AN ARTIFICIAL NEURAL NETWORK AS A SIMULATION METAMODEL TO DETERMINE PRODUCTION PARAMETERS IN A SPECIALIZED MANUFACTURING SETTING

by

Kevin Conrad

Submitted in partial fulfillment of the requirements
for the degree of Master of Applied Science

at

Dalhousie University
Halifax, Nova Scotia
August 2015

Table of Contents

List of Tables

List of Figures

Abstract

The material handling system of a specialized production line is evaluated using a discrete-event simulation model. The object-oriented simulation model is built to allow changes to machine programmable logic, machine interaction effects, operator behaviours, and changeover policies. A second simulation model also allows for generalized production control parameters to be specified. The limitations of the current system are outlined and several opportunities for improving daily throughput are identified. To further understand and analyze the system, an artificial neural network simulation metamodel is developed and trained. To reduce the solution domain in training, a conditional Latin hypercube design is used. A strong network structure is identified through experimentation, and the performances of several training algorithms are compared. Finally, a simulated annealing algorithm is used with the trained and validated metamodel to determine reasonable production parameters for the system.

List of Abbreviations Used

ANN: Artificial Neural Network

AS/RS: Automated storage and retrieval system

BFGS: Broyden-Fletcher-Goldfarb-Shanno

BSS: Base Stock System

CONWIP: CONstant Work In Process

EA: Evolutionary Algorithm

ES: Evolutionary Strategy

FIFO: First In First Out

GA: Genetic Algorithm

GP: Genetic Programming

LHD: Latin Hypercube Design

LM: Levenberg-Marquardt

MHS: Material Handling System

MProd: Manufactured Product

MRP: Material Requirements Planning

MSE: Mean Squared Error

NEAT: NeuroEvolution of Augmenting Topologies

NSERC: Natural Sciences and Engineering Research Council

OT: Over-Training

PAC: Production Authorization Card

PA: Production Authorization

RPROP+: Resilient Backpropagation

SCG: Scaled Conjugate Gradient

TK: Transit-Keeping machine

VIMS: Visual Interactive Modelling System

Acknowledgements

First, I would like to express my gratitude to my advisor, Dr. Eldon Gunn, for taking me on as his student. His expanse of knowledge, experience, and enthusiasm has proven to be invaluable throughout this past year. His measured feedback and advice have repeatedly demonstrated his sharp intelligence and in time, elucidated his wisdom. Thank you.

I am also grateful to Dr. Corinne MacDonald, Dr. John Blake, and Dr. Malcolm Heywood for taking the time to review my work and provide valued feedback as part of my Thesis Committee. Your questions, suggestions, and guidance are appreciated.

I would also like to acknowledge the contributions of several employees of the manufacturer mentioned throughout the thesis. They have been excellent facilitators throughout this project and have always found time for my questions. Unfortunately, to preserve confidentiality, I cannot mention these people by name, but they know who they are. Thank you for being friendly, patient and extremely competent.

This research was supported by NSERC Engage Grant EGP 469286-14.

Chapter 1: Introduction

The genesis of this thesis was a project with a manufacturer aimed at exploring the role of simulation in the company. As discussed below, this evolved into the detailed simulation of one particular line within the company. The line is complex and we chose to implement the simulation using the flexible, open source, and process-based discrete-event simulation language SimPy, a package based on the Python language (Python Software Foundation, 2015). A detailed description of this simulation model and a demonstration of its capabilities form a major part of this thesis. Once the project with the manufacturer was complete, we decided to investigate if this simulation could serve as the basis for analysis of production control procedures on this line. The framework we chose for this investigation was the Production Authorization Card (PAC) framework developed by MacDonald and Gunn (2011) based on the original work by Buzacott and Shanthikumar (1992). This involved extensive simulations, the design and creation of a simulation metamodel, the use of neural network techniques to fit several metamodels, and the use of a chosen metamodel with an optimization procedure to demonstrate the concept of optimizing the parameters of the implemented PAC System. Our implementation of the PAC System does not rigorously adhere to the MacDonald and Gunn framework but it does demonstrate the concepts on a real production line.

1.1 Background on the Manufacturer

In the summer of 2014, a manufacturer was investigating the feasibility of a new trolley configuration for one of the production lines at their manufacturing facility. The manufacturer makes a highly specialized product and does not wish for details of the process to be revealed. For this reason, products are referred to as MProds (manufactured products) throughout the thesis. There were some concerns that the current serial trolley configuration of the material handling system (MHS) for the production line could not meet the anticipated future demand. Dalhousie University was asked to evaluate the MHS to determine its capability for five future product line configurations.

The MHS consists of several overhead monorail trolleys, which are used to transfer MProds from a group of MProd-building machines to a specialized automated storage

and retrieval system (AS/RS), henceforth referred to as the Transit-Keeping machine (TK). The MHS then transfers MProds from the TK to one of the 30 curing machines that are part of the production line. Fig. 1 shows how MProds begin construction on MProd-building machines in group A, and then are sent to a paired machine in group B. Once construction is complete in machine group B, an overhead trolley (trolley A) transfers the MProd to a weight scale. If the MProd is within weight tolerances, a second overhead trolley (trolley B) transfers the MProd to the input of the TK. When a MProd is required by a curing machine in group C, one of three overhead trolleys (E, F, or P) transfers the MProd from the output of the TK to a staging area called pre-cure. There are three parallel pre-cure areas, which serve as staging areas for three parallel groups of 10 curing machines. When a MProd is required from one of the pre-cure areas, the appropriate trolley (E, F, or P) will transfer the MProd to a ground trolley aligned with the nearest curing machine, referred to as a chariot, which facilitates the exchange of MProds with the curing machines. The three chariots are not depicted in Fig. 1.

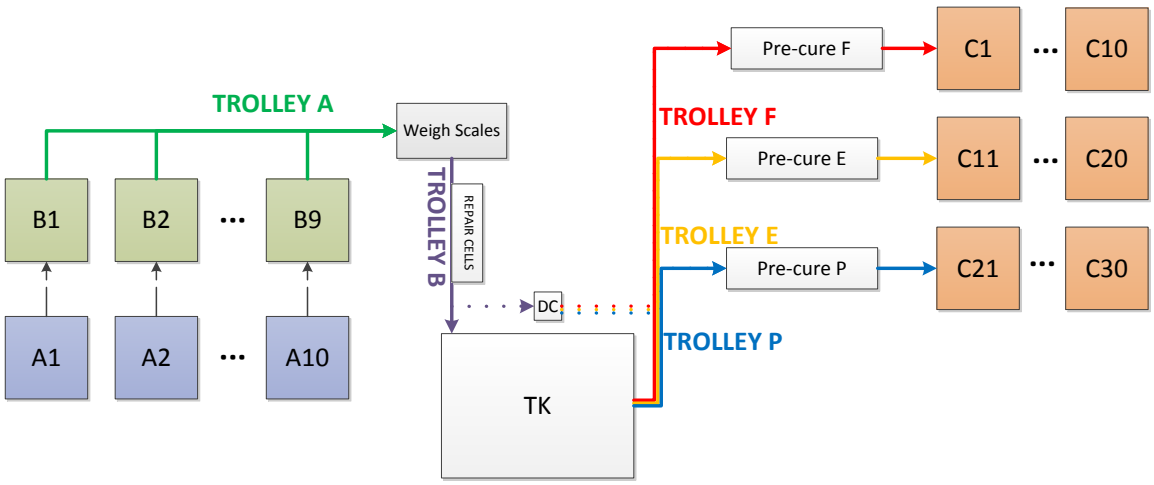

Figure 1: The production line

Several days of system observation and discussion with operators revealed that the operation of the MHS was complex, due to interactions between machines, the diversity of possible product mixes, and the potential and unknown impact of operator behaviour on the system. Further, since a detailed and accurate analysis was desired by the

company, it was decided that a discrete-event simulation model was the best approach for analysing the capability of the production line.

At the beginning of the project, trolley B and the TK were identified by experts at the company as possible bottlenecks for the production line. The TK was of particular interest due to the complexity of its movements, its interactions with other machines, and its programmable logic. Early simulations of the MHS indicated that trolley B and the TK would both operate at very high utilization under anticipated future demand. However, as development of the simulation model continued, several other issues in the production line became manifest. Operator behaviour and decision-making in MProd-building, the repair areas, and curing, as well as poor production control between curing and MProd-building were shown through simulation to cause variance in the system which leads to significantly lower system throughput. Simulation results indicated that although the MHS in the production line is capable of handling anticipated future demand under simulated conditions, the high utilization of the TK for some product mixes is a risk in the real system. In short, results showed that it is necessary to mitigate or eliminate the sources of variation elsewhere in the production line to ensure that the whole system is capable of meeting future production targets.

As indicated above, there were two closely related simulation models developed in Python to model system behaviours for this production line. The first simulation model, aimed at meeting the needs of the manufacturer, is used to determine the capability of the MHS for five anticipated future system configurations. A production control system is not included. The second simulation model incorporates the generalized production control scheme and a more sophisticated MProd-building process. Its purpose is to assess the impact of various production controls on the production line. An artificial neural network (ANN) is used to model the second simulation, and the ANN is used as the objective function of a simulated annealing algorithm to determine production control parameters. The methodologies and results of the computational experiments undertaken are described in detail in this thesis and provide a framework for future analysis of complex systems using artificial neural network simulation metamodels.

1.2 Thesis Outline

In Chapter 2, background methodologies are explained, providing a foundation for the modelling work performed in later chapters. First, discrete-event simulation is explained and a brief historical perspective is given on simulation. In particular, the merits of object-oriented design are examined. Chapter 2 also includes a discussion on the optimization of production coordination parameters, and describes Buzacott and Shanthikumar's generalized production control system, the production authorization card (PAC) System in detail. Finally, background information is given on simulation metamodelling and artificial neural networks (ANNs), including a brief overview of ANN training techniques.

In Chapter 3, the production line and the main simulation model are described in detail, including a description of the model classes and modifiable parameters. The data used in the model is also described. Statistically significant results and findings from the simulation model are given and represented visually, when possible. The simulation results show that the production line is capable of handling anticipated future production requirements without changes to the layout. However, the results also indicate that operator behaviours, changeover policies, production control policies, and stock levels have a significant impact on average daily throughput.

In Chapter 4, the second, modified simulation model is developed and used to investigate the effects of a production control system on the production line. First, the effect of structure on the performance of an artificial neural network (ANN) is examined in the development of a metamodel of the second simulation. To design a set of training experiments in Chapter 4, the ANN training set is derived using a Latin-hypercube design. The performance comparison of different network structures uses mean squared error (MSE), overtraining, and computational time as its primary measures. Ultimately, an ANN structure with two hidden layers is found to outperform a similarly complex structure with a single hidden layer. Six ANN training algorithms are compared for the two hidden layer structure, and the Levenburg-Marquardt algorithm is shown to train networks to have lower MSE than networks trained by other algorithms. Finally, an

approach to determining reasonable production parameters using a simulated annealing search is outlined.

Chapter 2: Literature Review on Background Methodologies

During the development of the first simulation model, research was focused on prior approaches to production line modelling and optimization. Historical approaches to simulation modelling were examined and the principles of object-oriented programming were also considered. As work with the first simulation model was nearing completion, the research became more focused on the issues of line control. After the development of the second simulation model, ANN training algorithms and experiment design techniques were researched to support the development of an ANN metamodelling framework.

2.1 Discrete-Event Simulation

Discrete-event simulation modelling is one of the two popular simulation approaches for operations research analysis of processes, the other approach being system dynamics (Beaumont and Pidd 1984). Often, either one of the two approaches are a valid simulation approach, although the fundamentals of each approach differs significantly. Discrete-event simulation represents entities in the model as they move through queues and processes at discrete moments in time, while in system dynamics, systems are modelled as stocks and flows in continuous time (Tako and Robinson 2010). Due to these differences, discrete-event simulation has traditionally been used in manufacturing and in the service industry, while systems dynamics is more often used for conceptual or larger scale models, as are found in the fields of economics, supply chain, or environmental management (Tako and Robinson 2008).

2.1.1 Event Scheduling Approach to Discrete-Event Simulation Modelling

In discrete-event simulation, the current state of the system can be described by a set of state variables, which change instantaneously as opposed to continuously (Law 2007). Discrete-event event simulation can be viewed as a list of events which are scheduled to occur at a set point in time. The simulation runs by advancing time to process the next scheduled event, which could trigger more events to be added to the event list. When the event list becomes empty, the simulation is over. The time at which state variables change values is determined using the simulation clock. In discrete-event simulation, the

clock does not advance continuously like a clock in the real world. Instead, the clock jumps forward to the next moment in time that an event is scheduled to occur on the event list (i.e. whenever the state of the system changes). The event is then processed, state variables may change, additional events are scheduled on the event list, and then the clock advances to the next scheduled event. For example, imagine a discrete-event simulation model of a car wash. An event could be the beginning of the first soap cycle, which takes 2 minutes to complete. The discrete event simulation model will process the first event, which begins the soap cycle, and change a state variable to record that the soap cycle is underway. A second event will be scheduled on the event list for 2 minutes in the future, when the soap cycle is going to be finished. Then, assuming that the completion of the soap cycle is the next event in the event list, the model will instantaneously advance the clock by 2 minutes. Scheduling events using an event list and using the event list to advance to discrete moments in time on the simulation clock allows discrete-event simulation to be very efficient. No processing time is wasted stepping through time when there are no events to process.

Sometimes it is useful to limit how many entities can be served at one time. In discrete-event simulation, it is possible to define several types of finite resources which serve this purpose. In the car wash example, if there are only two car wash stations, perhaps it would be convenient to define a resource with a capacity of two, so that only two cars can use the car wash at one time in the model. Other resource types include continuous resources, and resources with non-FIFO (first in first out) queues.

One of the advantages of discrete-event simulation modelling is that it can account for randomness in processes. Returning to the car wash example: in the real world, the soap cycle will not take exactly 2 minutes every time. The process time may depend on the car wash operator, or the process may have some random variation. The variation can be modelled using an appropriately chosen statistical distribution. A random number generator is used to sample from the distribution and return a process time. For the car wash example, this means that for one car, the soap cycle may take 2.1 minutes, while for the next car, the cycle may only take 1.8 minutes. Models which account for probabilistic

process in this manner are referred to as stochastic simulation models. Conversely, a simulation model without any probabilistic processes is referred to as a deterministic simulation model. Most discrete-event simulation models are stochastic models.

In discrete event-simulation, output measures and other statistics are recorded using statistical counters throughout the simulation model. Due to the stochastic nature of many discrete-event simulation models, the output measures are random variables. Thus, multiple runs, or replications of the model are required to construct confidence intervals on each statistic. It is therefore necessary to determine when to terminate one replication and begin another. One termination condition could be an empty event list. For example, if there is a simulation model of a car wash, and the car wash closes for the day, no more cars will arrive and the event list becomes empty. This termination condition works well for some models, but for others, such as a model of a 24-hour production system, it is necessary to specify an end time, which can be done by scheduling an end-of-simulation event at a time where enough data has been collected for that replication. There is a problem that can arise with systems that operate at a steady state, such as a 24-hour production system. When a replication is started, in most cases the system will not be at a steady state. In these cases, it is desirable to begin recording statistics only once the transient period is complete and steady state has been reached. There are various techniques for identifying the end of the transient period, also referred to as the warm-up period. For further discussion of discrete-event simulation modelling techniques, see Law (2007).

2.1.2 The Process Approach to Discrete-Event Simulation Modelling

The definition of discrete-event simulation using the event list and time advancement techniques as developed in Chapter 2.1.1 is a traditional approach to simulation modelling, referred to as the event-scheduling approach to simulation modelling. An alternative approach to simulation modelling is the process approach, which "views the simulation in terms of the individual entities involved" (Law 2007). Using the process approach, process and entities are defined as classes in the simulation model. In computer programming, a class is a programmed template for creating objects with similar

behaviours (Gamma 1995). Each object created using the class, commonly referred to as an instance of the class, may contain different parameter values which differentiate it from other instances of the same class, but it will contain the same functionality and attributes as the other instances. In simulation modelling, separate instances of each process class are defined as needed to structure the model, and instances of entity classes flow through the system and interact with other objects. For example, in a discrete-event simulation model of a manufacturing facility, such as the one introduced in Chapter 1, there is a class for curing machines. While each curing machine is different, they share the same basic attributes and functionality. An instance of the curing machine class is created for each curing machine which assigns its parameter values to differentiate it from the other curing machine class instances. Machines A and B are specified in a similar manner. Then, entities are created by an arrivals process, flow through the system and are processed by objects. A simulation model using the process approach can be viewed as a collection of objects which all operate within their own specified functions, yet interact with each other by exchanging entities and sending and receiving signals. However, even when modelling a simulation with the process approach, the simulation operates behind the scenes using the same mechanisms as the event-scheduling approach.

2.1.3 Object-Oriented Programming for Simulation Modelling

Object-oriented programming is an approach to computer programming where programs are composed of objects that interact with one another (Kindler and Krivy 2011). Recall that objects are instances of classes which can be composed of variables, functions, or data structures. Robinson (2005) advocates object-oriented programming approaches to keep code simple and interchangeable. It is clear that well-designed objects lead to code which is easier to understand and maintain. Although there is no guarantee that an object-oriented model runs faster, an object-oriented model is cleaner, easier to modify, and easier for a third party to read and understand. An object-oriented design is required for the process approach to discrete-event simulation models. Eldabi and Paul (2005) describe four steps to developing an object-oriented simulation model. First, the programmer should begin by identifying and building the major components of the model separately, without worrying about the inner detail and complexity. This manifests itself

in the form of the objects and classes in the simulation model. At this stage, design decisions should be considered given the system being modelled. For example, for the production line introduced in Chapter 1, the programmer would need to decide whether to model trolley A, the weigh scales, and trolley B as one class with several functions, or as three separate classes. Step two is to assign the behaviour of each entity, without including unnecessary details that risk overcomplicating the model at the conceptual stage. However, it is also advisable to ensure that all of the functionality for each entity is included. The third step is to add the detailed logic for each entity. The final step is to fine-tune the model parameters to get the desired system performance (Eldabi and Paul 2005).

2.1.4 Advancements in Discrete-Event Simulation

Robinson (2005) presents a brief history of discrete-event simulation. Four distinct periods are identified. In the late 1950s and 1960s, the first simulation programs were written and the first simulation software was released. Of particular interest is Simula, as it is considered the first object-oriented programming language. Simula later inspired the development of several of the object-oriented languages, including C++. In the 1970s, simulation software continued to improve, and the concept of visual interactive simulation was proposed. By the 1980s, many different simulation packages, such as Arena, Flexsim, and AutoMod had been developed and released, predecessors of commonly used packages today (Robinson 2005). At this stage, hardware was sufficiently advanced that simulation models could be run for reasonably complex systems. As hardware continued to improve in the 1990s, visual interactive modelling improved and was part of many simulation packages. With a visual interactive modelling system (VIMS), the user could create animations and develop the model visually. This meant that the user did not necessarily need to be a strong programmer to develop a simulation model, and could often create a simple model by dragging and dropping pre-constructed objects. Of course, models with complexity still require programming for most packages.

Introduced in the 1990s, simulation optimization is recognized as a significant step forward for discrete-event simulation for process design. Simulation optimization is the process of finding the combination of input variable values that gives the highest expected value for the model performance measures, without necessarily running the simulation model for each possible combination of inputs (Carson and Maria 1997). There are several approaches to simulation optimization, including gradient based search methods, stochastic optimization approaches, response surface methodologies, and heuristic approaches. Gosavi (2013) examines the various approaches to simulation optimization in detail. Response surface methodologies are further explored in Chapter 2.3.

There are dozens of discrete-event simulation packages available today, including open source packages in many common programming languages. Generally, commercial software offers enhanced features, such as a visual development environment, or built in tools for simulation optimization and animation. Swain (2013) has performed a survey of modern simulation packages to compare the strengths and weaknesses.

The modern, open source simulation package SimPy (*Welcome to SimPy*, 2014) is based on ideas from Simula and SIMSCRIPT, both of which are very early simulation packages from the 1960s (Nance 1993). For complex model components, it is often necessary to explicitly define certain functions even for VIMS packages. When there are many complex components which must be explicitly defined, it may be better to explicitly define the entire model. In SimPy, all functions must be defined by the programmer in the Python programming language. SimPy also offers a process-based approach to discrete-event modelling, which is advantageous given the object-oriented nature of the production system that is being modelled for this project. The built-in synchronization tools in SimPy are particularly useful for some of the interactions that must be replicated as part of the programmable logic in the system. In addition, SimPy is open source, which offers a considerable advantage in the context of this project because it is free, the project sponsor can distribute the model at will, and can modify behaviours of the model. Finally, SimPy is written in Python, so other open source packages can be integrated with

the simulation model to provide additional functionality, including an animation, a user interface, plots, and statistical analysis tools.

2.2 Optimization of Coordination System Parameters in Manufacturing

There have been many systems proposed for controlling and coordinating production. However, there is no guarantee that implementing a given system will be optimal for the manufacturing process under consideration. To address this issue, Buzacott and Shanthikumar (1992) propose a generalized approach to production coordination. The system, called the Production Authorization Card (PAC) Coordination System, provides the framework to achieve optimal coordination for all multiple-cell production systems. While there are other coordination systems which have been proven optimal or near-optimal for certain production systems, the PAC System is a notable approach because it does not require the designer to select a coordination scheme *a priori*. Buzacott and Shanthikumar point out that none of the coordination systems which currently exist actually provide a basic framework to compare them to other coordination schemes, or an approach to selecting the best coordination scheme for a given manufacturing system. The PAC System provides a generalized approach to this problem such that common coordination schemes are all special cases of the PAC System which can be derived with an appropriate choice of PAC parameters.

The PAC Coordination System works for a multiple-cell manufacturing system. This means that the manufacturing system must be modelled as a series of alternating product stores and cells, beginning with a product store which represents suppliers and ending with a product store which represents the customer. Cells send requisition tags to stores, informing stores that they require a certain part as soon as possible. Usually, the cell will have sent an order tag to the product store, informing them that they will be requisitioning a part in the future. If there is no advance warning, the cell sends the order and requisition tags together. Upon receiving an order tag from a cell, a product store can send a Production Authorization (PA) card to the cells that supply the parts that will be needed to meet the order. Sometimes, other conditions need to be met before sending a

PA card, such as waiting for a certain amount of time before sending the PA card or waiting until the number of PA cards issued for the product is below the maximum.

Order tags can be cancelled using cancellation notes. Requisition tags turn into material tags on arrival at the cell which requisitioned the part. A material tag also accompanies each part as it reaches the store. It serves as an identifier for the part, and knowing the number of material tags in a cell means that the inventory level is known for each part. Surplus tags wait with excess products at stores until they are cancelled with the next requisition tags. Finally, process tags are used to authorize material movement and part processing, and are generated in the cell upon the arrival of the part. The system is best explained through the use of cards and tags, but can be implemented electronically, and also easily simulated.

The PAC Coordination System has several parameters which can be adjusted to achieve an optimal coordination scheme. First, the requisition delay between a store receiving an order tag and its corresponding requisition tag can be adjusted. Second, the static inventory limits for each product store can be set at each store for each product. This is equivalent to the number of material tags at each store given that there has been no demand for a long time. This number has to be zero or positive. Third, Buzacott and Shanthikumar (1992) suggest that it is possible to introduce a delay between the product store receiving an order tag and generating a PA card. This is generally not used in any standard schemes. Fourth, there can be a limit imposed on the total number of PA cards issued for a given product at a given cell. This allows for more specific limits on work-in-process at the cell. Finally, restrictions can be imposed on the distribution of PA cards and requisition tags by creating a minimum and/or maximum batch size for each. For requisition tags, it is also possible to impose the restriction that all of the requisition tags in a batch must be fulfilled and sent back to the cell as a batch, rather than individually.

Buzacott and Shanthikumar (1992) describe under which parameter values the PAC Coordination System behaves like a make-to-order system, a base stock system (BSS), a material requirements planning controlled system, a locally controlled system, an integral

control system, and other common control schemes. However, they do not describe how to achieve optimal parameter values in the PAC Coordination System.

MacDonald and Gunn (2011), however, provide a framework for the analysis and parameter selection of the PAC System. MacDonald and Gunn begin by explaining that a discrete-event simulation model is necessary to measure the system performance of a complex, real-world production line. Using the simulation model, the paper suggests developing neural network metamodels as an approach to optimization. In the context of the PAC System, the neural network metamodel would be a surface which represents the value of a single performance measure as a function of all of the PAC parameters required for the production line. A metamodel is created for each performance measure. Exchange curves, representing the optimal tradeoffs can then be created for any pair of performance measures to make policy decisions.

To begin analyzing the system under study for the selection of optimal PAC parameters, it is important to define a domain within which the optimal value for each parameter will exist. Reducing the domain by eliminating redundant, inefficient, or unstable values reduces the number of simulation runs needed to construct the neural network metamodel later on. There are two main parameters which should be considered for each product store. There is k, the number of process tags at each store, and z, the initial (and maximum) inventory at each store. Other parameters like delays and batching could also be considered from Buzacott and Shanthikumar (1992).

2.3 Artificial Neural Networks as Simulation Metamodels

Simulation metamodels are mathematical functions that approximate the relationships between the key input parameters of a particular simulation model to the expected value of performance measures and outcomes (Friedman 1997). Approximate expected response models, or metamodels, provide a simpler, surrogate model to the simulation which can be evaluated much more quickly than a full simulation replication. They can be used as an aid in simulation optimization (Hurrion 1997), or as an operational tool for management to quickly understand the implications of any production decision which has

been included as an input variable to the model. Simulation metamodels have been applied in several fields, including risk analysis (Badiru and Sieger 1996), manufacturing systems design (Chen and Yang 2002), supply chain analysis (Cigolini et al. 2011), and production control (Kuo et al. 2007).

Wang and Shan (2007) describe several techniques for metamodelling. Polynomial regression, stochastic Kriging, multivariate adaptive regression splines, radial basis functions, support vector regression, artificial neural networks (ANNs), and genetic programming (GP) have all been used to create approximate response models. Polynomial regression has been shown to lose accuracy as the number of decision variables increases, unless it is based on a theoretical model (Durieux and Pierreval 2004). ANNs are the predominant approach in simulation metamodelling since they do not require any underlying assumptions or knowledge about the relationships between input parameters and performance measures, and they are generally more efficient than other approaches (Can and Heavey 2011). Computational experiments have been carried out by Sabuncuoglu and Touhami (2010) which indicate that ANNs are effective metamodels for determining the system performance of a job shop scheduling simulation model. Hurrion and Birgil (1999) showed that ANNs provided more accurate predictions more efficiently than regression metamodels. ANN simulation metamodels are used as part of a framework for optimal production control (MacDonald and Gunn 2011). For a discrete-event simulation model of an automated material handling system with 23 input parameters, an ANN was chosen as the metamodel (Kuo et al. 2007). Although there is some evidence that GP can generalize better than ANNs in some cases (Can and Heavey 2011), using ANNs appears to be a strong approach for metamodelling complex discrete-event simulations.

ANNs are structures of connected nodes and neurons, which are inspired by a simple analogy of the brain (Illingworth 1989). There are three layers of nodes: the input layer, where input parameters X_i enter the network, the hidden layer, where inputs are weighted, summed, and transformed at each neuron, and the output layer (see Fig. 2).

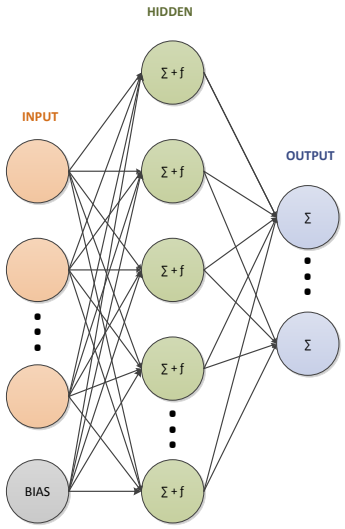

Figure 2: An ANN with a single hidden layer

At each neuron j in the hidden layer, there is an adaptive weight (W_{ij}). The adaptive weights are used to multiply the value of each input node i before summing the total of all weighted input values. Then, a bias b_j is added to each value and the value is transformed by an activation function f which has been set *a priori*. If there are multiple hidden layers, the weighting, summing, and transforming process is repeated, except that the inputs to the next hidden layer are the final values from the previous hidden layer. The final values of each of the final hidden layer neurons are weighted and summed at each output node k. For a feed-forward (acyclic) ANN with a single hidden layer, if v_j is the value at each hidden neuron j:

$$v_j = f\left(b_j + \sum_{\forall i} W_{ij} \times X_i\right) \qquad (1)$$

Then, the value of output node v_k is as follows (adapted from MacDonald and Gunn 2011):

$$v_k = \sum_{\forall j} W_{jk} \times v_j \qquad (2)$$

It is necessary to specify the number of neurons at each layer as well as the number of hidden layers *a priori*. In simulation metamodelling, the number of input nodes corresponds to the number of key parameters to be considered as part of the ANN. Similarly, the number of output nodes corresponds to the number of key measures in the simulation model. However, it is common practice to train a separate neural network for each performance measure, thus leaving a single output node (Barton 1998). There is no established procedure to specify number of hidden layers, as well as the number of neurons per layer. The iterative process of modifying the weights and biases of an ANN is referred to as a training algorithm (Hagan et al. 1996). The training algorithm uses a set of training data for which each sample in the set specifies the target system output for a full set of input parameter values. Supervised learning then takes place, in which the weights and biases are adjusted to improve the mean squared error of the ANN output relative to the target output. The entire training dataset is examined each training run, with the goal of minimizing the difference between the value at the output of the neural network and the expected output from the training dataset.

Multi-layer networks are more powerful than single-layer networks, and can be trained to approximate functions arbitrarily well (Hagan et al. 1996). However, if the ANN becomes too complex, it risks overtraining, which will limit its ability to accurately represent inputs which were not present in the training set (Sabuncuoglu and Touhami 2010). This is an important consideration in simulation metamodelling, since any useful application of the metamodel depends on its ability to generalize; that is to represent the entire input space reasonably well, not just the training set.

There are several classes of training algorithms for ANNs, which differ principally in the methodology used to modify the weights. Gradient-based training algorithms are "probably the most famous iterative methods for efficiently training neural networks in scientific and engineering computation" (Livieris and Pintelas 2013). Most gradient-based methods require a learning rate L to be specified before training begins. The learning rate modifies how quickly weights will converge in the direction of gradient g_n, where a weight at run n, W_n is updated using the following equation:

$$W_n = W_{n-1} - L \times g_n \qquad (3)$$

The neural network training process can be treated as an unconstrained large scale minimization problem. From this perspective, it is easy to see that the gradient descent algorithm in Eq. (3) is just the standard steepest descent algorithm, where the learning rate represents the step size. Although a higher learning rate, or step size, will cause error to reach a minimum more quickly, higher learning rates also decrease the likelihood that the training algorithm will find some minima (Mukherjee and Routroy 2012). Smaller learning rates will take longer to converge, but are likely to produce better results. For some algorithms, the learning rate changes as training progresses in order to produce convergence.

For the classic gradient-based algorithm, backpropagation, the sign and magnitude of the gradient are used to calculate the weights in the current period. Some other variations of gradient-based propagation algorithms such as Manhattan Propagation and Resilient Propagation (RPROP+) only use the sign of the gradient. The method of calculating the magnitude of the change to the weights varies between algorithms, but the key idea behind these approaches is that they are resistant to making changes to the weights which are undesirably large. For example, RPROP+ individually tracks magnitudes for each weight value in the ANN. The gradient is used to determine how each of the magnitudes should change over time.

Another way to resist large changes to the weights is to add a momentum coefficient M to Eq. (3) which helps to smooth the oscillation of weight values:

$$W_n = M \times W_{n-1} - L \times g_n \qquad (4)$$

As the value of the momentum coefficient approaches one, the adaptability of the ANN training is slowed significantly, while low or zero values of the momentum coefficient can cause instability. One sensible way to select a value for the momentum parameter

could then be to start near zero and let the momentum increase towards one as training progresses.

The quasi-Newton class of algorithms, inspired by Newton's method, uses the gradient to approximate derivatives and update the weights. There are several hybrid training algorithms which resemble gradient-based approaches, such as Quickprop and the Levenberg-Marquardt (LM) algorithm. The LM algorithm uses a combination of first-order and second-order methods which has been shown to converge in fewer iterations than backpropagation (McLoone and Irwin 1999). Algorithms such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm use true quasi-Newton approaches. The BFGS algorithm and its various adaptations use second-order derivative methods which at each iteration compute approximations to the Hessian matrix of size A x A, where A is the total number of weights and biases. Although these algorithms converge in fewer iterations than gradient-based algorithms, they can be memory inefficient for large ANNs (Mukherjee and Routroy 2012). The Scaled Conjugate Gradient (SCG) algorithm also uses an approximation to the second order derivative, but avoids the line search of the BFGS algorithm by using an approach similar to the LM algorithm (Moller 1993). The SCG algorithm is considered to be part of the conjugate gradient family of algorithms. Generalized Barzilai-Borwein algorithm (GBB) for unconstrained minimization problems avoids most of the expensive line search of the BFGS algorithm, while also incorporating a changing step length which is calculated at each iteration (Raydan 1997). Likas and Stafylopatis (2000) and McLoone and Irwin (1999) provide thorough derivations of several second-order quasi-Newton training algorithms.

Although second-order based algorithms have been shown to have a computational advantage over gradient-based algorithms for some problems, neither class of algorithm can guarantee that a global minimum error will be achieved.

Since neither gradient-based and quasi-Newton approaches can guarantee achieving a global minimum, evolutionary algorithms (EAs) have been explored as a means to overcome this limitation. Genetic algorithm (GA) based implementations of training

algorithms which did not use gradient information were examined by Mandischer (2002). He found that evolutionary strategies (ESs) could only compete with gradient-based approaches for small problems. While ESs without gradient information were useful for small problems or problems using activation functions which were non-differentiable, in general they took much longer and were less reliable than other approaches (Mandischer 2002).

One inventive ES is the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, which evolves both the weights and the topology of the ANN during training (Stanley and Miikkulainen 2002). The NEAT algorithm begins with a minimal ANN structure, then grows incrementally, using crossover and mutation to innovate. The incremental growth of topology mechanism is convenient because it removes the necessity of specifying the number of hidden layers and hidden nodes. Similarly to the ideas behind compositional GA (Watson and Pollack 2003), the NEAT algorithm improves solutions and generates more complex solutions simultaneously. The NEAT algorithm also has the ability to restart when it reaches a local minimum. Experimental comparisons have shown that the NEAT algorithm is faster and reaches the error threshold more frequently than other ESs (Stanley and Miikkulainen 2002).

Chapter 3: Development of the Simulation Model

3.1 Detailed Description of the Production Line

The production line begins in the preparation (prep) department. Here, base product is prepared for later stages of manufacturing. There are hundreds of different products produced in prep, providing all of the raw materials for the four principal production lines in the plant.

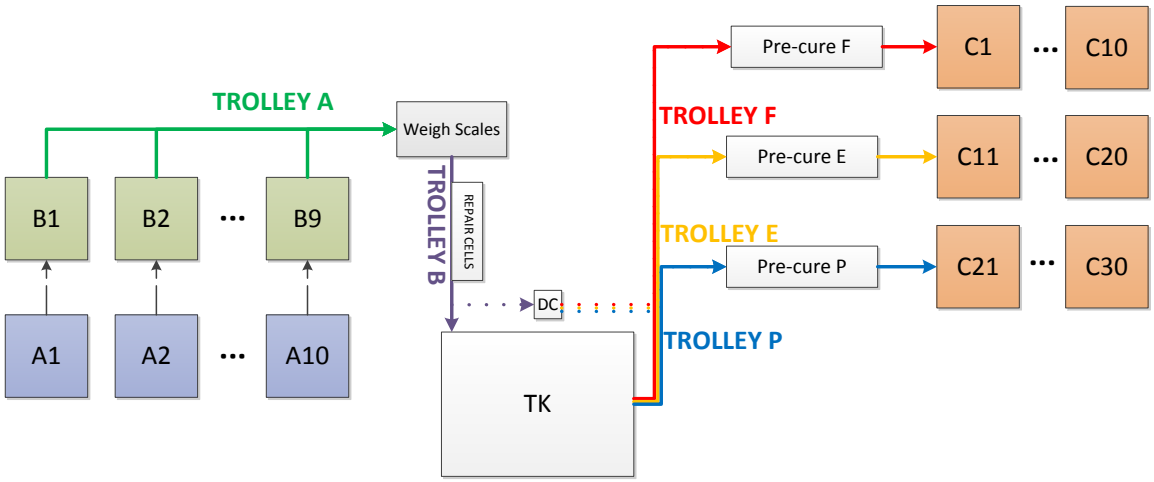

Figure 3: (reproduced) The production line

For the reader's convenience, the diagram in Chapter 1 is reproduced here for the production line under study. This line separates from the other three production lines at the MProd-building stage of production. There are two sets of MProd-building machines, where each machine in the first set is paired with another in the second set. At the first set, machine group A, an operator constructs the base structure of the MProd from base products. The MProd at this stage is referred to as a carcass. The semi-manual construction process takes 20-25 minutes per carcass on most of the machines in group A, although there is some variation in the level of machine assistance. There are ten machines in machine group A. All but one of the machines in group A can only construct one carcass at a time. However, there is one newer machine which can construct several carcasses simultaneously. This machine has a much higher rate of production than the other machines in group A, and is paired with two B machines as a result. After the carcass is constructed, further construction operations take place on the second set of machines, machine group B. Here, a different operator adds additional base product, and

reshapes the MProd. This semi-manual process requires an operator for the majority of the 10-15 minutes required per MProd. However, there is a stage in the process where the operator has 2-3 minutes of available time while the machine processes the carcass. After the process is completed, the MProd is now called a cover. Carcasses are transferred between A and B machines via a rotating storage point, called a tree. There is a tree between each A and B machine. After the processes at machine group B are complete, the covers are transferred to a tipper, which re-orients the cover for pickup from an overhead monorail trolley called trolley A.

Trolley A services all of the machines in machine group B first-come-first-serve, unless a specific instance of a cover has been manually configured to have priority over the other covers. Trolley A transports covers along a monorail to one of two available weigh scales. When Trolley A is within a certain distance of the weigh scales, trolley B cannot enter the area. In the simulation model, the presence of a trolley in the weigh scales area is tracked using a single variable. When the variable value changes from occupied to available, the trolley that is leaving the area sends a signal to the other trolley, alerting it that the variable has changed. When trolley A is not present, Trolley B picks up covers from the weigh scales and deposits good covers into the configured input slot of the TK. While trolley B is at the weigh scales, trolley A cannot enter the area. If the MProd was not within weight tolerance limits, or if the MProd-building operator designated the cover as defective at the MProd-building stage, the cover will not be dropped in the TK input. Defective covers are instead deposited into a repair area which has space for up to six covers without any operator intervention. If all six repair slots are filled, and another defective cover arrives, the cover will remain in the weigh scale area, reducing the weigh scales to one available slot. If a second defective cover arrives, the weigh scale area will become full, and the MHS will cease to operate until an operator intervention clears the repair area. Ideally, the repair operator empties the repair area before it becomes full, although there is no formal notification system to alert the repair operator when the repair area becomes full. When the repair operator arrives to intervene, trolley B is called upon to pick up and move covers to a tipper, where they can be repaired and sent to the TK, removed from the system for a future repair, or scrapped. Sometimes, if there are many

covers to be repaired, the TK is used to store defective covers temporarily (see Chapter 3.3.3 for how the repair circuit was modelled).

The TK operates on an internal monorail. It has a separate input and output and can store up to 124 covers. Although a cover is eligible to be stored in any empty location, TK pods are required to store and retrieve covers. A TK pod is a pallet which is designed to hold MProds so that they can be easily picked up and put down by the TK. Due to the requirement that an empty TK pod be present for trolley B to drop a cover into the TK input, there is the possibility of some delays to trolley B if it needs to wait for the TK to bring an empty TK pod. The output is slightly more complex than the input. The output occupies two slots, and it has the ability to act independently from the main TK trolley. For instance, if the TK drops a TK pod with a cover on the upper output slot for pickup from an external monorail trolley, then after the cover has been picked up, the output mechanism will relocate the empty TK pod to the bottom output slot for pick-up by the TK later. This mechanism has the advantage of freeing the upper section of the output for another TK pod and cover drop off much more quickly than if the output was static. The TK has four possible actions, in order of priority:

1. Retrieve TK pod with cover from storage slot and place on the upper output slot
2. Retrieve TK pod with cover from input and store in an empty slot
3. Retrieve empty TK pod from storage slot and place on the input
4. Retrieve empty TK pod from lower output slot and store in an empty slot

If, when executing action 3, there is an empty TK pod on the lower output slot, the TK will retrieve that TK pod and move it directly to the input. This will reduce the time required for a complete TK cycle. When the TK is highly utilized, a typical cycle will be actions 2-1-(4-3) and repeat, where actions 4 and 3 are combined as described.

When the TK places an empty TK pod on the input slot, it checks for a signal from trolley B that another cover is on the way. If trolley B has sent the signal that it is on the way with another cover, then the TK will wait 20 seconds for trolley B to arrive directly

above the input and begin depositing the cover. If trolley B has not arrived within 20 seconds, the TK will perform the next action on its list. Many times, the TK was observed leaving just a second or two before trolley B would have arrived. This is inefficient for several reasons: the TK was idle for 20 seconds unnecessarily, and the sped-up cycle, for which actions 4 and 3 are combined, is often interrupted for several cycles.

When the TK is performing action 1, it will select the next product code that will be required in curing once a pre-cure slot becomes available for the appropriate group of ten curing machines (E, F, or P). Further, it will always select the cover which fits the product code requirement and which has been stored for the longest period of time. The purpose of this rule is to help prevent occurrences where a cover is stored in the TK for a very long period of time and becomes structurally asymmetric due to sagging from its own weight, necessitating a repair. However, frequently there are covers that fit the product code requirement which would necessitate a much shorter cycle time. All of the TK programmable logic control code can be modified, most of it without much difficulty.

As alluded to above, covers are cured by one of 30 curing machines in machine group C arranged in parallel rows of ten called E line, F line, and P line. After curing, MProds exit the system to cool. After the TK places a cover on the output and the appropriate trolley (E, F, or P) retrieves the cover and places it in a staging area called pre-cure. Although only one trolley can access the TK output at one time, there are three parallel pre-cure areas, each of which has storage space for six covers. Trolley E services pre-cure area E and is also responsible for transporting covers to the E line of curing machines when requested. Similarly, trolleys F and P also service their respective pre-cure areas and curing machines. The TK will try to output covers such that all three pre-cure areas are full, with the next six covers to be requested by the next six available curing machines.

There is an alternate route to pre-cure which allows some covers to bypass the TK (see DC on Fig. 1). If this option is active, then trolley B will sometimes drop a cover into a direct cell outside of the TK, where it can be picked up by trolley E, F, or P. Generally,

this is only used when a shortage has occurred and MProds must be rushed to curing, but it is an option which could reduce TK cycles if configured correctly.

At a constant time interval before a curing machine will require a cover in pre-cure, the curing machine sends a request to its trolley (E, F, or P) to bring the cover from the pre-cure area to a small monorail transporter called a curing chariot. The curing chariot brings the cover to the appropriate curing machine. When the curing machine opens, there is an exchange of cover for cured MProd. The chariot does not need to wait for the operator to perform its part in the exchange. Automated curing machines, such as those in line P, perform the entire exchange without an operator intervention. Most of the curing machines in lines E and F require an operator to place a small metallic identifier inside the curing machine before approving the transfer of the new cover from a holding device into the curing machine. This process is a legislative requirement that only applies to MProds being sold to certain customers. Press operators tend not to service curing machines immediately for several reasons which are discussed in Chapter 4, meaning that curing machines occasionally remain idle for short periods throughout the day. MProds usually take 50-100 minutes to cure, depending on their size, composition, and other characteristics. The cure time does not vary, and is a known length of time for each MProd.

The MProd production line can build a diverse mix of products. Usually, 8-20 different types of MProds are produced at any given time. Machines in groups A and B are capable of building two or three different products with fast changeover times between products under most circumstances. Curing machines, however, contain a mould for a specific product code. Changing moulds takes 4-6 hours and does not usually occur more than once per month for each curing machine. These changeovers are not considered as part of the simulation model.

3.2 Detailed Description of the Simulation Model

When designing the simulation model, there was some effort to adhere to object-oriented programming standards. The class TK found in Appendix B is an example of one of the

classes that adheres to an object-oriented design standard. Appendix G gives program flow charts for most of the major classes. This design allows the model to be more easily used for other related purposes in the future. The company wished to be able to use the simulation model to evaluate decisions, so a user interface was created which allows a non-programmer to change several parameters. This user interface was written using the Python open source package PyQt4; screen shots can be seen in Appendix C. To evaluate performance and learn about system behaviour, statistical tracking was included throughout almost all of the objects.

3.2.1 Exclusions

Although it is not difficult to include machine breakdowns in the model, they are excluded from the runs for which results are reported. This was done for several reasons. First, breakdowns are rare on the production line, accounting for less than 1% of machine time for most machines. Second, the alternatives being considered for the MHS will have similar or worse reliability than the current system, so this assumption does not bias the results. Third, data was not readily available for breakdowns due to their infrequent occurrences and the lack of a standardized tracking process. This made it very difficult to determine a reasonable statistical distribution for breakdowns.

Machines in group C have a start-up time if the interior temperature drops below a certain value due to lack of use. The start-up time and the temperature threshold and drop-off rates depend on several variables, and distributions are not known for these values. Fortunately, machines in group C are usually highly utilized, so the probability of requiring a start-up time is very low. This start-up time was not included in the simulation model.

3.2.2 Structure

The simulation model is written using SimPy in Python. Every machine type is its own class. A list of classes in the model, along with a brief description of each is found in Table 1. MProds are modelled as tuples, which are passed between machines. A tuple is a Python object which behaves like an array of values. In Python terminology, tuples are

not mutable. This means that although changing a part of the tuple can be done, this requires a somewhat awkward series of steps. Contained in the tuple is the identifying information for the MProd. This consists of i) a serial number, ii) time of construction, iii) time of TK entry, and iv) product type. A better way to model MProds would have been as instances of a general MProd class. This would have simplified the modelling process, since MProds acquire new attributes as they travel through the production line, necessitating the program to rewrite and restructure tuples at multiple stages in the current implementation.

Table 1: List of classes in the simulation model

Class	Description
Animation	Contains the model animation.
MProd_Building_Operators	Contains MProd building operator parameters, including break times, shift changes, and other activity details (see Chapter 3.3.8).
Machine_A	Requires a resource res_Machine_A_Operator to construct carcasses as permitted/requested. Can be controlled using PAC System and changeover between products.
Machine_B	Requires a resource res_Machine_B_Operator to construct covers from carcasses on the tree. Can be controlled using PAC System and changeover between products.
Demand	As an alternative to using Machine_A and Machine_B, this class provides a high level of realistically distributed demand to trolley A.
g	Stores model configuration information.
PAC	Contains PAC parameters and settings which can be configured within the class and through the populate_initial_inventories() function.
DirectCell	Accepts MProds as a tuple from trolley B as an alternative path to pre-cure, bypassing the TK.
Repair_Operator	Repairs MProds in the repair areas and calls trolley B to move repaired MProds to the TK.
Inspector	The quality inspector occasionally finds imperfect covers in pre-cure and requests Trolley_EFP objects to place the imperfect covers in the reject cells for repair.
Trolley_B	Fetches MProds from the weight scales as a tuple and places the MProd on the TK input or in the repair area.

Class	Description
Trolley_A	Fetches MProds from tippers as a tuple and brings them to the weight scales.
Trolley_EFP	Fetches MProds from the TK output as a tuple and places them in pre-cure upon request from the Monorail_System. Fetches MProds from pre-cure and gives them to the curing chariot upon request from Curing_Machine objects managed through the Monorail_System.
TK	Performs 4 main tasks: Input jobs, output jobs, get empty TK pod for input, and put away output TK pod. There are parallel operators in this class. Task priority can be set by the user. Output jobs are served in a queue of requests from Curing_Machine objects managed through the Monorail_System as space becomes available in pre-cure.
Scale	Holds MProds as tuples between Trolley_A and Trolley_B.
Monorail_System	Prioritizes requests from the Curing_Machine objects and places the next request in the TK output queue when there is a pre-cure cell available, the necessary trolley is idle, and no other trolley is currently in the TK output area.
Curing_Machine	Can be set to manual or automatic. When in manual, requires a resource res_Operator to close the machine and cure a MProd. Exchanges cured MProds with covers on a Chariot. Sends cover requests to the Monorail_System.
Chariot	Transports covers after receiving from a Trolley_EFP object to a Curing_Machine. Exchanges MProds.
Curing_Press_Operator	Contains curing operator behaviours and service times (see Chapter 3.3.6).

There are also several functions which are independent of the classes. These functions serve to initialize the model, manage statistics, and other miscellaneous purposes. A high level process map for the important classes in the model is in Appendix G.

On the production line, one machine must frequently wait for the actions of another machine or for the actions of an operator. To communicate these triggers can be tricky to implement. In the model, triggers have been communicated using generic SimPy events, which causes the class to wait at the event statement for a trigger to that event from

another place in the code. The class is essentially dormant or passivated while it is waiting for a trigger from elsewhere in the code. To make this work, it is necessary to ensure that all other classes contain the appropriate triggers in the appropriate places in their logic. An alternative approach would be to enter a loop which checks for a resume condition at a set interval. Both approaches were tested and the first method is computationally much more efficient, although somewhat more difficult to implement.

Extensive statistics tracking has been built into the simulation model. The following were metrics are tracked for each machine:

- Working time
- Time spent waiting for jobs
- Time spent waiting due to downstream delays
- Utilization
- Throughput

Other tracked metrics include:

- For curing machines and machines A and B the amount of time spent waiting for an operator
- Repair time
- Total time in the system for each MProd
- Time that the line is stopped due to the repair circuit

These metrics allow users to understand what events are causing time to be lost to each machine. The statistics tracking is also useful for the debugging, verification, and validation of the simulation model. Optionally, stock levels at each stage or other metrics can be tracked over time, and plots can be generated from this data.

3.2.3 Parameters

There are many parameters which are configurable both through the user interface and directly in the Python code. Configurable parameters include:

- PAC Parameters
    - Simulation mode: Select either the mode for analyzing maximum throughput, or the mode which includes Machines A and B, and uses the PAC Coordination System.
    - Number of process tags per store: Limit the number of PA cards at the preceding cell.
    - Batch size: Issue PA cards in batches of this size.
    - Initial stock in the TK: The initial stock in the TK determines the theoretical maximum stock level of the TK using the PAC System
- Simulation Options
    - Number of replications: A higher number of replications will typically narrow the confidence intervals on results
    - Warm-up period: Define the length of the transient period for which statistics should not be recorded for each replication
    - Time per replication: Days per replication. This must be larger than the warm-up period to get statistical results
- Machine and Operator Options
    - Direct cell on/off: Turning on the direct cell allows trolley B to occasionally bypass the TK. The purpose is to reduce TK cycles.
    - Repair circuit on/off: Repair operator services defective MProds immediately when the repair circuit is off.
    - Number of pre-cure cells per line: Six is the default value.
    - Machine speed for each machine: Increasing speed will speed up a machine. Decreasing will slow them down.
    - Operator efficiency for each operator type: Increasing efficiency will speed up operators. Decreasing will slow them down.
    - Changeover policies: Select a different changeover policy for machines in machine bank A
    - TK logic: Change the priorities of the TK
    - Number of empty slots in the TK: A number from 1 to 10 is reasonable here. This effectively reduces the capacity of the TK

- Product Mix Options
    - Fully specify configuration of products to machines
- Debugging and Animation
    - General debugging on/off: Toggles text debug for trolleys, curing machines
    - Repair circuit debugging on/off: Toggles text debug for repair operations
    - Extended statistics printing on/off: Toggles text debug for statistics tracking
    - TK debugging on/off: Toggles text debug for the TK
    - Production control and machine groups A and B debugging on/off: Toggles text debug for machines A, B, and PAC System
    - Animation on/off: Toggles the animation on or off
    - Animation speed: Speed up or slow down the animation
    - Animation update interval: Change how often the animation updates
    - Plotting interval: Change how frequently data is recorded for plots

3.2.4 Animation

The simulation animation was also useful in showing that the model was working properly, and also served as a visual tool to help to communicate with management. The animation, written in Python using the open source package Tkinter, runs in parallel with the simulation. First, a canvas of objects which resembles the actual production line is created as the base for the animation. As the location of MProds in the simulation model changed, the animation updated to show which type of product was located on which product slot. This was not difficult to program, as all that was required was to animate variable values as they changed in the simulation. In the animation, each color represents a different product type, and trolleys are represented using circles instead of squares. A screen shot from the animation is shown below in Fig. 4. In the upper left corner of the animation, the tippers slots from machine group B are represented. Further down the line, where trolley A and trolley B meet, the two weigh scales are represented. A repair area is displayed before the TK. Every product slot in the TK is represented in the bottom left of the animation. Finally, on the right side of the animation, the three parallel pre-cure and

curing lines are shown. The main differences between the animation in Fig. 4 and the system diagram in Fig. 1 are that in the animation, machines A and B are only represented in terms of the tipper slots, and the TK is shown in more detail.

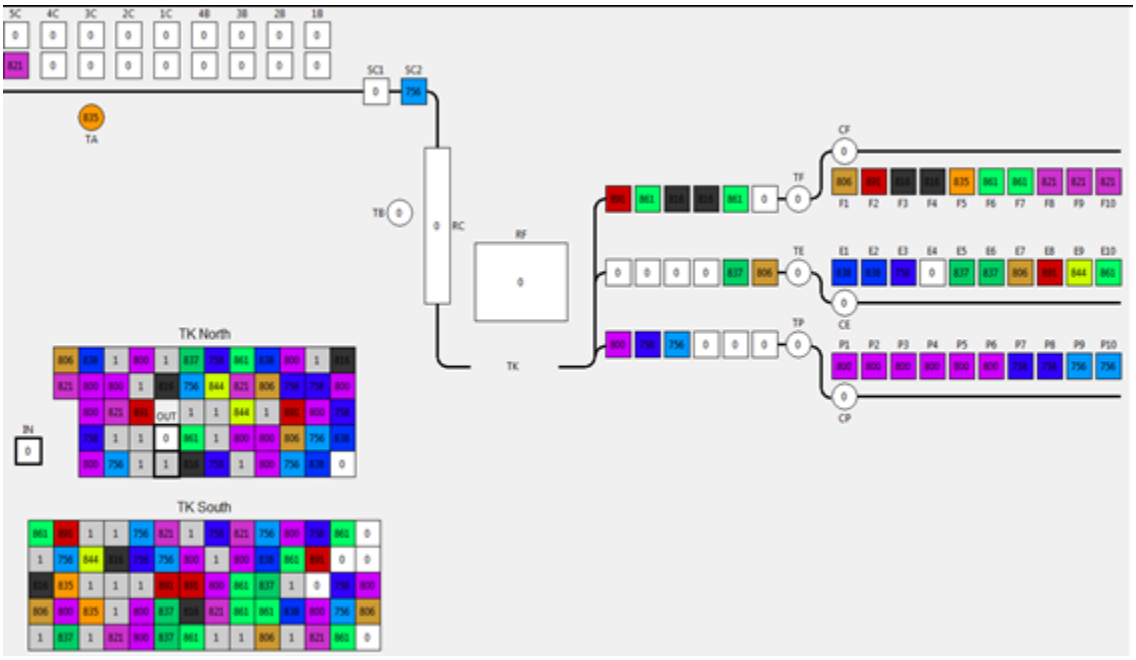

Figure 4: Simulation Animation

3.3 Simulation Model Data

3.3.1 Trolley A Demand

The simulation model can be run in two different modes. The first mode ignores the limitations of machine groups A and B, and produces MProds for trolley A to pick up at the tippers. This mode is intended to test the capability of the MHS. The second mode considers machine groups A and B in detail. The second mode also allows the production line to be controlled by PAC parameters to test various production control policies, and will be discussed in Chapter 4.

When the simulation is in the first mode, machine groups A and B do not exist. Instead, MProds appear at the tippers according to a discrete probability distribution, shown in Appendix A. This distribution was fitted using two months of tipper timestamp data acquired from a production tracking database. The data was aggregated by 15-minute

interval throughout the day and fitted to a discrete probability distribution because the multi-modal demand pattern would have been difficult to model using continuous distributions. An additional benefit to using a discrete probability distribution is that the 15-minute interval selected provides sufficient resolution to see the effects of regular breaks and shift change, but it is not so narrow that the data is noisy. The discrete probability distribution helps to represent the effects of breaks and shift changes on the production output from machine group B. To apply the distribution to each type of MProd in a product mix, the magnitude of the number of MProds to be produced in a given interval is scaled using the demand for that MProd in curing, such that MProds are produced at a higher production rate than the production rate in curing as seen in the following pseudocode:

```
now <- current simulation time
dailyAvg = wklyDemand/7
While simulation is running:
       prob = dailyAvg*discreteProbDist(now)
       If random.uniform(0,1) < prob OR Skip == True:
              Wait random.uniform(0,5) minutes
              If the TK is not at limit for this product AND there is room in
              the tipper:
                     Produce MProd
                     Skip = False
              Else:
                     Skip = True
              End If
              Wait until a total of 5 minutes have passed since the beginning
              of this iteration
       End If
End While
```

3.3.2 Trolley Service Times

Service times for trolleys A, B, E, F, and P were determined using PLC data from a previous study performed in 2013. To ensure that these times are still valid, observations were done with a stopwatch on all trolleys. In the simulation model, deterministic times are used for all trolley movements. Unfortunately, to preserve confidentiality, it is not possible to include real data from the time studies. However, some modified data is provided from the time studies to provide evidence that using deterministic times for trolley movements is reasonable, although not as precise as using a probabilistic

distribution. Trolley travel times are between 45 and 120 seconds in reality, depending on the trolley, the origin, and the destination. Data is provided for trolley B, since trolley B is the most important trolley to model accurately to its high level of utilization and its interactions with the TK. For an ordinary trolley B cycle, an MProd is transported from the scales to the TK input. The PLC study data for this specific movement for trolley B is summarized in Fig.5 below.

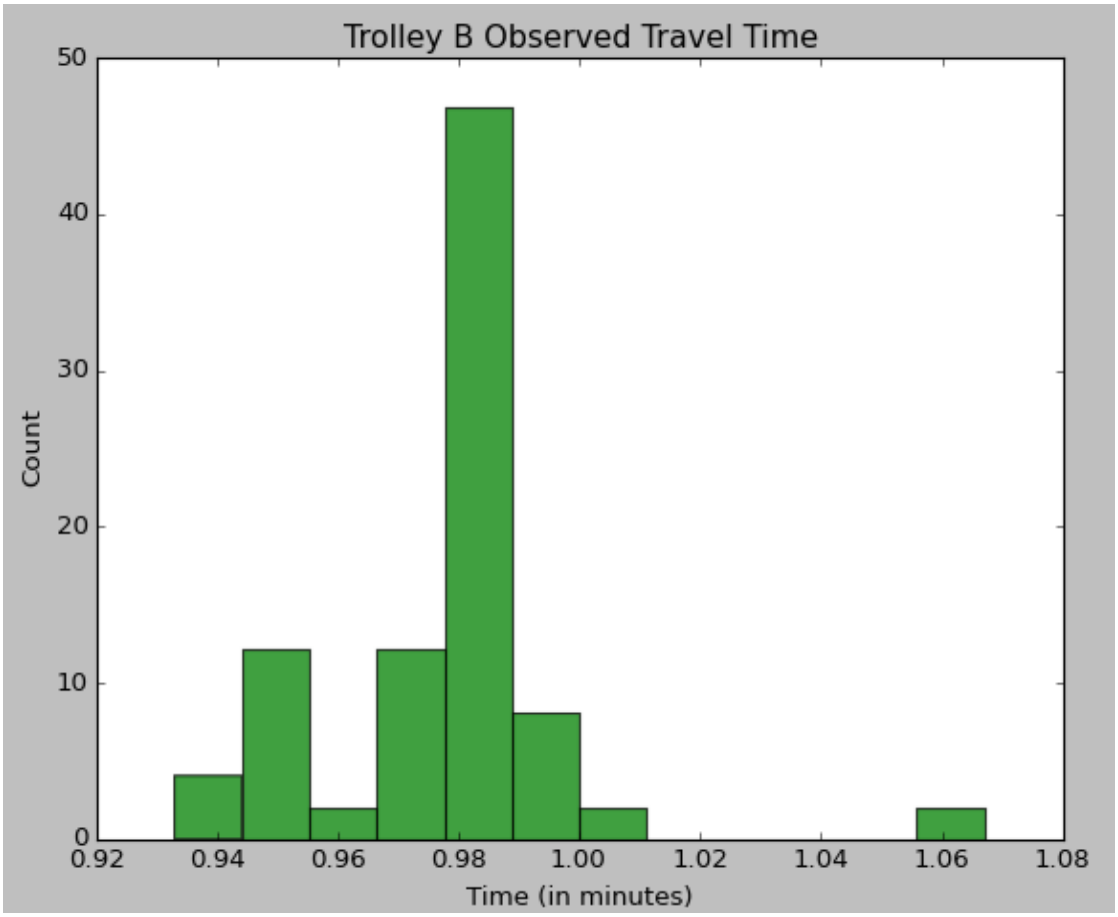

Figure 5: Observed travel time for trolley B between the weigh scales and the TK input

The data has a sample mean of 0.977, a sample standard deviation of 0.0213, and a coefficient of variation of 0.0218. Notice that the distribution appears bimodal. This is because the two pick-up locations for trolley B (the two weigh scales) are not differentiated in the study. The first peak represents the nearer of the two weigh scales to

the TK, while the second peak represents the furthest weigh scale from the TK. Trolley B travels more frequently to the furthest weigh scale because trolley A prefers to drop MProds in that slot. Trolley A will only drop a MProd to the nearer of the two weigh scale slots when the further one is occupied.

Excluding the single observation at 1.06 min as an outlier, possibly due to a sensor alignment issue when the trolley arrived at the input, and excluding travel times below 0.96 min, since they are likely observations for the nearer scale, the mean travel time becomes 0.981 min with a sample standard deviation of 0.0083 min. In the simulation model, the extra travel time to reach the further weigh scale is taken into account when trolley B travels to that scale. At this stage, it might be reasonable to model the travel time using the normal distribution, and this could have been done. However, the decision was taken to use deterministic times due to the low variation present in trolley travel times. Using the above example, and assuming a normal distribution, 99.74% of the time the true travel time will be within 1.5 seconds of the mean. To provide an indication of the level of variation present for trolleys other than trolley B, Table 2 provides a summary of standard deviation as a percentage of the travel time. This table contains data for the principal trolley movements. For example, trolley A may be travelling to any of the nine machines in group B, from either of the two weigh scales. This means that there are 18 different travel times possible for trolley A, all modelled deterministically. In this case, the time from either one of the two weigh scales to the third closest machine B is selected, because it has the most data points in the study. Similarly to trolley B, the standard deviation is probably overstated, due to the PLC study not differentiating between the two weigh scales. The slightly larger coefficient of variance for trolley F may be attributable to its smaller sample size.

Table 2: Sample standard deviation of trolley travel time

Trolley	Standard deviation as a percentage of the mean travel time
A	2.7%
B	0.84%
E	1.3%
F	3.1%
P	1.7%

3.3.3 Repair Circuit

There is very little historical data available for the repair circuit. To model the repair circuit, parameters were estimated from discussion with the repair circuit operator. Some of the parameter values were later validated through discussion with other operators, and a few parameters were estimated by consensus at a plant meeting. Since the purpose of modelling the repair circuit is to determine whether or not it can have an effect on daily throughput, the repair circuit in the model does not need to be a precise replication of the real repair circuit. Consensus values can at least provide an indication of the impact that the repair circuit has on production output. MProds were determined to require repairs with probability of 0.02 (2%). The repair operator is programmed to check the status of the repair cells every 45 minutes, except when the repair operator is doing repairs. If there are at least five repair cells out of six that are full, then the repair operator will begin to service the MProds. If it is during the day shift, the repair operator will repair all of the MProds and transfer them to the TK. If it is not the day shift, the repair operator will clear the MProds to another area until the day shift operator comes in to do repairs. Repairs sometimes require the use of trolley B, depending on the origin cell and destination cell when moving MProds during the repair process. These requests for trolley B take priority over its normal operations. The repair time for a single MProd is modelled using a triangular distribution, with a lower bound of five minutes, a median of ten minutes, and an upper bound of 20 minutes. Depending on the location of the MProd, a forklift may be needed. The time for a forklift to arrive is modelled using a uniform distribution with a lower bound of ten minutes and an upper bound of 40 minutes.

3.3.4 Transit-Keeping Machine

Initially, TK travel times were taken from a company study from 2013 that used PLC data. The TK was represented in the model as a simplified version. However, during the validation process, it became evident that the simplification of the TK was not giving sufficient information about the true behaviour of the system. For example, the TK can save some cycle time by transporting an empty TK pod from the output directly to the input to receive a new cover. Additionally, due to the placement of the TK input and output, as the TK becomes more full, cycle times increase (Fig. 4 shows the location of the TK input and output).

The TK was re-coded in the simulation model to mirror the PLC code precisely (see Appendix B). It can now be given some initial stock and its behaviour can easily be observed using the simulation animation (for further discussion, see Chapter 3.4). TK movement was timed with a stopwatch and is represented in the simulation model. A function was fit for the vertical and horizontal travel times in the TK. This function was developed using the observation that the TK can move vertically and horizontally simultaneously. It also moves in each direction using an independent mechanism from the other directional mechanism. Knowing this, vertical TK travel can be modelled separately from horizontal TK travel, and the actual travel time is the larger of the horizontal travel time and the vertical travel time. A linear function was fit to the vertical movement because it reaches a constant velocity very quickly, while a quadratic function was fitted to the horizontal movement because the acceleration has a larger role. The Python code used to determine the travel time is as follows, where x is the horizontal position and y is the vertical position:

```
a = -0.0019*((x[i]-x[j])**2) + 0.0409*(abs(x[i]-x[j])) + 0.0604
b = 0.0546*(abs(y[i]-y[j])) + 0.0452
d = 0.003*abs(x[i]-x[j])+0.25
      if abs(x[i]-x[j]) <= 9.0:
            TravelTime = max(a,b)/60.
      else:
            TravelTime = max(d,b)/60.
```

3.3.5 Machine Bank C

To model the operation of machine bank C, PLC data is used for press opening times, cover/MProd exchange times, and other machine activities. This data is taken from studies performed by the manufacturer and cannot be shared in this thesis. Cure times are deterministic, and are taken from a company database.

There are several processes modelled using the available data which are listed in Table 3 below. A Kolmogorov-Smirnov goodness of fit test was conducted for each distribution was conducted after first grouping samples into bins. For the goodness of fit test, the null hypothesis is that the data is consistent with the specified distribution, while the alternate hypothesis is that it is not. Using a significance level of 95%, a p-value of less than 0.05 would indicate that the null hypothesis should be rejected. For all distributions, the p-value is greater than 0.05, so we cannot reject the null hypothesis.

Table 3: Process time distributions for machine bank C

Process	**Distribution (minutes)**	**Notes**	**P-value**
Close curing machine	Normal(μ = 2.18, σ=0.305)	Minimum = 1.23 minutes	0.079
Cure	Deterministic	Varies by product	N/A
Open curing machine	Triangular(3, 3.55, 5.42)		0.294
Transfer MProd	Normal(μ = 0.718, σ=0.0744)	Minimum = 0.51 minutes	>0.15
Complete transferring MProd	0.1 + Exponential(μ = 0.0293)		0.18
Operator time	Normal(μ = 0.899, σ=0.245)	Minimum = 0.45 minutes	>0.15

Another parameter, the call-to-press time, specifies when a MProd is called from pre-cure to be transported to the curing chariot. The values used as the defaults throughout this thesis are selected to reflect how this setting is being used in the current system, which is why they are different from one another. The values appear to be a function of the

physical configuration of the machines; however it is not known exactly how these values are chosen. The values are shown in the following Table, where each cell represents a curing machine:

Table 4: Call-to-press defaults

Line	E	F	P
Call-to-press time (seconds)	130	70	163
	100	0	163
	100	43	163
	73	70	163
	70	70	163
	0	70	163
	0	70	163
	15	73	163
	100	73	163
	103	103	163

3.3.6 Operator Behaviours

To model the behaviour of curing press operators, data from a company time study was used in conjunction with several interviews with press operators. The company time study was used to determine how long the operator intervention should take. Since this is a company study, the data cannot be shared in this thesis. Real press operator behaviour is difficult to represent exactly in a simulated environment, considering the complexity and the variety of behaviours. However, several simplified operator behaviours have been modelled and the impacts of these behaviours can be examined through model results.

Three behaviours have been modelled for curing press operators:

- Type 0 operator: Operator who behaves as a resource in a simulation model. When a new task is requested, the operator immediately begins that task, and performs tasks in the order that they arrive.

- Type 1 operator: The second behaviour is similar to the first, except that there is a delay of one minute before starting a new task to account for travel time and inattentiveness.
- Type 2 operator: The third behaviour which was modelled is probably the closest to real operator behaviour. Operators wait for three curing machines to open or for ten minutes to pass after the first machine opens before beginning to service any of them. Once the start condition has been met, the operator services all curing machines until there are no remaining jobs in the queue, then the counter is reset.

In the simulation model, MProd-building operators are treated as a resource when the model is being run in the second mode with the detailed representation of build. When an operator is available, the operator will construct MProds. To test the impact of changeover decisions, five changeover policies were established for machine A operators. PA cards are discussed further in Chapter 4, but assume for now that a PA card is issued for a particular product code when an MProd exits the TK.

The changeover policies are:

- Policy 1: Switch only if the current product is out of PA cards and another product has at least one PA card
- Policy 2: Change products if another product has equal or more PA cards than the current product
- Policy 3: Change products if another product has one or more PA cards than the current product
- Policy 4: Change products if another product has at least two more PA cards than the current product
- Policy 5: Change products if another product has at least three more PA cards than the current product

For all of these policies, if multiple products are eligible to be changed to, the product with the highest number of PA cards is chosen. If they have an equal number of PA cards, the product is randomly chosen.

3.3.7 Product Mixes

Five realistic future product mixes were used to test the capability of the current system configuration. The mixes are typical of those created through discussion with the planning group and validated with help from the industrial engineering group. Again, for reasons of confidentiality, they may not represent any actual situation at the manufacturer. The five tested product mixes are included in Tables 5, 6, 7, 8, and 9. The product codes have been redacted and may not correspond with each other across tables. For example, P2 in Table 5 may not correspond with P2 in Table 6. In these tables, the cure time is a deterministic time, while the other time column is the sum of the means of several processes (see Table 3). The daily maximum column represents the maximum number of MProds that could be produced on an average day, given perfect operators and perfect product flow to curing. Only mixes 1, 2, and 4 can operate on the current production line. Mixes 3 and 5 were designed to test a configuration where MProds are produced for curing presses on another line. In the simulation model, these extra MProds exit the system after trolley A.

Table 5: Product mix 1

Product	Number of curing presses	Cure time (minutes)	Other time (minutes)	Daily maximum
P1	6	83	8.05	94.9
P2	3	77	8.05	50.8
P3	3	77	8.05	50.8
P4	2	81	8.05	32.3
P5	2	70	8.05	36.9
P6	2	77	8.05	33.9
P7	1	70	8.05	18.5
P8	2	75	8.05	34.7
P9	2	89	8.05	29.7
P10	1	95	8.05	14.0
P11	3	80	8.05	49.1
P12	3	95	8.05	41.9
Daily maximum throughput				**487.4**

Table 6: Product mix 2

Product	Number of curing presses	Cure time (minutes)	Other time (minutes)	Daily maximum
P1	9	79	8.05	148.9
P2	3	77	8.05	50.8
P3	4	77	8.05	67.7
P4	2	70	8.05	36.9
P5	3	77	8.05	50.8
P6	2	75	8.05	34.7
P7	1	70	8.05	18.5
P8	3	80	8.05	49.1
P9	3	95	8.05	41.9
Daily maximum throughput				**499.2**

Table 7: Product mix 3

Product	Number of curing presses	Cure time (minutes)	Other time (minutes)	Daily maximum
P1	9	79	8.05	148.9
P2	3	77	8.05	50.8
P3	4	77	8.05	67.7
P4	2	73	8.05	35.5
P5	2	70	8.05	36.9
P6	2	59	8.05	43.0
P7	3	65	8.05	59.1
P8	2	75	8.05	34.7
P9	1	70	8.05	18.5
P10	3	80	8.05	49.1
P11	3	95	8.05	41.9
Daily maximum throughput				**586.1**

Table 8: Product mix 4

Product	Number of curing presses	Cure time (minutes)	Other time (minutes)	Daily maximum
P1	6	77	8.05	101.6
P2	3	77	8.05	50.8
P3	3	77	8.05	50.8
P4	2	85	8.05	31.0
P5	2	81	8.05	32.3
P6	2	80	8.05	32.7
P7	3	70	8.05	55.4
P8	3	75	8.05	52.0
P9	3	80	8.05	49.1
P10	3	70	8.05	55.4
Daily maximum throughput				**511.0**

Table 9: Product mix 5

Product	Number of curing presses	Cure time (minutes)	Other time (minutes)	Daily maximum
P1	9	79	8.05	148.9
P2	3	77	8.05	50.8
P3	4	77	8.05	67.7
P4	3	73	8.05	53.3
P5	3	70	8.05	55.4
P6	3	59	8.05	64.4
P7	3	65	8.05	59.1
P8	2	75	8.05	34.7
P9	1	70	8.05	18.5
P10	3	80	8.05	49.1
P11	3	95	8.05	41.9
Daily maximum throughput				**643.7**

The simulation model has the capability to add additional product mixes for testing or production control parameter optimization. The new production mixes should be added using the user interface, and they can be saved for future use in a text file.

3.3.8 Machine Banks A and B

For the machines in banks A and B, the industrial engineering group standard times are used to determine work rates at each machine, while breaks, shift changes, and other events which are known to impact the work rate of operators are also included. In the simulation model, work rates are calculated separately from breaks and shift changes. Standard times vary from machine to machine due to variation between the machines themselves, as well differences in complexity of the products which are typically manufactured on each machine. Although the fixed standard times are not a perfect measurement of the true process time, the true process times are dependent on many factors which have not been fully studied. The standard times provide a time study based estimate which is believed to be reasonably accurate. In the simulation model, breaks and shift changes are modelled as work requests for operators. For example, at lunch time, a high priority request is sent to the top of all operators who are scheduled to go to lunch at this time. When the operator is finished their current task, they will then process the break task, which means they will not be at their post for the duration of the break. Operators for machine group A do not go on breaks, since breaks are covered for these operators. However, they still change shifts every 12 hours. All operators must attend morning meetings and fill out paperwork. Once again, durations for each of these events are from a company time study, from which data cannot be provided. In the simulation model, these events occur at scheduled times during the operator shift.

3.4: Simulation Model Results

3.4.1 Verification and Validation

To verify that the simulation model was being developed in line with expectations, weekly meetings took place with the company during the development phase of the model. During these meetings, assumptions were assessed by a team of stakeholders and process experts to ensure that they were reasonable in the context of the model. A process

map of the simulation model logic was shown to the system experts, and feedback was received to ensure that the process logic in the simulation model was accurate to the logic in the real system. The use of the extensive text-based debugging built into the model helped to verify that the model was behaving as it was expected to. For instance, when the TK logic was being verified, model output similar to the output in Fig. 6 was shown to operators. The output describes, line by line, the actions of trolleys and the TK during a typical TK cycle. The blank rectangles can be read as "TK pod", the terminology used in the model output is redacted. The TK begins by retrieving an empty TK pod from storage slot 62. The TK brings the empty TK pod to the input, slot 124, and places the empty TK pod in the input. Meanwhile, at the output, trolley E has arrived and is picking up a cover. Trolley B arrives with a new cover and places it on the input while the TK waits. The TK then takes the TK pod with the cover and stores it at slot 62. Then, the TK takes a TK pod with cover from slot 6 and places it on the upper output slot, 125. At this time, the empty TK pod from which trolley E picked up a cover earlier is now available at slot 126, which is the lower output slot. The TK identifies this opportunity to remove the empty TK pod from slot 126 and place an empty TK pod on the input, and does so.

```
000 days, 15 hours, 13 minutes, 50 seconds - New position: 62
000 days, 15 hours, 14 minutes, 08 seconds - [redacted] retrieved from: 62
000 days, 15 hours, 14 minutes, 14 seconds - New position: 124
000 days, 15 hours, 14 minutes, 19 seconds - Trolley 1 has arrived at output
000 days, 15 hours, 14 minutes, 23 seconds - TB arrives with cover.
000 days, 15 hours, 14 minutes, 32 seconds - [redacted] stored at: 124
000 days, 15 hours, 14 minutes, 32 seconds - Trolley just picked from output.
000 days, 15 hours, 14 minutes, 51 seconds - [redacted] retrieved from: 124
000 days, 15 hours, 14 minutes, 52 seconds - Output ready for new cover.
000 days, 15 hours, 14 minutes, 57 seconds - New position: 62
000 days, 15 hours, 15 minutes, 15 seconds - [redacted] stored at: 62
000 days, 15 hours, 15 minutes, 15 seconds - Reprioritizing...
000 days, 15 hours, 15 minutes, 21 seconds - New position: 6
000 days, 15 hours, 15 minutes, 40 seconds - [redacted] retrieved from: 6
000 days, 15 hours, 15 minutes, 50 seconds - New position: 125
000 days, 15 hours, 16 minutes, 08 seconds - [redacted] stored at: 125
000 days, 15 hours, 16 minutes, 08 seconds - Reprioritizing...
000 days, 15 hours, 16 minutes, 09 seconds - Trolley 2 has arrived at output
000 days, 15 hours, 16 minutes, 14 seconds - New position: 126
000 days, 15 hours, 16 minutes, 34 seconds - [redacted] retrieved from: 126
000 days, 15 hours, 16 minutes, 47 seconds - New position: 124
```

Figure 6: Debugging text output for a standard TK cycle

Although this is just a small segment of what was reviewed with operators, this type of text output demonstrates to operators that the model is behaving as expected for various

situations that can occur during TK cycles. This text-based verification technique was used to present many of the model components, such as the trolley movements and operator behaviours to the process experts at the company. The experts could then point out which details, if any, seems inaccurate. This helps ensure that the model ultimately ends up faithfully reproducing actual system logic.

Another verification technique which was used as part of the modelling process was to test the model output for a wide variety of inputs. Whenever a new feature or component was entered into the model, it was tested for several product mixes to ensure that it was behaving as expected.

The simulation animation was also useful in showing that the model was working properly. The animation is particularly useful for verifying that the logic in the TK was working correctly, since every product slot in the TK is shown. By changing the update interval and the simulation speed, it is possible to watch the production line operate at very slow speeds where each movement can be carefully observed by seeing the variables update in each product slot, or at very fast speeds so that general observations can be made on the behaviour of the system, such as observations on where product tends to be located in the TK over longer periods of time.

Validation of the model was done in several steps. First, it was necessary to ensure that the TK was processing MProds at its expected rate. When running at high production rates in the past, the TK was known to be able to process between 20 and 21 MProds/hour according to an unknown distribution. When running the simulation model using mix 1, the mix thought to be closest to mixes used in the past, a 95% confidence interval on the capability of the TK can be constructed. With 10 simulation replications, type 2 operators, PAC System controls, and no restrictions on maximum TK stock except for its capacity of 124, and the repair circuit in effect, the TK processed 19.5 +/- 0.5 MProds per hour. However, the TK was only utilized 94.6 +/- 2.0% of the time. If production rates had been slightly higher, the TK would have been nearer to the known processing rate.

The second validation check was for curing machine utilization. Using the same simulation parameters as above, which are meant to be as realistic as the simulation can be, the 95% confidence interval for achieved theoretical output was 95.72 +/- 2.1%. Historically, curing machines have been utilized between 90% and 92% of the time, according to an unknown distribution. The simulation model appears to be achieving better utilization than what is achieved by the real system. The difference between the values may be attributable to a combination of the exclusion of maintenance events, the exclusion of supply shortages in MProd-building, and possibly curing operator behaviours are more severe in reality than in the model. Another consideration is that mix 1 probably has a longer average cure time than historical mixes. This explains why in the model, the curing machines are operating at a higher utilization while the TK is processing less MProds than historical has indicated. Regardless, the model results are reasonably close to what is observed in reality, and while the model should not be used to obtain precise estimates for anticipated production output, it is useful for measuring the relative impact of changes to the system.

3.4.2 Warm-Up Period

To select a warm-up period, Welch's method (Welch 1983) is used to create an initial assessment. Welch's method requires a key statistic to be recorded at a set interval while the model is running. For this model, throughput was recorded every four hours. Multiple simulation replications are executed, and then the statistics are averaged across replications. A three-interval moving average is then calculated using the averaged statistics. For the warm-up experiment, five runs of 20 days were executed. Mix 1 was used, with type 2 operators, and the repair circuit enabled. This configuration represents the most challenging output that the simulation model is thought to be able to reproduce. The output is seen in Fig. 7.

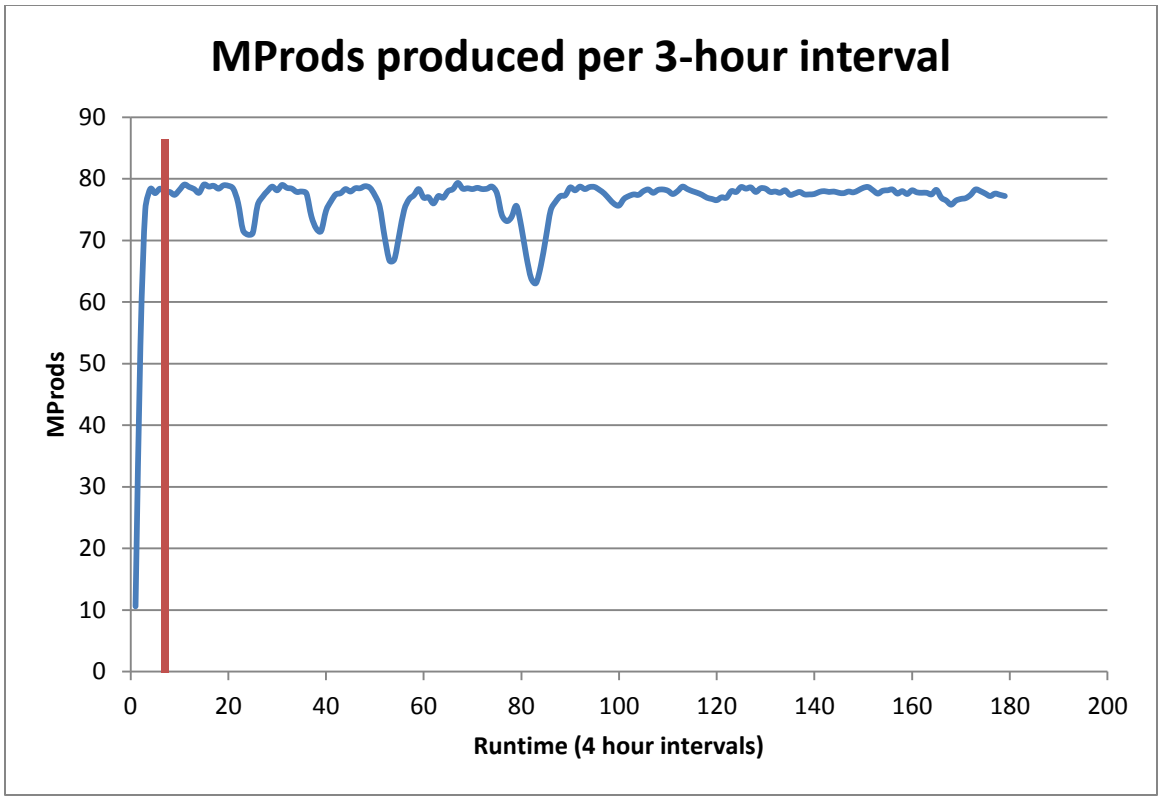

Figure 7: Warm-up period using Welch's method

The end of the warm-up period is shown using the red vertical line. On this chart, there are four notable drops in production. These drops are due to major repair circuit congestion for one of the five runs. They do not indicate that the system is still in a transient state; repair circuit congestion can occur at any point in a production run. It appears that the warm-up period is about 7 intervals, or 28 hours. However, the behaviour of the production line can vary, depending on production parameters, product mix, and initial conditions. Figures E1, E2, and E3 in Appendix E show how TK stock level traces can vary depending on production parameters. Due to this variance, a safety factor was applied and all statistical data in the first 96 hours is deleted for all of the simulation runs.

3.4.3 Run Length

To decide on an appropriate run length, several factors were considered. First, when simulation runs have the repair circuit activated, it is necessary to perform longer runs

since repair events should occur several times in order to produce a reasonable result. This can be seen in Fig. 7, where only four major repair events occurred in a 30 day replication. When the repair circuit was not active, long runs are not required because the system is quite stable. Thus, run lengths are chosen differently depending on the simulation parameters. For simulations without the repair circuit, often, a run length of only 14 days is chosen. For simulations with the repair circuit, the run length should be longer. Regardless of run length, several replications are performed to generate results and the warm-up period is deleted for each replication.

3.4.4 Machine Utilization

Results from the simulation model have shown some interesting system characteristics. One measurement of interest that the simulation model has established is that the TK has the highest machine utilization, with the exception of the curing machines. A simulation run with 30 replications and a warm-up period of 4 days was run in maximum throughput mode using Mix 1. The repair circuit is disabled to eliminate major repair events and operators are set to type 0. The 95% normal confidence interval widths are less than 0.005 for each machine, and are therefore not shown. The 95% confidence intervals for high utilization machines are in Table 10.

Table 10: Confidence intervals for machine utilization

Machine	95% Confidence Interval
Curing Presses	[0.989,0.993]
Trolley B	[0.905,0.912]
TK	[0.952,0.960]

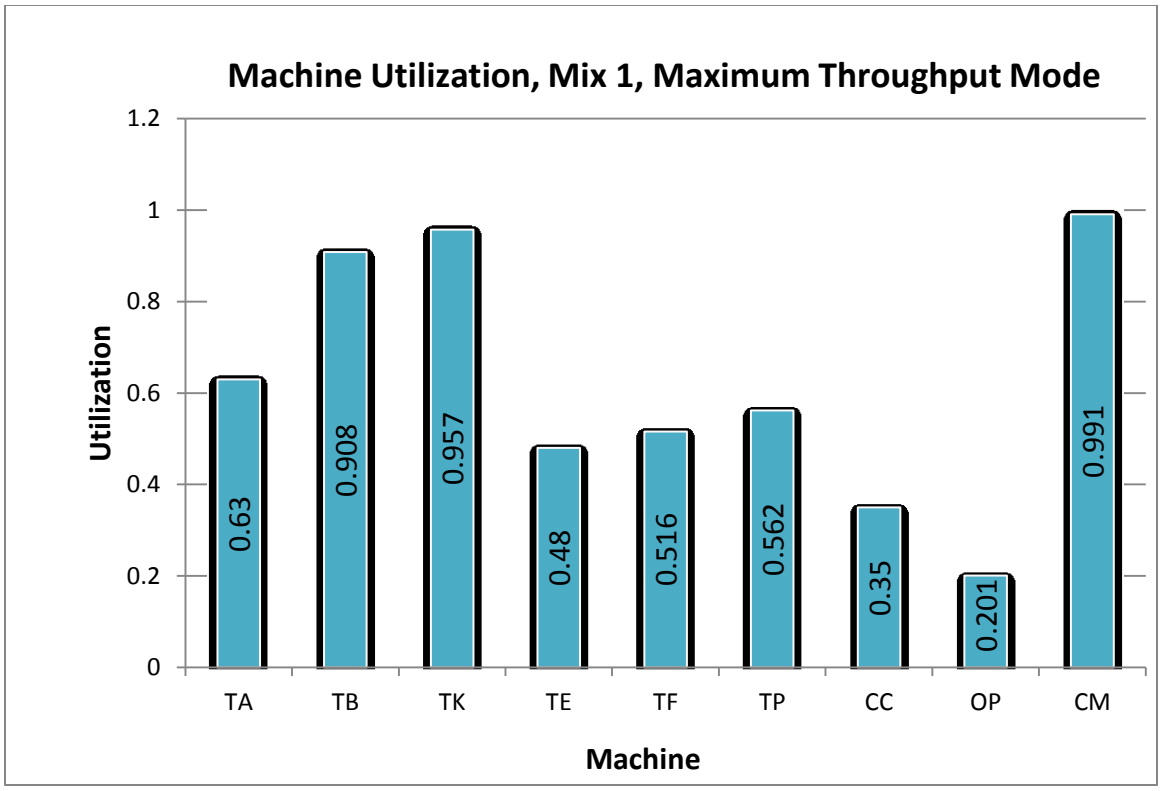

Figure 8: Average utilization of MHS machines and operators using Mix 1. From left to right: trolley A, trolley B, TK, trolley E, trolley F, trolley P, curing chariots, curing machine operator, curing machines.

There are a few important observations to be taken from Fig. 8. First, although the curing machines have the highest utilization of all, this is both desirable and expected. The curing machines are the highest value machines and should be operating at maximum capacity when possible. Curing machines also define the maximum throughput for the line. If curing machines are not reaching 100% utilization, the line is losing productive capacity. Of course, in reality, the curing machines do not reach utilization levels of 99% because of inefficiencies in the production line. As mentioned above, the repair circuit is disabled and operators are set to type 0 for this simulation run, so that the capability of the MHS could be tested to its limits. Under these conditions, the throughput results are higher than what would be achieved in the real system.

Trolley B appears to be very close to the TK in terms of utilization. However, upon further breakdown of trolley B time, approximately 5% of trolley B's time is spent

waiting for the TK to bring an empty TK pod to the input, so it is clear that the TK is the MHS bottleneck. Several opportunities were identified for improvement of the TK. These improvement opportunities include:

- Coordinating with MProd-building to maintain optimal inventory levels: establish a pull mechanism between curing and MProd-building to ensure the right products are built at the right time
- Changing the priorities of TK depending on TK inventory levels for each product code: if the inventory is low for a certain product code, perhaps the TK can break the FIFO rule and select an MProd to minimize travel time
- Using information on trolley status and timing to assist the TK in deciding which task to do next: reduce waiting for trolley B when it is not going to arrive on time, and make more efficient decisions when it comes to moving around empty TK pods.
- Anticipating future TK demand to optimize the storage location of each cover for smaller travel times later: this idea requires further study
- Optimizing the parameters governing the operation of the direct cell: this requires further study
- Repairing covers more frequently so that the TK is not needed for storage of out-of-weight covers

It is not known how much of an impact that improvement to the TK could have on TK utilization. Several of these opportunities will definitely reduce average TK cycle times, and it is possible that only changes to the PLC code will be needed to implement changes. However, since the company did not wish to continue exploring these opportunities at this time, they were not assessed using the simulation model. If desired, it is not difficult to make changes to TK logic in the simulation model to test these opportunities in the future.

3.4.5 Further Observations

One of the findings from the simulation model is that small changes to operator behaviour can have a statistically significant impact on system throughput. For example,

a type 2 operator performs significantly worse than a type 0 or type 1 operator, as shown in the simulation runs in Table 11. For these simulation runs, 25 replications of 14 days each were run using Mix 1, and the repair circuit disabled.

Table 11: Curing Operator Simulation Runs

Operator Type	95% Confidence Interval for Average Daily Throughput
Type 0	[480.9, 482.7]
Type 1	[474.3, 476.1]
Type 2	[468.9, 470.9]

There is a substantial opportunity for improvement, since discussion with operators and supervisors indicates that many operators try to align curing presses for various reasons. These observations are supported by company data. Management in the curing area have reported as much as a 10% difference between the performances of two work crews, although the reason for the difference is not known. These simulation findings show that curing operators can be having an important impact on total throughput.

The principal task of a curing operator is to place two small metallic identifiers on each side of the cover immediately before it is cured. The metallic identifiers imprint a serial number into the cured MProd, which is a legislated requirement for some MProds. For the MProds that do not require metallic identifier placements, the curing presses run on automatic mode, in which no operator intervention is required. Unlike manually operated curing presses, curing presses operating on automatic mode do not align presses and instead maintain a smooth flow of covers through pre-cure. A similar plant in another location of this corporation places the metallic identifiers on covers using a hot tool. This operation is performed by operators in machine bank B. This technique could be piloted in the production line under study so that rework percentage can be estimated and a financial case can be made to shift the metallic identifier placement process to the machine B operators in this plant. By moving this process, all curing machines can be placed on automatic, which eliminates the need for operator interventions. Note that operator type 0 behaviour in table 11 is similar to that of an automatic machine.

At the beginning of the project, there was a concern over the number of available pre-cure slots. It was thought that due to there being only six pre-cure slots servicing ten curing machines, there would be times when a cover could not pass through curing when it was needed due to congestion in the pre-cure area. This does occur when curing press operators align most of the presses on a line to open at the same time, a behaviour which causes other issues as well. However, the simulated behaviour of operators had machines aligning in groups of three, so the simulation showed that the number of pre-cure cells did not have much of an effect on throughput, as can be seen below in Fig. 9. In this experiment, the simulation is run in the second mode, the repair circuit is disabled and curing press operators are set to type 2. Ten replications were completed for each configuration and the 95% confidence interval on average daily throughput is shown using error bars. The results show that increasing the number of pre-cure cells to more than six will not be likely to improve the system throughput. There are no error bars for six pre-cure cells per line because all runs gave the same average daily throughput.

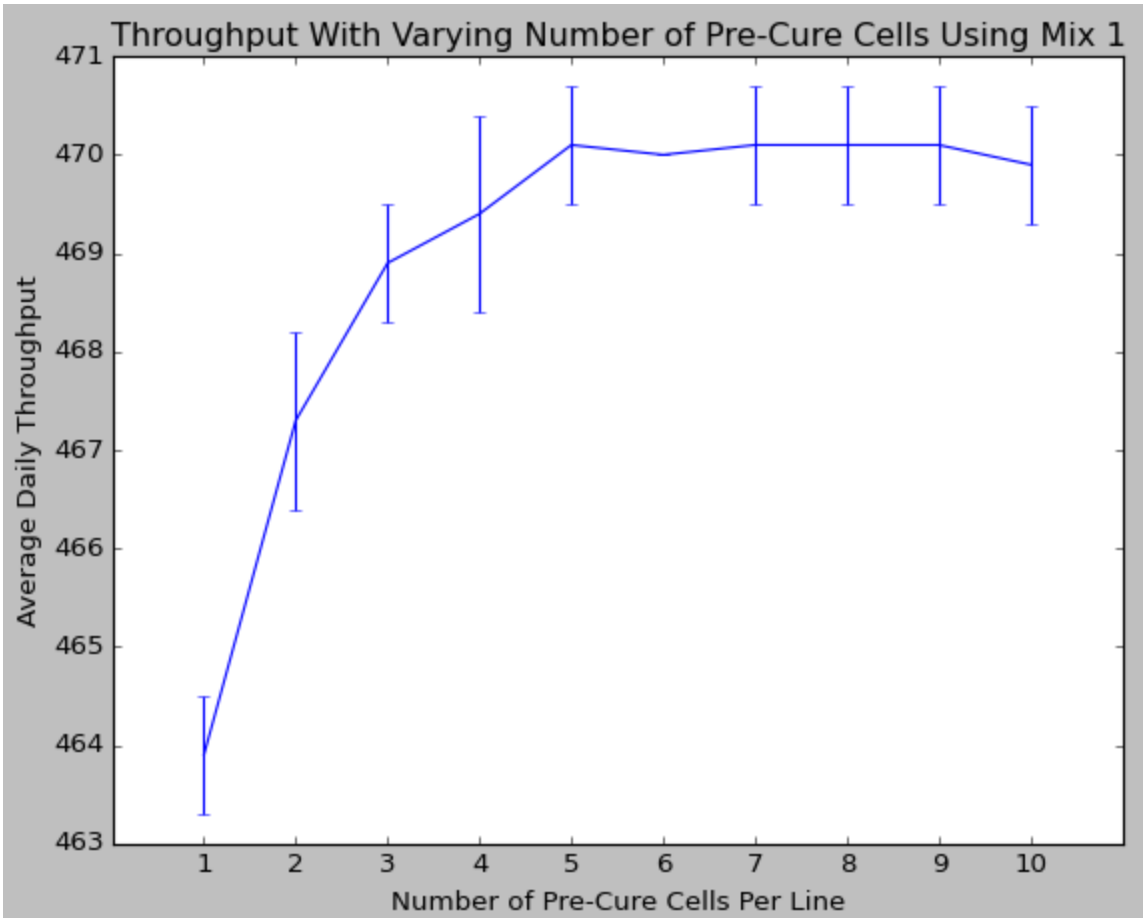

Figure 9: Average daily throughput vs. number of pre-cure cells per line

Another result of interest to management, the impact of TK stock level on the daily throughput in curing was quantified in the simulation model. Since the TK has longer cycle times as it becomes full, and since it is the bottleneck of the MHS, it has an impact on throughput. The TK capacity was limited to a fraction of its full capacity to estimate this impact, as seen in Fig. 10. In this experiment, maximum throughput mode is used, the repair circuit is disabled, and type 0 operators are used. Once again, ten replications are completed for each configuration and the 95% confidence interval on average daily throughput is shown with error bars.

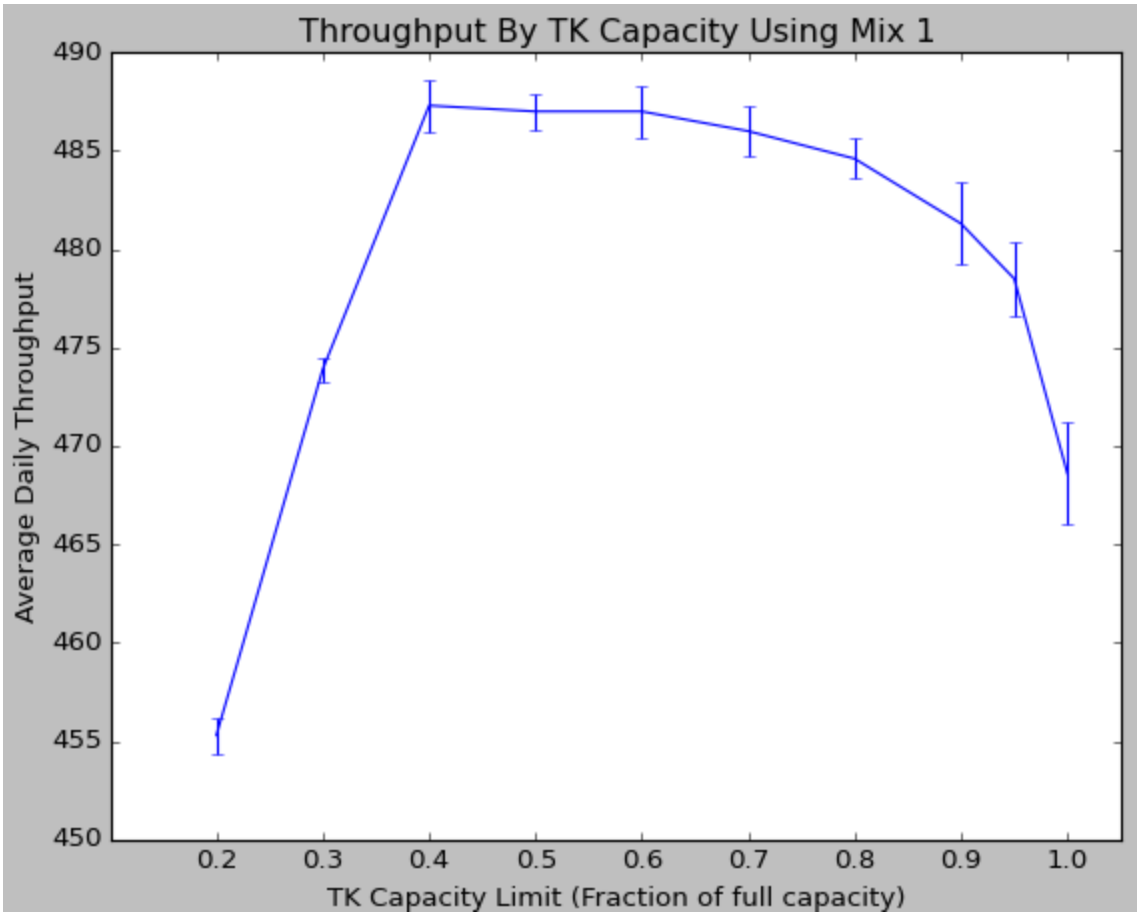

Figure 10: Average daily throughput vs. TK capacity

From the above figure, it is evident that the cycle time of the TK does have an effect on the average daily throughput, especially when the TK is allowed to fill near its capacity limit. This helps support the case that controlling the build rate and mix from MProd-building is an important objective for the production line.

Finally, the repair circuit will induce variation into the flow of MProds through the production line. A simulation run with the repair circuit active was completed for twenty simulation replications, with a replication length of 20 days, and a warm-up period of 4 days, using type 0 operators. A second simulation run was completed with the same parameters, except with the repair circuit disabled. A summary of these runs is shown in Table 12.

Table 12: Repair circuit simulation runs

Repair Circuit	Average Daily Throughput (MProds)	Variance (MProds^2)
Enabled	473.4	152.2
Disabled	481.9	0.052

Using a one-sided F test, the null hypothesis that the two configurations have equal variance is very firmly rejected. Although the company does not keep data on how long the production line goes down due to congestion at the repair circuit, the simulation model results suggest that the line stops an average of 2.8 hours per week, with a sample variance of 23.5 hours squared. This result, when compared to table 12, also indicates that most of the variation in the production line as modelled comes from the repair circuit. When the cost of repairing MProds more frequently is compared against the cost of lost production due to interference from the repair circuit, it is clear that the repair circuit should be carefully managed in the future.

Management at the company decided that although utilization levels for machines in the MHS are high under future anticipated production mixes, the system is still capable at these levels. The above experiments indicate that there are opportunities to improve the performance of the production line through the management of resources and coordination of production between MProd-building and curing.

Chapter 4: The Development of a PAC Approach

In Chapter 3, a detailed discrete-event simulation model of a production line was constructed using SimPy and the Python programming language. The purpose of the simulation model was to evaluate the throughput capability of the material handling system with the objective of identifying whether or not the production line was capable of handling an increase in throughput. Analysis of the system showed that the material handling system was capable of dealing with the anticipated future production requirements. However, findings from the simulation model indicate that operator behaviours, changeover policies, production control policies, and stock levels have a significant impact on average throughput.

In the current production line, the MProd-building operators are rewarded based on the number of MProds that they produce in a single shift. The computer system allows them to check inventory in the TK for the product codes that their machine is capable of constructing, and they are supposed to do this checking and ensure that the inventory level does not drop "too low". However, "too low" is not a standard quantity or level which is defined by management or agreed upon by all operators in MProd-building, nor is there a consistent framework for operators to make this judgment. Often, what happens is that operators construct many of the same MProd consecutively, perhaps even for the entire 12 hour shift. Operators are rewarded for this behaviour because it allows them to report high production numbers. This can cause inventory levels in the TK for some product codes to get very high, while others can get very low or even stock out. Additionally, since some raw materials are shared between product codes, sometimes by constructing many of the same product, the production line can stock out of a certain raw material, and the other product code that is supposed to be built cannot be built. How frequently this occurs is not documented; however it is thought to occur several times a day. The unbalanced stock levels in the TK have an effect on the utilization of curing machines and ultimately the total production of the line, even when product codes do not stock out. Operators in curing also have access to stock levels in the TK. When they observe stock getting low for a certain product; they may stop using one of the presses curing that product code even if there is sufficient stock to continue using the press for

another cycle. This action may be taken because the curing operator does not know why the stock is low; there may be five of that product code being constructed in MProd-building, but the curing operator does not have this information. In conversations with both curing operators and MProd-building operators, they described how they will infer the production rate of the other by watching how stock levels change in the TK, and adjust their production rate. This means that it is possible that a curing operator could shut down a press, and then a MProd-building operator could note the decreased production rate in curing and choose to build less of that product code. This type of miscommunication can impact the production rate of the line as a whole. The problems exist due to a lack of production control mechanisms, current incentives for the MProd-building operators, and a lack of direct communication between the two groups of operators.

To investigate the impacts of a production control policy on the system, a second simulation model was constructed as an extension of the first model which allows for production to be controlled by modifying certain PAC parameters and operator behaviours. The PAC approach was chosen for this production line because it was not known which production control scheme may have performed best. The obvious requirement for this production line is a pull mechanism from curing back to MProd-building to keep inventory levels at an appropriate level for all product codes in the TK. This pull mechanism should also regulate operator behaviours and improve communication down the line.

Recall that the PAC System requires a production line to be composed of cells and stores, with PA cards authorizing the manufacture of products in cells. The production line can be modelled under the PAC framework such that machine group A and machine group B are considered the two production cells, while the inventory trees between group A and group B as well as the TK are considered the two product stores. Curing machines are represented in terms of a demand for product from the system. Information flow is configured in the simulation model such that PA cards are required to authorize the manufacture of a product in a cell. In the simulation model, it is possible to modify the

number of process tags at each cell, the initial inventory in the TK for each product code, and the batch size between any two cells. Modifying the number of process tags at each cell allows for limits to be set for work-in-process. Modifying the initial inventory in the TK initializes the system and sets the maximum stock for each product in the TK. The maximum stock for each product in the TK is limited to the initial stock because PA cards are only generated for MProd-building when a MProd exits the TK. Modifying the batch size requires requisition tags to accumulate in a store to the specified batch size before they are sent to the preceding cell. In the model, it is also possible to change non-PAC parameters, such as the changeover policy governing the MProd-building machines, operator behaviours, and the number of operators for machine groups A and B.

To learn more about how these parameters affect average throughput, an artificial neural network (ANN) metamodel is constructed as an estimator of the expected value function of the second simulation model. While the output of a simulation run is a random variable dependent on its inputs, the output of a metamodel is function of its inputs. Using an ANN metamodel allows us to search the input space for strong combinations of input parameters, from which production control policies can be derived. At the end of Chapter 4, simulated annealing on the ANN inputs is used to identify some policies which might be adopted by the manufacturer.

4.1 Experiment Design

There are 25 simulation parameters which are taken as inputs for the ANN metamodel. These parameters could have been selected using screening experiments for significance, but instead were informally chosen based on which factors were thought to have the most impact on results from simulation runs. Ten of the inputs are binary policy variables which correspond to five mutually exclusive changeover policies in machine bank A, one binary variable which determines whether or not machine bank A and B can share operators, one binary variable which determines whether or not operator breaks in machine bank C are covered, and three mutually exclusive binary variables which determine operator behaviours in machine bank C. The binary variables for the five governing the changeover policy and the three governing bank C operators are modelled

as two variables in the experiment design phase, although they are treated separately in the neural network. Twelve of the inputs correspond to the initial stock level of each of the 12 products in the TK. The remaining 3 inputs determine the total capacity of the TK, and the number of operators in machine banks A and B. Reasonable ranges have been set for each input variable, as seen in Table 13.

Table 13: Simulation parameters and ranges

Parameter	Range	Number of ANN inputs
Initial stock level of products in TK	[3,35], integer	12
Number of unused slots in TK	[0,10], integer	1
Machine bank A changeover policy	[0,1], binary	5
Shared operators between machine banks A and B	[0,1], binary	1
Number of bank A operators	[4, 10], integer	1
Number of bank B operators	[4,9], integer	1
Bank C operator behaviour	[0,1], binary	3
Bank C operator breaks covered	[0,1] binary	1

For the experiments in this chapter, the initial inventory on the trees between machine group A and machine group B is set to zero, the number of process tags at each cell is set high enough that work-in-process in not limited and PA cards are issued as MProds exit the TK, and the batch size is set to one. This is done to simplify the ANN training task. By making these changes, the system behaves like a base stock system (BSS) with additional cell internal rules, which are the changeover policies listed in Chapter 3.3.6. A diagram of the system behaviour under these conditions is shown in Fig. 11.

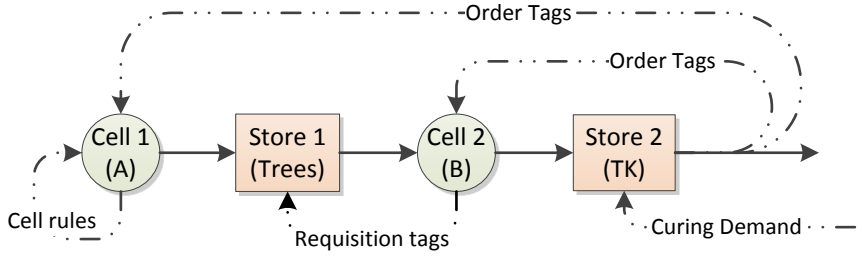

Figure 11: Product and information flow in the production system with ANN configuration

A single performance measure is used since due to the high value of the products being manufactured most costs are insignificant compared to the value of producing additional products per day. Therefore, the performance measure used was the average daily throughput, henceforth referred to as throughput. Note that if it is desired, it is still possible to consider trade-offs between the number of operators and throughput, since the number of operators is included as an input parameter. The single output value and all parameters were normalized to the [0,1] range for ANN training.

To generate a training dataset, several experiment design techniques were considered. A full factorial design is not feasible, since even if only two levels were considered for each non-binary variable, 2^{17} * 5 changeover policies * 3 operator behaviours = 1,966,080 simulation runs would be required. A partial factorial design appears to be feasible, although factorial experiment design for ANN simulation metamodelling have been shown to be inferior to random and space-filling experiment designs (Alam et. al 2004, Hurrion and Birgil 1999). There is an additional consideration for the specific task at hand: not all combinations of initial stock levels of productions in the TK are feasible. There are 12 products which require storage space in the TK, and since the maximum stock level of each product is in the range [3, 35], 99.96% of random combinations of maximum stock levels will exceed the 124 product capacity of the TK. When the TK capacity is exceeded, blockages occur on the production line. Since it would not be desirable to waste simulation time on scenarios which would not occur in reality, it is necessary to limit the domain of the experiment design. Thus, the approach taken is as follows.

First, a very large number of samples (40 million) are generated using a Latin hypercube design (LHD). Latin hypercube sampling (LHS) is a space-filling experiment design technique with some inherent randomness which was found to have a very good ANN predictive ability for simulation metamodelling (Alam et al. 2004). LHS divides the domain of all N variables M times, where M is the total number of design points (McKay et. al 1979). Figure 12 shows a possible LHD for N = 2 variables and M = 5 design points. Note that there is a sample for each row and column of the matrix. The LHD

design used in the present experiment iteratively generates samples using a heuristic, such that the smallest distance between any two samples in the sample space is maximized. The algorithm used is part of the open source package Python Design of Experiments (Baudin 2012).

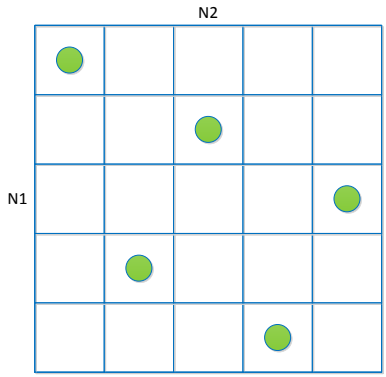

Figure 12: Latin hypercube sampling for 5 samples, 2 variables

To use LHS with discrete variables, it is necessary to convert from continuous values back to discrete values after the samples are generated. To do this, for variables with less than M=32 partitions, it is necessary to round. So for a binary variable, any value from 0.5 to 1 rounds to 1, and any value from 0 to 0.5 rounds to 0. Once the samples have been generated and converted back to discrete values, a subset *S* of the total samples are selected using the selection rule below, which ensures that only feasible samples are selected for subset *S*. In Eq. (5) below, input parameters for each generated sample *x* are denoted by the subscript *v*, where *v* corresponds to parameters in the sequence that they are presented in Table 13. For example, for a sample *x*, the value of its parameter "initial stock level of product 11" in the TK in x_{11}, and while the value of its parameter "number of unused slots in the TK" is x_{13}.

$$if \sum_{v=1}^{12} x_v \leq 124 - x_{13}\ ;\ \ x \in S \qquad \forall\, x \qquad (5)$$

In other words, if the sum of all of the product stock levels in the TK is less than the 124 product capacity of the TK, then the sample should be included; otherwise, it should be

discarded. Also note that since the full sample space obeys a LHD, then the subset *S* will also approximately obey a LHD.

By limiting the domain of the training set in this way, the training set will have a much higher point density in the regions of interest than would have otherwise been possible with the same number of samples. This should lead to a more accurate ANN in the regions of interest. However, there will also be an inherent bias towards training data for which parameters 1 through 12 sum to values closer to $124 - X_{13}$, since these samples are much more likely to have existed in the full sample space. Using the 7,986 samples in the full dataset, the histogram in Fig. 13 demonstrates the extent of this bias.

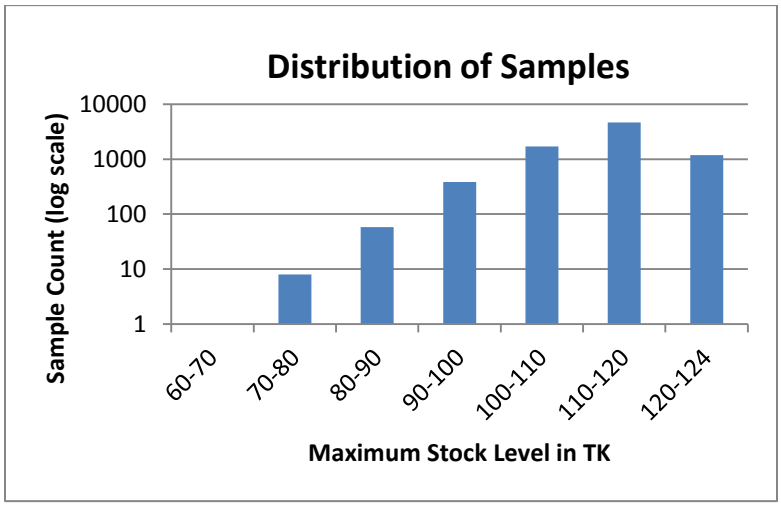

Figure 13: Distribution of samples by maximum stock level in the TK

It is clear that most samples have a maximum stock level above 100. While this may seem like a problematic bias, in simulation runs there are usually 15-30 PA cards pending for machine bank A, which in effect means that the TK stock is usually 30-40 below its maximum. Consider the case where the TK has a maximum stock level of 100. In this case, the maximum number of PA cards that can be pending is 100. In this situation, there are no MProds in the TK, in transit to the TK, or in machine group B. All MProds are either in pre-cure, curing, or have exited the system. However, this circumstance would not occur unless groups A and B stopped working for a long period of time. Normally, there will be far fewer PA cards pending, since groups A and B are working. The TK

stock level is further decreased by the number of MProds under construction, in the repair area, or in transit to the TK. Therefore, the observation that the TK stock tends to remain 30-40 below its maximum seems sensible. It is suspected that the optimal region for any given production mix will have a maximum stock level above 100, so the way that the samples have been chosen means that the ANN will have very good resolution in this region.

The simulation model was run for a single replication for all 7,986 training samples in subset *S*. While fewer design points could have been chosen in order to do more replications for each design point, it has been shown through experiments that distributing the simulation effort over several points in each region with a single replication on each point may result in better ANN simulation metamodels (MacDonald and Gunn 2012). The approach taken here differs by only simulating a single point in each region. This is possible due to the almost deterministic nature of the simulation model when the repair circuit is disabled, as seen in Table 12. Still, perhaps a better approach may have been to generate several separate LHD training sample sets so that there would be several samples for each region.

At an average run time of 12 seconds per replication, the total simulation effort required 26.6 hours of CPU time. The simulation model was set up to run replications automatically, changing parameter values each replication and storing the result in a text file as seen in the following pseudocode:

```
Generate training set and store in text file
Read in training set text file
For each row in training set
       Change parameters as specified
       Do simulation replication
       Record output
End For
Write output to text file
```

The training set is randomly partitioned by MATLAB as seen in Table 14 for each ANN training run, where the test and validation portions are each 15% of the total number of samples. Only the training set is used for training the ANN, while the test set is used to

determine training stop conditions. The validation set is independent, and is used to evaluate how well the ANN can generalize for other samples generated using the same LHD process. The throughput result of each simulation model replication in the training set is used as the desired value of the ANN during training.

Table 14: Distribution of samples

Data set	Samples
Training	5,590
Test	1,198
Validation	1,198

4.2 Initial Comparison of Training Algorithms

To select a good training algorithm for the task at hand, several training algorithms were compared using one hidden layer with 10 hidden nodes (see Fig. 14). Recall that the NEAT algorithm does not require a structure to be specified since the network structure is constructed as part of training process. A comparison of ANN training algorithms for this task is found in Chapter 4.6. For now, assume that a single run is completed with each algorithm using random initial weights and biases. The logistic function is used to transform values at the hidden layer as seen in Eq. (6).

$$f(x) = \frac{1}{1 + e^{-x}} \qquad (6)$$

Although this may not be the ideal ANN structure for this task, by establishing a structure it is possible to perform a simple comparison between training algorithms before selecting one and then further refining the structure.

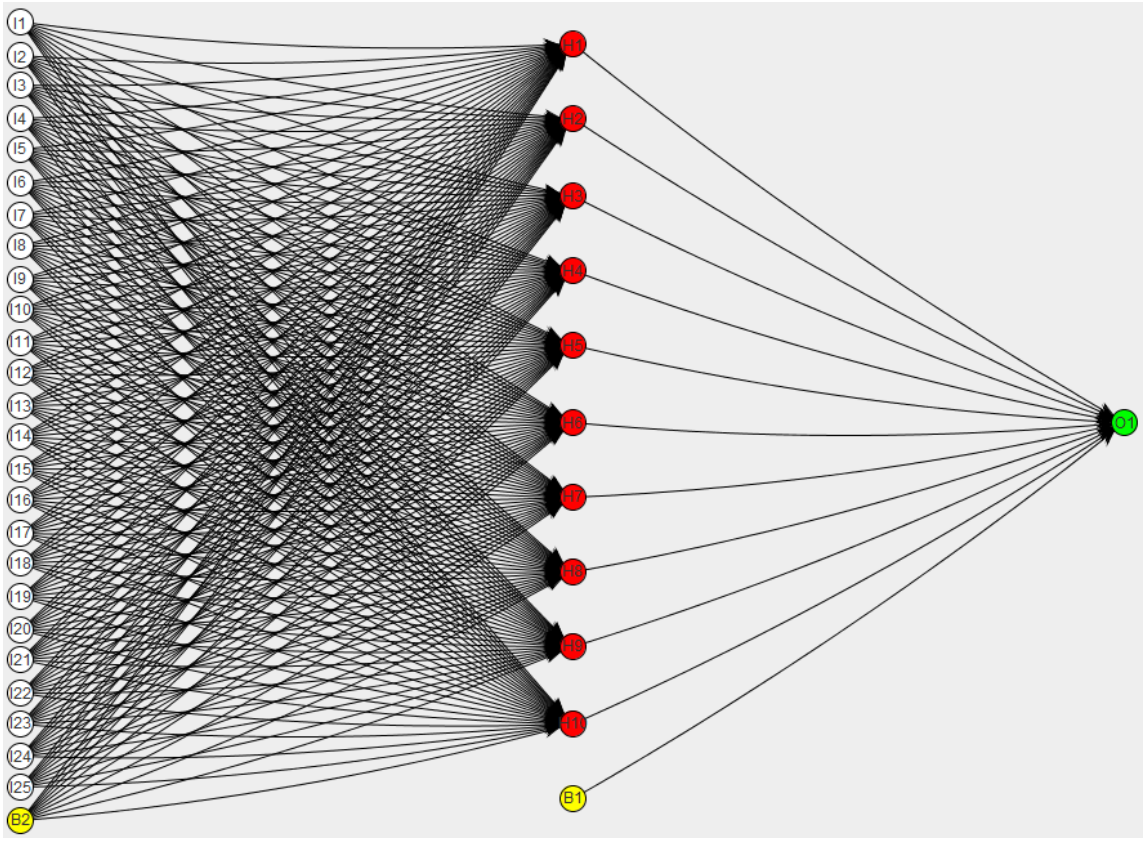

Figure 14: ANN with one hidden layer of ten hidden nodes

The stopping criterion for GA and NEAT training was to end after 20 minutes of training. For RPROP+, BFGS, and LM, the stopping criterion was when MSE in the test set begins to increase; this is an indication that continuing may cause the ANN to overtrain. This method is suggested as a possible stopping criterion by Hagan et al. (1996). These stopping criteria were used for all of the experiments in this chapter. The implementation of the GA and NEAT in ENCOG (Encog Machine Learning Framework) is used for comparison with implementations of RPROP+, BFGS algorithm, and the LM algorithm in MATLAB (MATLAB & SIMULINK). The experiments were performed on a quad-core Intel i5-2500K processor running at 3.3 GHz and the results are shown in Table 15.

Table 15: Comparison of training algorithms

Algorithm	Iterations	Time (s)	Training MSE	Validation MSE	Parameter settings
RPROP+	470	2	5.2E-04	5.7E-04	MATLAB default
BFGS	297	10	8.1E-04	1.0E-03	MATLAB default
LM	106	14	2.1E-04	2.5E-04	MATLAB default
GA	77	1,200	3.8E-04	3.5E-04	Mutation rate = 0.1 Crossover rate = 0.0025
NEAT	6	1,200	1.5E-02	1.5E-02	Population size = 10 Generations = 10

From the time taken to train, it appears that the evolutionary training algorithms are less efficient for this task than gradient-based or quasi-Newton training algorithms. The complex, multi-layer networks generated by the NEAT algorithm could have been used here to define the network architecture. Instead, a different approach is taken in the following section. The GA, gradient-based and quasi-Newton approaches have validation MSE values within the same order of magnitude. Without performing additional experiments, it cannot be said with confidence which of these three training algorithms is best for this task. The LM algorithm was selected to continue to examine and refine the structure of the ANN because it achieved the lowest validation MSE from the single run experiment, and did so within a reasonable amount of time.

4.3 Comparison by Number of Hidden Nodes

To determine a good number of hidden nodes for this ANN, six runs of the LM algorithm using random initial weights and biases were completed for 1, 2, 3, 5, 7, 10, 15, 25, and 50 hidden nodes. The results of these runs including sample standard deviation in brackets are found in Table 16. Trace plots of the first run for select numbers of hidden nodes are found in Appendix D.

Table 16: Comparison by number of hidden nodes

Number of hidden nodes	Iterations	Time(s)	Training MSE (MSE_T)	Validation MSE (MSE_V)	$\frac{MSE_V - MSE_T}{MSE_V}$
1	18(3.7)	1.2(0.4)	1.29E-2(1.79E-4)	1.29E-2(5.86E-4)	-5.90E-3(5.00E-2)
2	47(30.8)	3.3(2.5)	6.64E-3(1.31E-3)	6.84E-3(1.25E-3)	3.00E-2(6.00E-2)
3	49.7(21.1)	3.7(1.9)	4.30E-4(1.95E-3)	4.30E-3(1.95E-3)	5.80E-4(4.30E-2)
5	83.5(46.4)	7.8(4.5)	8.46E-4(5.88E-4)	8.76E-4(5.98E-4)	4.10E-2(3.90E-2)
7	1412(54.9)	15(6.1)	4.20E-4(1.50E-4)	4.32E-4(1.49E-4)	2.90E-2(2.50E-2)
10	227(291)	40(58)	3.20E-4(2.37E-4)	3.53E-4(2.56E-4)	9.50E-2(4.00E-2)
15	143(103)	34(26)	1.84E-4(4.27E-5)	2.14E-4(4.39E-5)	1.40E-1(5.00E-2)
25	153(82.8)	61(36)	1.18E-4(4.79E-5)	1.68E-4(4.48E-5)	3.20E-1(1.10E-1)
50	76(26)	72(27)	1.18E-4(2.12E-5)	1.98E-4(4.44E-5)	3.90E-1(7.00E-2)

In comparing the results of this experiment, the performance of the trained ANN on the validation set must be considered, while care must be taken that overtraining is avoided and that computation remains reasonably low. For all of the different numbers of hidden nodes tested, the training time would be considered reasonable under most circumstances. The performance of the trained ANN on the validation set is of particular interest, since the validation set represents how well the ANN can generalize. It is clear from the quartile box plot in Fig. 15 that one, two and three hidden nodes are worse than the others in terms of the performance of the validation set. Clearly, having three or less hidden nodes does not provide enough complexity to accurately model the underlying behaviour of the simulation model.

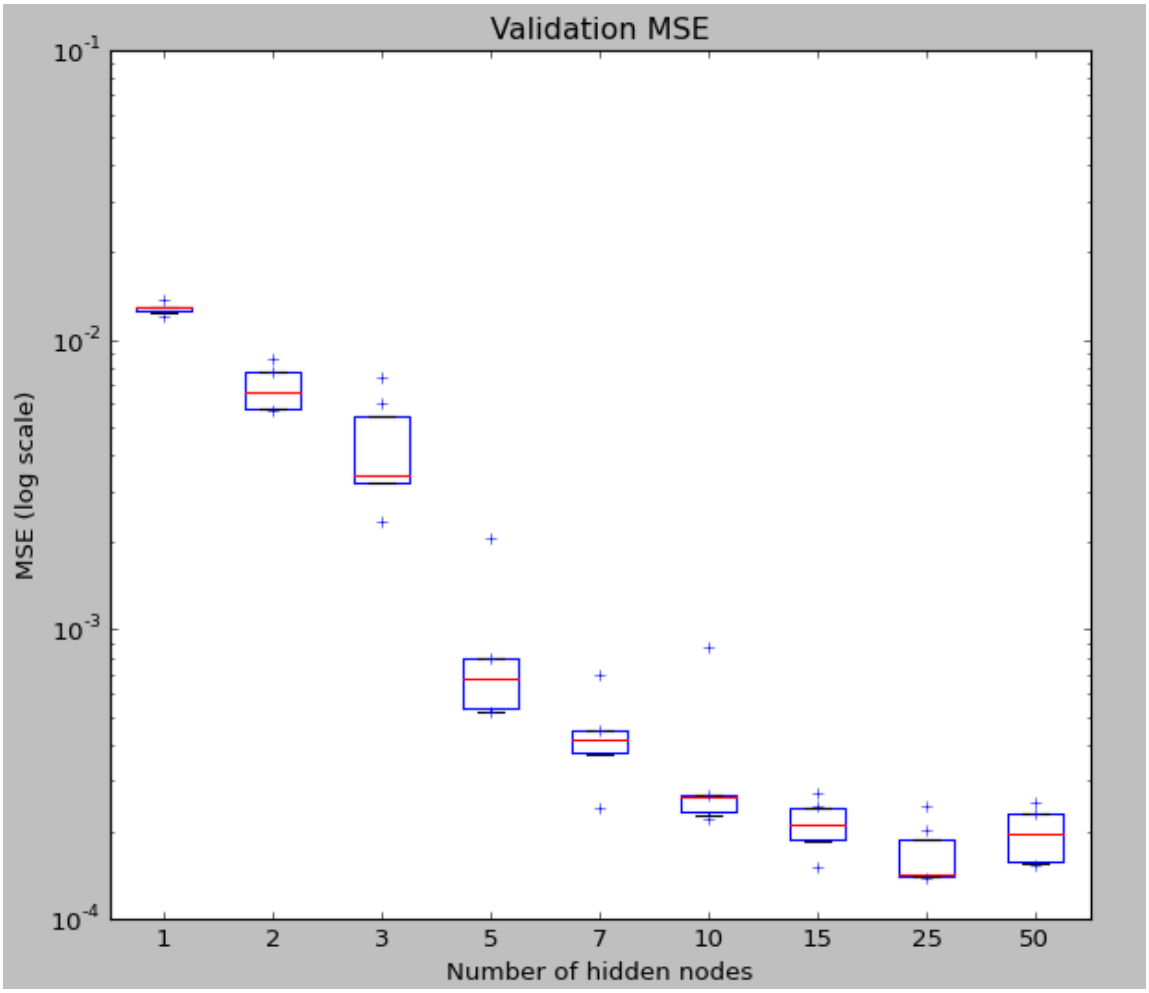

Figure 15: Validation MSE by number of hidden nodes

It is interesting to note that the mean validation MSE drops as the number of hidden nodes rises, with the exception of 10 and 15 hidden nodes, and 25 and 50 hidden nodes. In these cases the difference in mean validation MSE are not significantly different. To measure overtraining, the following formula is used:

$$Overtraining\ (OT) = \frac{MSE_V - MSE_T}{MSE_V} \qquad (7)$$

A comparison of the measure OT by number of hidden nodes is shown in Fig. 16.

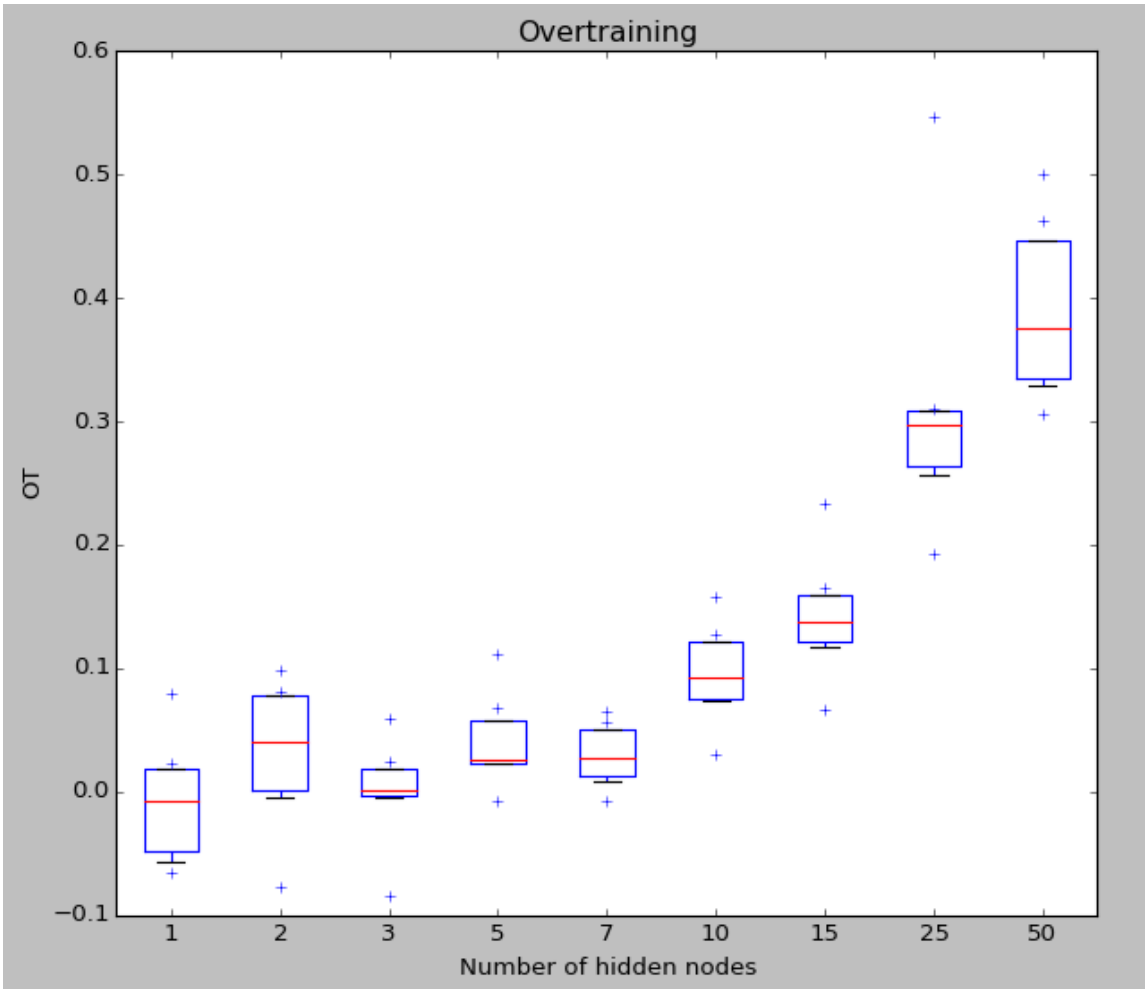

Figure 16: Overtraining by number of hidden nodes

For seven or less hidden nodes, there is little overtraining. For 10 and 15 hidden nodes, there is some overtraining, but the trade-off with improved performance may be worthwhile. However, the argument could be made that fewer hidden nodes is better so that overtraining is avoided entirely. As the number of hidden nodes continues to rise, it becomes clear that the ANN is overtraining significantly.

To select a good number of hidden nodes for the task at hand, we must consider that needs of the manufacturer. To make a good production decision, especially considering the high value of the products being manufactured, absolute error of no worse than 1.0% is desired. Ideally then, a validation MSE of 0.0001 or better should be the target, while also seeking to minimize overtraining. Five or less hidden node ANNs will not achieve

the MSE target, while 25 or more hidden node ANNs will tend to overtrain. It seems then that 7, 10, or 15 hidden nodes are reasonable ANN structures for this task. Since the 15 hidden node ANNs were not significantly better in validation MSE than the 10 hidden node ANNs, while the 10 hidden node ANNs were significantly less overtrained, a 10 hidden node architecture is preferable to 15 hidden nodes. A 10 hidden node ANN was ultimately selected, since the validation MSE comes significantly closer to the MSE target than the seven hidden node ANN. It would also have been reasonable to select the seven hidden node architecture to minimize overtraining.

4.4 Comparison by Number of Hidden Layers

It is common practice to use a single hidden layer for ANN simulation metamodelling. To test whether or not this standard assumption is valid, 10 training runs were executed using the LM algorithm for an ANN structure with a single hidden layer with 10 hidden nodes, and also for an ANN structure with two hidden layers with 7 hidden nodes per layer as in Fig. 17.

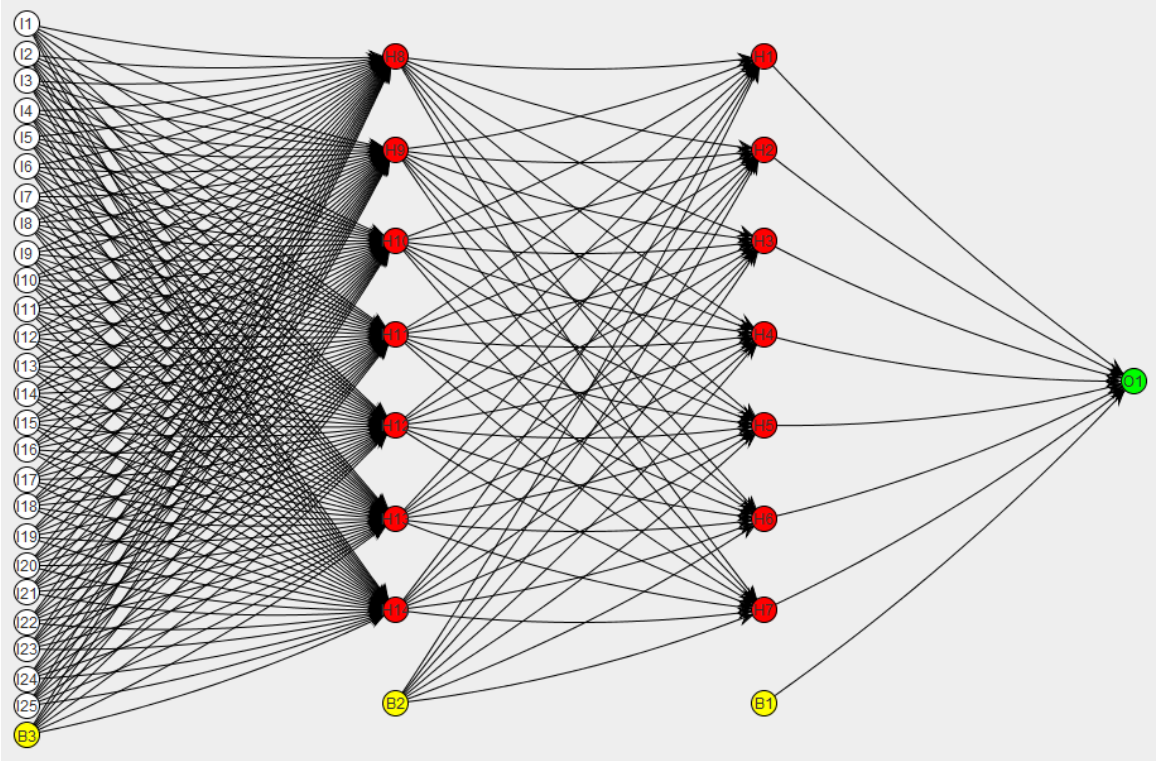

Figure 17: Two layer ANN structure

Each ANN structure has a similar number of connections, including biases, as seen in Table 17.

Table 17: ANN structures for comparison

Number of hidden layers	Nodes per hidden layer	Total number of connections
1	10	271
2	7	246

The results of the experiment are listed in Table 18.

Table 18: Comparison by number of hidden layers

Number of hidden layers	Iterations	Time(s)	Training MSE (MSE_T)	Validation MSE (MSE_V)	$\frac{MSE_V - MSE_T}{MSE_V}$
1	143(76)	19(14)	3.53E-4(1.45E-4)	3.70E-4(1.44E-4)	5.10E-2(3.70E-2)
2	164(68)	29(13)	1.16E-4(1.61E-5)	1.23E-4(1.65E-5)	6.00E-2(3.60E-1)

Although there is no significant difference in the number of iterations required, the time required for training, or the overtraining measure, the two hidden layer architecture performs almost three times better in validation MSE. The box plot in Fig. 18 also illustrates the superior consistency of the two hidden layer architecture results; the sample standard deviation is lower by an order of magnitude.

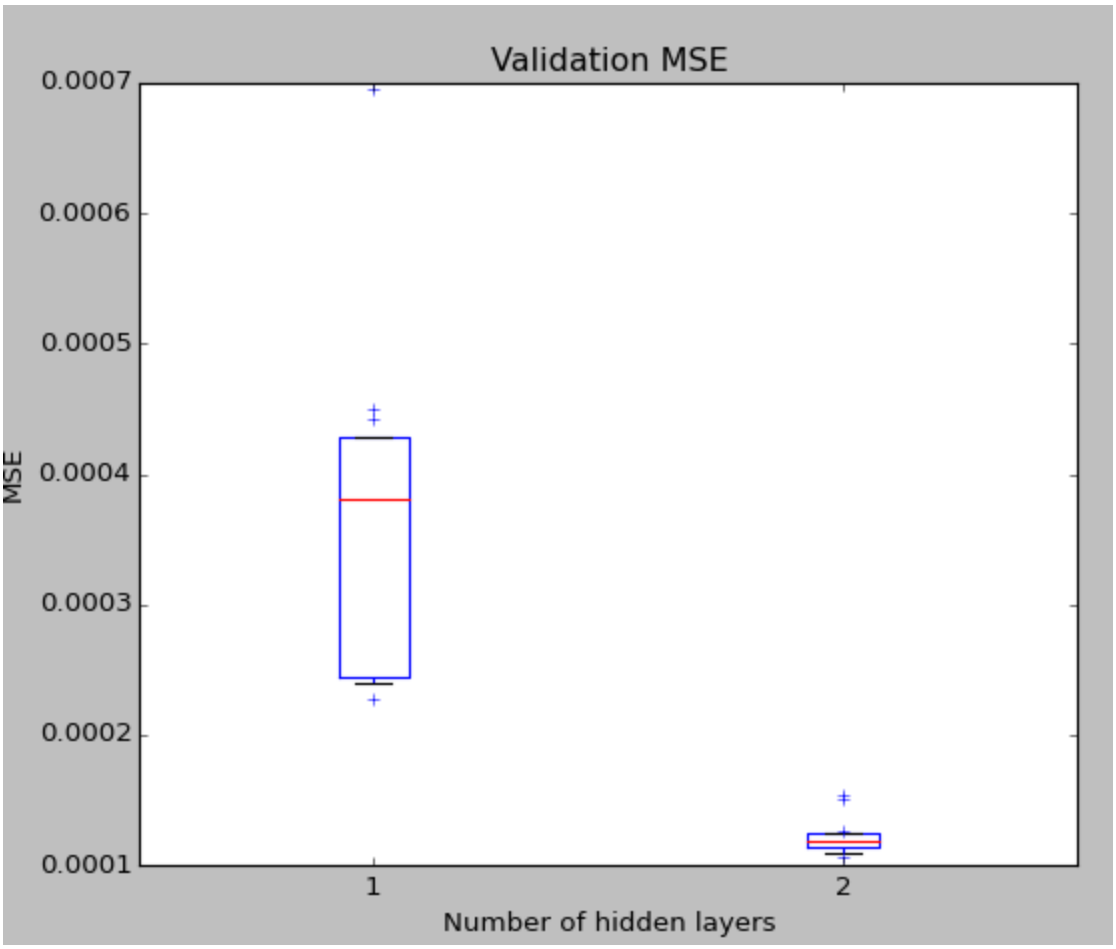

Figure 18: Validation MSE by number of hidden layers

For this particular task, it is evident that a two hidden layer ANN is a superior metamodel than a single hidden layer ANN of similar complexity. It is worth nothing that the simulations here are essentially deterministic with very little uncertainty. It remains to be seen if similar conclusions would hold if the simulations were much noisier.

4.5 Metamodel Selection

An ANN metamodel was trained using the two hidden layer, seven hidden nodes per layer architecture using simulation results based on product mix 1 (see table 5). The trace of MSE during training is found in Appendix D. The validation MSE for this ANN is 0.00011, which is very close to the target MSE of 0.0001. A histogram of the error in Fig. 19 shows that for the majority of samples, absolute error is within the desired bound of 1%.

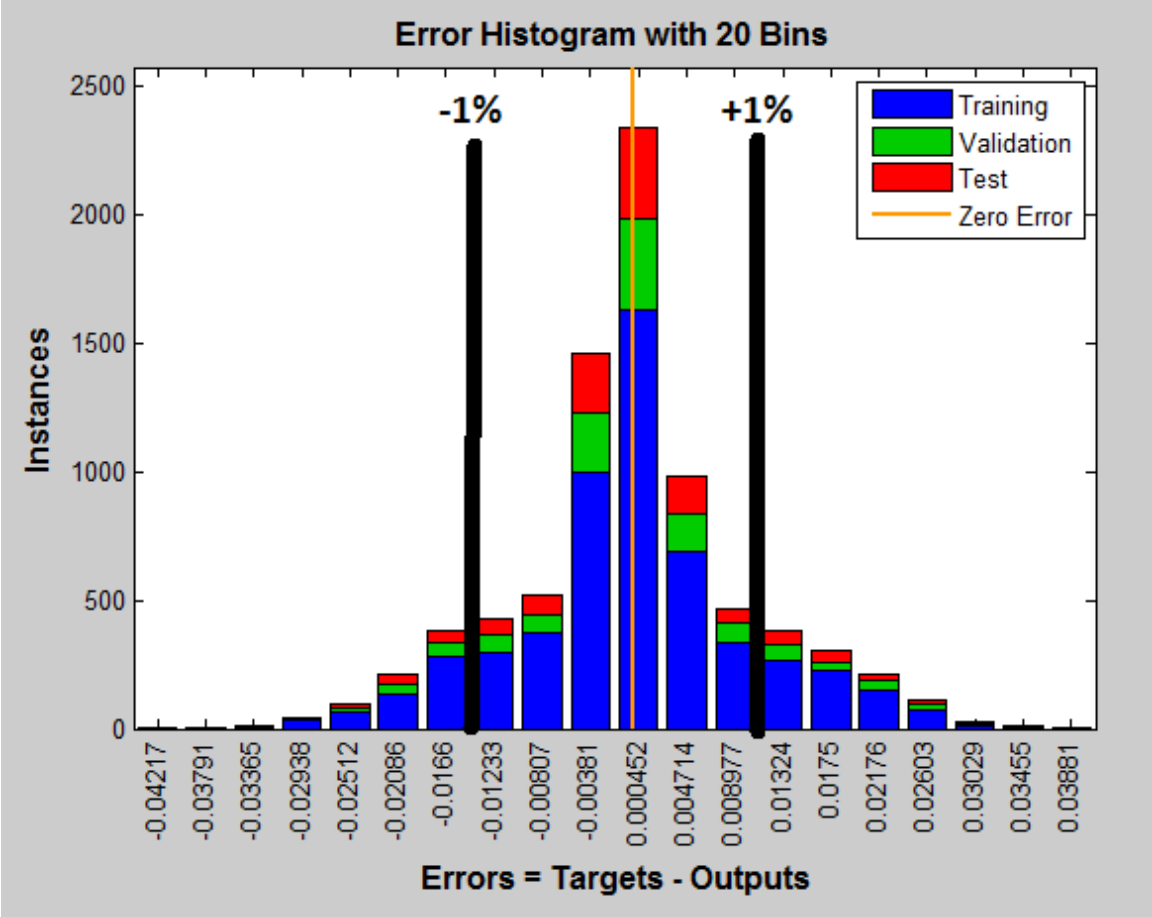

Figure 19: Error histogram for two hidden layer ANN structure

Thus, this is the ANN used for analysis in Chapter 4.7.

4.6 Comparison by Training Algorithm

To select the best training algorithm for the task at hand, a structure with two hidden layers and seven hidden nodes at each layer is used for all runs, except the runs with evolving topology (NEAT).

At each hidden node, a logistic activation function is used (see Eq. 5). For this study, ten runs were completed using RPROP+, SCG, LM, and BFGS, while two runs were performed for GA and NEAT. At the start of each run, all weights and biases are randomly assigned. The stopping condition for the GA and NEAT algorithms is 1,200

seconds of computation time, while the other algorithms stop training when the test MSE begins to rise. The implementation of the GA and NEAT in ENCOG is used for comparison with implementations of RPROP+, BFGS algorithm, and the LM algorithm in MATLAB.

Table 19: Training algorithm experiment results

Training algorithm	Iterations	Time(s)	Training MSE (MSE_T)	Validation MSE (MSE_V)	Parameter settings
LM	139(34)	22(7)	1.10E-4(3.57E-5)	1.18E-4(7.12E-5)	MATLAB default
BFGS	360(85)	21(6)	3.33E-4(1.11E-4)	3.40E-4(9.66E-5)	MATLAB default
RPROP+	781(298)	5(2)	3.80E-4(8.10E-5)	4.18E-4(1.01E-4)	MATLAB default
SCG	383(128)	5(2)	4.58E-4(1.76E-4)	4.69E-4(1.80E-4)	MATLAB default
GA1	77	1,200	3.80E-04	3.53E-04	Mutation rate = 0.1 Crossover rate = 0.0025
GA2	52	1,200	7.02E-04	6.60E-04	Mutation rate = 0.1 Crossover rate = 0.0025
NEAT1	6	1,200	1.52E-02	1.54E-02	Population size = 10
NEAT2	15	1,200	6.27E-03	6.37E-03	Population size = 10

It is clear from the results in Table 19 that the evolutionary training algorithms are less efficient for this task than gradient-based or second-order training algorithms. Perhaps if training had been allowed to continue, the NEAT approach would have achieved similarly low MSE, albeit much more slowly than the others. However, the GA training approach has a validation MSE in the same order of magnitude as the other approaches. Still, the GA algorithm takes much longer to train, and therefore it is inferior to the gradient-based or second-order training methods in its current implementation. Another approach could have been to define a structure using the NEAT approach and then use a gradient-based approach to train the network. Validation MSEs for the RPROP+, SCG, LM, and BFGS algorithms are shown as box plots in Fig. 20.

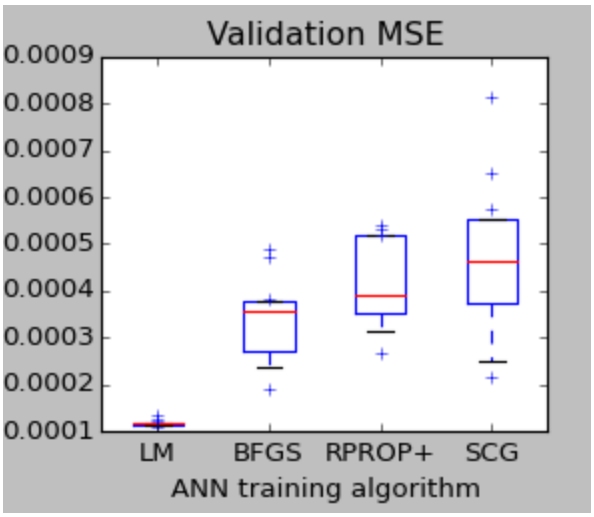

Figure 20: Validation MSE for gradient-based and second-order algorithms

The LM algorithm is far superior to the BFGS, RPROP+, and SCG algorithms in terms of validation MSE. The LM also has a smaller sample standard deviation, indicating that it is also less dependent on the starting conditions. The LM algorithm achieves its impressive performance with fewer iterations than the other algorithms, but these iterations take longer to complete. Thus, computation time per run (rather than per iteration) is compared on the box plot in Fig. 21.

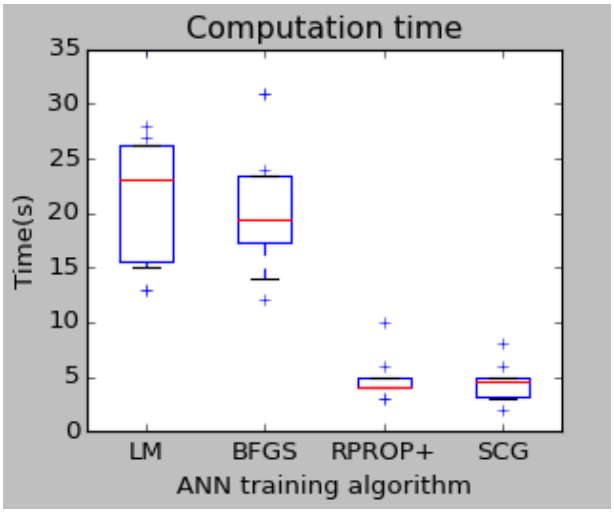

Figure 21: Validation MSE for gradient-based and second-order algorithms

RPROP+ and SCG have an advantage in computation time over LM and BFGS when you consider how the length of time required to reach the stopping criterion. However, this advantage does not sufficiently compensate for the inferior validation MSE results, especially when each run takes less than a minute. For this task, the LM algorithm is sufficiently fast and clearly outperforms the other algorithms. However, care should be taken when selecting the LM algorithm as the number of connections in the ANN – the amount of memory required by the LM algorithm rises proportionally to the square of the number of connections. This is not an issue for the chosen network structure since this number is quite small.

4.7 Neural Network Metamodel Results

Once the ANN has been trained using the network structure specified in chapter 4.6, the ANN is then used for further analysis. In previous work, MacDonald and Gunn (2011) used exchange curves to evaluate the trade-offs between two competing performance measures. For example, inventory levels may be considered against reorder quantities, throughputs, or other variable which have competing values. However, for this system, inventory and staffing costs are insignificant compared to the value of producing additional throughput, so exchange curves do not help to determine production parameters. Instead, a simulated annealing algorithm [see Kirkpatrick et al. (1983) for original work, Bertstimas and Tsitsiklis (1994) for a description of the method and its behaviour] is used to generate solutions, and then the feasibility and accuracy of these solutions are assessed. Simulated annealing is a search technique for finding solutions that uses local improvement procedures, can escape local minima/maxima, and has the larger strategic purpose of performing a robust search of solution space (Gendreau and Potvin 2010). The ANN is used as the objective function to evaluate generated solutions and the neighbouring structure.

Using a simulated annealing algorithm directly for each of the 25 input parameters will not lead to converging solutions across runs due to the size of the search space. However, it is possible to fix some of the inputs to reduce the size of the search space. Consider in

inventory control, the economic ordering quantity Q is proportional to the square root of demand, D, as seen in Eq. (8).

$$Q = \sqrt{WD} \qquad (8)$$

In this equation, W is a problem-specific parameter composed of costs. Using this principle, the stock level for each product code in the TK is assumed to be approximately proportional to the square root of the demand for that product in curing, although other possible rules could have been used to generate alternative tables. Under this assumption, all of the stock levels can be controlled by a single parameter, Stock_level, as seen in Table 20. The value of the variable Stock_level is an index to point to a set of maximum stock levels for products in the TK. Values are rounded to the nearest integer.

Table 20: Stock level variable for simulated annealing

Stock_level	1	2	3	4	5	6	7	8	9
Max stock, P1	8	10	11	12	13	14	15	16	17
Max stock, P2	6	7	8	9	9	10	11	12	12
Max stock, P3	6	7	8	9	9	10	11	12	12
Max stock, P4	5	5	6	7	7	8	9	9	10
Max stock, P5	5	6	7	7	8	9	9	10	10
Max stock, P6	5	6	6	7	7	8	9	9	10
Max stock, P7	4	4	5	5	5	6	6	7	7
Max stock, P8	5	6	6	7	8	8	9	10	10
Max stock, P9	5	5	6	6	7	8	8	9	9
Max stock, P10	3	4	4	4	5	5	5	6	6
Max stock, P11	6	7	8	8	9	10	11	11	12
Max stock, P12	6	6	7	8	8	9	10	10	11
TK max stock	**64**	**73**	**82**	**89**	**95**	**105**	**113**	**121**	**126**

Restricting the stocks levels in this manner, possibly considering a few alternative ways of constructing the stock level table could have been done as part of the LHD to simplify the ANN training task. The first 13 parameters described in table 13 could have been replaced with two variables, one for the stock levels themselves and another for the choice of which method is used to generate the stock levels. The ANN would have been likely to be much more accurate in the search region where the maximum stock levels of

each product are approximately proportional to each other. It is preferable to restrict stock level parameters in some way for future ANN experiments. Regardless of the technique used to reduce the search space, simulated annealing can now be used to search the region for the best solution. Initially, when the search space was not limited, the simulated annealing algorithm failed to converge. In the following experiment, when the search space was limited, the results improved significantly.

Twenty simulated annealing runs were carried out with random initial conditions. The parameter for the number of empty TK slots was fixed at 0, and a single parameter was used for stock levels as illustrated in Table 20. The parameters for changeover policy, number of operators, shared operators, and operator behaviours were free to change across their specified ranges. The simulated annealing algorithm considered neighbouring solutions to be any two solutions where only one parameter differs anywhere in its range. Thus, in the simulated annealing algorithm, only one parameter can be changed at a time. The pseudocode for the simulated annealing algorithm is found in Fig. 22. The full MATLAB code is found in Appendix F along with a table containing all of the experiment results.

```
Initialize temperature, first random solution
For each of 200 iterations
        For each of the 7 parameters
                For each of 5 randomly generated random solutions
                        If the generated neighbouring solution is better than the current solution
                                Replace current solution with generated solution
                        Else
                                Replace current solution with generated solution with probability
                                e^((Objective of current soln - Objective of generated soln)/temp)
                End For
                Update temperature
        End For
End For
Report final solution
```

Figure 22: Pseudocode for the simulated annealing algorithm

A summary of the experiment results is shown in Table 21 below.

Table 21: Summary of simulated annealing runs

	Stock Parameter	Changeover Policy	Operator behaviour	Shared operators	Machine A operators	Machine B operators	Curing operator breaks covered
Range	1-9	1-5	1-3	0-1	4-10	4-9	0-1
Most common occurrence	1	4	1	1	10	5	1
Count of most common occurrence	20	20	20	20	9	9	19

Each of the simulated annealing runs converged to similar solutions. The main difference between runs is the number of machine A and machine B operators. All twenty runs ended with the stock level parameter set to a value of 1. This indicates that lower stock levels in the TK are preferred. This is a sensible result, since with lower stock levels, the TK cycle time is reduced and the production line runs more smoothly. However, lower stock levels may be a concern if variation is introduced in MProd-building or in curing, due to a higher probability of stock outs in the TK. This is probably why the most efficient curing operator behaviour and shared operators in MProd-building are preferred for all twenty runs. In addition, covering the breaks of curing operators is preferred for 19/20 runs. These parameter choices reduce the variation in the system, and they appear to be preferred along with lower stock levels to achieve the best results.

Another interesting result is that while a higher number of machine A and machine B operators appears to be preferred, the ANN metamodel does not necessarily select the maximum number of operators each time. This is probably because an operator is not needed for each machine, so the ANN is unable to identify exactly how many operators are ideal because it does not matter once the total number of operators reaches a certain value. The total number of operators between machine A and machine B is exactly 15 for all twenty runs. The results indicate that even though there are 19 machines to operate, only 15 operators are required for the production line to operate at full capacity.

The simulated annealing runs gave an average daily throughput of 479.19 MProds with a sample standard deviation of 0.075 MProds. Considering that the maximum observed average daily throughput for all of the 7,986 sample simulation runs was 483 MProds, the results appear to be both realistic and useful. If the stock levels had not been restricted, perhaps the simulated annealing algorithm would have been able to find better solutions. However, it would have been less likely to converge across runs. A direction for future research could be to examine whether restricting stock levels at the experiment design stage would improve ANN accuracy in these regions of interest. Further, it would be worthwhile to examine several different approaches to restricting stock levels in the TK. In this thesis, stock levels are chosen to be proportional to the square root of demand. However, linearly proportional stock levels would be another option. Minimum inventory levels could have also been established, which could be a third way to develop a stock level policy.

Chapter 5: Conclusion

In this thesis, two closely related discrete-event simulation models are developed using SimPy and Python. The first detailed model is used to evaluate the capability of the MHS of a production line with five anticipated future production configurations. The limitations of the current system are outlined and several opportunities for improving daily throughput are identified. The second simulation model adds additional detail to the build process and uses the PAC System to impose production control on the line. The object-oriented simulation model is built to allow changes to machine programmable logic, machine interaction effects, operator behaviours, and changeover policies.

An ANN is then trained as a simulation metamodel. ANNs are a powerful tool for the metamodelling of complex simulations; however, care must be taken in the design of experiments to maximize the usefulness of the metamodel. In this case, by concentrating the samples in the regions of interest, carefully determining the ANN structure, and selecting a strong training algorithm, a sufficiently accurate metamodel was trained using simulation data. When considering how to structure an ANN for simulation metamodelling, this work demonstrates that it is important to consider validation MSE, the risk of overtraining, and real world requirements. If the number of hidden nodes is too low, the structure of the ANN will not be sufficiently complex for accurate training. Conversely, too many hidden nodes lead to overtraining and rapidly diminishing returns on computational time. It has been demonstrated that for this real-world task, an ANN structure with two hidden layers is superior to an ANN structure with a single hidden layer using an approximately equal number of connections.

Using the trained ANN as the objective function, simulated annealing is used to search the input space for strong input parameters. The results of the twenty simulated annealing runs tended to converge to similar results for all runs, indicating that a global maximum result has likely been achieved. This result indicates a preference for parameters that could help to guide production decisions in the real system.

This research has expanded the knowledge of the capabilities and applications of neural networks as metamodels, an area that has been of significant interest to members of the Department of Industrial Engineering (MacDonald and Gunn 2011). Additionally, the Department has gained further insight into the power of designing a simulation model using Python, SimPy, and open source visualization tools. This work has been an opportunity to apply concepts of production control systems generated by the application of the PAC System to the production line under study. This is one of the first actual applications that we are aware of.

From the perspective of the manufacturing company, this work has contributed significantly to company knowledge of discrete event simulation modelling. Prior to this work, opportunities to apply simulation modelling were not always recognized because the modelling effort was not fully embedded in their actual production control and design decision processes. Now, as a result of this work, a group of engineers has been established to continue to apply simulation to other production lines and processes at the manufacturing facility. The company has gained insight into how simulation adds to the knowledge of current processes; instead of being focussed on one manufacturing cell, to produce a detailed and accurate model the modeller must completely understand the entire production line. Throughout this project, unexpected opportunities for improvement were identified and then subsequently quantified using the simulation model. These opportunities include: relocating the small metallic identifier placement process to tire-building, enhancing communication between tire-building and curing by establishing a production control system, establishing stock level targets for each product in the TK, managing the repair circuit, and improving the programmable logic of the TK. Research has demonstrated some of the value that simulation modelling presents, not just as a tool for analysis, but as an important part of continuously improving the efficiency and effectiveness of the production line.

This work has outlined a complete framework for the optimization of production control parameters in a real manufacturing facility. It has been demonstrated that an ANN can be used as a simulation metamodel for specialized or complex manufacturing systems

controlled by the PAC System. Further, the work has shown how the metamodel can be used as the objective function for a simulated annealing search to determine production parameters for the line. While this is significant, there are several opportunities to improve upon and expand this body of work.

Future research could investigate the effectiveness of several methods for fixing stock levels relative to one another before training and during the search process. It would be worthwhile to examine the impact of using screening experiments to select variables to use in the neural network. It would also be interesting to see whether using non-deterministic travel times for the trolleys have any impact on simulation results.

For similar work in the future, researchers may wish to try using the generated NEAT structures and applying gradient-based algorithms to those structures to see if these structures can produce a more accurate network. It would also be worthwhile to try adding additional layers to the neural network to determine at what stage the network begins to overtrain. Applying the entire methodology to additional product mixes is also a logical next step for research to continue. Researchers or engineers working with this model may wish to modify the number of process tags at each cell to accommodate production control systems other than the base stock system. The simulation model can be used to test changes to TK logic and functionality. The model could also be enhanced to include prep and to model customer demand. By adding new production mixes to the model, the simulation model can continue to provide guidance for future production decisions.

References

Alam, Fasihul M., Ken R. Mcnaught, and Trevor J. Ringrose. "A Comparison of Experimental Designs in the Development of a Neural Network Simulation Metamodel." *Simulation Modelling Practice and Theory* 12.7-8 (2004): 559-78.

Badiru, Adedeji B., and David B. Sieger. "Neural Network as a Simulation Metamodel in Economic Analysis of Risky Projects." *European Journal of Operational Research* 105.1 (1998): 130-42.

Barton, R. R. "Simulation Metamodels." Proceedings of the 1998 Winter Simulation Conference, Washington, D.C. (1998): 167-174.

Baudin, Michael. *Python Design of Experiments*. Computer software. *PyDOE: The Experimental Design Package for Python*. Vers. 0.3.7. (2012) <http://pythonhosted.org/pyDOE/>

Beaumont, Chris, and M. Pidd. "Computer Simulation in Management Science." *The Journal of the Operational Research Society* 35.7 (1984): 679.

Bertsimas, Dimitris, and John Tsitsiklis. "Simulated Annealing." *Statistical Science Statist. Sci.* 8.1 (1993): 10-15.

Buzacott, J. A., and L. E. Hanifin. "Models of Automatic Transfer Lines with Inventory Banks a Review and Comparison." *A I I E Transactions* 10.2 (2007): 197-207.

Buzacott, John A., and J. George Shanthikumar. "A General Approach For Coordinating Production In Multiple-Cell Manufacturing Systems." *Production and Operations Management* 1.1 (1992): 34-52.

Can, Birkan, and Cathal Heavey. "A Comparison of Genetic Programming and Artificial Neural Networks in Metamodelling of Discrete-event Simulation Models." *Computers & Operations Research* 39.2 (2011): 424-36.

Carson, Yolanda, and Anu Maria. "Simulation Optimization." *Proceedings of the 29th Conference on Winter Simulation - WSC '97* (1997)

Chen, Mu-Chen, and Taho Yang. "Design of Manufacturing Systems by a Hybrid Approach with Neural Network Metamodelling and Stochastic Local Search."*International Journal of Production Research* 40.1 (2002): 71-92.

Cigolini, Roberto, Margherita Pero, and Tommaso Rossi. "An Object-oriented Simulation Meta-model to Analyse Supply Chain Performance." *International Journal of Production Research* 49.19 (2011): 5917-941.

Durieux, Séverine, and Henri Pierreval. "Regression Metamodelling for the Design of Automated Manufacturing System Composed of Parallel Machines Sharing a Material Handling Resource." *International Journal of Production Economics* 89.1 (2004): 21-30.

Eldabi, Tillal, and Ray J. Paul. "Evaluation of Tools for Modelling Manufacturing Systems Design with Multiple Levels of Detail." *The International Journal of Flexible Manufacturing Systems* 13 (2001): 163-76.

Encog Machine Learning Framework. Computer software. Vers. 3.1. Heaton Research, Inc., n.d. Web.

Friedman, L. W. "The Simulation Metamodel." *Journal of the Operational Research Society* 48.8 (1997): 850.

Gamma, E. *Design Patterns: Elements of Reusable Object-oriented Software*. Reading: Addison-Wesley Pub., 1995

Gendreau, Michel, and Jean-Yves Potvin. *Handbook of Metaheuristics*. New York: Springer, 2010.

Ghaffari, A., H. Abdollahi, M.r. Khoshayand, I. Soltani Bozchalooi, A. Dadgar, and M. Rafiee-Tehrani. "Performance Comparison of Neural Network Training Algorithms in Modelling of Bimodal Drug Delivery." *International Journal of Pharmaceutics* 327.1-2 (2006): 126-38.

Gosavi, Abhijit. *Simulation-based Optimization: Parametric Optimization Techniques and Reinforcement Learning*. Boston: Kluwer Academic, 2003.

Hagan, Martin T., Howard B. Demuth, and Mark H. Beale. *Neural Network Design*. Boston: PWS Pub., 1996.

Harris, Ford W. "How Many Parts to Make at Once." Operations Research 38.6 (1990): 947-50.

Hur, Sun, Young Hae Lee, Si Yeong Lim, and Moon Hwan Lee. "A Performance Estimation Model for AS/RS by M/G/1 Queuing System." *Computers & Industrial Engineering* 46.2 (2004): 233-41.

Hurrion, R. D., and S. Birgil. "A Comparison of Factorial and Random Experimental Design Methods for the Development of Regression and Neural Network Simulation Metamodels." *Journal of the Operational Research Society* 50.10 (1999): 1018-033.

Hurrion, R. D. "An Example of Simulation Optimisation Using a Neural Network Metamodel: Finding the Optimum Number of Kanbans in a Manufacturing System." *Journal of the Operational Research Society* 48.11 (1997): 1105-112.

Illingworth, W.t. "Beginner's Guide to Neural Networks." *IEEE Aerospace and Electronic Systems Magazine* 4.9 (1989): 44-49.

Kindler, Eugene, and Ivan Krivy. "Object-oriented Simulation of Systems with Sophisticated Control." *International Journal of General Systems* 40.3 (2011): 313-43.

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing." *Science* 220.4598 (1983): 671-80. Web.

Kuo, Yiyo, Taho Yang, Brett A. Peters, and Ihui Chang. "Simulation Metamodel Development Using Uniform Design and Neural Networks for Automated Material Handling Systems in Semiconductor Wafer Fabrication." *Simulation Modelling Practice and Theory* 15.8 (2007): 1002-015.

Law, Averill M. *Simulation Modelling and Analysis*. Boston: McGraw-Hill, 2007.

Livieris, Ioannis E., and Panagiotis Pintelas. "A New Conjugate Gradient Algorithm for Training Neural Networks Based on a Modified Secant Equation."*Applied Mathematics and Computation* 221 (2013): 491-502.

MacDonald, Corinne, and Eldon A. Gunn. Ed. C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Urmacher. “Allocation of Simulation Effort for Neural Network vs. Regression Metamodels.” *Proceedings of the 2012 Winter Simulation Conference* (2012).

MacDonald, Corinne, and Eldon A. Gunn. "A Framework for Analysis of Production Authorization Card-Controlled Production Systems." *Production and Operations Management* 20.6 (2011): 937-48.

Mandischer, M. "A Comparison of Evolution Strategies and Backpropagation for Neural Network Training." *Neurocomputing* 42.1-4 (2002): 87-117.

MATLAB & SIMULINK. Vers. 2014a. Natick, MA: Mathworks, 2014. Computer software.

McCloone, Sean, and George Irwin. "A Variable Memory Quasi-Newton Training Algorithm." *Neural Processing Letters* 9.1 (1999): 77-89.

Mckay, M. D., R. J. Beckman, and W. J. Conover. "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code." Technometrics 21.2 (1979): 239.

Moller, Martin F. "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning." *Neural Networks* 6. (1993): 525-533.

Mukherjee, Indrajit, and Srikanta Routroy. "Comparing the Performance of Neural Networks Developed by Using Levenberg–Marquardt and Quasi-Newton with the Gradient Descent Algorithm for Modelling a Multiple Response Grinding Process." *Expert Systems with Applications* 39.3 (2012): 2397-407.

Nance, Richard E. "A History of Discrete Event Simulation Programming Languages." *ACM SIGPLAN Notices* 28.3 (1993): 149-75.

Python Software Foundation. 2015. Python Language Reference, version 2.7. Available at http://www.python.org.

Raydan, Marcos. "The Barzilai and Borwein Gradient Method for the Large Scale Unconstrained Minimization Problem." *SIAM Journal on Optimization SIAM J. Optim.* 7.1 (1997): 26-33.

Robinson, S. "Discrete-event Simulation: From the Pioneers to the Present, What Next?" *Journal of the Operational Research Society* 56.6 (2004): 619-29.

Sabuncuoglu, Ihsan, and Souheyl Touhami. "Simulation Metamodelling with Neural Networks: An Experimental Investigation." *International Journal of Production Research* 40.11 (2002): 2483-505.

Smith, Jeffrey S. "Survey on the Use of Simulation for Manufacturing System Design and Operation." *Journal of Manufacturing Systems*22.2 (2003): 157-71.

Stanley, Kenneth O., and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies." *Evolutionary Computation* 10.2 (2002): 99-127.

Swain, James J. "Simulation: A Better Reality? Imagine the World as It Is and What It Might Be: Biennial Survey of Discrete-event Simulation."*OR/MS Today* 40.5 (2013): 48-52.

Tako, A. A., and S. Robinson. "Comparing Discrete-event Simulation and System Dynamics: Users' Perceptions." *Journal of the Operational Research Society J Oper Res Soc* 60.3 (2008): 296-312.

Tako, A. A., and S. Robinson. "Model Development in Discrete-event Simulation and System Dynamics: An Empirical Study of Expert Modellers." *European Journal of Operational Research* 207.2 (2010): 784-94.

Wang, G. Gary, and S. Shan. "Review of Metamodelling Techniques in Support of Engineering Design Optimization." *Journal of Mechanical Design* 129.4 (2007): 370.

Watson, Richard A., and Jordan B. Pollack. "A Computational Model of Symbiotic Composition in Evolutionary Transitions." *Biosystems* 69.2-3 (2003): 187-209.

Welch, P. "The Statistical Analysis of Simulation Results." *Computer Performance Modeling Handbook*. New York: Academic, 1983. 268-328.

"Welcome to SimPy." *Overview — SimPy 3.0.8 Documentation*. Web. https://simpy.readthedocs.org/en/latest/. 1 September 2014.

Appendix A: Comparison of Trolley A Demands

When the simulation is in the first mode, it uses a discrete probability distribution to determine how likely it is that a MProd is produced during each 15-minute interval. The distribution is fitted using historical data. The discrete probability distribution is shown in Fig. A1 below.

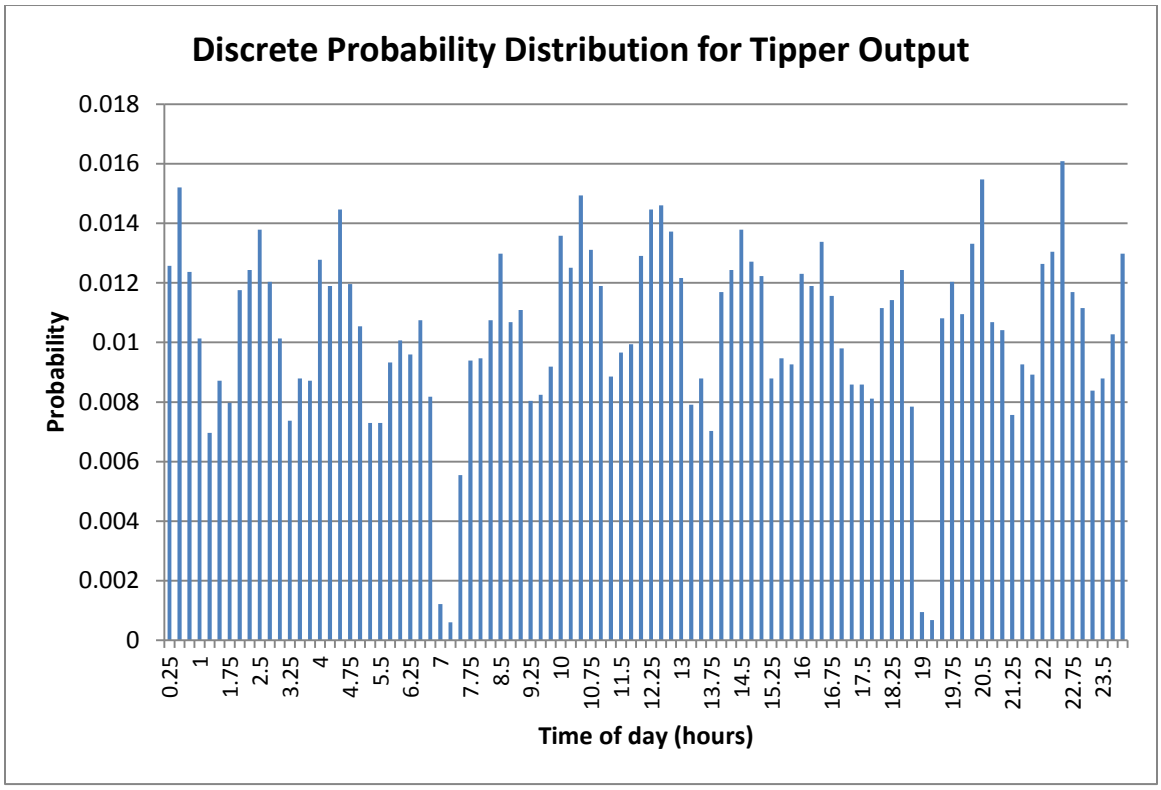

Figure A 1: Discrete probability distribution for tipper output divided into 15-minute intervals

Appendix B: TK Class Python Code

```
class TK(object):
    def __init__(self,env):
        self.env = env
        self.name = "TK"
        self.count = 0

        ## Define output job queue
        self.outputJobs = []
        q.heapify(self.outputJobs)

        ## Define all possible product stations. 0 means no TK pod,
        ## 1 means an empty TK pod, and a tuple indicates a cover.
        self.s = 
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,

1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,

1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0]
        self.sLift = 0

        ## Define other important variables
        self.position = 0
        self.travelTime = TKTravelTimes
        self.slotList_in = TKSlotPriority_in
        self.slotList_out = TKSlotPriority_out
        self.TBStatus = 0
        self.wait1 = self.env.event()
        self.process = env.process(self.main_lift())
        self.next_easy = False
        self.outputrunning = False
        self.efpjobs = 0
        self.waitfortrolley = self.env.event()
        self.waitforB = self.env.event()
        self.res_pc = [0,0,0]
        self.busytrolley = [False, False, False]

        ## Define stat tracking variables
        self.throughput = 0
        self.workingtime = 0.0
        self.downtime = 0.0
        self.idletimeA = 0.0
        self.idletimeB = 0.0

        ## Set up heaps for each dim code
        self.Tkheaps = []
        self.dimdict = {}
        for i in range(len(g.dim_codes)):
            self.Tkheaps.append([])
            q.heapify(self.Tkheaps[i])
            self.dimdict.update({g.dim_codes[i]:self.Tkheaps[i]})
```

```
        if debug_TK:
            print "TK initialized."

        ## Set up priority options
        if not useGUI:
            self.p_output_covers = 1
            self.p_input_covers = 2
            self.p_output_TKpod = 3
            self.p_input_TKpod = 4
        else:
            self.p_output_covers = v[55]
            self.p_input_covers = v[56]
            self.p_output_TKpod = v[57]
            self.p_input_TKpod = v[58]

        self.pdict = {}
        self.pdict.update({self.p_input_covers:self.check_for_input_job})
        self.pdict.update({self.p_output_covers:self.check_for_output_job})

self.pdict.update({self.p_input_TKpod:self.check_if_input_TKpod_needed})
        self.pdict.update({self.p_output_TKpod:self.check_for_output_TKpod})
        self.ptaskdict = {}
        self.ptaskdict.update({self.p_input_covers:self.do_input_job})
        self.ptaskdict.update({self.p_output_covers:self.do_output_job})

      self.ptaskdict.update({self.p_input_TKpod:self.get_empty_TKpod_for_input
      })

self.ptaskdict.update({self.p_output_TKpod:self.put_away_output_TKpod})

    def output_mechanism(self):
        """
        TKpods with covers destined for pickup from trolleys E/F/P trigger
        the output mechanism for repositioning.

        After pickup, the output mechanism places the empty TKpod on the lower
        shelf for pickup.
        """
        self.waitfor126 = self.env.event()
        self.waitfortrolley = self.env.event()
        self.waitforself = self.env.event()
        if self.outputrunning == True:
            yield self.waitforself
        self.outputrunning = True
        ## Get in position for pickup
        yield self.env.timeout(46/3600./speedfactor_TK)
        ## Pickup
        if self.trigger_trolleys_output() == True:
            yield self.env.timeout(0.212/60./speedfactor_TK)
        else:
            yield self.waitfortrolley
            yield self.env.timeout(0.212/60./speedfactor_TK)
        ## Place empty TKpod on lower shelf
        if debug_TK:
```

```
            print "%s - Trolley just picked from
output."%(timify(self.env.now))

        yield self.env.timeout(20/3600./speedfactor_TK)
        if self.s[g.lower_output_slot] != 0:
            yield self.waitfor126
        self.s[g.upper_output_slot] = 0
        ## Tell the MS and TK that the output is available again
        if debug_TK:
            print "%s - Output ready for new cover."%(timify(self.env.now))
        try:
            MS.wait1.succeed()
        except RuntimeError:
            pass
        try:
            self.wait1.succeed()
        except RuntimeError:
            pass
        ## Finish placing TKpod
        yield self.env.timeout(26/3600./speedfactor_TK)
        self.s[g.lower_output_slot] = 1
        try:
            self.wait1.succeed()
        except RuntimeError:
            pass
        try:
            PriorityTrolleys[1].tkfullwait.succeed()
        except RuntimeError:
            pass
        ## Go back to ready position
        yield self.env.timeout(10/3600./speedfactor_TK)
        self.outputrunning = False
        try:
            self.waitforself.succeed()
        except RuntimeError:
            pass

    def main_lift(self):
        """
        Primary TK logic stream.
        """
        if dCell:
            global direct

        while True:
            if debug_TK:
                print "%s - Reprioritizing..."%timify(self.env.now)

            if self.next_easy:
                yield self.env.process(self.ptaskdict[self.p_input_covers]())
                continue

            if self.pdict[1]():
                yield self.env.process(self.ptaskdict[1]())
                continue
```

```
            if self.pdict[2]():
                yield self.env.process(self.ptaskdict[2]())
                continue

            if self.pdict[3]():
                yield self.env.process(self.ptaskdict[3]())
                continue

            if self.pdict[4]():
                yield self.env.process(self.ptaskdict[4]())
                continue

            ## If there are no jobs to do, wait until something changes
            self.wait1 = self.env.event()
            if debug_TK:
                print "%s - TK is waiting..."%timify(self.env.now)
            yield self.wait1
            continue

    def do_output_job(self):
        job = q.heappop(self.outputJobs)

        ## See if it's in the direct cell
        if dCell and DC.MProd!=None and job[5][4][2] == DC.MProd[4][2]:
            self.send_order_tags(job[3])
            direct = True
            out_MProd = job[5]
            ## Don't take a cover from the TK
            q.heappush(TK.dimdict[job[3]], out_MProd)
            ## Tell the trolley to go get the MProd
            q.heappush(queue_Trolleys[job[4]],(job[0],job[1],
            out_MProd[2],out_MProd[3], out_MProd[4]))
            self.efpjobs += 1
            self.res_pc[job[4]] += 1
            self.trigger_trolleys(job[4])
            yield self.env.timeout(0.00000000001)

        else:
            ## If the trolley for the first job is unavailable, pick another
job
            if self.busytrolley[job[4]] == True:
                job = self.choose_trolley_strategically(job)
            self.send_order_tags(job[3])
            ## Go to the slot of the dim code requested using FIFO
            out_MProd = job[5]
            destination = self.next_dim_code_slot(job)
            yield self.env.process(self.travel_to(destination))
            if debug_TK:
                print "%s - New position:
%d"%(timify(self.env.now),self.position)
            ## Pick up the cover
            yield self.env.process(self.unstore_cover())
            ## Tell the next trolley to come
            q.heappush(queue_Trolleys[job[4]],(job[0],job[1],
```

```
            out_MProd[2],out_MProd[3], out_MProd[4]))
            self.efpjobs += 1
            self.res_pc[job[4]] += 1
            self.trigger_trolleys(job[4])
            ## Transport cover to the output
            yield self.env.process(self.travel_to(g.upper_output_slot))
            if debug_TK:
                print "%s - New position:
%d"%(timify(self.env.now),self.position)
            ## Place cover on output
            yield self.env.process(self.store_cover(1))
            self.env.process(self.output_mechanism())
            self.throughput += 1

    def do_input_job(self):
        ## Go to the input
        yield self.env.process(self.travel_to(g.input_slot))
        if debug_TK:
            print "%s - New position: %d"%(timify(self.env.now),self.position)

        if self.TBStatus == 2:
            yield self.env.process(self.post_input_TKpod_placement_logic())
        else:
            ## Pick up the TKpod
            yield self.env.process(self.unstore_cover())
            ## Transport it to nearest empty slot
            destination = self.nearest_empty(g.upper_output_slot)
            yield self.env.process(self.travel_to(destination))
            if debug_TK:
                print "%s - New position:
%d"%(timify(self.env.now),self.position)
            ## Put it away
            yield self.env.process(self.store_cover(self.sLift))

    def put_away_output_TKpod(self):
        ## Go to the lower output cell
        yield self.env.process(self.travel_to(g.lower_output_slot))
        if debug_TK:
            print "%s - New position: %d"%(timify(self.env.now),self.position)
        ## Get the empty TKpod
        yield self.env.process(self.unstore_cover())

        if self.s[g.input_slot] == 0:
            ## Travel to the input
            yield self.env.process(self.travel_to(g.input_slot))
            if debug_TK:
                print "%s - New position:
%d"%(timify(self.env.now),self.position)
            ## Place empty on input
            yield self.env.process(self.store_cover(1))
            self.trigger_tb()
            yield self.env.process(self.post_input_TKpod_placement_logic())

        ## Otherwise, bring empty TKpod to storage
        else:
```

```
            ## Go to empty slot as specified by logic in self.nearest_empty
            destination = self.nearest_empty(g.input_slot)
            yield self.env.process(self.travel_to(destination))
            if debug_TK:
                print "%s - New position:
%d"%(timify(self.env.now),self.position)
            ## Store TKpod
            yield self.env.process(self.store_cover(1))

    def get_empty_TKpod_for_input(self):
        ## Travel to empty TKpod according to logic in self.nearest_TKpod
        destination = self.nearest_TKpod(g.input_slot)
        yield self.env.process(self.travel_to(destination))
        if debug_TK:
            print "%s - New position: %d"%(timify(self.env.now),self.position)
        ## Get the empty TKpod
        yield self.env.process(self.unstore_cover())
        ## Travel to the input
        yield self.env.process(self.travel_to(g.input_slot))
        if debug_TK:
            print "%s - New position: %d"%(timify(self.env.now),self.position)
        ## Place empty on input
        yield self.env.process(self.store_cover(1))
        self.trigger_tb()

        yield self.env.process(self.post_input_TKpod_placement_logic())

    def check_for_input_job(self):
        if self.TBStatus == 2 and self.s[g.input_slot] == 1:
            return True
        try:
            int(self.s[g.input_slot])
            if self.s[g.input_slot] != 0 and self.s[g.input_slot] != 1:
                print "Error 111"
                raise AttributeError
            return False
        except TypeError:
            ## Do input job if it exists
            return True

    def check_for_output_job(self):
        if len(self.outputJobs) > 0 and self.s[g.upper_output_slot] == 0 and (
        self.efpjobs == 0):
            return True
        else:
            return False

    def check_for_output_TKpod(self):
        if self.s[g.lower_output_slot] == 1:
            return True
        else:
            return False
    def check_if_input_TKpod_needed(self):
        if self.TBStatus > 0 and self.s[g.input_slot] == 0:
            return True
```

```
    else:
        return False

def post_input_TKpod_placement_logic(self):
    if self.TBStatus > 0:
        self.waitforB = self.env.event()
        ## Wait 20 seconds to see if TB shows up
        self.env.process(self.release_lift())
        start_wait = self.env.now
        yield self.waitforB
        ## If it still hasn't shown up, do another job
        if self.TBStatus == 1:
            self.idletimeA+=self.env.now-start_wait
        ## If it's here and unloading, be a little patient
        elif self.TBStatus == 2:
            self.waitforB = self.env.event()
            yield self.waitforB
            self.next_easy = True
            self.idletimeA+=self.env.now-start_wait
        else:
            try:
                int(self.s[g.input_slot])
            except TypeError:
                self.next_easy = True
                self.idletimeA+=self.env.now-start_wait
            else:
                print "Error 6719"
                raise AttributeError
    else:
        yield self.env.timeout(0.00000000001)

def next_dim_code_slot(self,job):
    next_cover = job[5][4]
    for i in range(len(self.s)):
        try:
            if self.s[i][4] == next_cover:
                return i
        except TypeError:
            continue
    print next_cover
    print "Error 423"
    raise AttributeError

def nearest_empty(self,near_this_slot):
    """
    Returns location of nearest empty TK slot to the slot specified.
    """
    if near_this_slot == g.input_slot:
        for i in self.slotList_in:
            if self.s[i] == 0:
                return i
        print "No empty slots found!"
        raise AttributeError
    elif near_this_slot == g.upper_output_slot:
        for i in self.slotList_out:
```

```
            if self.s[i] == 0:
                return i
        print "No empty slots found!"
        raise AttributeError

def nearest_TKpod(self,near_this_slot):
    """
    Returns location of nearest empty TKpod to the slot specified.
    """
    if near_this_slot == g.input_slot:
        for i in self.slotList_in:
            if self.s[i] == 1:
                return i
        print "No empty TKpods found!"
        raise AttributeError
    elif near_this_slot == g.upper_output_slot:
        for i in self.slotList_out:
            if self.s[i] == 1:
                return i
        print "No empty TKpods found!"
        raise AttributeError

def store_cover(self,job):
    """
    Store a TKpod in one of the TK storage slots.
    """
    p = self.position
    if self.s[p] != 0:
        print "Error 91"
        raise AttributeError

    elif self.sLift == 1:
        self.s[p] = 1
        self.sLift = 0
    else:
        try:
            int(job)
        except TypeError:
            self.s[p] = (job[0],self.env.now,"out",job[3],job[4])
            self.sLift = 0
            q.heappush(self.dimdict[job[3]],(
                        job[0],self.env.now,"out",job[3],job[4]))
            try:
                MS.wait2.succeed()
            except RuntimeError:
                pass
    if p == g.upper_output_slot:
        self.s[p] = self.sLift
        self.sLift = 0

    time = random.triangular(17,18,20)/3600.
    yield self.env.timeout(time/speedfactor_TK)
    self.workingtime += time

    if debug_TK:
```

```
            print "%s - TKpod stored at:
%d"%(timify(self.env.now),self.position)

    def unstore_cover(self):
        """
        Remove a TKpod from the TK storage slot at the current position.
        """
        p = self.position

        if self.s[p] == 0 or self.sLift != 0:
            print "Error 486", self.s[p], self.sLift
            raise AttributeError

        elif self.s[p] == 1:
            self.s[p] = 0
            self.sLift = 1

        elif len(self.s[p]) > 1:
            self.sLift = self.s[p]
            self.s[p] = 0

        else:
            print "Error 486b"
            raise AttributeError

        if self.next_easy:
            if p != g.input_slot:
                print "next_easy error"
                raise AttributeError
            else:
                time = random.triangular(10,11,12)/3600.
                yield self.env.timeout(time/speedfactor_TK)
                self.workingtime += time
                self.next_easy = False
        else:
            time = random.triangular(17,18,20)/3600.
            yield self.env.timeout(time/speedfactor_TK)
            self.workingtime += time

        if p == g.lower_output_slot:
            try:
                self.waitfor126.succeed()
            except RuntimeError:
                pass

        if debug_TK:
            print "%s - TKpod retrieved from:
%d"%(timify(self.env.now),self.position)

    def choose_trolley_strategically(self,job):
        q.heappush(self.outputJobs, job)
        qcopy = deepcopy(self.outputJobs)
        for i in range(len(self.outputJobs)+2):
            try:
                job = q.heappop(qcopy)
```

```
            if self.busytrolley[job[4]] == True:
                continue
            else:
                popped = []
                for j in range(i+1):
                    if j == i:
                        job = q.heappop(self.outputJobs)
                        for k in popped:
                            q.heappush(self.outputJobs, k)
                    else:
                        popped.append(q.heappop(self.outputJobs))
                break
        except IndexError:
            job = q.heappop(self.outputJobs)
            break
    return job
def trigger_trolleys(self,m):
    if m == 0:
        try:
            TE.wait1.succeed()
        except RuntimeError:
            pass
        try:
            TE.wait2.succeed()
        except RuntimeError:
            pass
        try:
            TE.wait3.succeed()
        except RuntimeError:
            pass
    elif m == 1:
        try:
            TF.wait1.succeed()
        except RuntimeError:
            pass
        try:
            TF.wait2.succeed()
        except RuntimeError:
            pass
        try:
            TF.wait3.succeed()
        except RuntimeError:
            pass
    elif m == 2:
        try:
            TP.wait1.succeed()
        except RuntimeError:
            pass
        try:
            TP.wait2.succeed()
        except RuntimeError:
            pass
        try:
            TP.wait3.succeed()
        except RuntimeError:
```

```
            pass
def trigger_trolleys_output(self):
    a = False
    try:
        TE.waitforoutput.succeed()
        a = True
    except RuntimeError:
        pass
    try:
        TF.waitforoutput.succeed()
        a = True
    except RuntimeError:
        pass
    try:
        TP.waitforoutput.succeed()
        a = True
    except RuntimeError:
        pass
    return a

def release_lift(self):
    yield self.env.timeout(10./3600.)
    try:
        TK.waitforB.succeed()
    except RuntimeError:
        pass
def trigger_tb(self):
    try:
        PriorityTrolleys[1].waitforinput.succeed()
    except RuntimeError:
        pass

def travel_to(self,end):
    time = self.travelTime[self.position][end]
    self.position = end
    self.workingtime += time
    yield self.env.timeout(time/speedfactor_TK)

def send_order_tags(self, dim):
    PAC.orderTags[PAC.sTK].append(dim)
    create_PA_cards(PAC.sTK,PAC.cBNS,dim)
```

Appendix C: Simulation Model User Guide

This section is intended to provide instructions for installation, and provide an overview of the functionality of the simulation model.

Setup

The file "prod-v1.0.py" is the version of the simulation model that has been delivered to the client. There are two necessary accompanying files: "Interface.py" and "config.txt". All three of these files should be in a .zip folder in the electronic files accompanying this thesis. If "config.txt" is missing, create it as a blank text file.

The model has been designed for use with Python 2.7 on a Windows 7 machine; however it should still be functional using Macintosh and Linux operating systems. Several open source packages are required to run the model. These are:

- SimPy 3.0.5
- Numpy
- Scipy
- Matplotlib
- PrettyTable
- PyQt4
- Several packages which come default with Python 2.7.6
- All dependencies of the above packages

Once all of the above packages are installed and working correctly, the simulation model, user interface, and animation will operate as designed. To install Python, download the installer at this URL: https://www.python.org/download/releases/2.7.8/ and follow the installation instructions for your machine. Configure your machine so that 'python' is a path variable (see https://docs.python.org/2/using/windows.html). Then, install pip from here: https://pip.pypa.io/en/latest/installing.html. Pip will make it much easier to install the packages that are needed to run the simulation model. For example, to install the

latest version of SimPy, all that is needed is to open the command line and type 'pip install simpy'. Most packages will install this way, but if there is an error, please refer to the documentation for the specific package for detailed instructions.

Operation

Settings in the simulation model can be modified either through a user interface or directly in the Python code. It is recommended that the user interface is used to modify settings unless the user is a strong programmer. Modifying the source code may affect simulation results. To turn on the user interface, change the value of the global variable "useGUI" on line 53 to True.

Settings are distributed between six tabs in the user interface. These screens are shown in the proceeding diagrams, although they have been redacted to preserve confidentiality. Hovering the mouse over each setting in the user interface will show a tooltip with a description of the setting and occasionally some recommendations for ranges.

On the first tab shown in Fig. C1, it is important to select which simulation model is desired. Select either the mode for analyzing maximum throughput, or the mode which includes Machines A and B, and uses the PAC Coordination System. Also in this tab:

- Number of replications: A higher number of replications will make results more statistically significant.
- Warm-up period: It is recommended to use a warm-up period of at least 4 days.
- Time per replication: Days per replication. This must be larger than the warm-up period to get statistical results. Anywhere from 2 weeks to 6 weeks is reasonable.
- Direct cell on/off: Should reduce the utilization of the TK
- Repair circuit on/off: Repair operator services defective MProds immediately when the repair circuit is off.

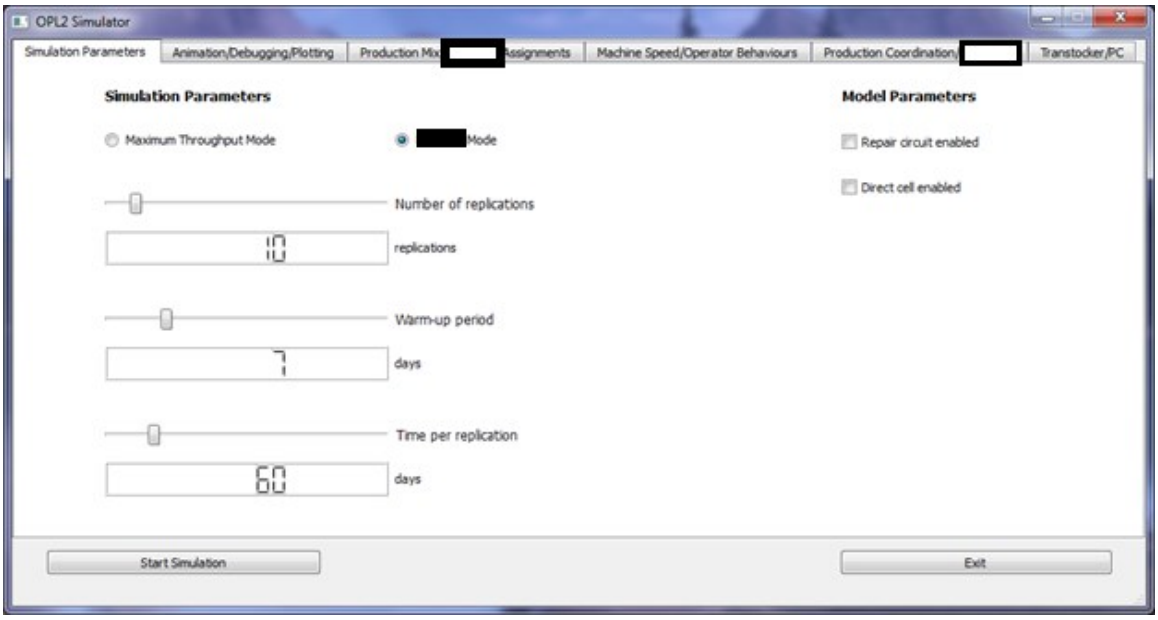

Figure C 1: First tab from the simulation user interface

Another important setting is on the second tab, shown in Fig. C2. Here, the user should select whether or not the animation is enabled. Also in this tab:

- Animation speed: Speed up or slow down the animation
- Animation update interval: Change how often the animation updates
- Plotting interval: Change how frequently data is recorded for plots
- General debugging on/off: Toggles text debug for trolleys, curing machines
- Repair circuit debugging on/off: Toggles text debug for repair operations
- Extended statistics printing on/off: Toggles text debug for statistics tracking
- TK debugging on/off: Toggles text debug for the TK
- Production control and machine groups A and B debugging on/off: Toggles text debug for machines A, B, and PAC System

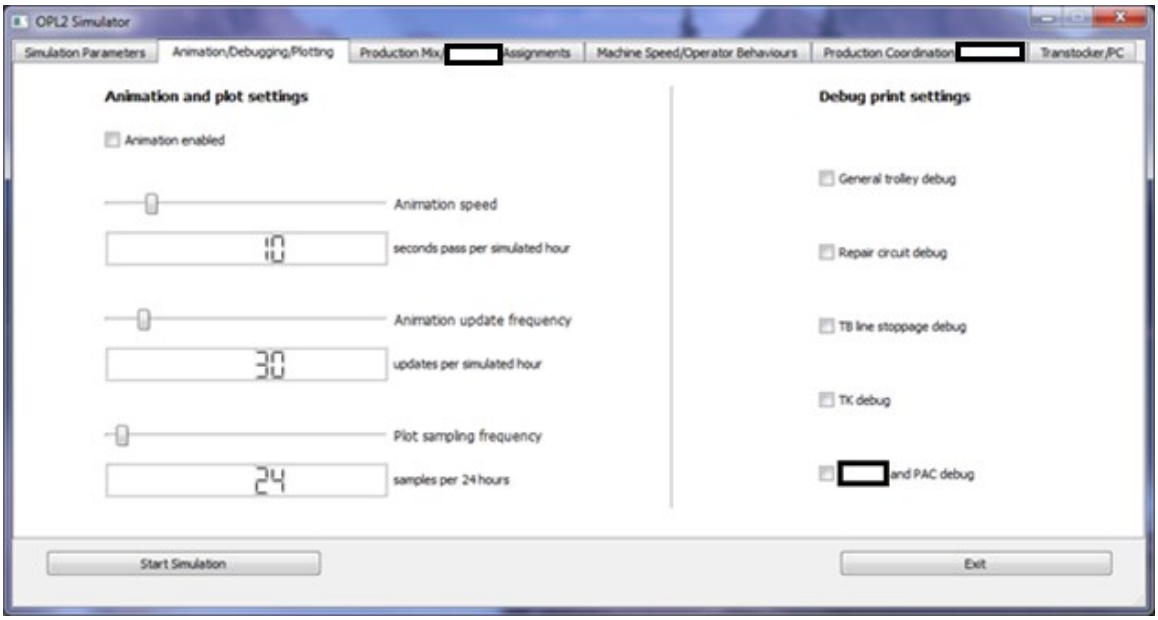

Figure C 2: Second tab from the simulation user interface

On the third tab, it is necessary to select a production mix. By default, there will not be a production mix selected. If there are no production mixes in the "config.txt" file, the user must fully specify and save a new production mix. Once a production mix has been specified, and all other simulation settings are set as desired, click the "Start Simulation" button in the bottom left. This will begin the simulation. Figure C3 shows a correctly specified production mix. Instructions are included in the scroll box on the left side of the tab.

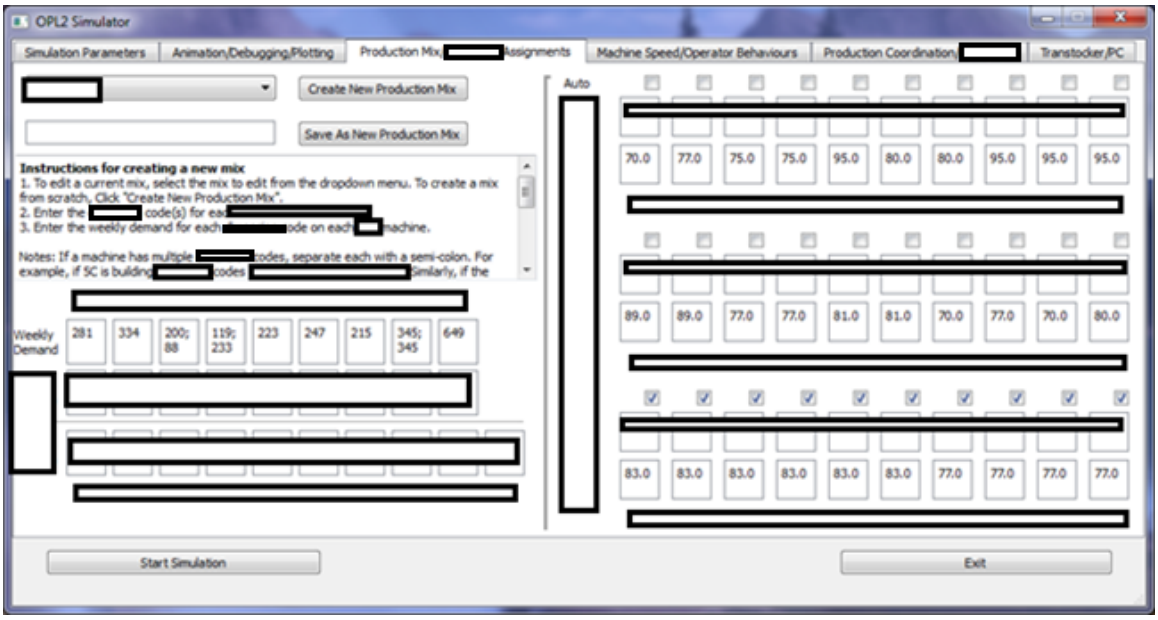

Figure C 3: Third tab from the simulation user interface

The settings in the fourth tab are:

- Machine efficiency for each machine: Increasing efficiency will speed up machines. Decreasing will slow them down.
- Operator efficiency for each operator type: Increasing efficiency will speed up operators. Decreasing will slow them down.
- Operator breaks covered: If operator breaks are not covered, curing press operators will go for lunch breaks, leaving presses unattended.
- Operator behaviour: Alter the behaviour of curing press operators

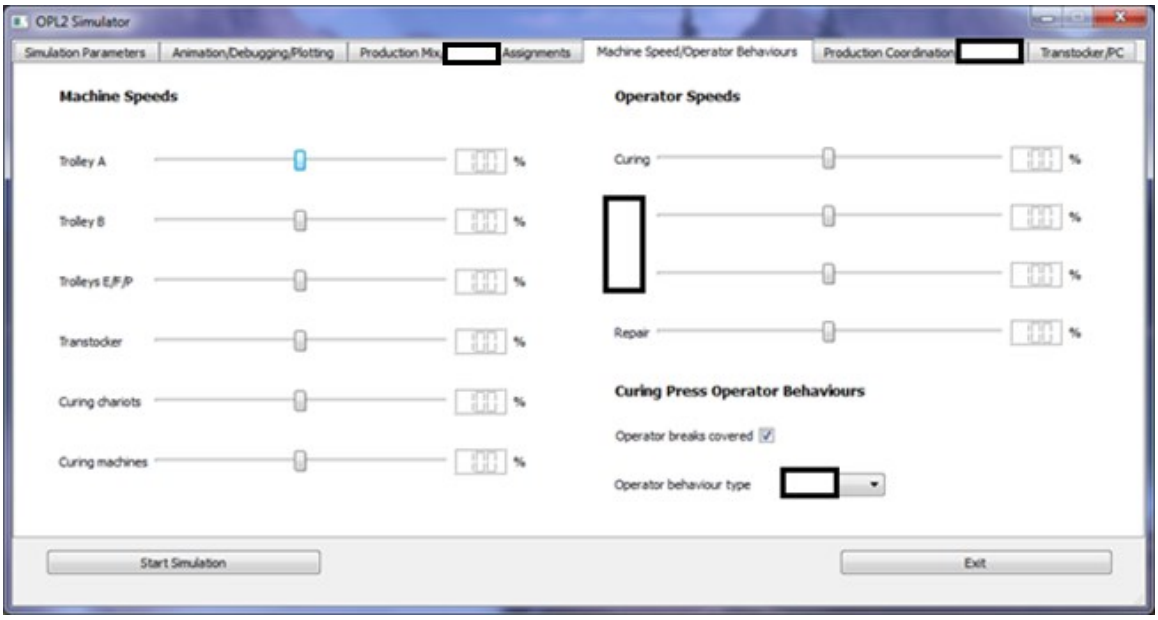

Figure C 4: Fourth tab from the simulation user interface

The settings in the fifth tab are:

- TK capacity: Lowering this will keep the TK less full.
- Number of empty slots in the TK: A number from 1 to 10 is reasonable here. This effectively reduces the capacity of the TK
- Initial stock in the TK: Lowering this will keep the TK less full. This setting is locked because it serves the same purpose as the TK capacity setting.
- Batch size: Issue PA cards in batches of this size.
- Machine A/B Setup: Set process times and changeover times for each machine
- Number of operators: Set number of operators for machines A and B
- Changeover policies: Select a different changeover policy for machines in machine bank A

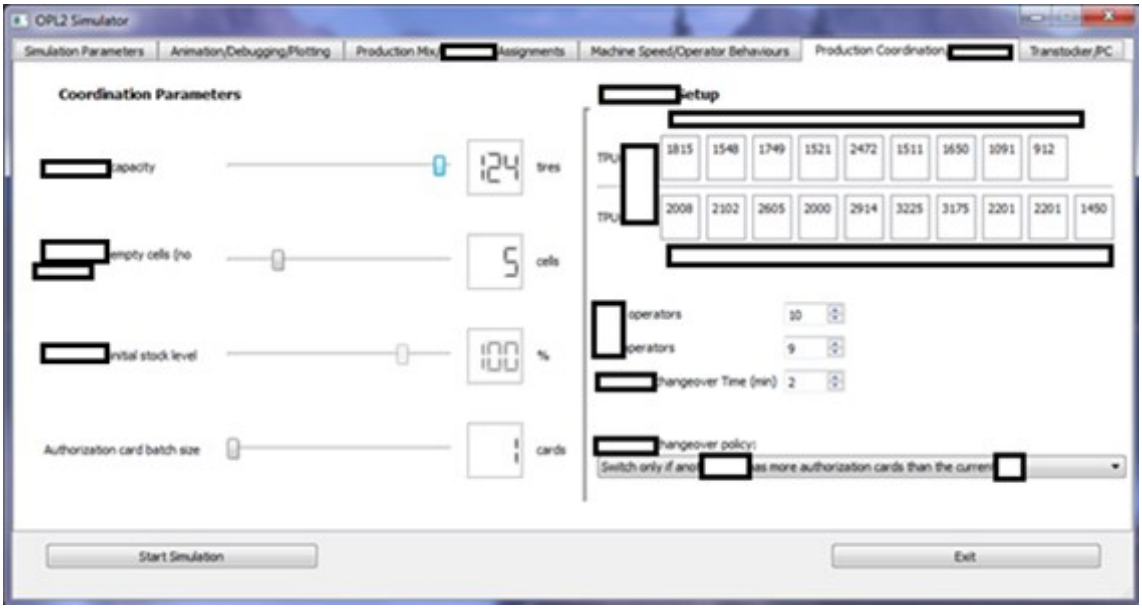

Figure C 5: Fifth tab from the simulation user interface

The settings in the sixth tab are:

- TK logic: Change the priorities of the TK
- Number of pre-cure cells per line: Six is the default value.

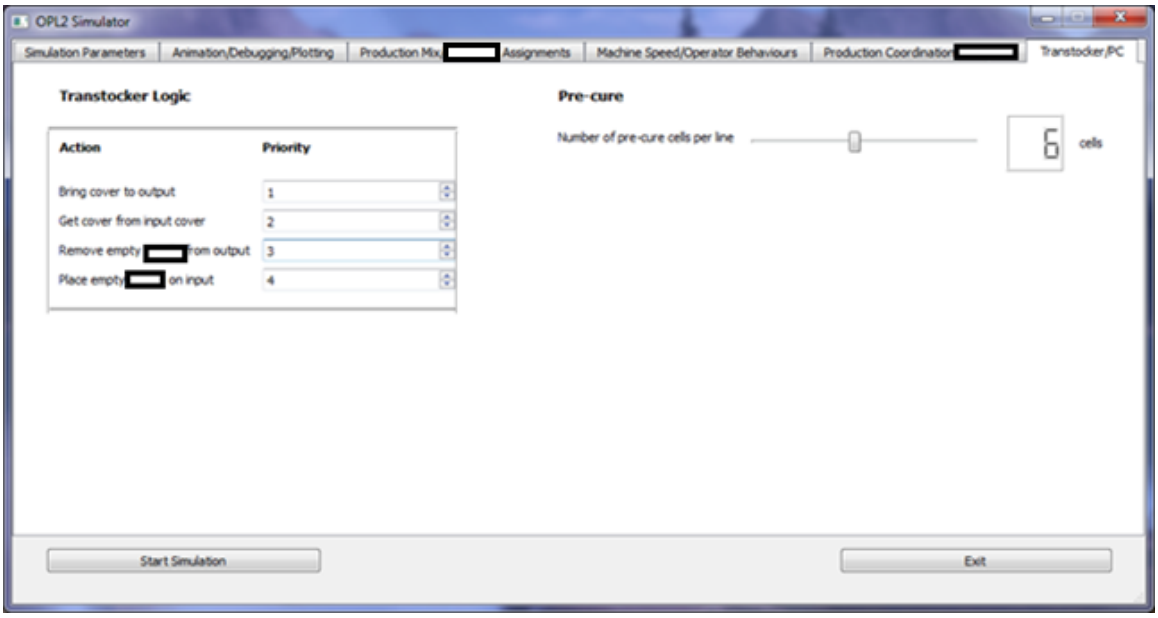

Figure C 6: Sixth tab from the simulation user interface

When the simulation is complete, statistics will be outputted in the Python Interpreter window or in the command line. It is recommended to use the Interpreter window, because tables do not format well in the command line. Only statistics from the most recent replication are displayed for individual machines, although the daily throughput numbers for each replication are displayed above the first table as a list in square brackets. The tables at the bottom of the output display some run settings, as well as confidence intervals on several key parameters across all replications.

Appendix D: Select Trace Plots of Training Runs

In this section, select trace plots are shown for neural network training runs.

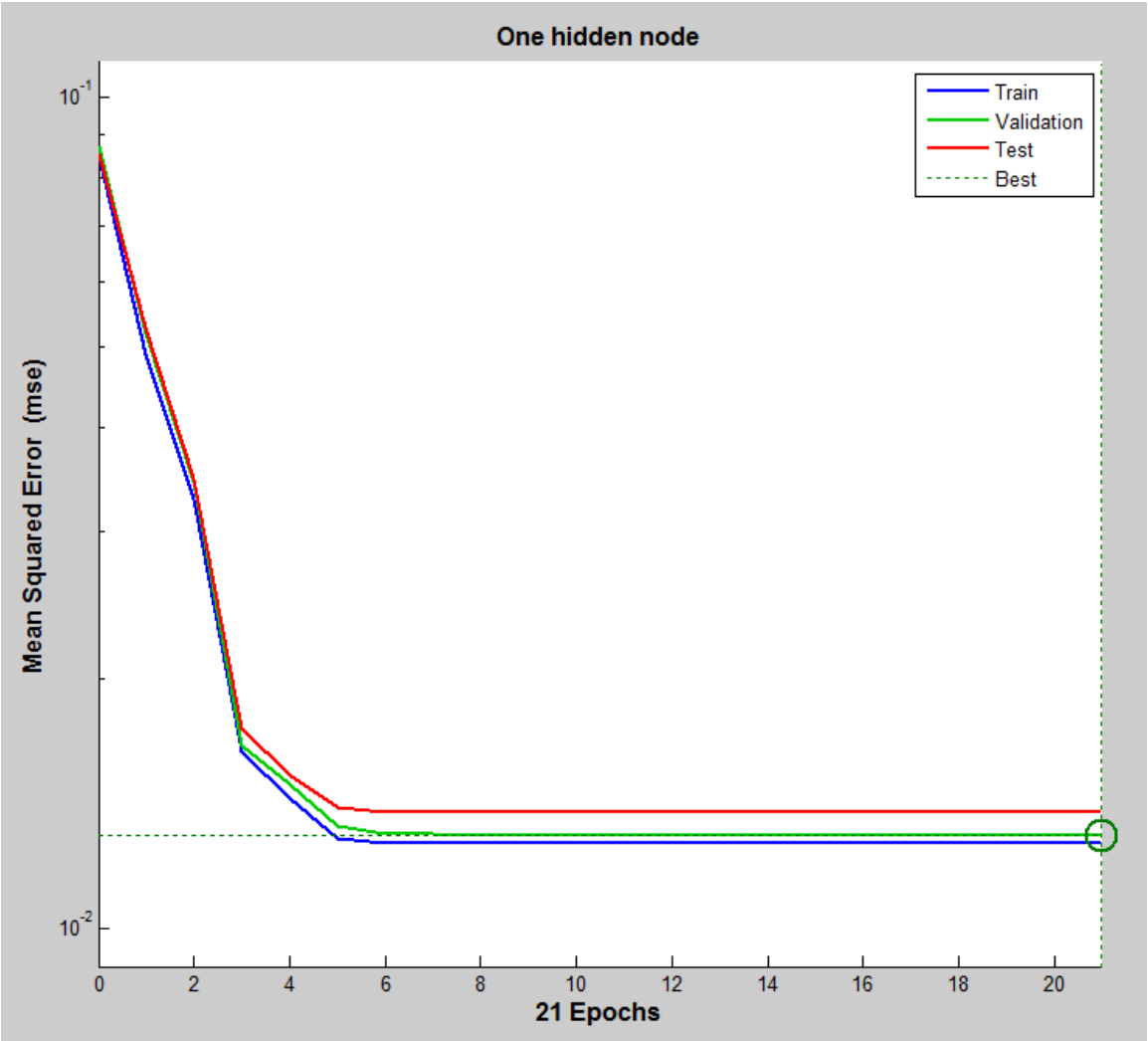

Figure D 1: Training trace plot for one hidden node

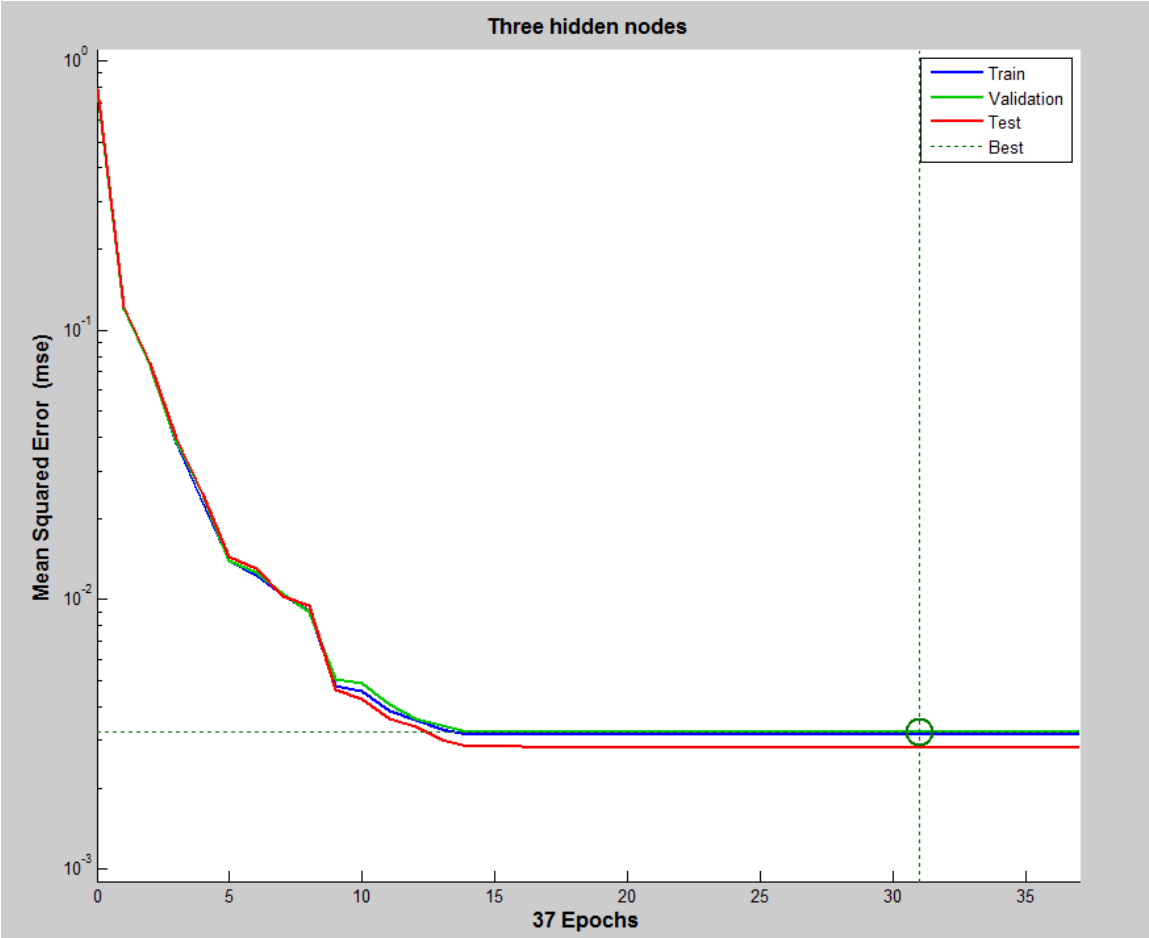

Figure D 2: Training trace plot for three hidden nodes

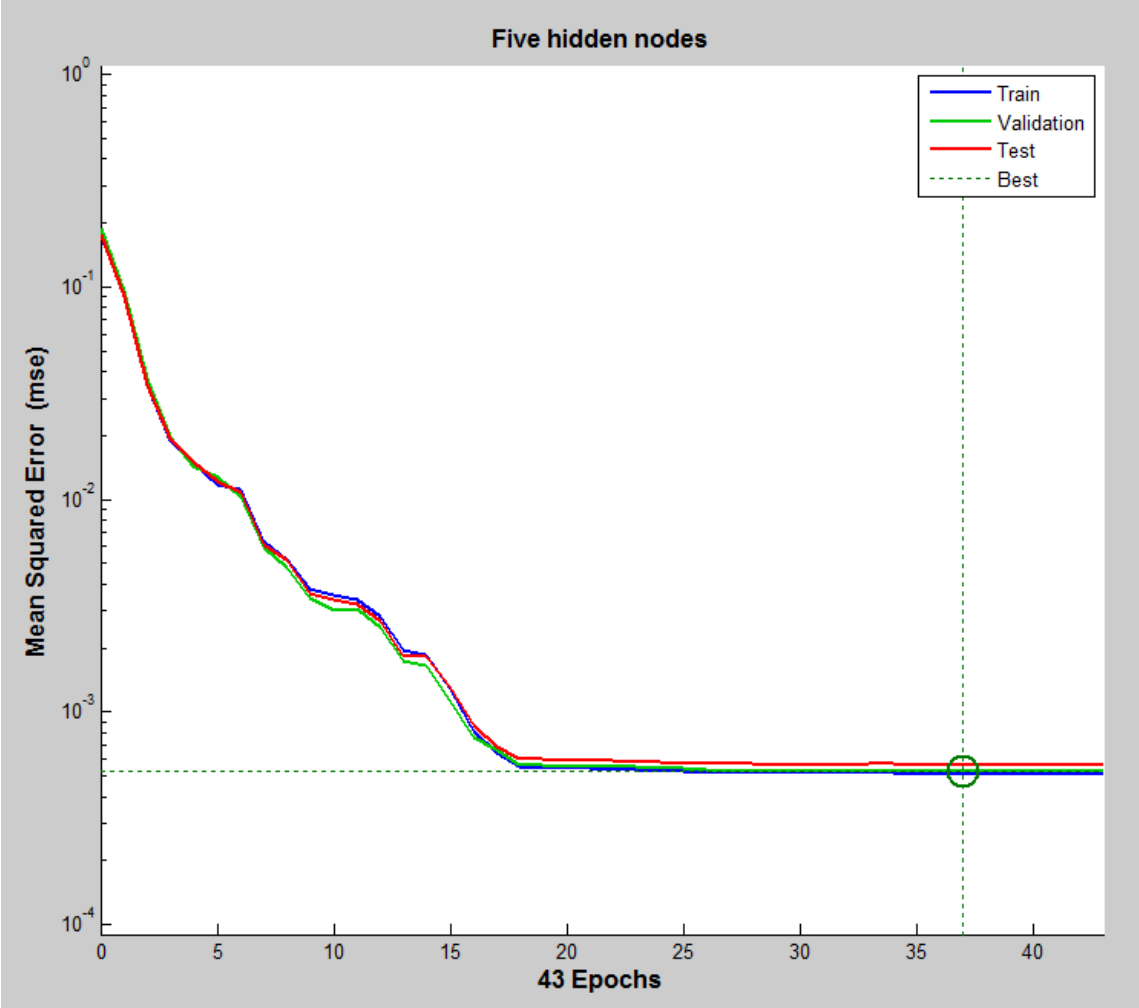

Figure D 3: Training trace plot for five hidden nodes

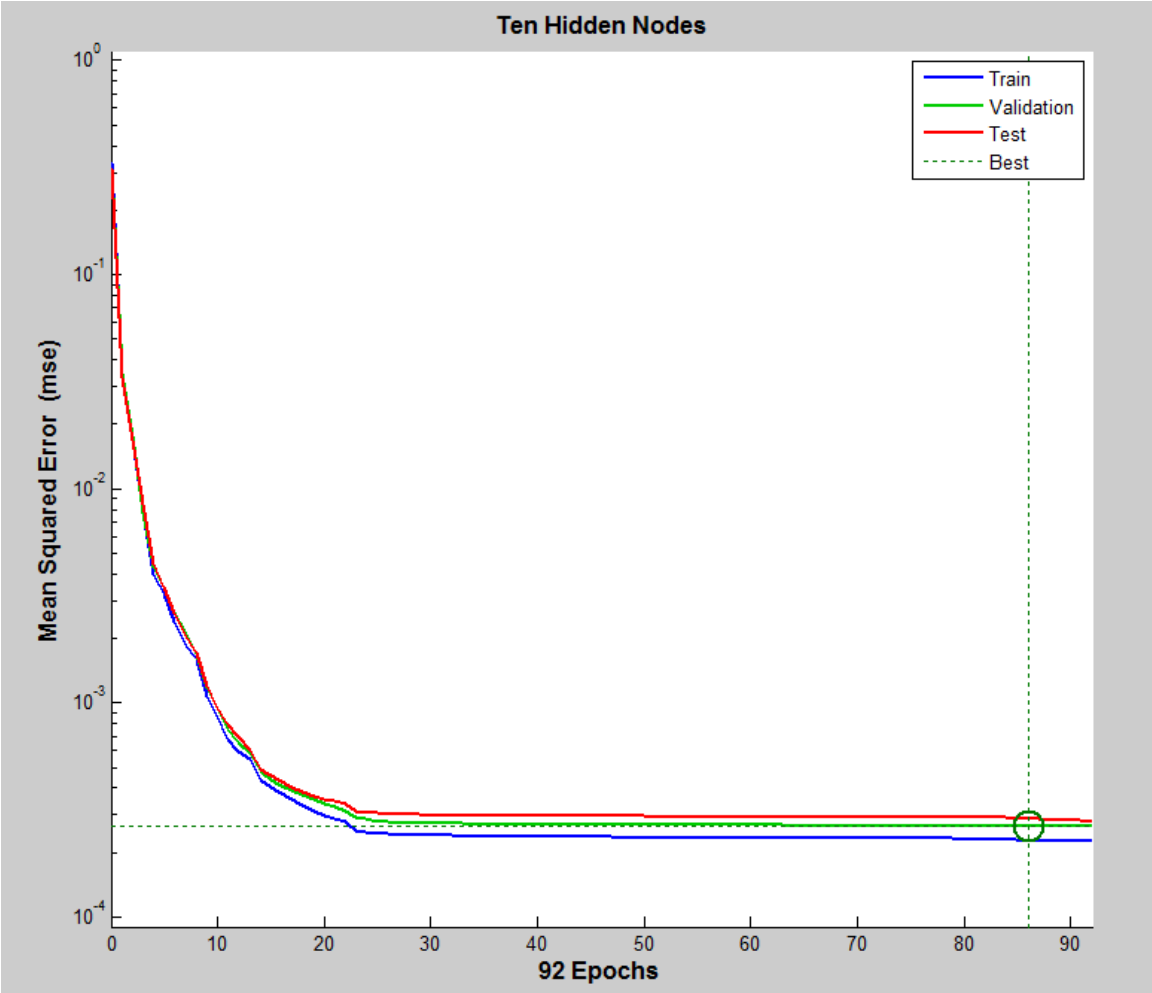

Figure D 4: Training trace plot for ten hidden nodes

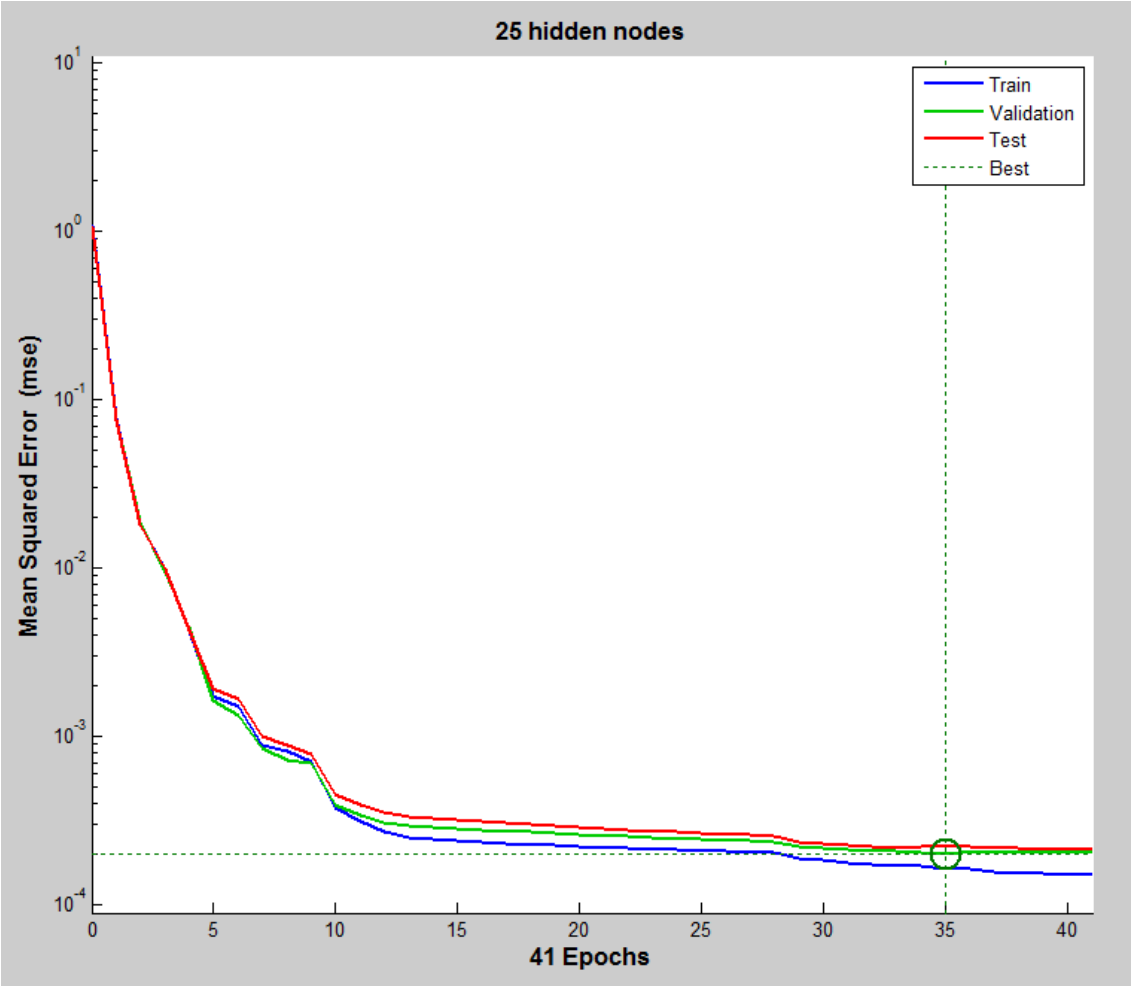

Figure D 5: Training trace plot for 25 hidden nodes

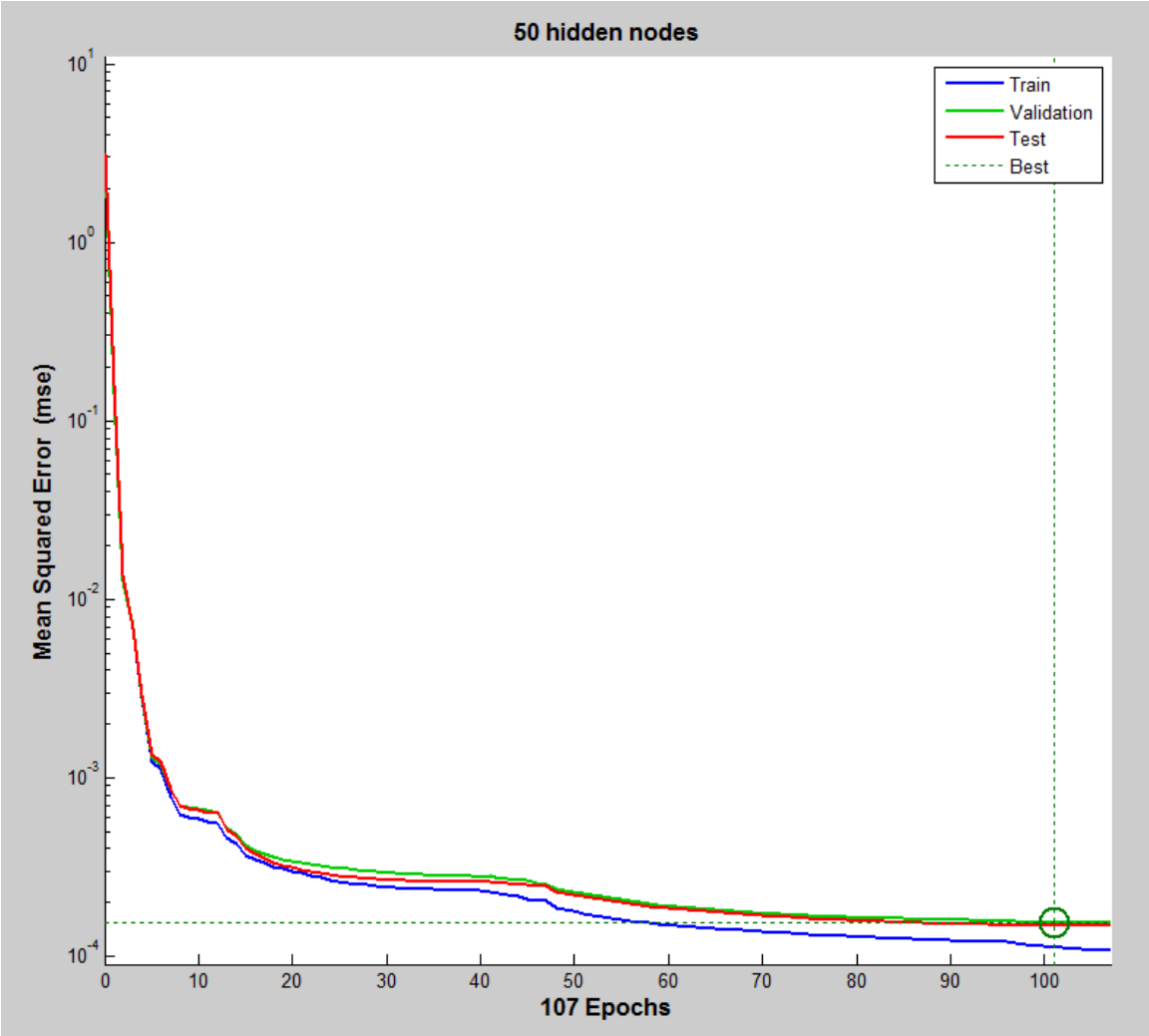

Figure D 6: Training trace plot for 50 hidden nodes

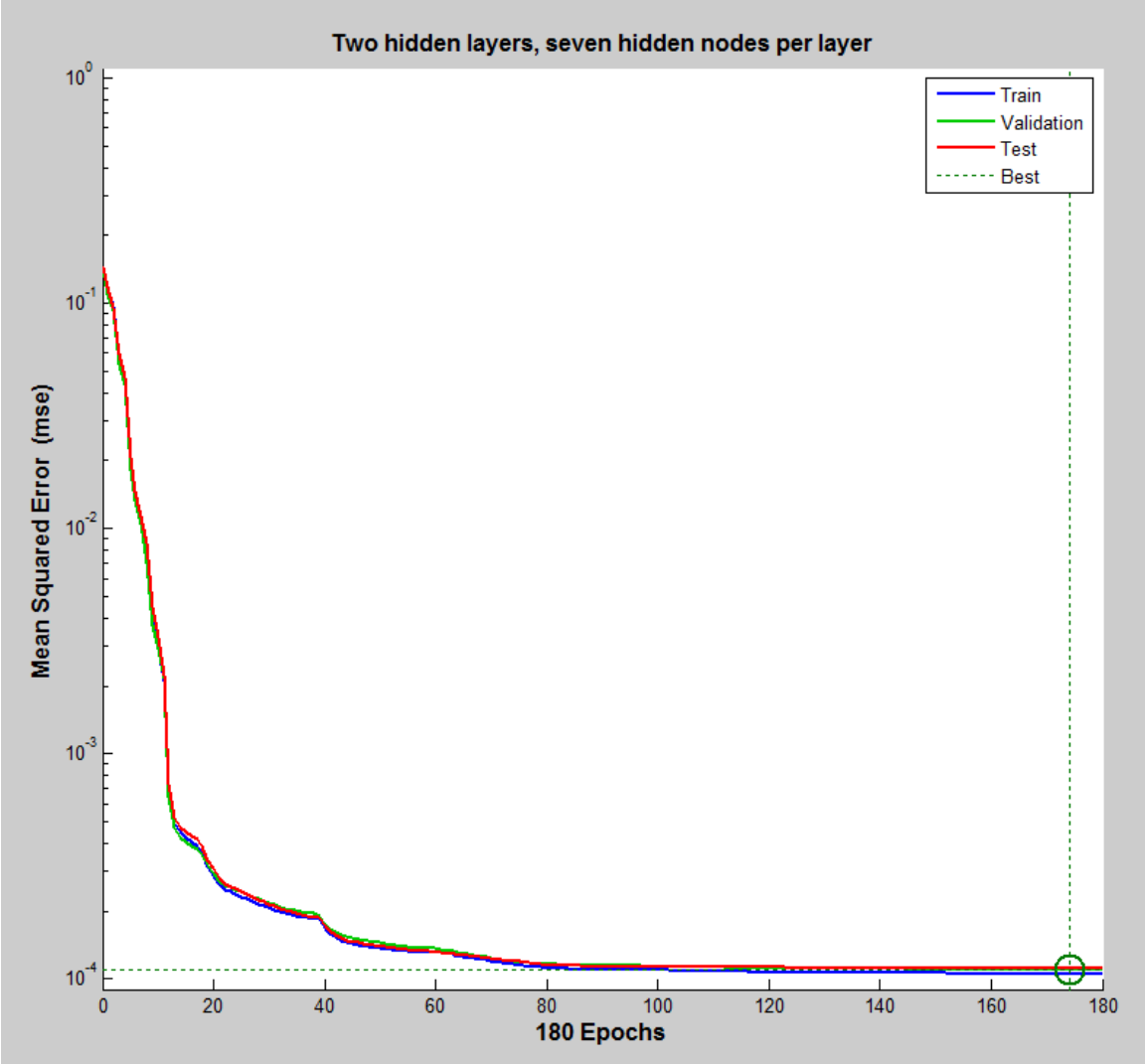

Figure D 7: Trace plot for training with two hidden layers, seven hidden nodes per layer

Appendix E: Trace Plots for Determining Appropriate Warm-Up Period

In this section, three trace plots of the stock level in the TK are shown under different scenarios to show how the production rate in MProd-building and production rate in curing affect TK stock levels in the transient period of the simulation. In Fig. E1, MProd-building can produce MProds at a higher rate than curing, so there is a fairly steady stock level in the TK almost immediately. In Fig. E2, MProd-building can produce MProds at a very similar rate as curing. In this scenario, the stock level does not become steady for several days. In Fig. E3, curing can out-produce MProd-building, so the TK achieves a steady stock level quickly.

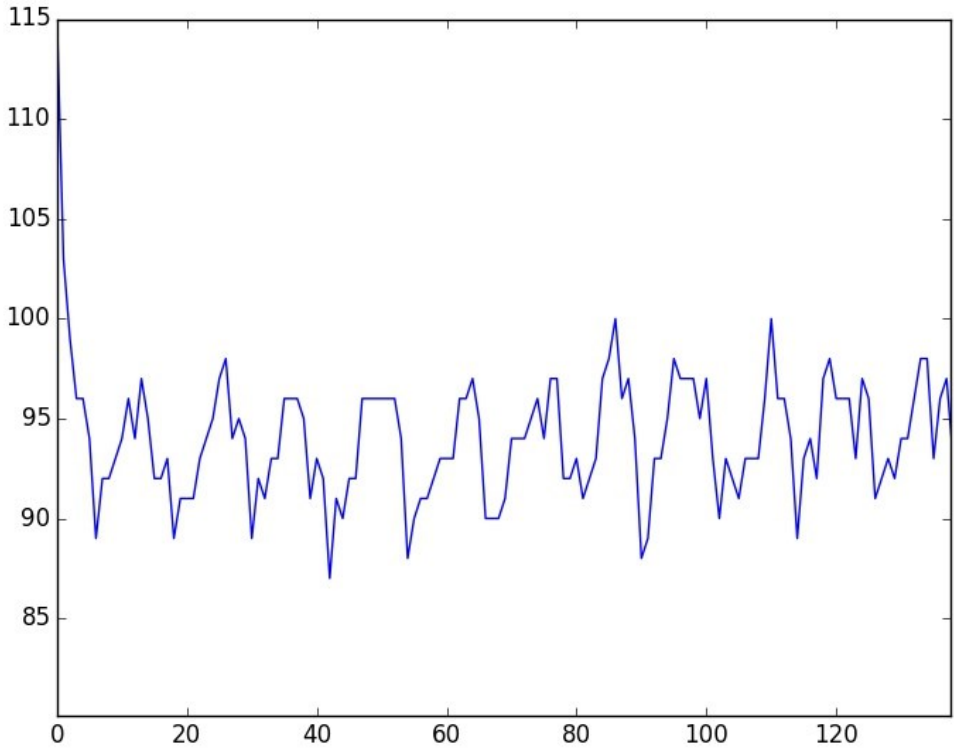

Figure E 1: TK stock level is shown as a function of simulation hours. MProd-building easily keeps pace with curing demand, resulting in a short warm-up period.

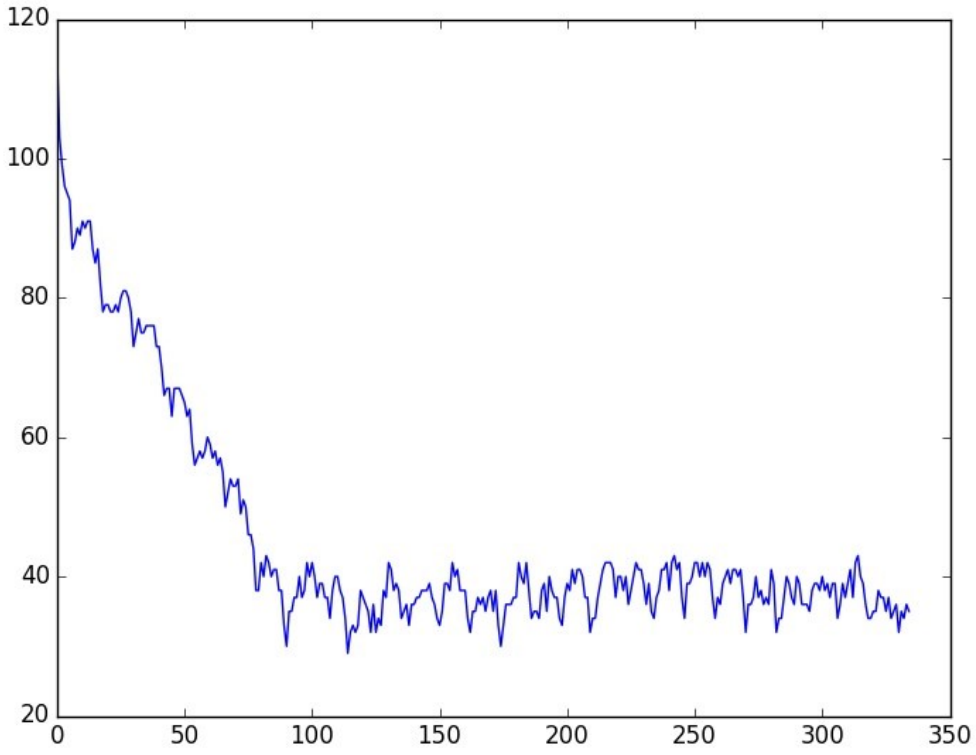

Figure E 2: TK stock level is shown as a function of simulation hours. MProd-building barely keeps pace with curing, resulting in a longer warm-up period.

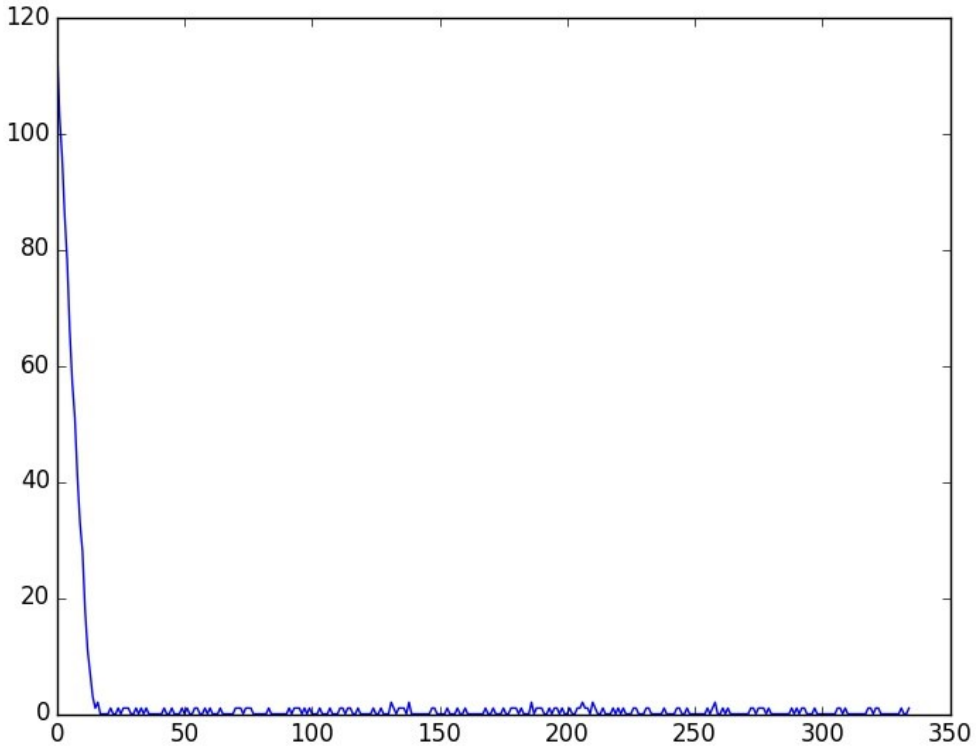

Figure E 3: TK stock level is shown as a function of simulation hours. MProd-building cannot keep pace with curing demand, resulting in a short warm-up period.

Appendix F: Simulated Annealing Code and Full Results

The results from the simulated annealing runs are shown in full in the table below. The code which was used to generate solutions is included after the results.

Table F 1: Full simulated annealing results

Stock Parameter [1-9]	Changeover Policy [1-5]	Operator behaviour [1-3]	Shared operators [0-1]	Machine A operators [4-10]	Machine B operators [4-9]	Curing operator breaks covered [0-1]	Output
1	4	1	1	8	7	1	-0.98568
1	4	1	1	10	5	1	-0.9859
1	4	1	1	10	5	1	-0.9859
1	4	1	1	10	5	1	-0.9859
1	4	1	1	8	7	1	-0.98568
1	4	1	1	9	6	1	-0.98575
1	4	1	1	7	8	1	-0.98496
1	4	1	1	10	5	1	-0.9859
1	4	1	1	10	5	1	-0.9859
1	4	1	1	10	5	1	-0.9859
1	4	1	1	10	5	1	-0.9859
1	4	1	1	9	6	1	-0.98575
1	4	1	1	9	6	1	-0.98575
1	4	1	1	7	8	1	-0.98496
1	4	1	1	8	7	1	-0.98568
1	4	1	1	10	5	1	-0.9859
1	4	1	1	8	7	0	-0.98554
1	4	1	1	9	6	1	-0.98575
1	4	1	1	10	5	1	-0.9859
1	4	1	1	8	7	1	-0.98568

```
function y = simanneal()
    % Load in trained network
    load('netonly.mat', '-mat');
    net = net;

    % Set up other preliminaries
    stockDict =[8   10  11  12  13  14  15  16  17
            6   7   8   9   9   10  11  12  12
            6   7   8   9   9   10  11  12  12
            5   5   6   7   7   8   9   9   10
            5   6   7   7   8   9   9   10  10
            5   6   6   7   7   8   9   9   10
            4   4   5   5   5   6   6   7   7
            5   6   6   7   8   8   9   10  10
            5   5   6   6   7   8   8   9   9
            3   4   4   4   5   5   5   6   6
            6   7   8   8   9   10  11  11  12
            6   6   7   8   8   9   10  10  11];


    % Initialize SA
    vars = [randi([1 9]) randi([1 5]) randi([1 3]) randi([0 1])...
        randi([4 10]) randi([4 9]) randi([0 1])];
    varmax = [9 5 3 1 10 9 1];
    varmin = [1 1 1 0 4 4 0];
    w = vars;
    evaluate(w);
    output_w = output;
    bestw = w;
    bestoutput = output_w;
    k = 0;
    T = 0.50;
    for tk = 1:200
        for v = 1:7
            for m = 1:5
                if output_w < bestoutput
                    bestw = w;
                    bestoutput = output_w;
                end
                wp = w;
                wp(v) = randi([varmin(v) varmax(v)]);
                evaluate(wp);
                output_wp = output;
                if output_wp < output_w
                    output_w = output_wp;
                    w = wp;
                elseif rand < exp((output_w-output_wp)/T)
                    output_w = output_wp;
                    w = wp;
                end
            end
            T = 0.93*T;
        end
    end

    y = horzcat(bestw, bestoutput);
```

```
    function evaluate(soln)
        % Evaluate ANN
        mix = stockDict(:,soln(1));
        mix = transpose(mix);
        mix = (mix-3)/32.;
        changeover = [0 0 0 0 0];
        changeover(soln(2)) = 1;
        behaviour = [0 0 0];
        behaviour(soln(3)) = 1;
        sharedops = [soln(4)];
        ops1 = [soln(5)];
        ops1 = (ops1-4)/6.;
        ops2 = [soln(6)];
        ops2 = (ops2-4)/5.;
        opsbreaks = [soln(7)];


        input = transpose(horzcat(mix,[0],changeover,...
            sharedops, ops1, ops2,behaviour, opsbreaks));
        output = -net(input);
    end

end
```

Appendix G: Process Maps of Major Model Classes

In this section, the programmed flows of the major classes in the simulation model have been represented as process maps. While the process maps are fairly high level, the purpose of the maps is to show how these classes operate in the context of the model, not to exhaustively list all of the variable changes or interactions between classes. The classes that are excluded from this section are not relevant to the operation of the production line, and instead support other model functionalities, such as animation or statistical tracking.

The classes in this section are as follows:

- Machine_A
- Machine_B
- Trolley_A
- Trolley_B
- Repair_Operator
- TK
- Monorail_System
- TrolleyEFP
- Inspector
- Curing_Chariot
- Curing_Machine
- Curing_Press_Operator

Figure G 1 shows the programmed logic of the Machine_A class in the simulation model. When a PA card arrives for an instance of Machine_A for which the relevant product code is ready to be built, then if there is space on the tree and an operator is available, construction of the carcass begins immediately. If there is not space on the tree or if there is no available operator, the machine waits until both of these conditions are met. If there are no PA cards for the relevant product code and the machine is idle, then a product change may take place if conditions are met.

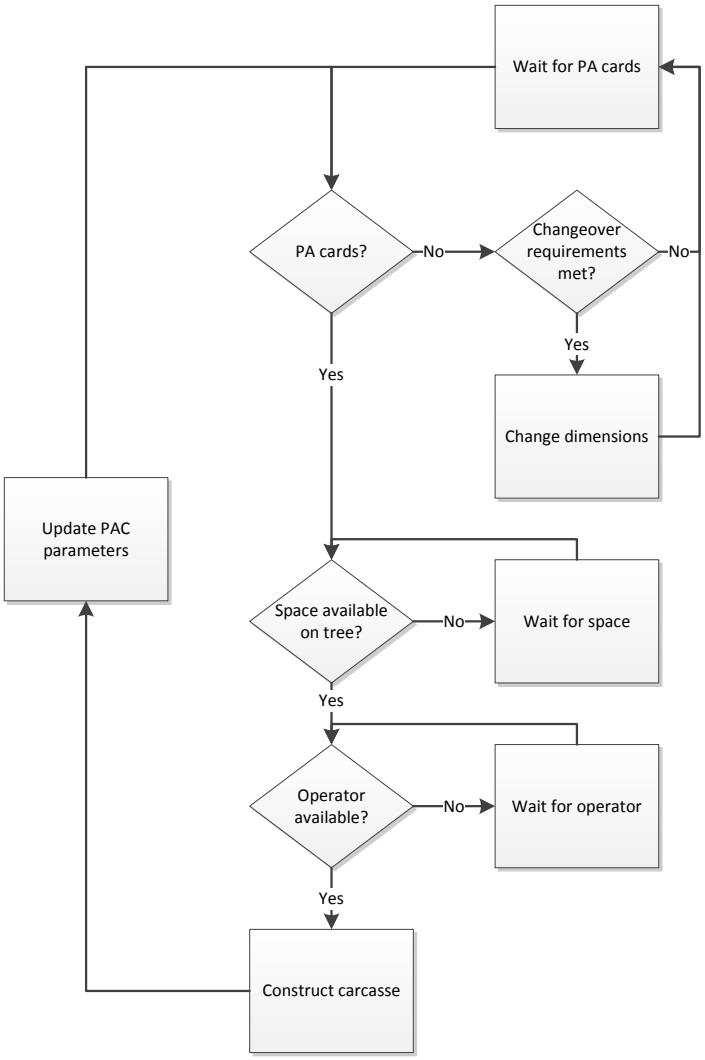

Figure G 1: Machine_A Process Map

The programmed logic of Machine_B in Fig. G2 is similar to the logic of the Machine_A class, except that a carcass must be available to begin the process. Instead of checking for space on a tree, instead there must be space on the tipper sling before construction begins.

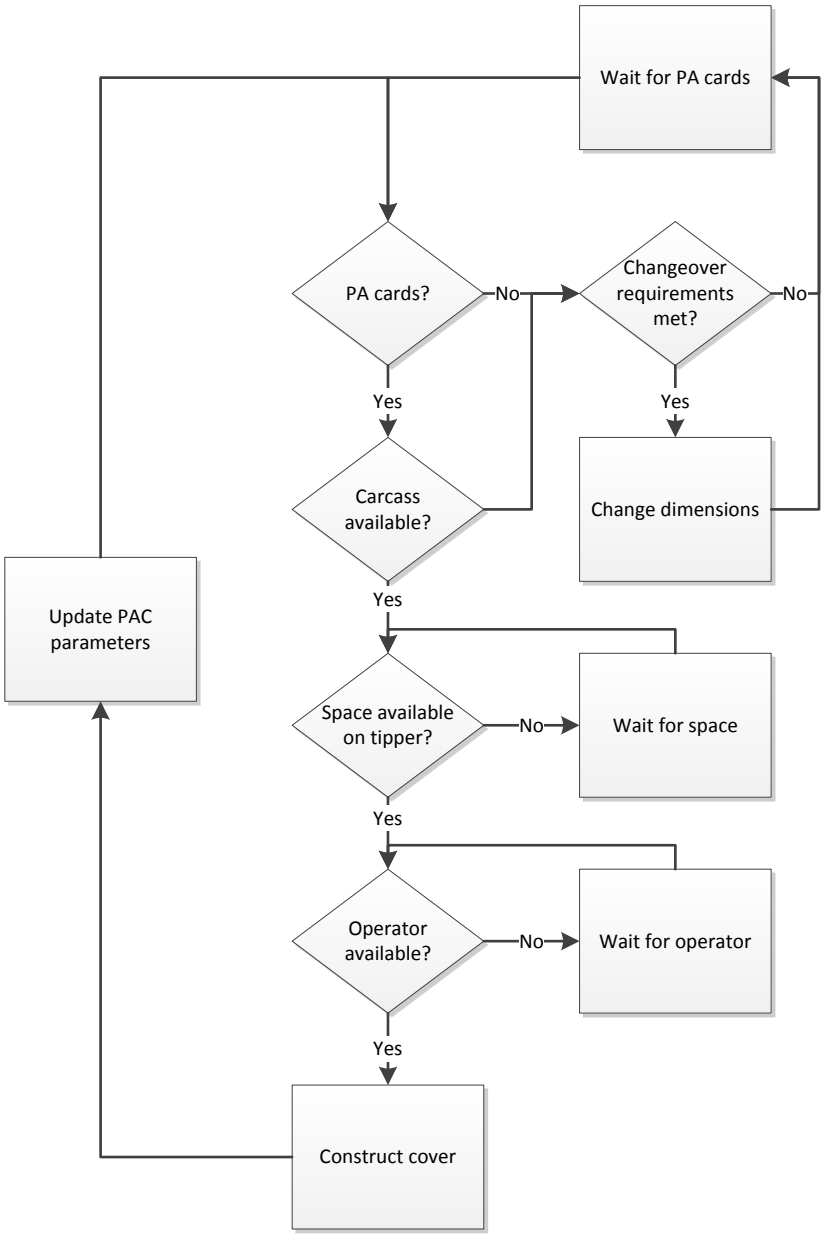

Figure G 2: Machine_B Process Map

The programmed logic of Trolley_A in Fig. G3 shows how trolley A takes jobs from its queue then travels to pick up the cover. It can only approach the scales when trolley B is not at the scales. Additionally, there must be space at the scales, otherwise trolley A will remain idle until space is cleared.

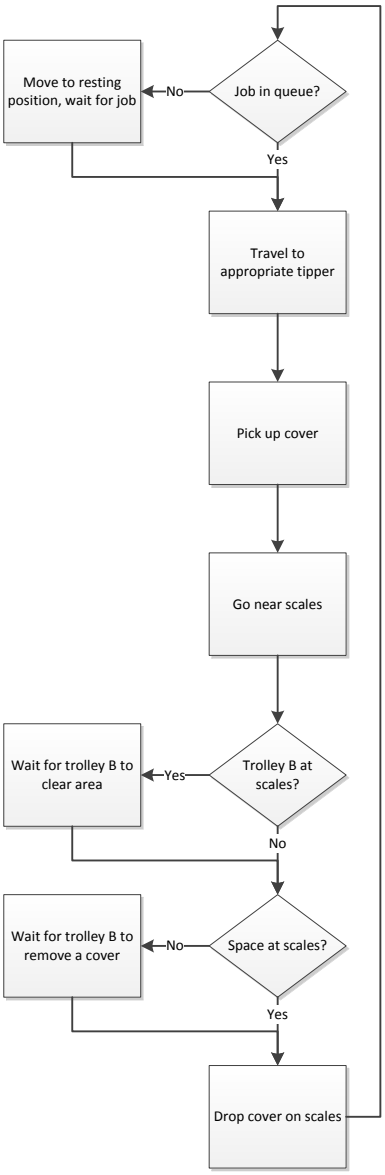

Figure G 3: Trolley_A Process Map

The programmed logic of Trolley B in Figure G4 is more complex than Trolley_A. When a job is selected from the queue, the type of job must be determined. If the job is from the scales, then trolley A must be clear of the area. If the job is from the repair area, then the cover must be picked up in that location. Once the MProd is picked up and there is space in the TK, then the trolley travels to the input, waits for an empty TK pod to be present, and drops off the MProd. However, if the MProd was defective, rather than proceeding to the TK, the trolley drops the MProd in the repair area. If there is no space in the repair area, the trolley waits for space to be cleared. Once the MProd is dropped in either the repair area or the TK, the next job is processed.

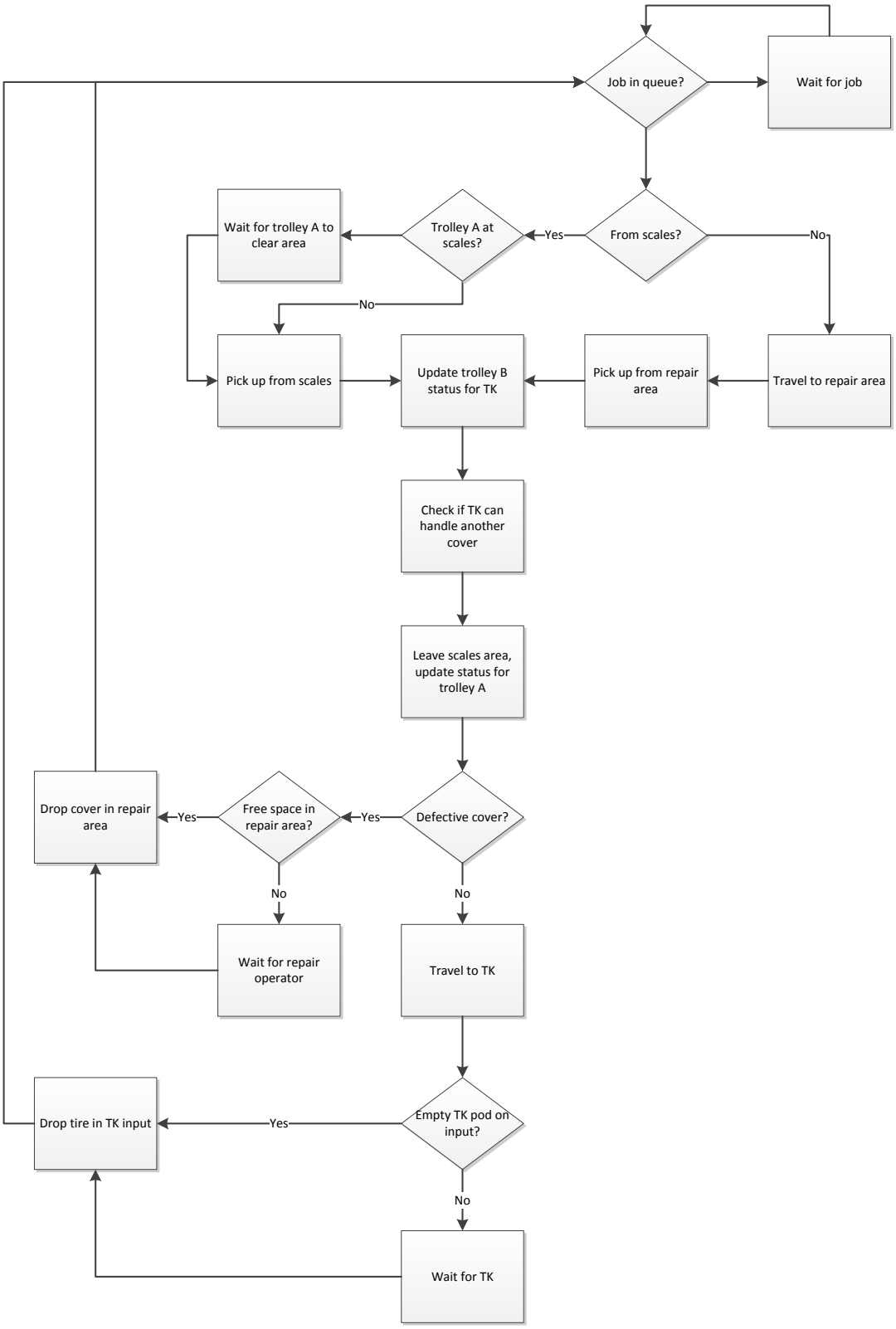

Figure G 4: Trolley_B Process Map

Occasionally the Repair_Operator class checks to see if enough MProds are defective to begin repairs. When this repair condition is met, the operator repairs covers at an interval determined by the necessity of forklifts, trolley B, and the complexity of the repair.

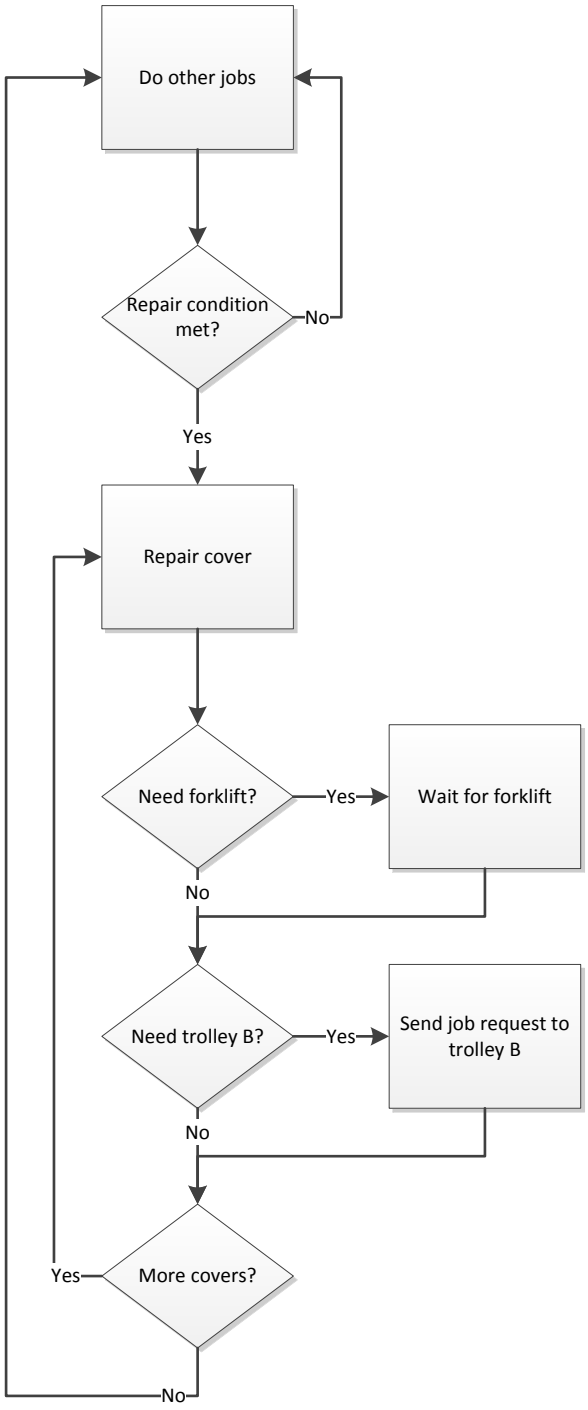

Figure G 5: Repair_Operator Process Map

The programmed logic of the TK class in Fig. G6 follows the main stream of decisions represented by the decision blocks in the centre of the figure. If the TK is at the input and an input cover has arrived, then the TK does the input job, storing the cover in the nearest slot to the input. If instead, there is an output job in the queue, the TK will do this job first. If the cover is already in the direct cell, then the TK returns to the beginning of its logic path. Otherwise, the TK travels to the oldest MProd that meets the product requirements, retrieves it, and delivers it to the output. After dropping off the TK pod and cover at the output, the parallel output process begins, where either trolley E, F, or P will pick up the MProd, at which time the empty TK pod is transported to the lower output slot by a separate mechanism. If there are no output jobs in the queue, and there is an input job in the queue, then the TK travels to the input, picks up the input cover once trolley B has dropped it off, and stores it in the empty slot nearest the input. The TK will only wait 20 seconds after arrival at the input for trolley B to arrive, after which it will return to the beginning of its logic path. If there are no input or output jobs in the queue, then the TK will remove an empty TK pod from the bottom of the output if it is present. The TK will store this TK pod in an empty slot, or preferably, on the output if it is empty and a cover is on its way. If there is no empty TK pod on the lower output slot, the TK will retrieve an empty TK pod from storage and place it on the output as part of the input job process.

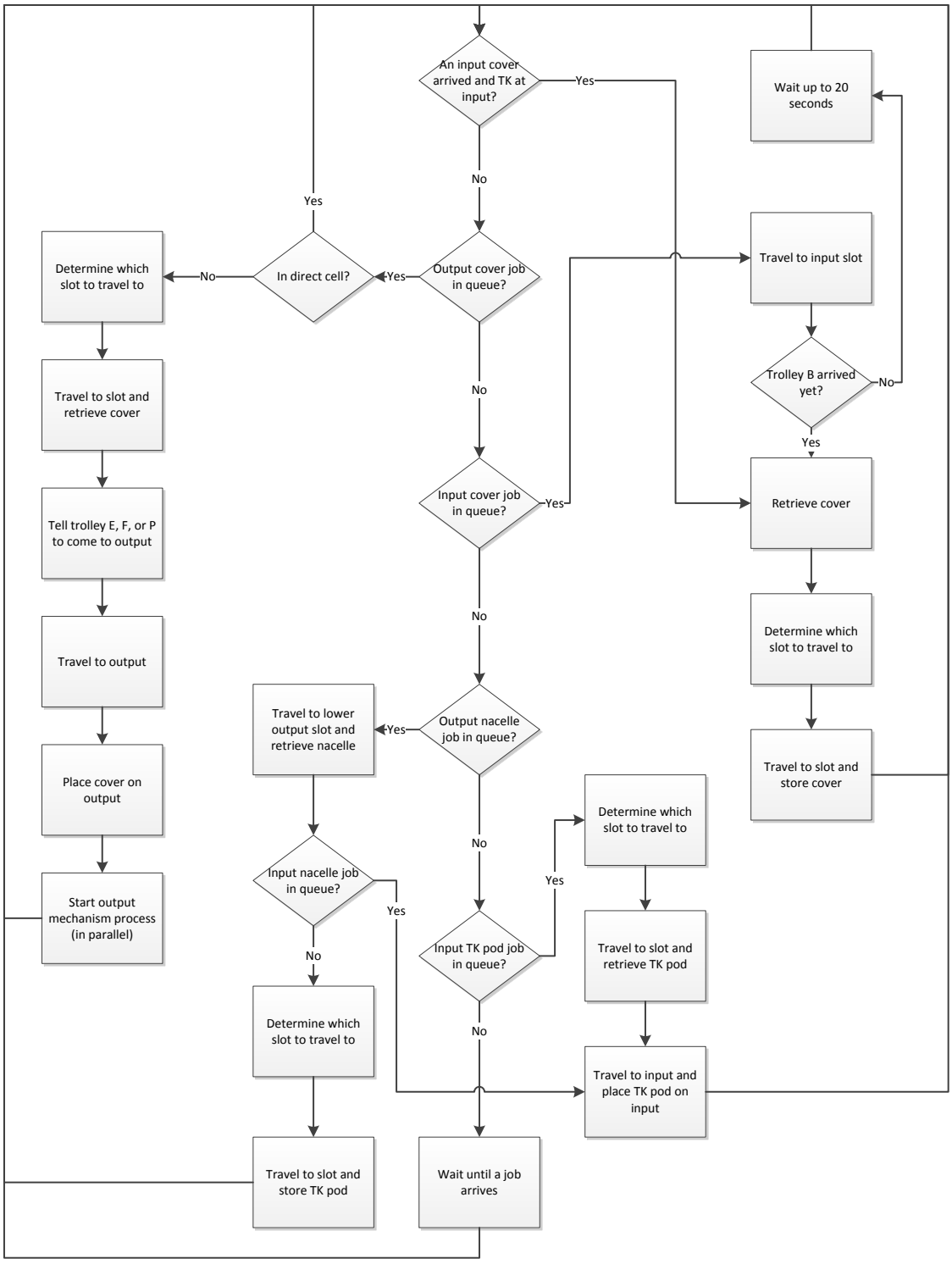

Figure G 6: TK Process Map

The Monorail_System class in Fig. G7 acts as a queue manager for the TK. When there is space in pre-cure and no other output jobs in the TK queue, the job is sent to the TK.

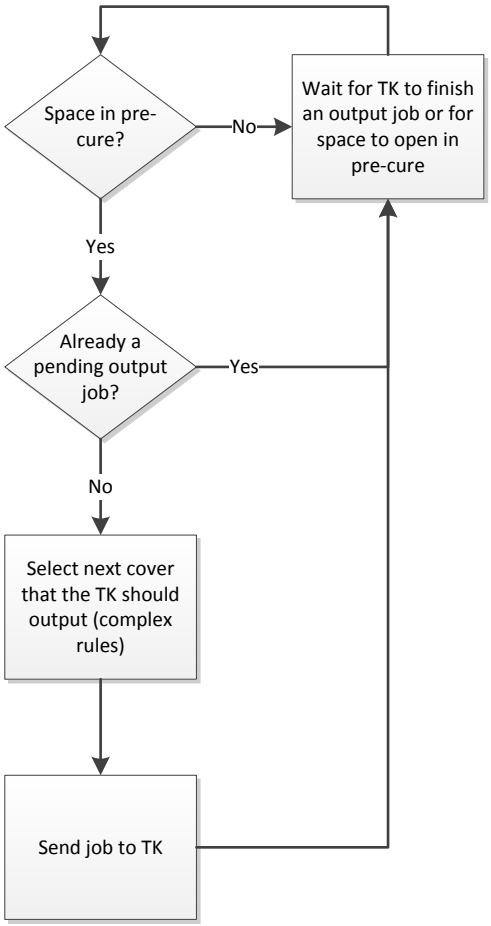

Figure G 7: Monorail_System Process Map

The programmed logic of the TrolleyEFP class in Fig. G8 applies to trolleys E, F, and P. When there is a job in the queue, the trolley must determine if it is a job from pre-cure to curing or a job from the TK output to pre-cure. If the destination is curing, then the trolley picks up the cover from pre-cure, transports it to the chariot drop point, where it will wait until the chariot is in position for drop off. If the destination is pre-cure, the trolley first verifies that no other trolley is in the output area before travelling to the output, picking up the cover, and dropping it off in pre-cure.

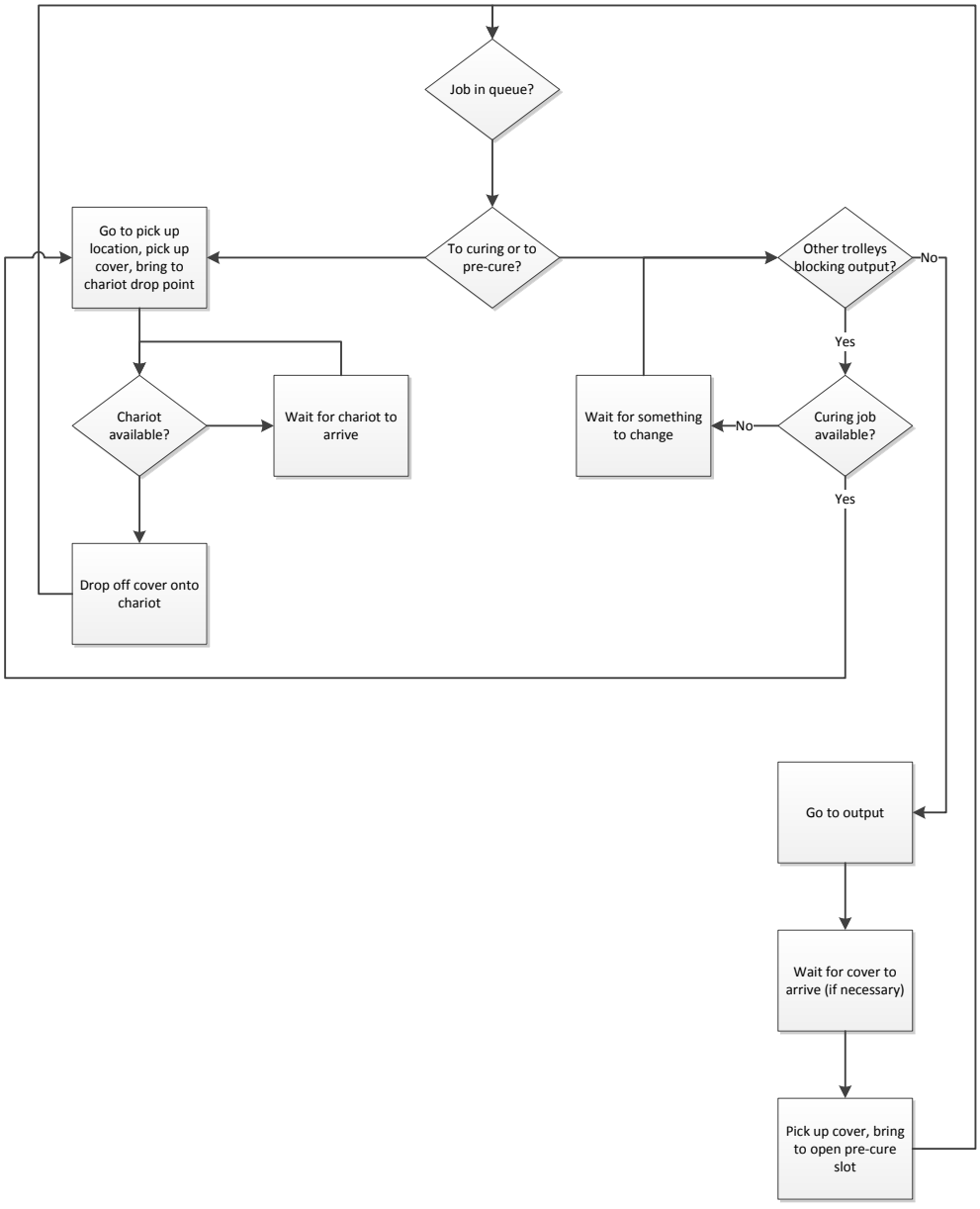

Figure G 8: TrolleyEFP Process Map

The Inspector class randomly selects covers to be rejected from pre-cure into the repair area. Inspections occur three times daily.

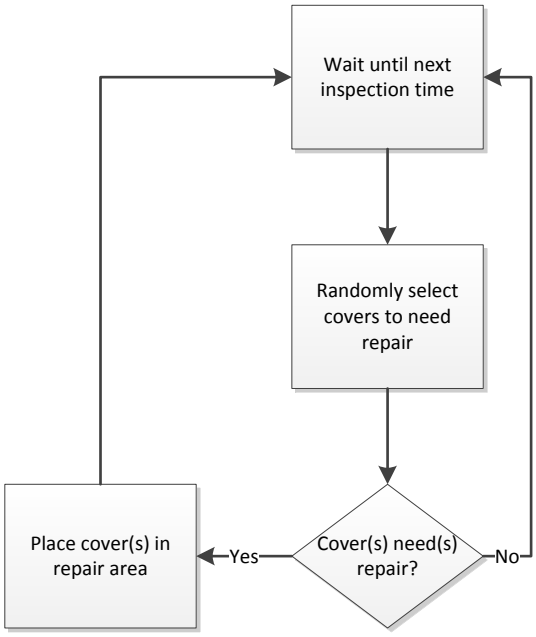

Figure G 9: Inspector Process Map

There are three instances of the Curing_Chariot class, one for each of the E, F, and P lines. The chariots wait for a job in the queue before travelling to the drop point and receiving a cover from either trolley E, F, or P. The chariot then travels to the appropriate curing machine where there is an exchange of cover for cured MProd.

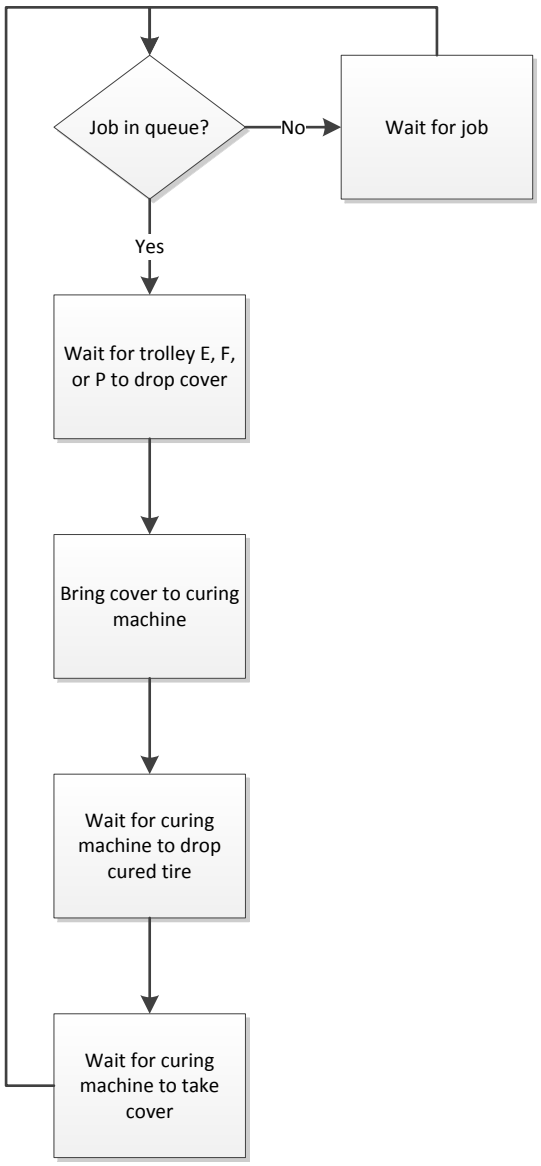

Figure G 10: Curing_Chariot Process Map

The programmed logic of the Curing_Machine class in Fig. G11 applies to all 30 instances of the class in lines E, F, and P. Soon after beginning to cure a MProd, the curing machine informs the monorail system when curing will be complete. Curing continues, and at a fixed interval before curing is complete, the curing machine places a curing MProd request to either trolley E, F, or P. When the curing process is complete, the machine opens and the cured MProd is given to the chariot, once it has arrived. If the curing machine requires an operator, the operator is needed to begin the process of lowering the new cover into the curing machine. Once the cover is in place, the machine closes and the curing cycle begins again.

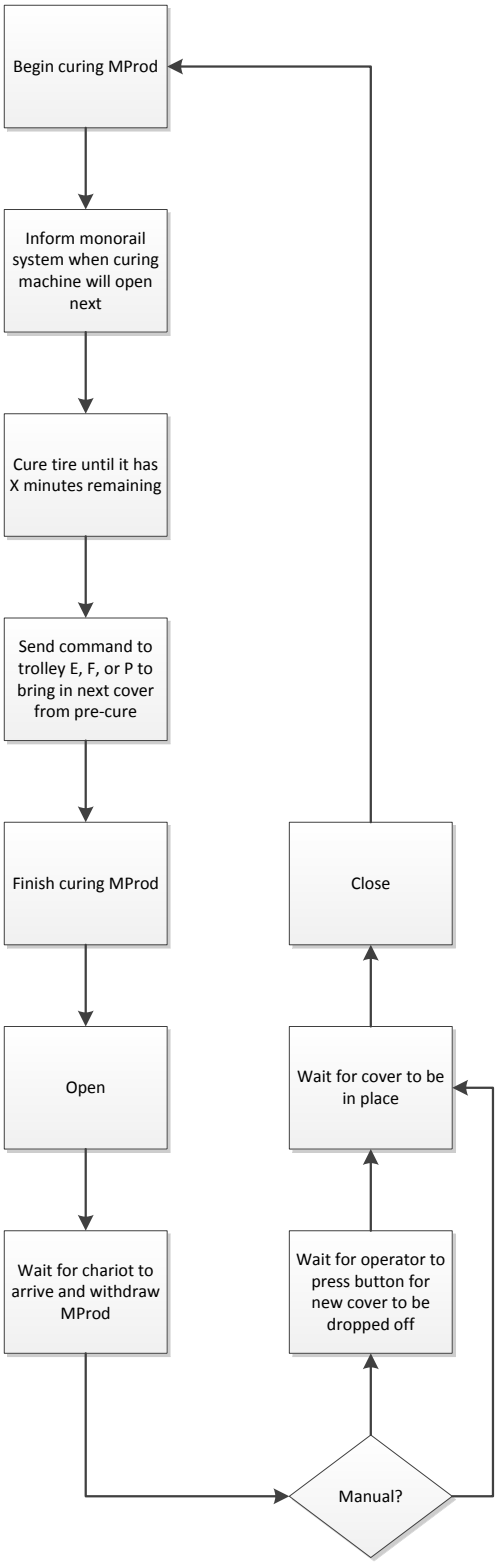

Figure G 11: Curing_Machine Process Map

Each of the three programmed operator behaviours are found in the Curing_Press_Operator class in Fig. G12. Type 0 operators service presses immediately. Type 1 operators take 1 minute to react to and travel to the press before servicing it. Type 2 operators wait for three open presses or ten minutes to pass since the first press has opened before servicing the presses. At this time, they do all jobs in the queue, including jobs that arrive while the operator is servicing presses.

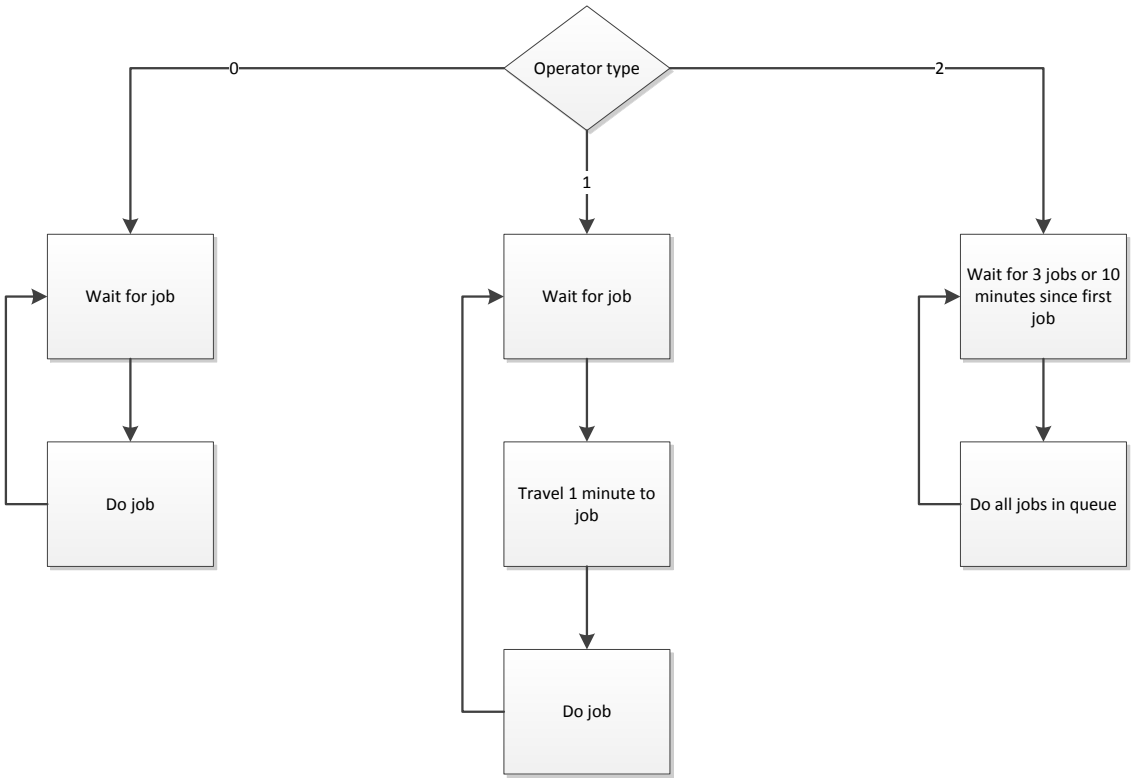

Figure G 12: Curing_Press_Operator Process Map