.22866
```

The probability that M8 is working on Part 1 is: 0.5616
 The probability that M8 is working on Part 2 is: 0.0
 The probability that M9 is working on Part 1 is: 0.0
 The probability that M9 is working on Part 2 is: 0.2287

Reference Multi-Part Simulation Output File:

| Name: | Avg: | Std: | 95% conf.int: |
|-----------------|-----------|----------|-----------------|
| Part.1.ProdRate | 0.560099 | 0.001297 | +/- 0.000464273 |
| Part.2.ProdRate | 0.248545 | 0.001477 | +/- 0.000528556 |
| Buffer.1.1 | 8.127161 | 0.027168 | +/- 0.00972181 |
| Buffer.2.1 | 27.480517 | 0.068778 | +/- 0.024612 |
| Buffer.3.1 | 26.731263 | 0.087106 | +/- 0.0311703 |
| Buffer.4.1 | 27.203810 | 0.067049 | +/- 0.0239932 |
| Buffer.5.1 | 27.920136 | 0.055365 | +/- 0.0198121 |
| Buffer.6.1 | 27.312571 | 0.057490 | +/- 0.0205727 |
| Buffer.1.2 | 10.322719 | 0.005149 | +/- 0.00184255 |
| Buffer.2.2 | 39.146440 | 0.165345 | +/- 0.0591681 |
| Buffer.3.2 | 15.969940 | 0.196340 | +/- 0.0702595 |
| Buffer.4.2 | 14.179274 | 0.192955 | +/- 0.069048 |
| Buffer.5.2 | 15.457897 | 0.187256 | +/- 0.0670085 |
| Buffer.6.2 | 3.355527 | 0.054975 | +/- 0.0196725 |

Basic Simulation Input and Output Files

Input File:

1
 3
 0.1 0.0 10
 0.1 0.0 10
 0.1 0.0 10
 0.1 0.0 10
 0.1 0.0

Output File:

Buffer 1 has average size: 1.0

Buffer 2 has average size: 1.0

Buffer 3 has average size: 1.0

Buffer 4 has average size: 1.0

The probability that M1 is working on Part 1 is: 1.0

The probability that M2 is working on Part 1 is: 1.0

The probability that M3 is working on Part 1 is: 1.0

The probability that M4 is working on Part 1 is: 1.0

The probability that M5 is working on Part 1 is: 1.0

REFERENCES

Gershwin, Stanley. *Manufacturing Systems Engineering*. Eaglewood Cliffs, NJ: Prentice Hall, 1994.