Simulation and Comparative Evaluation of Flexible Automotive Assembly Layouts

by

Michael Theologos Kelessoglou

B.S., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

Author .....
Department of Electrical Engineering and Computer Science
August 5, 2016

Certified by .....
Julie A. Shah
Associate Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Simulation and Comparative Evaluation of Flexible Automotive Assembly Layouts

by

Michael Theologos Kelessoglou

Submitted to the Department of Electrical Engineering and Computer Science
on August 5, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Equipment malfunction, logistics errors, and other disturbances in automobile factories can be very costly for manufacturers, as they often result in assembly line downtime until the problem is resolved. In this thesis, we examine a new concept for a Flexible Assembly Layout, enabled by new mobile robot technology, that aims to decrease the cost of such errors by removing the affected car from the line while the error is being fixed. In order to compare it to the layout commonly used today, we develop Discrete Event Simulations, based on data from real factories, for both. The simulations contain fast heuristic schedulers that adjust their generated schedules in real time in response to errors. Our comparison shows that, for a representative factory with representative error frequencies, the Flexible Layout is able to produce cars in 29.4% less time than the Conventional Layout, thanks to its robustness to errors.

Thesis Supervisor: Julie A. Shah
Title: Associate Professor

Acknowledgments

I would like to thank my advisor, Prof. Julie Shah. She always made sure I had a plan to achieve my goals. She was always optimistic and encouraging. Even with her hectic schedule, she was always available when I needed guidance.

I would also like to thank the other members of the Interactive Robotics Group for creating a friendly social environment.

Finally, I would like to thank my parents for their love and support.

Contents

Chapter 1

Introduction

Markets today require automobile companies to strive for highly customizable products. Assembly lines that include diverse products require sophisticated planning, as robot and worker agents will have to perform different sets of tasks on different cars. Although modern assembly lines are efficient, they are often not very robust. A disturbance caused by equipment malfunction, logistics issues, or worker related issues can halt the entire line until the problem is resolved.

In this thesis, we develop discrete event simulation models for two automotive assembly layouts, in order to evaluate them. The first layout we model is the *Conventional Layout*, in which the cars are placed on a conveyor belt that moves at a constant speed, while agents on both sides of the belt perform assembly operations on the cars in front of them. The second layout we model is the *Flexible Layout*, a new concept, in which the cars and agents are placed on mobile platforms and move together through a segment of the line. While this means that the agents have to be capable of performing a larger variety of tasks, it also lowers the cost of errors — whether mechanical, logistic, or worker-related — by allowing the affected car to move to the side and be overtaken by the other cars. When similar errors occur in the Conventional Layout, the entire line stops until the error is resolved. To compare the models, we simulate them under various amounts of errors and measure their performance. We do this in order to obtain information that will help determine whether investing in the Flexible Layout concept could be profitable.

The remainder of this thesis is structured as follows:

Chapter 2 covers prior work in the areas of automobile factory simulation and scheduling problems. We first review the concept of the *digital factory* as motivation for the new factory concept. We present work using Discrete Event Simulation and motivate the use of this modeling technique for the work conducted in this thesis. We cover Tercio [1], the scheduling algorithm which inspired the scheduling algorithms we used in our simulation, and explain why we had to modify it for the problem we solved. Finally, we review a study on optimal car sequencing for automobile assembly lines, and discuss how the topic of this thesis differs.

Chapter 3 covers our simulation architecture and scheduling algorithm for the Conventional Layout. In order to specify a mathematical formulation of the problem, we define the concepts of task, temporal constraint, car, agent, band, station, cycle time, error, and schedule. The temporal constraints between tasks that we encounter in our problem are of three types. First, *delay* constraints specify a lower bound on the time difference between one task's completion and another task's start of execution. Second, *immediate* constraints require that one task begins executing immediately after another is completed. Third, *simultaneous* constraints require that two tasks begin executing at the same time. A band is a section of the assembly line. Each band has its own conveyor belt. Our simulations model a single band. The band is separated into equal-length segments called stations. Each station can be occupied by at most one car at a time and is associated with a set of agents who perform tasks on the car occupying the station. The cycle time is the time each car spends at each station if there are no errors in the band and is constant since the stations all have the same length and the cars all move with the same speed.

The inputs to our problem are the sets of cars, agents, stations, and errors and the cycle time. Our system's main outputs are a schedule, which in this case means an assignment of every task to a set of agents that will execute it and the time at which they will be executed, and the list of times at which the cars move to the next station. The schedule must obey the temporal constraints given as input and the spacial constraints of the layout.

We present the system we use to solve the problem, which is comprised of two components, the Pre-Processor and the Simulation-Scheduler. The Pre-Processor applies some approximations that allow us to solve the otherwise computationally intractable scheduling problem. Specifically, it views tasks bound by *simultaneous* constraints as a single multi-agent task, which assumes that the two tasks bound by the constraint have the same duration. It also views tasks bound by *immediate* constraints as a single longer task, which assumes that the two tasks are executed by the same agent. We discuss the cases in which these approximations are appropriate. The Simulation-Scheduler is a single algorithm that carries out two functions. The Simulation part models the state of the line. The Scheduler part makes decisions about when each task should execute. The simulation is a Discrete Event Simulation. All relevant events are placed in a queue which returns the earliest one. The algorithm loops until all tasks have been scheduled. In each loop, after the earliest event from the queue is removed and processed, all cars are moved to the next station if the conditions are met, and the idle agents are assigned to tasks. Finally, we report the runtime and memory usage of our algorithm.

Chapter 4 covers our simulation architecture and scheduling algorithm for the Flexible Layout. In order to specify a mathematical formulation of the problem, we define the concepts of task, temporal constraint, car, agent, band, makespan, parking spot, error, log entry, and car-schedule. The temporal constraints are the same as in the Conventional Layout. Unlike in the Conventional Layout, errors have varying effects depending on their type. The system's inputs are the cars, agents, errors, and some parameters related to the size and shape of the band. The outputs are the schedules for the completion of tasks on each car and a log that keeps track of all the significant events in the simulation.

We present the system we use to solve the problem, which is comprised of three components, the Pre-Processor, the Car-Level Scheduler, and the Band-Level Simulation-Scheduler. The Pre-Processor serves the same purpose as the one in the Conventional Layout. The Car-Level Scheduler, given the tasks and temporal constraints of a car and the agents assigned to it, generates a schedule for the completion of those tasks.

The algorithm for generating the schedule is discrete-event-based. It loops until all tasks have been scheduled. In each loop, it processes the earliest event and then idle agents are assigned to tasks. If a schedule is becomes infeasible thanks to the occurrence of an error, a new one is generated that accounts for the error's effect. The Band-Level Simulation-Scheduler keeps track of the state of the band, assigns agents to cars, and calls the Car-Level Scheduler to generate schedules for them. We use Discrete Event Simulation. The algorithm loops until all cars have left the band. In each loop, the earliest event is processed, and then idle agents are assigned to cars. We first explain how the algorithm works in the absence of errors. Next, we explain the complications added by errors, including how we keep track of parking and the case in which the line stops moving if a car experiencing a specific type of error is unable to find parking. Finally, we report the runtime and memory usage of our algorithm.

Chapter 5 presents the method and results of the empirical evaluation of the two layouts. We first review the layouts we are comparing and the inputs to the two respective simulations. Next, we explained how we selected each of the inputs for our comparison. For the Conventional Layout, the inputs were determined by data drawn from a real automobile assembly line. For the Flexible Layout, we selected most of the inputs such that the parameters of the two layouts we were comparing were similar. An exception to this is band length. We made the Flexible Layout shorter than the Conventional Layout because we wanted the time agents spent on the moving platforms to only slightly exceed the time they spent working on tasks for the cars on the platforms. We provide data on how we selected the length we used in our comparison to minimize agent idle time.

We ran simulations on both layouts under various amounts of errors to observe their performance. The tasks and errors used were randomly selected from probability distributions based on observations of real factories. We reported the total runtime of all our simulations, which was less than 2 hours. The results show that the Flexible Layout finishes the same amount of cars 17.1% faster in the case of no errors and 29.4% faster in the case of an average amount of errors, as observed in real factories.

We provide explanations of these results. The Flexible Layout performs better in the case of no errors because its scheduling problem is less constrained, since the tasks are not restricted to a particular station, as they are in the Conventional Layout. The Flexible Layout responds better to errors because the affected car can park and allow the rest of the cars to overtake it, whereas in the Conventional Layout, the entire band stops until the error is resolved.

Chapter 6 presents the conclusions of this thesis and suggestions for future work. The results show that the Flexible Layout can yield considerable improvement in throughput and resilience to errors, but a full cost-benefit analysis should be performed before determining whether an investment in it could be profitable.

Chapter 2

Prior Work

Fast changing market demands and increasing global competitiveness require modern factories to be flexible and efficient in order to deliver highly customized products [2, 3]. Computer software methods, in particular simulation, have enabled the management of complexity, the speedup of innovation time, and the reduction of manufacturing costs [4]. Customized products require sophisticated scheduling to streamline assembly of varying products on a single line. In this chapter, we present previous work in the areas of simulation and scheduling, related to assembly lines.

2.1 Simulation

A *digital factory* is an integrated approach for virtual manufacturing, including CAD and simulation models of machines, equipment, work cells, lines and plants. Kuehn [5] discusses the components and benefits of the digital factory, which involves modeling, simulating, and monitoring all stages of the production process using software tools. Within this concept simulation is a key technology and can be applied in virtual models on various planning levels and stages to improve the product and process planning. Software vendors offer software solutions to implement this approach.

Computer simulations can relate either to models based on continuous variables or to discrete-event descriptions [6]. Continuous-Variable Simulation is best used to model dynamic systems whose behavior is governed by multiple differential and

algebraic equations. Discrete Event Simulation is best used to model systems whose behavior is determined by certain important events. In this thesis, we use Discrete Event Simulation because the factory assembly line better fits the second description.

Plant, line and process simulation can be performed using Discrete Event Simulation techniques [5], which provide the following benefits:

- Exploration of solutions that minimize the investment cost for production lines while meeting the required production demands
- Detection and elimination of problems that otherwise would require cost- and time-consuming correction measures during production ramp-up
- Improved performance of existing production systems by implementing measures that have been verified in a simulation environment prior to implementation

Discrete Event Simulation has been applied to the virtual analysis of manufacturing systems and has succeeded in reducing times and costs without the need for physical experimentation [7].

Tecnomatix Plant Simulation by Siemens PLM Software [8] is a comprehensive discrete-event simulation program. It allows quick and realistic analysis of production systems leading to optimized performance. It has been used by many automobile companies to plan better assembly layouts. Mahindra Vehicles has used it to develop simulations to determine the optimal buffer storage capacity between lines and the optimal routing from the paint line to the assembly lines [9]. Sichuan BMT Welding Equipment & Engineering used it to reduce their time-to-market [10]. It has been used to evaluate, via simulations, two potential variants of optimization of the production line of an automobile-door manufacturer [11].

Michalos et. al. use simulation models to evaluate factory layouts in which autonomous mobile manipulators and detachable robot grippers allow for reconfiguration on the fly [12]. Fereira et. al. use a discrete event simulation model to study the effects of varying the buffer sizes between different segments of an automobile assembly line [13].

The simulation models in our work give particular emphasis to the following:

- fine granularity of tasks — of the order of 5000 for one car — and their scheduling
- consideration of operational errors and thorough analysis of their various types and effects on the assembly line

2.2 Scheduling

The factory we simulate in this thesis produces highly customizable cars. As a result, the set of assembly operations that must be performed on each car is different. Because of this, there is not one fixed sequence in which the operations are executed on each car. Instead, the simulation must also include a scheduling algorithm that decides the agent that each task is performed by and the time at which it is performed. The schedule generated must obey certain constraints specifying the relative timing of specific pairs of tasks which is necessary for the correct completion of the tasks. For example, a screw must be retrieved before it is attached and tightened.

Gombolay et. al. [1] develop Tercio, a centralized task assignment and scheduling algorithm. Tercio performs agents assignment and task sequencing separately. Its task sequencer is inspired by real-time processor scheduling techniques and returns schedules that are satisficing, but also near-optimal for well-structured problems. Tercio was able to scale better than previous approaches to hybrid task-assignment and scheduling [14], by solving problems of up to 10 agents and 500 tasks in less than 20 seconds on average.

There are four main reasons why we developed our own scheduling algorithms to use in our simulation, instead of using Tercio. First, some tasks of the cars we model are bound by *simultaneous* temporal constraints, meaning that they must execute at the same time. Tercio can produce schedules that obey delay and deadline constraints, but it does not handle *simultaneous* constraints. Second, the only type of deadline constraints found between pairs of tasks in the cars we model are *immediate*

constraints, meaning that one task must execute immediately after the other task. Accommodating such constraints is simpler computationally than accommodating deadline constraints, which are a generalization of *immediate* constraints. Thus, by creating specialized scheduling algorithms that only work for the *immediate* subset of deadline constraints, we can achieve faster runtimes. Third, there are certain aspects in the structure of the scheduling problems we want to solve that would be inconvenient to incorporate into Tercio, as it was not designed specifically for these problems. For example, in the Conventional Assembly Layout, cars stay at each station for the same amount of time, even if their tasks for a station are completed earlier. Fourth, due to the size of scheduling problems we want to solve, we are forced to make a different tradeoff between computational complexity and quality of schedules produced. While Tercio scales well for problems of 500 tasks, the problems we had to solve were of the order of 1 million tasks. Also, the quality of the schedules is not as important to us, since our ultimate goal is to emulate a scheduler to compare factory layouts, not to develop a close-to-optimal scheduling algorithm.

Mazur et. al. [15, 16] solve the problem of finding an optimal sequence of cars to maximize assembly line throughput. Their problem differs from ours in a few significant ways. First, for us, the sequence of cars is a given and fixed, while for them it is the variable they are solving for. Second, their model considers the total makespan of each car's tasks at a station, while our model has finer granularity, considering each task separately, and scheduling it individually. Third, their model does not include unforeseen disturbances that can occur during production, while ours does.

Chapter 3

Conventional Automotive Assembly Layout

3.1 Overview

Figure 3-1: BMW Plant in Spartanburg, SC

The most widely used automotive assembly layout is usually attributed to Henry Ford, who installed a conveyor belt that enabled mass production of the *Model T* [17]. Figure 3-1 shows a picture from inside a state-of-the-art factory.

In its modern form, the layout consists of multiple linear sections, each containing their own belt, separated by buffer zones [18]. These sections are called bands and each one corresponds to a different stage in the assembly process. Figure 3-2 shows an illustration of a segment of a band. Human or robot agents, situated on both sides of the belt, perform assembly operations on car as they pass in front of them. We will be simulating assembly lines for highly customizable cars, each requiring a unique set of tasks. Because of this, agents may have to perform different tasks on different cars. For agents to know what tasks they need to perform in the limited time that the car is in front of them, they are given a schedule. In fact, the entire sequence of cars to be produced in a day is determined a few days before. This allows for the correct parts to be available and for a schedule for the entire assembly process to be generated [19, 20].

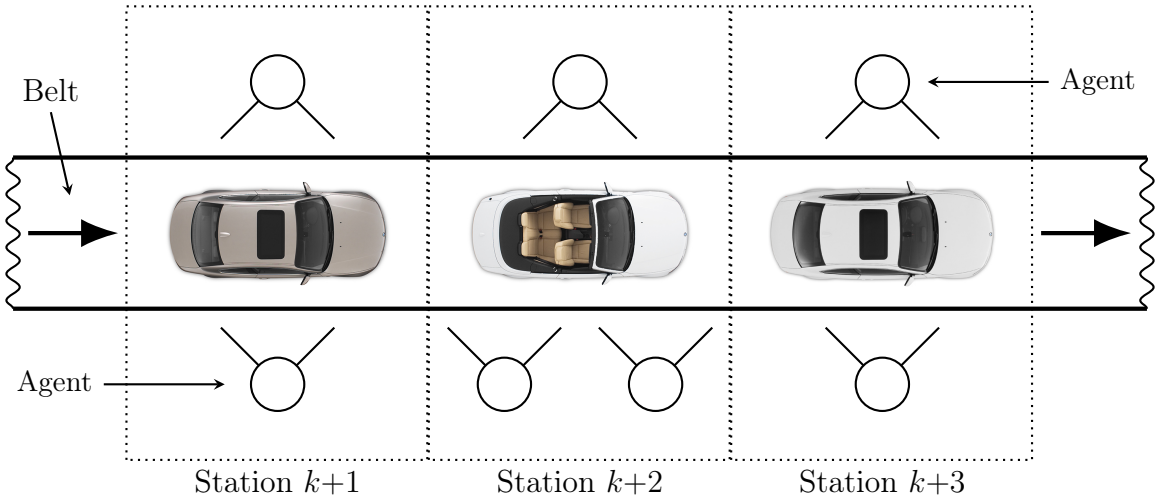

Figure 3-2: Conventional Automotive Assembly Layout

While this assembly process is efficient, it suffers greatly when something goes wrong. If an agent cannot complete its tasks on a car before the car moves on, the agent must stop the movement of the entire band until it can finish them. Whether the cause of the disturbance is worker-related, machinery-related, resource-related,

or something else, the effect on the band is the same. All cars stop moving and all agents that have finished their tasks remain idle. An error lasting long enough can cause a loss of productivity outside the band as well. Specifically, bands behind the one the error occurred on will experience blocking, meaning that they will not be able to continue moving because the cars they output have nowhere to go. On the other hand, bands ahead of the one the error occurred on will experience starvation, as they will process all the cars available to them, and will then have to remain idle. These effects on other bands can be somewhat alleviated by introducing buffer zones between bands, such that each band can output a few more cars before it blocks or process a few more cars before it starves. Buffer zones, however, still don't deal with errors that take a long time to fix nor do they address the lack of productivity on the band experiencing the error. The concept for a factory layout that aims to minimize the cost of errors will be introduced in the next chapter.

In this chapter, we will develop a simulation and scheduling algorithm for one band in this factory layout. The simulation will keep track of the state of the band, while the scheduler makes the decisions that an online[1] scheduler would have to make in real time. This will allow us to compare the new concept to it and help determine whether it would be worthwhile.

3.2 Definitions

Task $w < c_w, d_w, s_w, A_w, ts_w, te_w >$

The process of car assembly is broken down into small steps that we call tasks. In our model, the amount of time a task takes to complete is an attribute of the task and not dependent on the time at which the task is executed or the set of agents executing it. In this factory layout, the agent or set of agents that perform each task is predetermined. We define the set of all tasks W. A task $w \in W$ is characterized

[1] An online scheduler has to develop quickly and adjust a schedule in response to events that it could not have predicted as the schedule is deployed. This is in contrast to an offline scheduler that has more time to develop a schedule before the schedule is actually deployed.

by the following traits:

- the car c_w for which it needs to be executed
- the duration d_w that it takes to complete
- the station s_w at which it needs to be executed
- the set A_w of agents that are required to execute it. All agents in A_w must be from the station s_w.
- the time ts_w at which it begins being executed. Note that ts_w is unspecified until the task is scheduled.
- the time te_w at which it is finished. Note that te_w is unspecified until the task is scheduled. When ts_w has been specified, te_w is given by

$$te_w = ts_w + d_w.$$

Temporal Constraint $\delta_{l_{ij}} \leq t_j - t_i \leq \delta_{u_{ij}}$

For some pairs of tasks, the times at which they are executed need to have a specific relation. We call this relation a temporal constraint between the two tasks. The temporal constraints we will be dealing with in our model are Simple Temporal Constraints (STC), which means that they are of the form $t_j - t_i \leq \delta$ for two timepoints t_i, t_j [21]. We encounter them in pairs of the form $t_j - t_i \leq \delta_{ji}$ and $t_i - t_j \leq \delta_{ij}$, where $\delta_{ij} \leq 0 \leq \delta_{ji}$. Together these constraints specify an upper bound $\delta_{u_{ij}} = \delta_{ji}$ and a lower bound $\delta_{l_{ij}} = -\delta_{ij}$ for the time difference between t_i and t_j. In this chapter and the next, when we mention Temporal Constraints, we are referring to pairs of STC's of this form, which we write as:

$$\delta_{l_{ij}} \leq t_j - t_i \leq \delta_{u_{ij}}.$$

We define the set of these Temporal Constraints D. In our model, which is based on a real factory, there are three types of simple temporal constraints. For two tasks

A and B with start times ts_A and ts_B and end times te_A and te_B respectively, the types of ways in which B could be constrained by A are the following:

Delay constraint: B must be executed after A has finished and the time between A finishing and B starting must be lower bounded by the delay. In terms of STC notation, these constraints are of the form $d \leq ts_B - te_A \leq \infty$ or just $ts_B \geq te_A + d$, where d is the delay. Our ability to schedule B depends on the completion of A, so we call A the releasing task and B the constrained task.

Immediate constraint: B must begin executing immediately after A has finished. In terms of STC notation, these constraints are of the form $0 \leq ts_B - te_A \leq 0$ or just $ts_B = te_A$.

Simultaneous constraint: B must begin executing at the same time as A. In terms of STC notation, these constraints are of the form $0 \leq ts_B - ts_A \leq 0$ or just $ts_B = ts_A$.

Car $c < n_c, W_c, D_c >$

We define the set of all cars C. A car $c \in C$ is characterized by the following traits:

- the unique number n_c that describes its order in the sequence of all cars
- the set of tasks W_c that need to be performed on it
- the set of temporal constraints D_c between its tasks

Agent $a < s_a, i_a >$

Agents are the units that perform the tasks. Whether they are human workers or robots doesn't matter in our simulation. In this layout, agents are fixed to a specific station in the band and can only perform some types of tasks. This is why tasks must specify which agents are required to execute them. We define the set of all agents A. An agent $a \in A$ is characterized by the station s_a that its location in the band corresponds to and a unique identifier i_a to distinguish it from other agents in s_a.

Band

The assembly line is separated into linear sections which we call bands. Each band performs different kinds of tasks according to what agents, tools, and resources it has available. We will simulate the events that occur on a single band. In this layout, the band consists of a moving belt with agents stationed at both sides. Cars are placed on the belt and move in front of the agents, who perform tasks on the car in front of them. During normal operation, the belt moves at a constant speed v.

Station s

In our model, we separate the band into sections of equal length l we call stations. A station represents the portion of the band on which a set of agents[2] operates. At most one car can occupy each station at a time.

Cycle Time t_c

Since the speed of the cars in a band is constant during normal operation and the length of all the stations is the same, the amount of time each car spends at each station is constant. We call this the cycle time t_c and it is given by $t_c = l/v$. During normal operation, each car moves to the next station every t_c.

Error $e < c_e, ts_e, d_e, te_e, o_e >$

Any event that can cause the band to stop moving is an error. Though there can be many different causes for an error[3], they all have the same effect on the band, so we do not distinguish between them in our model. If an error is resolved by the time the cycle time is completed, then it has no effect on the band. If an error does not resolve in time, cars remain at their current stations and the band does not move forward until the error is resolved. In our model, for the sake of simplicity, each car can only

[2]typically consisting of two agents, one on each side of the band

[3]The cause of an error can be worker-related, machinery-related, logistics-related, or information-system-related, among others.

experience one error at a time[4]. We define the set of all errors E. An error $e \in E$ is characterized by the following traits:

- the car c_e it occurs on
- the time ts_e at which it occurs. Note that ts_e is initially unspecified and is determined by our simulation.
- the duration d_e it takes to resolve. The countdown for resolution starts immediately when the error occurs.
- the time te_e at which the error is resolved. Note that te_e is unspecified until ts_e is specified. When ts_e is specified, the resolution time is given by $te_e = ts_e + d_e$.
- the time offset o_e after the resolution of the previous error on car c_e at which e occurs[5]. The time offset is used in the simulation to specify the error's start time. The reason we specify the time at which an error occurs based on the time at which the previous error on the same car is resolved is to ensure the property that there is no overlap of errors on the same car. The process of randomly generating errors that do not overlap becomes simpler when the only condition that must be met is that the time offsets must be non-negative.

Schedule S

One of our goals in this chapter is to generate a schedule for the assembly work. In this model, a schedule S consists of a mapping $M: w \rightarrow ts_w$ of every task to a time at which it should be executed. The agents that will perform the tasks are predetermined and thus do not need to be specified in the schedule.

[4]In a real factory, cars can experience multiple errors at the same time, but errors are rare enough that this almost never happens. It would not be worth it for us to complicate our model to account for this extremely rare occurrence.

[5]For the first error occurring on a car, the time offset refers to the amount of time after the car enters the band after which the error occurs.

3.3 Problem Formulation

Inputs

The inputs to our system are the following:

- the set C of cars
  - the union of W_c over all $c \in C$ is the set W of tasks
  - the union of D_c over all $c \in C$ is the set D of temporal constraints
- the set A of agents
- the number n_s of stations
- the cycle time t_c
- the set E of errors that will occur

Outputs

The outputs of our system are the following:

- the schedule S
- the list T of moving times at which the cars in the band moved to the next station
- a mapping $M: e \rightarrow ts_e$ of all errors to the times at which they occurred

Input-Output Constraints

Certain relations must exist between our input and output as well as between different parts of our output for our system to be true to our model for the Conventional Layout. They are listed below, along with formulations using quantities defined above:

- All tasks are scheduled.

$$\exists ts_w, \forall w \in W$$

- No task is in progress while cars are moving to the next station. Another way to say this is that for every moving time, no task can start before it and end after it. This means that the difference between the moving time and the task's start time must have the same time as the difference between the moving time and the task's end time, so their product must be non-negative.

$$(t - ts_w)(t - te_w) \geq 0, \forall w \in W \text{ and } \forall t \in T$$

- No error is in progress while cars are moving to the next station. Another way to say this is that for every moving time, no error can start before it and end after it. This means that the difference between the moving time and the error's start time must have the same time as the difference between the moving time and the error's end time, so their product must be non-negative.

$$(t - ts_e)(t - te_e) \geq 0, \forall e \in E \text{ and } \forall t \in T$$

- Each station is occupied by the appropriate car in the appropriate time frame. For $0 \leq t < T[0]$, car 0 is at station 0 and all other cars are outside the band. For $T[0] \leq t < T[1]$, car 0 is at station 1 and car 1 is at station 0. In the general case, if $T[i-1] \leq t < T[i]$, then car j is at station $i - j$. If $i - j < 0$, then the car has not yet entered the band. If $i - j \geq n_s$, then the car has left the band.

$$T[i-1] \leq ts_w < T[i] \Rightarrow s_w = i - n_{c_w}, \forall w \in W$$

  From this it also follows that each station is occupied by at most one car at a time.

- No two moving times are separated by less than the cycle time

$$|t_1 - t_2| \geq t_c, \forall t_1, t_2 \in T$$

- All cars move to the next station if cycle time has passed since the last move time unless an error is in progress or some cars have not finished their tasks at their current stations

$$T[i] - T[i-1] > t_c \Rightarrow \begin{cases} \exists e \in E: ts_e < T[i-1] + t_c < te_e \\ \text{or} \\ \exists w \in W: s_w = i - n_{c_w} \text{ and } te_w > T[i-1] + t_c \end{cases}$$

- An agent can be working on at most one task at any given time.

$$A_{w_1} \cap A_{w_2} \neq \emptyset \Rightarrow te_{w_1} \leq ts_{w_2}, \forall w_1, w_2 \in W \text{ where } w_1 \neq w_2 \text{ and } ts_{w_1} \leq ts_{w_2}$$

- All temporal constraints must be satisfied.

$$\delta_{l_{ij}} \leq t_j - t_i \leq \delta_{u_{ij}}, \forall t_i, t_j \text{ bound by a constraint in } D$$

- The errors occur at times that obey their time offsets. Specifically, if we let e_i be the ith error that occurs on car c:

$$ts_{e_i} = te_{e_{i-1}} + o_{e_i}$$

3.4 System Overview

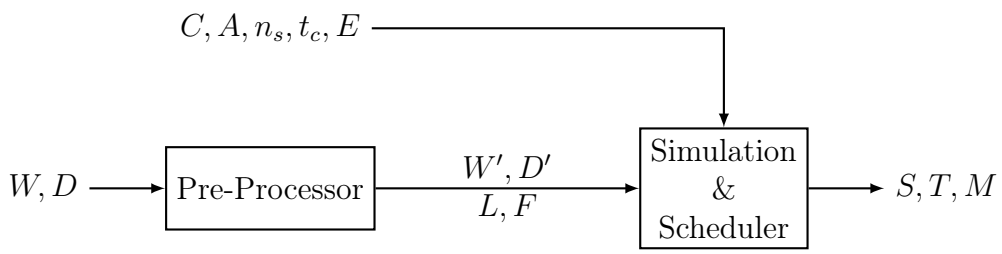

Figure 3-3: System Overview

The system we develop in this chapter has two main components: the Pre-

Processor and the Simulation-Scheduler. The Pre-Processor applies some approximations that simplify the scheduling problem, allowing us to use an efficient scheduling algorithm. The Simulation and Scheduler can be conceptually thought of as two different entities that perform different tasks, but for convenience and efficiency we implemented them in one algorithm. The Simulation keeps track of the state of the band, including the cars occupying each station, the occurrence and resolution of errors, and the availability of the agents. The Scheduler makes real-time decisions about which task each agent should perform at any given time. Although the Simulation has access to more information, the Scheduler only uses information about the cars and the present state of the band, information that would also be available to a real online scheduler, which is what our Scheduler is emulating. Figure 3-3 shows the inputs and outputs of each component.

3.5 Pre-Processor

Optimal scheduling for problems that include delay, immediate, and simultaneous constraints is not solvable in polynomial time, as it is a generalization of the Multi-Processor Scheduling Problem[6], which is NP-complete [22]. We are not necessarily aiming to produce optimal schedules, as our ultimate goal is to compare factory layouts, not develop an optimal scheduler. In order to generate a good enough schedule in reasonable time, we make some approximations. Specifically, the Pre-Processor eliminates simultaneous and immediate constraints by merging tasks that are bound by such constraints.

Tercio[1], which inspired our scheduling algorithm, does not deal with simultaneous constraints. We are willing to make certain approximations, which hold true in the common case, that make satisfying simultaneous constraints simple. Specifically, we assume that tasks bound by simultaneous constraints have the same duration. If this holds true, then the two tasks will be executed by two agents as if they were a

[6]The original Multi-Processor Scheduling Problem is given a partial ordering of the tasks and does not include other constraints. Thus, it is a special case of our scheduling problem in which there are only delay constraints where the delay is 0.

single multi-agent task. Our assumption allows us to merge pairs of tasks bound by simultaneous constraints in the following manner. If we have two tasks u and v, we merge them into one task w with $d_w = \max(d_u, d_v)$ and $A_w = A_u \cup A_v$. All temporal constraints that used to apply to one of the initial tasks apply to the merged task. In the case where our assumption is not true, we do introduce an additional constraint because we will occupy the agents for the shorter of the two tasks after it has finished until the longer task has also finished. Also, if any other tasks have a precedence constraint on the shorter of the two tasks and not the longer one, then they will have to wait for the longer one. However, the assumption we made almost always holds true.

Tercio deals with deadline constraints, which are a superset of immediate constraints. Much of the complexity of Tercio is derived from trying to accommodate those deadline constraints. The problem we are trying to solve does not require this complexity, since satisfying an immediate deadline does not allow a multitude of options. If we schedule an agent to perform a task that is an immediate constraint to another task, we know that the other task will be performed immediately after the first is completed and most likely by the same agent. In the common case, the same agent will be performing both tasks, so the effect is as if there is one longer task rather than two shorter ones. Our assumption allows us to merge tasks bound by immediate constraints in the following manner. If we have two tasks u and v, we merge them into one task w with $d_w = d_u + d_v$ and $A_w = A_u \cup A_v$. All temporal constraints that used to apply to one of the initial tasks apply to the merged task. In the common case[7] according to real factory data, this does not introduce additional constraints, but if another task has a dependence on the first of the two tasks and not the second, or the second of the two tasks has a dependence on another tasks which the first of the two tasks does not share, then an additional constraint is introduced to the problem. In the first case, the other task will have to wait for the entirety of the merged task time, instead of just the first of the two tasks. In the second case,

[7] Usually, two tasks that are bound by an immediate constraint share all other precedence constraints.

we will have to wait for the other task to finish before we begin the merged task, instead of starting it such that we leave enough time for it to finish before we are on the second part of the merged task.

There is one case in which we do not merge tasks bound by immediate constraints. This exception is the case in which the two tasks are performed by agents in different stations. Such a merging would complicate the scheduling process, but more importantly introduce the inefficiency of one car occupying two stations at the same time. Our workaround for this is fairly simple. We observe that when tasks in two consecutive stations are bound by an immediate constraint, the task from the earlier station must be the last one completed in its station, and the task from the later station must be the first one executed in its station. Thus, the Pre-Processor encodes these constraints by specifying two lists L and F, of tasks that must be completed last in their station and executed first in their station respectively.

We always merge simultaneous constraints first before merging immediate constraints because the other order could result in a merged task with unclear relations to other tasks. The following example shows why merging constraints in the reverse order can result in unclear behavior. Consider tasks a, b, and c in Figure 3-4a. Tasks b and c are bound by a simultaneous constraint. Both b and c must immediately follow a. If we first merge the simultaneous constraint, we get a merged task bc which must immediately follow a, as shown in Figure 3-4b. We can then merge a and bc to get the fully merged task. On the other hand, if we merge immediate constraints first, we will get a merged task ab whose relation to c is both simultaneous and immediate, as shown in Figure 3-4c. Merging ab and c would be very complicated due to the information lost during the merging of a and b.

To summarize the Pre-Processor's function, it takes the set of tasks W and the set of temporal constraints D as inputs and outputs a new set of tasks W' and a new set of temporal constraints D' that account for the merged tasks, as well as the set L of tasks that must be completed last in their station and the set F of tasks that must be completed first in their station.

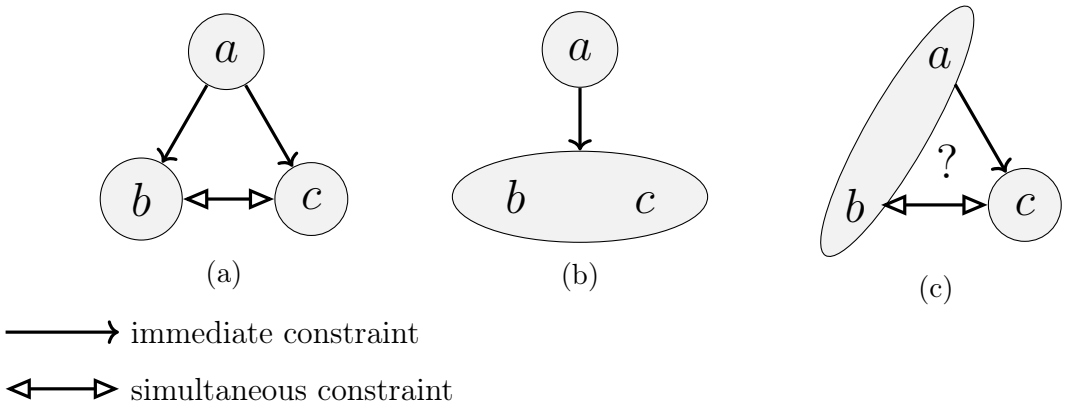

Figure 3-4: Constraint Merging Example

3.6 Simulation and Scheduler

3.6.1 Approach: Discrete Event Simulation

A Discrete Event Simulation models a system as a state machine whose state can only change because of certain discrete instantaneous events [23]. It processes these events in the order they occur to get the state of the system at any given time. This is in contrast to Continuous Simulation which continuously keeps track of the state of the system. A Continuous Simulation breaks down time into small periods and calculates the changes in the system that occur over every such period. Because of this, Continuous Simulation can provide detailed information about the system dynamics that is very useful for modeling systems with multiple rapidly changing parameters. However, its drawback is the higher computational cost due to the large number of time periods for which calculations are made, as opposed to the smaller number of time periods that separate discrete events.

We elected to use Discrete Event Simulation, as it better fits our model. Even though the state of the line doesn't remain constant between the discrete events, it changes in a predictable and easy to calculate way. Cars stay in their station for a predictable amount of time and agents complete their tasks in a predictable amount of time. The types of events that we use in our simulation are the following:

Agent becomes available (AE): An agent becomes available when it finishes a

task it was assigned to. An event $z \in AE$ of agent a_z becoming available occurs at time t_z if:

$$\exists w \in W : a_z \in A_w \text{ and } t_z = te_w$$

Delay constraint is satisfied (DE): A delay constraint is satisfied after the specified delay time has elapsed since the completion of the releasing task. An event $z \in DE$ of a delay constraint of d_z being satisfied between two tasks u and w occurs at time t_z if:

$$t_z = te_u + d_z$$

Station finishes its tasks (SE): A station finishes its tasks when the last task to complete by the car occupying it completes. An event $z \in SE$ of a station s finishing its tasks occurs at time t_z if:

$$\exists w \in W : te_w = t_z \text{ and } te_u \leq t_z, \forall u \in W_w$$

We define W_w as the set of all tasks $u \in W$ for which $s_u = s_w$ and $c_u = c_w$.

Error occurs (ESE): An error occurs at the time it is assigned by the simulation. An event $z \in ESE$ of an error e occurring occurs at time t_z if:

$$t_z = ts_e$$

Error is resolved (EEE): An error is resolved after its duration has elapsed since its occurrence. An event $z \in EEE$ of an error e being resolved occurs at time t_z if:

$$t_z = te_e$$

Cycle time expires (CE): The cycle time expires when it has elapsed since the last movement of cars to the next station. An event $z \in CE$ occurs at time t_z if:

$$\exists t \in T : t_z = t + t_c$$

We place these events in a queue that always returns the one with the earliest timestamp to be processed.

3.6.2 Algorithm

We will now describe our SIMULATE algorithm (Algorithm 1), which also incorporates the Scheduler. As shown in Figure 3-3, the inputs to SIMULATE are the Pre-Processor's outputs, W', D', L, and F, discussed in Section 3.5, as well as the inputs to the system C, A, n_s, t_c, and E, discussed in Section 3.3. The outputs of SIMULATE are S, T, and M, also discussed in Section 3.3.

Algorithm 1 Conventional Layout Simulation

```
1:  procedure SIMULATE(C, W', D', L, F, A, n_s, t_c, E)
2:      INITIALIZE
3:      while rt > 0 do
4:          z ← Queue.remove()
5:          time ← t_z
6:          HANDLEEVENT(z)
7:          if |DS| = n_s and ec = 0 and ct then
8:              T.add(time)
9:              move cars to the next station
10:             add ESE event for first error of car that just entered the band
11:             empty DS except for unoccupied stations
12:             ct ← false
13:             add CE event to Queue
14:         end if
15:         for a ∈ AA do
16:             if CS[s_a] ≠ nil then
17:                 w ← GETBESTHEURISTIC(AW[CS[s_a]](a))
18:                 if all agents in A_w available then
19:                     schedule w at time in S
20:                 end if
21:             end if
22:         end for
23:     end while
24:     return S, T, M
25: end procedure
```

The data structures used in SIMULATE are initialized in the INITIALIZE procedure (Algorithm 2). In the beginning of the simulation, there are no cars in the band, and

Algorithm 2 Conventional Layout Initialize Procedure

1: **procedure** INITIALIZE
2: create array CS of length n_s ▷ maps stations to the cars occupying them
3: **for** $i \leftarrow 0...n_s - 1$ **do**
4: $CS[i] \leftarrow$ nil
5: **end for**
6: $ec \leftarrow 0$ ▷ error count
7: $ct \leftarrow$ **false** ▷ boolean describing whether cycle time has expired
8: create map $h: w \rightarrow h(w)$ ▷ $h(w)$ is a heuristic
9: **for** $w \in L$ **do**
10: $h(w) \leftarrow -\infty$
11: **end for**
12: **for** $w \in F$ **do**
13: $h(w) \leftarrow \infty$
14: **end for**
15: create map $CW: w \rightarrow CW(w)$ ▷ $CW(w)$ is the set of tasks that have a dependence on w
16: create map $RD: w \rightarrow RD(w)$ ▷ $RD(w)$ is the set of unsatisfied dependencies of w
17: create array of lists XC ▷ $|C|$ different lists, $XC[n_c]$ holds the sequence of all errors that will occur on car c
18: **for** $e \in E$ **do**
19: $XC[n_{c_e}]$.add(e)
20: **end for**
21: create set DS ▷ set of stations that have finished their tasks this cycle
22: **for** $i \leftarrow 0...n_s - 1$ **do**
23: DS.add(i)
24: **end for**
25: create set AA ▷ set of agents not currently assigned to a task
26: **for** $a \in A$ **do**
27: AA.add(a)
28: **end for**
29: create array of maps $AW[i]: a \rightarrow AW[i](a)$ ▷ $AW[i](a)$ is the set of tasks of car i assigned to a whose precedence constraints have been satisfied
30: **for** $w \in W$ **do**
31: **if** $RD(w) = \emptyset$ **then**
32: $AW(a_w)$.add(w)
33: **end if**
34: **end for**
35: create empty Queue
36: Queue.add(CE event at time 0)
37: $rt \leftarrow |W'|$
38: **end procedure**

Algorithm 3 Conventional Layout Handle Event Procedure

```
1: procedure HandleEvent(z)
2:     if z ∈ AE then
3:         AA.add(a_z)
4:     else if z ∈ DE then
5:         w ← z.constrained
6:         RD(w).remove(tc_z)
7:         if RD(w) = ∅ then
8:             AW(a_w).add(w)
9:         end if
10:    else if z ∈ SE then
11:        DS.add(s_z)
12:    else if z ∈ ESE then
13:        ec + +
14:        add EEE event to Queue
15:        M[e] ← time
16:    else if z ∈ EEE then
17:        ec − −
18:        if XC[n_cz] ≠ ∅ then
19:            e ← XC[n_cz].remove()
20:            add ESE event for e to Queue
21:        end if
22:    else if z ∈ CE then
23:        ct ← true
24:    end if
25: end procedure
```

the only event in the queue is a CE event at time 0 (lines 35–36). At all times, we keep track of the car occupying each station (CS, initialized in lines 2–5), the number of ongoing errors (ec, initialized in line 6), whether the cycle time has expired since the last move time (ct, initialized in line 7), the set of unsatisfied dependencies of each task (RD, initialized in line 16), the set of stations that have finished their tasks in the current cycle (DS, initialized in lines 21–24), the set of available agents (AA, initialized in lines 25–28), the set of tasks each agent is able to execute (AW, initialized in lines 29–34), and the number of tasks that have not yet been scheduled (rt, initialized in line 37). For a task to be ready to execute, its car needs to be at the station where its agent is located and all its precedence constraints must be satisfied. For each task, we calculate the priority heuristic (line 8), whose goal is to

give the agents as many options of tasks to perform at any given moment as possible in order to increase agent utilization. The heuristic we use is the number of inter-agent precedence constraints and is further explained in Section 3.6.4. We set the heuristic of all tasks in L to $-\infty$ since we want them executed last in their station and the heuristic of all tasks in F to $+\infty$ since we want them executed first in their station (lines 9–14).

The simulation consists of a loop that runs until all tasks in W' have been scheduled (line 3). The loop consists of three parts. In the first part (lines 4–6), the algorithm processes the earliest event from the queue and any other events with the same timestamp as that event. In the second part (lines 7–14), the algorithm checks whether the conditions for the line to move are met and moves the line if they are. In the third part (lines 15–22), the algorithm assigns available agents to the available tasks with the highest value heuristic. We will now discuss these parts in more detail.

Events are processed by the HandleEvent procedure (Algorithm 3). Different types of events are processed in different ways:

- AE events add their agent to the set of available agents (line 3).
- DE events remove their dependency from the set of unsatisfied dependencies of the constrained task (line 6), and add that task to the set of available tasks if its set of unsatisfied dependencies is empty (lines 7–9).
- SE events add their station to the set of stations that have finished their tasks (line 11).
- ESE events increment the error count (line 13) and add the EEE event for their resolution to the queue at $time + d_e$ (line 14). They also assign $time$ to e in M (line 15).
- EEE events decrement the error count (line 17) and add the next ESE event for the car to the queue at $time + o_e$, if there is another error for that car in E (lines 18–21).

- CE events set the boolean indicating that the cycle time has expired since the last move to true (line 23).

The condition that has to hold for the cars to move to the next station is that all stations must have finished their tasks, the error count must be 0, and the cycle time must have expired (line 7). If these conditions are met, the line moves. When the line moves, we add the *time* to T (line 8). The algorithm shifts the array that keeps track of the car at each station to the right and fills the empty spot created with the next car to enter the band (line 9). It adds the ESE event for the first error that car will encounter to the queue, if there is one (line 10). It empties the set of stations that have finished their tasks, with the exception of vacant stations, either because the last car has left them or the first car has not yet reached them (line 11). It also sets the boolean of whether the cycle time has expired to false (line 12) and adds the next CE event to the queue at $time + t_c$ (line 13).

After the events have been processed and the moving condition has been checked, the last thing the algorithm tries to do before it loops again is assign tasks to the available agents. It iterates over all available agents (line 15), and for those that are not at vacant stations it selects the task with the best heuristic of those available (line 17), checks that any other agents needed for the task are available, and, if so, schedules it (line 19). After it schedules a task, it removes the assigned agents from the set of available agents and checks if there are no tasks left to schedule at the station, and if so puts an SE event in the queue at *time*.

3.6.3 Queue Implementation

We implement our queue of events as a min-heap. A heap is a tree-based data structure that satisfies the heap property, which is a relation between a parent and each of its children. In our case, this property is that a parent event will have a timestamp that is lower than or equal to those of its children. It follows from the heap property that the event at the root will have the lowest timestamp, hence min-heap, and will thus be the earliest of these events to occur and the first one to be

processed.

3.6.4 Heuristic

The heuristic we use is that of inter-agent precedence constraints, the most relevant heuristic to our situation of those used in Tercio. Consider a situation with two agents a_1, a_2 and three tasks w_1, w_2, w_3. Tasks w_1 and w_2 are assigned to a_1 while w_3 is assigned to a_2. The only temporal constraint in the system is that w_3 must be executed after w_2 has completed. If we schedule a_1 to execute w_1 before w_2, then a_2 will have to wait for a_1 to finish both its tasks before it can begin working on w_3. On the other hand, if we schedule w_2 before w_1, then a_2 can work on w_3 while a_1 works on w_1, resulting in better agent utilization and a schedule that finishes all the tasks in less time. What makes w_2 a better choice than w_1 in this case is the fact that it is a precedence constraint for a task on a different agent. Satisfying such constraints first gives other agents more options for what tasks to execute, resulting in less idle time. Because of this, the heuristic gives priority to the tasks whose completion will satisfy the most inter-agent precedence constraints.

As in the case of the event queue, our heuristic is implemented with heaps, specifically max-heaps. Instead of iterating through the entire set of tasks an agent could execute to find the one with the best heuristic, we use a max-heap whose root will have the best heuristic thanks to the heap property. This allows us to find the best heuristic in $O(\log n)$ time rather than $O(n)$ time for sets of n tasks.

3.7 Runtime and Memory Analysis

The Pre-Processor has to iterate over all temporal constraints to do the merging, and must produce new lists of tasks and constraints, so its runtime and space requirements are $O(|W| + |D|)$.

The number of times the simulation algorithm loops is upper bounded by the number of total events processed. To find the total number of events, we go over how many were processed for each type. Since the number of agents per task is upper

bounded by a constant[8], there are $O(|W|)$ AA events. There are $O(|D|)$ DE events. Each station will finish each car once, so there are $O(n_s|C|)$ SE events. There are $O(|E|)$ ESE and EEE events. There will be $|C|$ moves before the last car enters the band and n_s moves after that to get it to the last station, so there are $O(|C| + n_s)$ ME events. Thus, the number of total events, and consequently the number of times the algorithm loops, is

$$O(|W| + |D| + |E| + n_s|C|).$$

Since we implemented our queue as a min-heap, removing the earliest event takes $O(\log n)$ time, where n is the size of the queue[9] The processing of AE, DE[10], SE, and CE events takes constant time. ESE and EEE events add other events to the queue, which is a $O(\log n)$ operation where n is the size of the queue, since we implemented our queue as a min-heap. Thus, the total amount of time contributed by the first part of the loop is

$$O((|W| + |D| + |E| + n_s|C|) \log n).$$

The check of whether to move the line is a constant time operation. As stated earlier there will be a total of $O(|C| + n_s)$ movements. Each movement includes shifting the array of cars at each station, which takes $O(n_s)$, and adding an ESE and a CE event to the queue, which takes $O(\log n)$. Thus, the total time contributed by the second part of the loop is

$$O((|W| + |D| + |E| + n_s|C|) \log n + (|C| + n_s)(n_s + \log n)).$$

In the third part of each loop, we iterate over all available agents. In practice, throughout most of the simulation, few agents will be idle, but in the worst case

[8]Usually there are at most two agents per task, since there are typically at most two agents per station.

[9]Reading the earliest event is a constant time operation, since it is just the root of the heap tree, but removing it leaves a broken tree, which takes $O(\log n)$ time to repair.

[10]Processing a DE event involves removing a constraint from the list of unsatisfied constraints of a task. We can view this as a constant time operation if we assume that the number of precedence constraints on any single task is small compared to the total number of precedence constraints, which is a reasonable assumption.

the size of the available agents set is $|A|$. The cost of each check is constant time. The total number of tasks scheduled is $|W|$, and for each one, we pick it as the best heuristic out of the set of available tasks, and add the AA events to the queue, which is $O(\log n)$. Thus, the total amount of time contributed is

$$O((|W| + |D| + |E| + n_s|C|)|A| + |W| \log n).$$

After summing all the parts of the algorithm we analyzed, we find that the runtime of the algorithm, including the Pre-Processor, is

$$O((|W| + |D| + |E| + n_s|C|)(|A| + \log n) + (|C| + n_s)(n_s + \log n)).$$

In terms of space, the largest potential requirement is the event queue, so the space required is $O(n)$. We can upper bound n by the total number of events, which is $O(|W| + |D| + |E| + n_s|C|)$.

Chapter 4

Flexible Factory Concept

4.1 Overview

The flexible factory layout aims to reduce the cost of errors. In the conventional factory layout, an error causes the line to back up, as the car experiencing the error will impede the progress of the cars behind it. Each station can only be occupied by one car at a time, and there is no way for the cars behind it to overtake it. The new layout attempts to solve this problem by decoupling tasks from specific stations and physical locations. Specifically, cars are placed on mobile platforms that carry all the tools and resources necessary for all the tasks in the band. When a new car enters the band, it is placed on such a platform and assigned the required number of workers if they are available. It then begins moving through the band, as the assigned workers perform the tasks. The car does not need to be at a specific station or location for any tasks. Figure 4-1 shows an illustration of a segment of a band in this layout.

One might wonder why the car should move at all if it has everything it needs on its own platform. The reason is that by the time the car finishes its tasks for the band, it will need to have traveled to the entrance of the next band. To handle errors in this layout, there are parking spots next to the assembly line that cars can park at if they experience errors. This allows cars from behind to overtake, so the delay caused by the error only affects the car that experienced it.

Our goal in this chapter is to develop both a simulation of this factory layout and

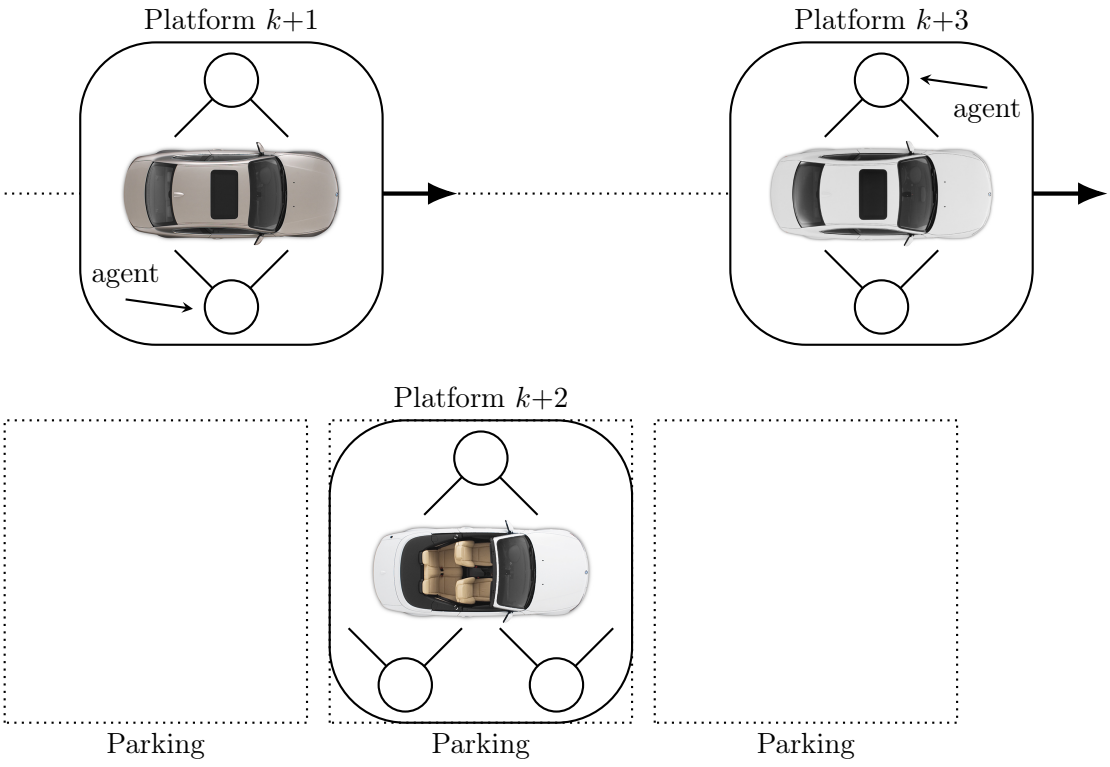

Figure 4-1: Flexible Automotive Assembly Layout

a scheduler that will assign agents to cars and tasks. The simulation will be used to evaluate and compare this layout to the conventional layout. The scheduler need not be optimal, but should perform reasonably well in the common case.

4.2 Definitions

Task $w < c_w, d_w, n_w, ts_w, te_w >$

As was the case for the conventional factory layout, the amount of time a task takes to complete is an attribute of the task and not dependent on the time at which the task is executed or the set of agents executing it. In this factory layout, all agents in a band can perform the same tasks. We define the set of all tasks W. A task $w \in W$ is characterized by the following traits:

- the car c_w for which it needs to be executed

- the duration d_w that it takes to complete
- the number of agents n_w that are required to execute it
- the time ts_w at which it begins being executed. Note that ts_w is unspecified until the task is scheduled.
- the time te_w at which it is finished. Note that te_w is unspecified until the task is scheduled. When ts_w has been specified, te_w is given by

$$te_w = ts_w + d_w.$$

Temporal Constraint $\delta_{l_{ij}} \leq t_j - t_i \leq \delta_{u_{ij}}$

The temporal constraints we encounter in this model are the same ones discussed in Section 3.2: Delay constraints, Immediate constraints, and Simultaneous constraints. We define the set of these Temporal Constraints D.

Car $c < t_c, n_c, W_c, D_c >$

We define the set of all cars C. A car $c \in C$ is characterized by the following traits:

- the time t_c at which it arrives at the start of the band
- the number of agents n_c that need to be assigned to it
- the set of tasks W_c that need to be performed on it
- the set of temporal constraints D_c between its tasks.

Agent a

Agents are the units that perform the tasks. Unlike in the Conventional Layout, all agents in a band are capable of performing all the tasks in the band. Sets of agents are assigned to cars at the beginning of the band, perform all the tasks on their cars

based on a schedule provided, and return to the beginning of the band to be assigned again once the car they were assigned to leaves the band.

Band

The assembly line is separated into linear sections which we call bands. Each band performs different kinds of tasks according to what agents, tools, and resources it has available. We will simulate the events that occur on a single band.

Makespan m_c

The *makespan* m_c of a car c is the time difference between the time when its first task begins being executed and the time its last task is completed. The makespan is a result of the scheduling process.

Parking Spot p

A parking spot is a space next to the line at which the mobile platforms can park. Each spot can be occupied by at most one car at a time. Each parking spot is only accessible from some areas of the band. We are given a function that given a location x on the band, returns a set of parking spots $f(x) \to P$ that are accessible from x.

Error $e < c_e, g_e, ts_e, d_e, te_e, o_e >$

Any event that causes a car's makespan or travel time to increase is an error. Unlike in the Conventional Layout, there are different groups of errors that affect the line in different ways. The grouping resulted from discussions with an automotive manufacturer and is based on errors occurring in real assembly lines. In our model, for the sake of simplicity, each car can only experience one error at a time[1]. We define the set of all errors E. An error $e \in E$ is characterized by the following traits:

[1]In a real factory, cars can experience multiple errors at the same time, but errors are rare enough that this almost never happens. It would not be worth it for us to complicate our model to account for this extremely rare occurrence.

- the car c_e it occurs to
- the group g_e it belongs to
- the time ts_e at which it occurs. Note that ts_e is initially unspecified and is determined by our simulation.
- the duration d_e it takes to resolve. The duration has a slightly different meaning for different groups of errors. For errors of Group 2 ($g_e = 2$), the duration is the time difference between the car parking and the error being resolved. For all other groups ($g_e \neq 2$), the duration is the time difference between the error occurring and it being resolved.
- the time te_e at which the error is resolved. Note that te_e is unspecified until a parking time is specified for errors of Group 2 or ts_e is specified for all other groups.
- the time offset o_e after the resolution of the previous error on car c_e at which e occurs[2]. The time offset is used in the simulation to specify the error's start time. The reason we specify the time at which an error occurs based on the time at which the previous error on the same car is resolved is to ensure the property that there is no overlap of errors on the same car. The process of randomly generating errors that do not overlap becomes simpler when the only condition that must be met is that the time offsets must be non-negative.

Depending on its group, an error may display one or more of the following properties:

Requires Immediate Parking: The car experiencing the error needs to park immediately, and the countdown for the error's resolution doesn't start until it has parked. If the car is not able to park, it and all the cars behind it in the band stop moving until parking becomes available. This means that every position in the band should have access to at least one parking spot, since otherwise the

[2]For the first error occurring on a car, the time offset refers to the amount of time after the car enters the band after which the error occurs.

entire band would permanently stop moving if this kind of error occurred on a car located at a position with no access to parking.

Requires Agent to Fix: One of the agents assigned to the car must fix the error, and will thus be unavailable to perform tasks for the duration of the error.

Causes Blocked Task: One of the tasks that was being performed when the error occurred becomes blocked for the duration of the error, which means that progress towards that task is lost and that it cannot be performed until after the error is resolved.

Can Move Between Parking Spots: After the car experiencing the error has parked, it can re-enter the line before the error is resolved and park later, if a higher priority error needs its parking spot.

Increases Makespan By Duration of Error: The car's makespan increases due to re-work that needs to occur.

Table 4.1 shows the properties that correspond to each error group used in our model.

	Group 1	Group 2	Group 4	Group 7	Group 8	Group 9
Requires Immediate Parking		•				
Requires Agent to Fix	•		•			
Causes Blocked Task			•		•	•
Can Move Between Parking Spots	•			•	•	•
Increases Makespan By Duration of Error	•			•		

Table 4.1: Symptoms of the different groups of errors in our model

Log Entry l

Our simulation must output a series of events that occur as a result of the scheduling and errors. It does this by keeping a log of all the events it needs to output. The advantage of recording a log is that we can use it to reconstruct intermediate states for the band and also filter it to view only certain kinds of entries or only entries regarding certain types of cars if we want to. The types of events that are recorded in the log are the following:

Assignment of agents to a car: When a car is assigned agents, it begins moving in the band, and its agents begin executing its tasks. These entries l_a are characterized by:

- the time t_{l_a} of the assignment
- the car c_{l_a} which was assigned the agents
- the set of agents A_{l_a} that were assigned

A car finishing its tasks: These entries l_t are characterized by:

- the time t_{l_t} at which the tasks are finished
- the car c_{l_t} whose tasks are finished

Occurrence of an error: These entries l_e are characterized by:

- the time t_{l_e} at which the error occurs
- the car c_{l_e} on which it occurs
- the group g_{l_e} that the error belongs to
- the duration d_{l_e} of the error

Resolution of an error: These entries l_r are characterized by:

- the time t_{l_r} at which the error is resolved
- the car c_{l_r} on which it had occurred

A car parking: These entries l_p are characterized by:

- the time t_{l_p} at which the car parks
- the car c_{l_p} that is parking
- the parking spot p_{l_p} which is used

A car leaving a parking spot: These entries l_u are characterized by:

- the time t_{l_u} at which the car leaves the parking spot
- the car c_{l_u} that is leaving the parking spot
- the parking spot p_{l_u} that is being left

A car leaving the band: When a car reaches the end of the band, it leaves and the agents assigned to it move back to the beginning of the band. These entries l_b are characterized by:

- the time t_{l_b} at which the car leaves the band
- the car c_{l_b} that is leaving the band

For two entries l_1, l_2, when we write $l_1 < l_2$, we mean that l_1 appears before l_2 in the log, which, since the log is in chronological order, means that $t_{l_1} \leq t_{l_1}$.

Individual Car Schedule S_c

Besides the high level events that are recorded in the log, our system must also produce a schedule for each individual car. Each schedule S_c for a car c consists of:

- an ordering O_c of the car's tasks that matches the chronological order in which they were performed
- a mapping $M_t \colon w \to ts_w$ from each task to the time at which it was performed
- a mapping $M_A \colon w \to A_w$ from each task to the set of agents that performed it.

4.3 Problem Formulation

Inputs

The inputs to our simulation are the following:

- the set C of cars
  - the union of W_c over all $c \in C$ is the set W of tasks
  - the union of D_c over all $c \in C$ is the set D of temporal constraints
- the speed v of the cars. All cars that are moving in the band move at the same speed. This means that if we know how long a car has been moving in the band, we can tell what its position is.
- the number n_a of agents in the band
- the time r it takes agents to move back to the start of the band. When a car exits the band, the agents assigned to it get off and move back to the start of the band. Once they reach the start of the band, they are available to be assigned to another car.
- the length $x_{\max}$ of the line
- the mapping $f: x \to P$ from location along the line to a set of accessible parking spots
- the set E of errors that will occur

Outputs

The outputs from the simulation are the following:

- a mapping $M_s: c \to S_c$ from each car to its schedule
- the log L, which includes an entry l for every recorded event in chronological order. We also define the subsets of L according to the Log Entry types:

- L_a, the set of entries about the assignment of agents to a car
- L_t, the set of entries about a car finishing its tasks
- L_e, the set of entries about the occurrence of an error
- L_r, the set of entries about the resolution of an error
- L_p, the set of entries about a car parking
- L_u, the set of entries about a car leaving a parking spot
- L_b, the set of entries about a car leaving the band

Input-Output Constraints

Certain relations must exist between our input and output as well as between different parts of our output for our simulation to be true to our model for the Flexible Layout. They are listed below, along with formulations using quantities defined above:

- All tasks are scheduled.

$$\exists ts_w, \forall w \in W$$

- Each car is assigned agents at least once.

$$\forall c \in C: \exists l_a \in L_a \text{ with } c_{l_a} = c$$

- Each car is assigned agents at most once. Along with the previous constraint, this ensures that each car is assigned agents exactly once.

$$c_{l_{a1}} \neq c_{l_{a2}}, \forall l_{a1} \neq l_{a2} \in L_a$$

- Each car is assigned agents after it has entered the band.

$$\forall l \in L_a: t_{l_a} \geq t_{c_{l_a}}$$

- Each car leaves the band at least once.

$$\forall c \in C: \exists l_b \in L_b \text{ with } c_{l_b} = c$$

- Each car leaves the band at most once. Along with the previous constraint, this ensures that each car leaves the band exactly once.

$$c_{l_{b1}} \neq c_{l_{b2}}, \forall l_{b1} \neq l_{b2} \in L_b$$

- A schedule is generated for every car.

$$\forall c \in C: \exists S_c \in M_s$$

- Every task is scheduled.

$$\forall c \in C: \forall w \in W_c: \exists ts_w \in M_t \text{ and } \exists A_w \in M_A \text{ in } S_c$$

- Every task is allocated enough agents.

$$\forall w \in W: |A_w| \geq n_w$$

- Each individual car schedule must include at most as many agents as were assigned to the car.

$$\forall l \in L_a: |A_l| \geq |A_{S_{c_l}}|$$

- Each car should be assigned as many agents as required.

$$\forall l \in L_a: |A_l| = n_{c_l}$$

- No agent should be assigned to more than one car at once.

$$\forall l_1 < l_2 \in L_a \colon A_{l_1} \cap A_{l_2} \neq \emptyset \Rightarrow \exists l \in L_b \colon l < l_2 \text{ and } c_l = c_{l_1}$$

- No agent should be assigned to more than one task at once.

$$\forall S_c \in M_s \colon \forall w_1, w_2 \in W_c \colon A_{w_1} \cap A_{w_2} \neq \emptyset \Rightarrow te_{w_1} \leq ts_{w_2} \text{ or } te_{w_2} \leq ts_{w_1}$$

- Each car finishes its tasks after the makespan has elapsed since it was assigned agents.

$$\forall l_a \in L_a, \forall l_t \in L_t \colon c_{l_a} = c_{l_t} = c \Rightarrow t_{l_t} = t_{l_a} + m_c$$

- Each car leaves the band after it has traversed it.

$$\forall l_a \in L_a, \forall l_b \in L_b \colon c_{l_a} = c_{l_b} = c \Rightarrow t_{l_b} \geq t_{l_a} + x_{max}/v$$

  If we define the amount of time car c was parked t_p, and the amount of time it was immobilized by a car ahead of it in the band stopping t_s, then

$$t_{l_b} = t_{l_a} + t_p + t_s + x_{max}/v$$

- Each car leaves the band after it has finished its tasks.

$$\forall l_t \in L_t, l_b \in L_b \colon c_{l_t} = c_{l_b} \Rightarrow l_t < l_b$$

- Each car can experience at most one error at a time. Another way to say this is that there will be an error resolution entry for a car in the log between every pair of error occurrence entries for that car.

$$\forall l_1 < l_2 \in L_e \colon c_{l_1} = c_{l_2} = c \Rightarrow \exists l \in L_r \colon c_l = c \text{ and } l_1 < l < l_2$$

- All errors are resolved. We have already established this for all errors except the last one for each car in the previous constraint. To establish it for the last error of each car, we add

$$\forall l_e \in L_e \colon \exists l_r \in L_r \colon c_{l_e} = c_{l_r} \text{ and } l_e < l_r$$

- Each error should resolve only once and a car should experience an error before an error is resolved for that car. We establish this with two equations very similar to the last two constraints.

$$\forall l_1 < l_2 \in L_r \colon c_{l_1} = c_{l_2} = c \Rightarrow \exists l \in L_e \colon c_l = c \text{ and } l_1 < l < l_2$$

$$\forall l_r \in L_r \colon \exists l_e \in L_e \colon c_{l_e} = c_{l_r} \text{ and } l_e < l_r$$

- All errors on a car are resolved before it exits the band.

$$\forall l_r \in L_r, l_b \in L_b \colon c_{l_r} = c_{l_b} \Rightarrow l_r < l_b$$

- All Temporal Constraints must be satisfied.

$$\delta_{l_{ij}} \leq t_j - t_i \leq \delta_{u_{ij}}, \forall t_i, t_j \text{ bound by a constraint in } D$$

- Cars must exhibit symptoms of the errors they occur. An error e must be resolved d_e after it has occurred if $g_e \neq 2$, or d_e after its car has parked if $g_e = 2$. An error belonging to Group 2 will immobilize the car experiencing it and all cars behind it in the band until that car is able to park. A car experiencing an error of Group 2 or Group 4 will not be able to leave its parking spot until the error has been resolved. A car experiencing an error of Group 1 or Group 7 will need to spend additional time on its tasks equaling the error's duration. Errors of Group 1 and Group 4 require one of the agents assigned to the car to fix them, which means that a new schedule must be generated for the car to

account for the agent's absence for the duration of the error. Errors of groups 4, 8, and 9 will cause a task that was being performed when the error occurred to be blocked[3] for the duration of the error.

- In order for our scheduler to be realistic, we must make sure that it doesn't use information that wouldn't be available to it in reality. Specifically, even though errors are an input to our simulation, we should only process them when we reach the time at which they occur in the simulation. In other words, our scheduler should not be able to take future errors into account.

4.4 System Overview

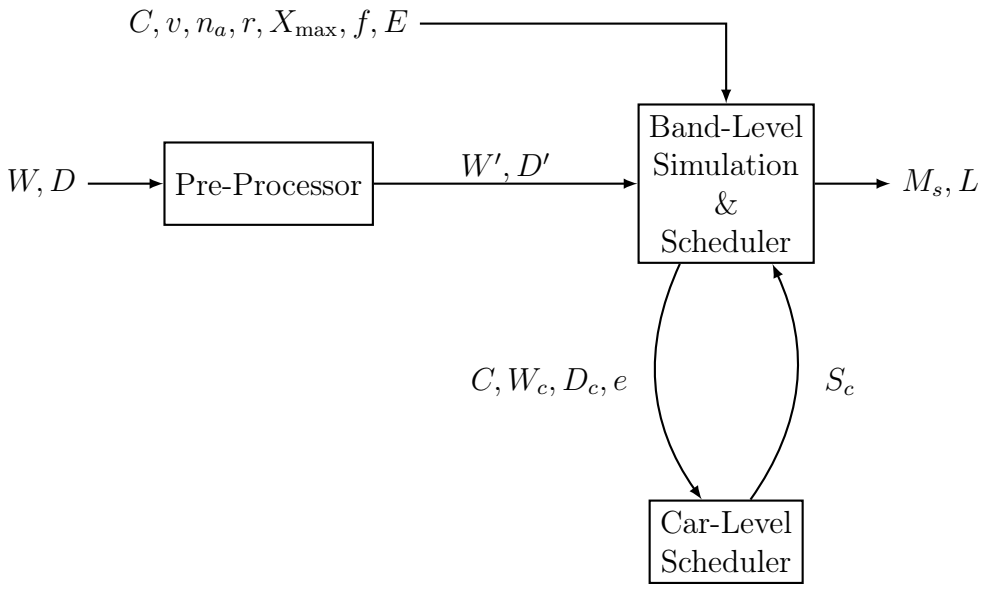

Figure 4-2: System Overview

The system we develop in this chapter has three main components: the Pre-Processor, the Car-Level Scheduler, and the Band-Level Simulation-Scheduler. The Pre-Processor applies some approximations that simplify the scheduling problem, allowing us to use a much faster scheduling algorithm in the Car-Level Scheduler. The

[3] A task being blocked for a period of time means that it can't be performed during that period of time. Any progress that had been made on the task at the time when it was blocked is lost. Blocking a task also blocks all tasks that have it as a precedence constraint.

Car-Level Scheduler produces a schedule for each individual car. These schedules are unaffected and thus can be produced independently of the rest of the simulation, with the exception of some types of errors. If an error disrupts the schedule of a car, the Car-Level Scheduler produces a new schedule that accounts for it. The Band-Level Simulation and Scheduler can be conceptually thought of as two different entities that perform different tasks, but for convenience and efficiency we implemented them in one algorithm. The Band-Level Simulation keeps track of the state of the band, including the locations of the cars and agents, the errors that occur, and the use of parking spots. The Band-Level Scheduler makes real-time decisions about parking in response to errors and the assignment of agents to cars. Although the Band-Level Simulation has access to more information, the Band-Level and Car-Level schedulers only use information about the cars and the present state of the band, information that would also be available to a real online scheduler, which is what our schedulers are emulating. Figure 4-2 shows the inputs and outputs of each component.

4.5 Pre-Processor

The Pre-Processor in this chapter is very similar to the one described in Section 3.5. It makes the same approximation of merging tasks bound by Simultaneous constraints followed by merging tasks bound by Immediate constraints. The merging is slightly different because the model for tasks we use in this chapter is different from the one in Chapter 3, as tasks specify the number of agents that must execute them rather than a specific set of agents. When merging two tasks u and v bound by a simultaneous constraint, the number of agents required for the merged task w is $n_w = n_u + n_v$. When merging two tasks u and v bound by an immediate constraint, the number of tasks required for the merged task w is $n_w = \max(n_u, n_v)$. Unlike in the case of the Conventional Layout, there are no exceptions to the merging. To summarize the Pre-Processor's function, it takes in the set of tasks W and the set of temporal constraints D, merges over Simultaneous and Immediate constraints in that order, and outputs the set of merged tasks W' and the set of reduced constraints D'.

4.6 Band-Level Simulation and Scheduler

The Band-Level Simulation and Scheduler keeps track of the state of the band, makes decisions about parking and assignment of agents to cars, and calls the Car-Level Scheduler to generate schedules for the completion of tasks on each car.

4.6.1 Approach: Discrete Event Simulation

Our approach is to use Discrete Event Simulation, discussed in Section 3.6.1. In this model, the events that cause the state of the band to change are different from those in Chapter 3, so we use different events in our simulation as well. The types of events that we use in our simulation are the following:

Agent becomes available (AE): An agent becomes available when it returns to the beginning of the band after the car it was last assigned to exits the band. An event $z \in AE$ of agent a_z becoming available occurs at time t_z if:

$$\exists c \in C\colon a_z \in A_c \text{ and } \exists l_b\colon c_{l_b} = c \text{ and } t_z = t_{l_b} + r$$

Car enters the band (EBE): An event $z \in EBE$ of car c_z entering the band occurs at time t_z if:

$$t_{c_z} = t_z$$

Car finishes tasks (TE): When the agents assigned to a car have spent an amount of time working on it equal to its makespan, the car's tasks are completed. An event $z \in TE$ occurs at time t_z if that is the time at which c_z finishes its tasks.

Car leaves the band (LBE): When a car reaches the end of the band, it leaves.

Error occurs (ESE): An error occurs at the time it is assigned by the simulation. An event $z \in ESE$ of an error e occurring occurs at time t_z if:

$$t_z = ts_e$$

Error is resolved (EEE): An error is resolved after its duration has elapsed since its occurrence. An event $z \in EEE$ of an error e being resolved occurs at time t_z if:

$$t_z = te_e$$

Parking check (PE): When a car experiences an error that requires it to park or when the time it requires to finish its tasks exceeds the time it requires to reach the end of the band, it tries to park.

Leave parking check (LPE): After a car has parked long enough for its error to resolve or for its time required to reach the end of the band to be greater than or equal to its time required to finish its tasks, it leaves its parking spot.

4.6.2 Simulation without Errors

In order to explain how our simulation works, we will first explain how it would work if there were no errors before explaining it fully in the next section. In a world with no errors, there is also no reason to park, so the only relevant events to place in the queue are an agent becoming available (AE), a car entering the band (EBE), a car finishing its tasks (TE), and a car exiting the band (LBE).

The inputs to the SimulateNoErrors algorithm (Algorithm 4) are the Pre-Processor's outputs, W' and D', as well as the inputs to the system C, v, n_a, r, and $x_{\max}$, discussed in Section 4.3. The outputs of SimulateNoErrors are M_s and L, also discussed in Section 4.3.

The data structures used in SimulateNoErrors are initialized in the InitializeNoErrors procedure (Algorithm 5, line 2). At the beginning of our simulation, all agents are available, there are no cars in the band, and the only events in the queue are those for each of the car arrivals (lines 3–6). At all times we keep track of the set of available agents (AA, initialized in line 7), the agents assigned to each car (CA, initialized in line 8), the makespan of each car (MC, initialized in lines 9–13), a FIFO[4] queue of the cars waiting to be assigned agents (AC, initialized in line 14),

[4] First In First Out; the cars are put into a FIFO queue to make sure that cars that arrive earlier

Algorithm 4 Flexible Layout Simulation in the Absence of Errors

```
1: procedure SimulateNoErrors(C, W', D', v, n_a, r, x_max)
2:     InitializeNoErrors
3:     while d < |C| do
4:         z ← Queue.remove()
5:         time ← t_z
6:         HandleEventNoErrors(z)
7:         while |AA| > 0 and |AC| > 0 do
8:             c ← AC.first()
9:             if |AA| < n_c then
10:                break
11:            else
12:                AC.removeFirst()
13:                A_c ← AA.remove(n_c)
14:                CA[c] ← A_c
15:                log l_a entry for assignment of A_c to c at time
16:                add TE event for c at time + MC[c]
17:                add LBE event for c at time + t_b
18:            end if
19:        end while
20:    end while
21:    return M_s, L
22: end procedure
```

Algorithm 5 Flexible Layout Simulation Initialization in the Absence of Errors

```
1: procedure InitializeNoErrors
2:     initialize M_s and L to be empty
3:     create empty Queue
4:     for c ∈ C do
5:         add EBE event for c at t_c
6:     end for
7:     AA initialized to contain n_a agents
8:     CA initialized as empty array of size |C|
9:     MC initialized as empty array of size |C|
10:    for c ∈ C do
11:        M_s[c] ← CarScheduler.Schedule(W'_c, D'_c, n_c)
12:        MC[c] ← M_s[c].makespan
13:    end for
14:    AC initialized as empty FIFO
15:    d ← 0
16:    t_b ← x_max/v
17: end procedure
```

are assigned agents earlier.

Algorithm 6 Flexible Layout Event Handler in the Absence of Errors

```
 1: procedure HandleEventNoErrors(z)
 2:     if z ∈ AE then
 3:         AA.add(a_z)
 4:     else if z ∈ EBE then
 5:         AC.add(c_z)
 6:     else if z ∈ TE then
 7:         log l_t entry for completion of tasks of c_z at time
 8:     else if z ∈ LBE then
 9:         log l_b entry for car c_z at time
10:         d++
11:         for a ∈ CA[c_z] do
12:             add AE for a at time + r
13:         end for
14:     end if
15: end procedure
```

and the number of cars that have left the band (d, initialized in line 15). We also define the time required to traverse the band $t_b = x_{\max}/v$ (line 16). The Car-Level Scheduler is called to generate the schedules in M_s (line 11).

The simulation runs the following loop until all cars in C have left the band. In each loop, we first process the earliest event from the queue (line 6). Then we try to assign agents to cars until there are no agents available or no cars waiting (lines 7–19). For each car we assign agents to, we remove it from AC (line 12), remove the assigned agents from the pool of available agents (line 13), log the event (line 15), and create events for that car finishing its tasks[5] (line 16) and reaching the end of the band[6] (line 17).

Events are processed by the HandleEvent procedure (Algorithm 6). Different types of events are processed in different ways:

- AE events add their agent to the set of available agents (line 3).
- EBE events add their car to the queue of available cars (line 5).
- TE events add an entry to the log about their car finishing its tasks (line 7).

[5] The timing of this event is given by the makespan of the car's internal schedule.

[6] The timing of this event is given by the time it takes to traverse the band, which is the same for all cars and is equal to $x_{\max}/v$.

- LBE events add an entry to the log (line 9), increment the counter of cars that have left the band (line 10), and add AE events for the agents that were assigned to the car becoming available again (lines 11–13).

4.6.3 Simulation with Errors

Errors can disrupt the model described above in three main ways. First, they can change a car's makespan by blocking a task or occupying an agent. Second, they can cause the car to park, either because the error's resolution requires it or because the increased makespan would cause the car to reach the end of the band before finishing its tasks if it didn't park. Third, if a car experiences an error that requires immediate parking, but is not able to park, it stops moving, as do all the cars behind it in the line. When disturbances like these occur, the simulation's previous prediction for the timing of TE and LBE events is no longer accurate, which makes the events that are in the queue corresponding to those predictions stale. We could remove these stale events from the queue, but it is more efficient computationally to instead remember that they are stale and ignore them when they are processed. The disturbances always result in the events happening later, not earlier, so we know that we will process all the stale events before we process the valid event. Thus, it is sufficient to keep counters of how many stale events of each type are in the queue. When an error makes a prediction obsolete, the corresponding counter is incremented. When a TE or LBE event is processed, if the counter is positive, it means we just removed a stale event from the queue, so the corresponding counter is decremented and the event is ignored. If the counter is zero, it means the event we are processing is accurate, so we log it. Also, since there will be parking events, the time spent working on tasks will not necessarily match the time spent moving in the band, so we will need to keep track of the progress of each car in those two areas separately.

The inputs to the SIMULATE algorithm (Algorithm 7) are the Pre-Processor's outputs, W' and D', as well as the inputs to the system $C, v, n_a, r, x_{\max}, f$, and E, discussed in Section 4.3. The outputs of SIMULATE are M_s and L, also discussed in Section 4.3.

Algorithm 7 Flexible Layout Simulation with Errors

```
1: procedure SIMULATE(C, W', D', v, n_a, r, x_max, f, E)
2:     INITIALIZE                                          ▷ Differs with Errors
3:     while d < |C| do
4:         z ← Queue.remove()
5:         time ← t_z
6:         HANDLEEVENT(z)
7:         while |AA| > 0 and |AC| > 0 do
8:             c ← AC.first()
9:             if |AA| < n_c then
10:                break
11:            else
12:                AC.removeFirst()
13:                A_c ← AA.remove(n_c)
14:                CA[c] ← A_c
15:                UT[c] ← time                            ▷ Differs with Errors
16:                log l_a entry for assignment of A_c to c at time
17:                add TE event for c at time + MC[c]
18:                if |FC| = 0 then                        ▷ Differs with Errors
19:                    add LBE event for c at time + t_b
20:                end if                                  ▷ Differs with Errors
21:                if XC[c] ≠ ∅ then                       ▷ Differs with Errors
22:                    e ← XC[c].remove()                  ▷ Differs with Errors
23:                    add ESE event for e at time + o_e   ▷ Differs with Errors
24:                end if                                  ▷ Differs with Errors
25:            end if
26:        end while
27:    end while
28:    return M_s, L
29: end procedure
```

The data structures used in SIMULATE are initialized in the INITIALIZE procedure (Algorithm 8, line 2). SIMULATE keeps track of all the data structures that SIMULATENOERRORS keeps track of (line 2). It additionally keeps track of the following:

- the time each car has spent moving in the band (MP, initialized in line 3)
- the time each car has spent working on its tasks (TP, initialized in line 4)
- the number of stale LBE events for each car in the queue (IM, initialized in line 5)

Algorithm 8 Flexible Layout Simulation Initialization with Errors

```
1: procedure INITIALIZE
2:     INITIALIZENOERROR
3:     initialize MP as array of zeroes with size |C|
4:     initialize TP as array of zeroes with size |C|
5:     initialize IM as array of zeroes with size |C|
6:     initialize IT as array of zeroes with size |C|
7:     initialize ET as array of zeroes with size |C|
8:     initialize UT as array of -1's with size |C|
9:     initialize MD as array of false with size |C|
10:    initialize EC as empty map
11:    initialize PC as empty map
12:    initialize CP as empty map
13:    initialize FC as ∅
14:    create array of lists XC ▷ |C| different lists, XC[n_c] holds the sequence of all
       errors that will occur on car c
15:    for e ∈ E do
16:        XC[c].add(e)
17:    end for
18: end procedure
```

Algorithm 9 Flexible Layout Event Handler with Errors

```
1: procedure HANDLEEVENT(z)
2:     if z ∈ AE ∪ EBE then
3:         HANDLEEVENTNOERRORS(z)
4:     else if z ∈ TE then
5:         HANDLEEVENTTE(z)
6:     else if z ∈ LBE then
7:         HANDLEEVENTLBE(z)
8:     else if z ∈ ESE then
9:         HANDLEEVENTESE(z)
10:    else if z ∈ EEE then
11:        HANDLEEVENTEEE(z)
12:    else if z ∈ PE then
13:        HANDLEEVENTPE(z)
14:    else if z ∈ LPE then
15:        HANDLEEVENTLPE(z)
16:    end if
17: end procedure
```

- the number of stale TE events for each car in the queue (IT, initialized in line 6)

Algorithm 10 Flexible Layout TE Event Handler

```
1: procedure HandleEventTE(z)
2:     if IT[c_z] > 0 then
3:         IT[c_z] − −
4:     else
5:         log l_t entry for completion of tasks of c_z at time
6:     end if
7: end procedure
```

Algorithm 11 Flexible Layout LBE Event Handler

```
1: procedure HandleEventLBE(z)
2:     if IM[c_z] > 0 then
3:         IM[c_z] − −
4:     else
5:         log l_b entry for car c_z at time
6:         d + +
7:         for a ∈ CA[c_z] do
8:             add AE for a at time + r
9:         end for
10:        MD[c_z] ← true
11:    end if
12: end procedure
```

- the time each car has left before the error it is currently experiencing is resolved (ET, initialized in line 7). If the car isn't experiencing an error, its entry in ET is zero.

- the time at which each car last updated its entries in MP, TP, and ET (UT, initialized in line 8)

- a boolean for each car indicating whether that car has left the band (MD, initialized in line 9)

- a map from each car experiencing an error to the error it is experiencing (EC, initialized in line 10)

- a map from each occupied parking spot to the car occupying it (PC, initialized in line 11)

Algorithm 12 Flexible Layout ESE Event Handler

```
1: procedure HandleEventESE(z)
2:     Update(c_z)
3:     log l_e event for e_z on c_z at time
4:     EC[c_z] ← e_z
5:     if ReschedulingRequired(e_z) then
6:         M_s[c_z] ← CarScheduler.Reschedule(W'_c, D'_c, n_c, M_s[c_z], e)
7:         MC[c_z] ← M_s[c_z].makespan
8:         IT[c_z] + +
9:         add TE event for c_z at time + MC[c_z] − TP[c_z]
10:    end if
11:    if ImmediateParking(e_z) then
12:        add PE event for c_z at time
13:    else
14:        add EEE event for c_z at time + d_{e_z}
15:        if ShouldPark(c_z) then
16:            add PE event for c_z at time
17:        end if
18:    end if
19: end procedure
```

Algorithm 13 Flexible Layout EEE Event Handler

```
1: procedure HandleEventEEE(z)
2:     Update(c_z)
3:     log l_r event for c_z at time
4:     if XC[c_z] ≠ ∅ then
5:         e ← XC[c_z].remove()
6:         add ESE event for e at time + o_e
7:     end if
8:     EC.remove(c_z)
9: end procedure
```

- a map from each parked car to the parking spot at which it is parked (CP, initialized in line 12)

- the set of all cars that are blocking the progress of cars behind them because they are experiencing an error that requires them to park, but they cannot find parking (FC, initialized in line 13). Since each car in FC blocks all cars behind it, the only car that matters is the one that has progressed the furthest because all the cars that are blocked by the other cars are also blocked by it. Thus,

Algorithm 14 Flexible Layout PE Event Handler

```
1: procedure HandleEventPE(z)
2:     Update(c_z)
3:     P ← f(MP[c_z]v)
4:     for p ∈ P do
5:         if p ∉ PC then
6:             PC[p] ← c_z
7:             CP[c_z] ← p
8:             if not Blocked(c_z) then
9:                 IM[c_z] + +
10:            end if
11:            log l_p entry for c_z parking at p
12:            if c_z ∈ EC and ImmediateParking(c_z) then
13:                add EEE event for c_z at time + d_{EC[c_z]}
14:            end if
15:            add LPE event for c_z to queue
16:            return
17:        end if
18:    end for
19:    if c_z ∉ EC or not ImmediateParking(c_z) then
20:        add new PE event for c_z later
21:        return
22:    end if
23:    for p ∈ P do
24:        if CanRelocate(PC[p]) then
25:            Relocate(c_z, PC[p], p)
26:            return
27:        end if
28:    end for
29:    initialize OB to be an array of booleans of length |C|
30:    for c ∈ C do
31:        Update(c)
32:        OB[c] ← Blocked(c)
33:    end for
34:    FC.add(c_z)
35:    for c ∈ C do
36:        if not OB[c] and Blocked(c) and c ∉ CP then
37:            IM[c] + +
38:        end if
39:    end for
40: end procedure
```

Algorithm 15 Flexible Layout LPE Event Handler

```
1: procedure HandleEventLPE(z)
2:     Update(c_z)
3:     p ← CP[c_z]
4:     log l_u entry for c_z leaving spot p
5:     PC.remove(p)
6:     CP.remove(c_z)
7:     if not Blocked(c_z) then
8:         add LBE event for c_z
9:     end if
10:    for c ∈ FC do
11:        if p ∈ f(MP[c]v) then
12:            for c' ∈ C do
13:                Update(c')
14:            end for
15:            PC[p] ← c
16:            CP[c] ← p
17:            initialize OB to be an array of booleans of length |C|
18:            for c' ∈ C do
19:                OB[c] ← Blocked(c')
20:            end for
21:            FC.remove(c)
22:            for c' ∈ C do
23:                if OB[c'] and not Blocked(c) and c ∉ CP then
24:                    add LBE event for c'
25:                end if
26:            end for
27:        end if
28:    end for
29: end procedure
```

we find it useful to define FC.furthest() which returns the car in FC with the largest entry in MP.

Simulate mostly differs from SimulateNoErrors in the types of events used and the way events are handled. Besides the event handler (HandleEvent, Algorithm 9, called in line 6) which we will discuss later, there are a few differences in what happens when agents are assigned to a car c. $UT[c]$ is set to *time* (line 15), so that future progress in tasks and position can be calculated based on the time elapsed since the agents were assigned. The LBE event for c is only generated if FC is empty

Algorithm 16 Flexible Layout Simulation Progress Update Function

```
1: procedure UPDATE(c)
2:     if INBAND(c) then
3:         Δt ← time − UT[c]
4:         if c ∉ CP and not BLOCKED(c) then
5:             MP[c] ← MP[c] + Δt
6:         end if
7:         TP[c] ← TP[c] + Δt
8:         if c ∉ FC then
9:             ET[c] ← max(0, ET[c] − Δt)
10:        end if
11:        UT[c] ← time
12:    end if
13: end procedure
```

Algorithm 17 Checks if a car is blocked by another car

```
1: procedure BLOCKED(c)                          ▷ is c blocked by another car?
2:     return INBAND(c) and |FC| > 0 and MP[c] < MP[FC.farthest()]
3: end procedure
```

Algorithm 18 Checks if a car is in the band

```
1: procedure INBAND(c)                           ▷ is c in the band?
2:     return UT[c] ≥ 0 and not MD[c]
3: end procedure
```

Algorithm 19 Checks if a car should part to avoid reaching the end without finishing its tasks and resolving its errors

```
1: procedure SHOULDPARK(c)                       ▷ should c park now?
2:     return t_b − MP[c] < MC[c] − TP[c] or t_b − MP[c] < ET[c]
3: end procedure
```

Algorithm 20 Checks if an error disrupts the car's internal schedule

```
1: procedure RESCHEDULINGREQUIRED(e)
2:     return g_e ∈ {1, 4, 7, 8, 9}
3: end procedure
```

(line 18) because if it is not, it means the cars behind a car in FC can't move, and since c is located at the start of the band, it will be immobilized. If any errors in E are to occur on c, the ESE event for the first is added to the queue (lines 21–24).

Before we describe how events are processed, we will first introduce some helper

Algorithm 21 Checks if a parked car can relocate

```
1: procedure CanRelocate(c)
2:     if c ∉ EC then
3:         return true
4:     else
5:         e ← EC[c]
6:         return g_e ∈ {1, 7, 8, 9}
7:     end if
8: end procedure
```

Algorithm 22 A parked car and a non-parked car switch places

```
1: procedure Relocate(c_i, c_o, p)
2:     CP.remove(c_o)
3:     PC[p] ← c_i
4:     CP[c_i] ← p
5:     if not Blocked(c_i) then
6:         IM[c_i] + +
7:     end if
8:     log l_u entry for c_o leaving spot p
9:     log l_p entry for c_i parking at p
10:    if c_i ∈ EC and ImmediateParking(c_i) then
11:        add EEE event for c_i at time + d_{EC[c_i]}
12:    end if
13:    add LPE event for c_i to queue
14:    add PE event for c_o later
15:    if not Blocked(c_i) then
16:        IM[c_i] + +
17:    end if
18:    if not Blocked(c_o) then
19:        add LBE event for c_o
20:    end if
21: end procedure
```

Algorithm 23 Checks if an error requires the car experiencing it to park immediately

```
1: procedure ImmediateParking(e)
2:     return g_e = 2
3: end procedure
```

functions that we will use.

InBand (Algorithm 18, lines 1–3) takes in a car as input and outputs a boolean indicating whether the car given is in the band. InBand returns true for cars that

have been assigned agents, but have not yet left the band. It recognizes cars that have been assigned agents by the fact that their entries in UT are non-negative. It recognizes cars that have not left the band by the fact that their entries in MD are false.

Blocked (Algorithm 17, lines 1–3) takes in a car as input and outputs a boolean indicating whether the car is immobilized because a car in FC has blocked the line. For a car to be blocked, it has to be in the band, and there has to be a car in FC that is ahead of it.

Update (Algorithm 16, lines 1–13) takes in a car as input and updates its entries in MP, TP, EC, and UT. It first checks if the car is in the band, and only performs the updates if it is. It then calculates the amount of time that has passed since the last update Δt (line 3). Δt is added to the car's entry in TP (line 7), since agents will have been working on its tasks. If the car is not parked and not blocked, Δt is added to its entry in MP (line 5), since it has been moving in the band. If the car is not in FC, then it will have been making progress on an error if it had one, so its entry in ET is set to the maximum of zero and the old entry minus Δt (line 9).

CanRelocate (Algorithm 21, lines 1–8) takes in a parked car and outputs a boolean indicating whether the car is able to leave its parking spot. If the car is not experiencing an error or if it belongs to one the groups 1, 7, 8, 9, then it is able to relocate. Otherwise, it is not.

ReschedulingRequired (Algorithm 20, lines 1–3) takes in an error as input and outputs a boolean indicating whether that error causes a disruption that will require the generation of a new schedule by the Car-Level Scheduler. Errors from groups 1, 4, 7, 8, and 9 exhibit this behavior.

ImmediateParking (Algorithm 23, lines 1–3) takes in an error as input and outputs a boolean indicating whether that error requires immediate parking. If a car experiencing this kind of error is not able to find parking, it will stop moving and prevent all cars behind it in the line from moving until it can find parking. Errors from group 2 exhibit this behavior.

ShouldPark (Algorithm 19, lines 1–3) takes in a car (which is not experiencing

an ImmediateParking error) as input and outputs a boolean indicating whether the car should park. There are two reasons for which the car should park. First, if the time the car needs to finish its tasks exceeds the time it will stay in the band if it doesn't park. Second, if the time the car needs to resolve the error it is currently experiencing (if any) exceeds the time it will stay in the band if it doesn't park. The reason that the car has to park in both these cases is that each car has to finish all its tasks and resolve its errors before exiting the band, as mentioned in Section 4.3.

Relocate (Algorithm 22, lines 1–21) takes in a parked car, the spot at which it is parked, and a non-parked car that has access to that spot, and performs the operations of the parked car leaving the spot and the non-parked car parking there. First, it updates PC and CP accordingly. It adds entries to the log for the parking and leaving parking. It increments the IM counter for the car parking and adds an LBE event for the car leaving if they are not blocked. It adds an EEE event for the car parking, since it was a car that needed parking to be able to resolve its error. It also adds a PE event for the car leaving the parking to try to park once it has moved to an area where different parking spots are accessible.

Events are processed by the HandleEvent procedure (Algorithm algo:flexible-3). AE and EBE events are processed in the same way as the no-error case. The rest of the events are processed by the following procedures:

HandleEventTE (Algorithm 10): TE events add an entry to the log about their car finishing its tasks only if the IT counter is zero (line 5). Otherwise, they are ignored and the counter is decremented (line 3).

HandleEventLBE (Algorithm 11): LBE events check the IM counter. If it is zero, they are processed in the same way as in the no-error case (lines 5–9) and also update MD (line 10). Otherwise, they are ignored and the counter is decremented (line 3).

HandleEventESE (Algorithm 12): ESE events call Update (line 2), add an entry to the log for the error occurring (line 3), and add an entry for their car that points to the error in EC (line 4). If ReschedulingRequired returns

true for the error, the Car-Level Scheduler is called to produce a new schedule (line 6), the car's entry in IT is incremented (line 8), and a new TE event is added for the car, according to the makespan of the new schedule (line 9). If ImmediateParking returns true for the error, a PE event is added for the car at *time* (line 12). If ImmediateParking returns false for the error, a EEE event is added for the car at $time + d_e$ (line 14). The reason we do not generate the EEE event for ImmediateParking errors is that their resolution time depends on when they park, which is not yet determined. Also, if ShouldPark returns true for the car, a PE event is added for the car at *time* (lines 15–new:ESE-shouldp-endif).

HandleEventEEE (Algorithm 13): EEE events call Update (line 2), add an entry to the log for their car resolving its error (line 3), add an ESE event to the queue for the next error their car will encounter if there is one (lines 4–7), and remove their car from EC (line 8).

HandleEventPE (Algorithm 14): PE events first call Update (line 2). Then they iterate over the accessible parking spots until they find a vacant one (lines 3–18). If they find a vacant one, they park the car there by updating PC and CP, incrementing the car's IM counter if the car isn't blocked, and add an entry to the log for the car parking. If the car is experiencing an ImmediateParking error, an EEE event is added to the queue for the error's resolution. A LPE event is also added to the queue. If no vacant spaces are found and the car is not experiencing an ImmediateParking error, then a PE event is added to the queue, and the search for immediate parking stops (lines 19–22). If no vacant spaces are found and the car is experiencing an ImmediateParking error, then we iterate over the accessible parking spots in search of a car that can relocate (lines 23–28). If we find such a car, we make it leave the parking spot and park our car there, by calling the Relocate procedure. Finally, if no accessible spot is found that is occupied by a car that can relocate, it means that the car stops moving, so we add it to FC and increment IM for all the

non-parked cars that become blocked by it (lines 30–39).

HandleEventLPE (Algorithm 15): LPE events call Update (line 2), update CP and PC, and add an entry to the log for the car leaving the parking spot (line 4). If the car is not blocked, a LBE event is added to the queue for it. If a car in FC has access to the newly vacated parking spot, we park it there, updating CP and PC, adding an entry to the log for the car parking, and check if any previously blocked and non-parked cars are no longer blocked, and if so, add LBE events for them to the queue (lines 12–new:LPE-FC-end).

4.7 Car-Level Scheduler

The Car-Level Scheduler generates the schedules for the completion of tasks on each car and modifies those schedules when the occurrence of errors makes them infeasible. Once agents have been assigned to a car, it doesn't share them with other cars nor do its tasks depend on those for other cars. This means that unlike in the Conventional Layout, we can solve the task scheduling problem for each car individually. Note that the schedules generated have their own convention for time and agents. The time is such that the schedule begins at time zero. When viewing such a schedule at the Band-Level, one must add the time at which the agents were assigned to the car to all the times to convert them to Band-Level Simulation Time. One must also map the token agents used in the Car-Level Scheduler to the agents assigned to the car by the Band-Level Scheduler.

4.7.1 Discrete Events

Like the Band-Level Simulation and Scheduler, the Car-Level Scheduler also uses discrete events placed in a queue that always returns the earliest one. It uses two types of events, which are:

Agent becomes available (AE): An agent becomes available when it finishes a task it had been working on. An event $z \in AE$ of agent a_z becoming available

occurs at time t_z if:

$$\exists w \in W' \colon a_z \in A_w \text{ and } t_z = ts_w + d_w$$

Task dependency is resolved (DE): When a Temporal Constraint's delay has elapsed since the completion of its releasing task, the constraint is satisfied and the constrained task will be ready to execute if it has no remaining unsatisfied precedence constraints. An event $z \in DE$ for a constraint between releasing task rw and constrained task cw with delay d occurs at t_z if:

$$t_z = ts_{rw} + d_{rw}$$

4.7.2 Scheduling Algorithm

The SCHEDULE algorithm (Algorithm 24) takes in a car's tasks $W_c{}'$, temporal constraints $D_c{}'$, and number of agents assigned n_c as inputs. Its output is a schedule S_c for the completion of the car's tasks, comprised of the ordering of tasks O_c, the mapping from tasks to time of execution M_t, and the mapping from tasks to the set of agents executing them M_A.

The scheduling algorithm works as follows. Besides the queue, we also keep track of the agents who are not busy (AA, initialized in line 2), the tasks whose dependencies have been satisfied (AT, initialized in line 3), the unsatisfied dependencies of each task (RT, initialized in line 3), and the number of tasks that have been scheduled (dt, initialized in line 8). Until we have scheduled all the tasks, we repeat the following loop, shown in SCHEDULELOOP (Algorithm 25). We process the earliest event in the queue and any other events that occur at the same time (lines 2–11), and then we try to schedule tasks until we have no available agents remaining or no tasks whose dependencies have been met (lines 12–25). When we schedule a task, we create and place in the queue events for the release of each of the agents that were assigned to the task (lines 19–21), as well as for the satisfaction of each dependency on the task (lines 22–24). If we have to choose one of many tasks to schedule, we use a heuristic

that prioritizes tasks that many other tasks depend on. This heuristic is similar to the one used by Tercio and aims to minimize the time agents spend idle by increasing the number of tasks that are ready to be executed at any given time. How we process an event from the queue depends on the type of the event. Events indicating the release of an agent add the agent to the pool of available agents. Events indicating the satisfaction of a dependency remove that dependency from the list of unsatisfied dependencies of the constrained task. If the list becomes empty, it means that the task is ready to execute, so it is added to the pool of ready-to-execute tasks.

Algorithm 24 Individual Car Scheduling

```
1: procedure SCHEDULE(W_c', D_c', n_c)        ▷ initialized to include one no-op event
2:     AA initialized to contain n_c agents
3:     AT initialized to contain all tasks that do not depend on other tasks
4:     RT initialized such that RT[w] gives a list of all tasks that task w depends
   on
5:     CT initialized such that CT[w] gives a list of all tasks that depend on task w
6:     Queue initialized to contain one no-op event at time 0
7:     O_c, M_t, M_A initialized empty
8:     dt ← 0
9:     while dt < |W_c'| do
10:        SCHEDULELOOP
11:    end while
12:    return S_c(O_c, M_t, M_A)
13: end procedure
```

4.7.3 Rescheduling Algorithm

The RESCHEDULE algorithm (Algorithm 26) takes in a car's tasks $W_c{}'$, temporal constraints $D_c{}'$, number of agents assigned n_c, the previous schedule S_c, and the disrupting error e as inputs. Its output is a new schedule $S_c{}'$ that takes into account the disruption caused by the error.

Some types of errors require changes in the internal schedule of the car experiencing them. There are three ways in which this can occur. The first is by a task being blocked[7] for the duration of the error, which is the case for errors from groups 8

[7]Unable to be performed

Algorithm 25 Individual Car Scheduling Loop Procedure

```
1: procedure ScheduleLoop
2:     z ← Queue.remove()
3:     time ← t_z
4:     if z ∈ AE then
5:         AA.add(a_z)
6:     else if z ∈ DE then
7:         RT[cw].remove(rw)
8:         if RT[cw] = ∅ then
9:             AT.add(cw)
10:        end if
11:    end if
12:    while AA ≠ ∅ and AT ≠ ∅ do
13:        w ← AT.getBestHeuristic()
14:        A_w ← AA.remove(n_w)
15:        O_c.add(w)
16:        M_t[w] ← time
17:        M_A[w] ← A_w
18:        dt + +
19:        for a ∈ A_w do
20:            add AA event for a to Queue at time + d_w
21:        end for
22:        for w′ ∈ CT[w] do
23:            add DE event for satisfaction of constraint between w and w′
24:        end for
25:    end while
26: end procedure
```

and 9. The second is by the error requiring an agent to fix it, thus making the agent unavailable for the duration of the error, which is the case for errors from Group 1. The third is a combination of the first two, a blocked task and a busy agent, which is the case for errors from Group 4.

The algorithm for adjusting a car's schedule in response to such an error is as follows. The schedule up until the point in time when the error occurred remains as is, since it has already been executed by the time of the rescheduling. For the tasks that have not yet been executed, we use the same algorithm that we used for the original scheduling with a few modifications to take into account the new constraints.

In the case of a blocked task, we add a fake dependence to it and an event to the queue to remove that dependence after the error is resolved. Because of this

Algorithm 26 Individual Car Rescheduling

1: **procedure** RESCHEDULE($W_c{}', D_c{}', n_c, S_c, e$) ▷ initialized to include one no-op event
2: O_c, M_t, M_A initialized to include all tasks scheduled in S_c before e occurred
3: $time \leftarrow ts_e$
4: $dt \leftarrow |O_c|$
5: AA initialized to contain all agents not busy at the time e occurred
6: AT initialized to contain all tasks that do not depend on tasks that have not been scheduled
7: RT initialized such that $RT[w]$ gives a list of all non-completed tasks that task w depends on
8: CT initialized such that $CT[w]$ gives a list of all tasks that depend on task w
9: $Queue$ initialized to contain the AA and DE events corresponding to the completion of all in-progress events
10: **if** $g_e \in \{1, 4\}$ **then**
11: $a \leftarrow AA$.remove()
12: Add AE event for a at $time + d_e$
13: **end if**
14: **if** $g_e \in \{4, 8, 9\}$ **then**
15: $w_b \leftarrow AT$.removeBlocked(e)
16: $RT[w_b]$.add(w_f) ▷ w_f is a fake task, added as a dependence to prevent w_b from being scheduled
17: add DE event for resolution of w_b's dependence on w_f at $time + d_e$
18: **end if**
19: **while** $dt < |W_c{}'|$ **do** ▷ The loop is exactly the same as the one in SCHEDULE
20: SCHEDULELOOP
21: **end while**
22: **return** $S_c{}'(O_c, M_t, M_A)$
23: **end procedure**

dependence, the task will never be added to the ready-to-execute tasks pool and thus will not be executing for the duration of the error, thus effectively being blocked.

In the case of an error that requires an agent to solve it, we first must decide which agent will solve the error. If there is an idle agent, we select it. If not, we select the agent that has made the least progress on their current task, since that progress will be lost. Once the agent has been selected, we remove it from the pool of available agents and we add an event to the queue that will add it back to the pool after the error is resolved. This way, this agent will not be assigned to any tasks while it is busy fixing the error.

When we have an error that blocks a task and requires an agent to solve it, we select one of the agents that were working on the blocked task to solve the error. Recall that the tasks being blocked were in progress when the error occurred, so the agents that were performing them will be idle after the task is blocked. Beyond that, we use the exact same methods we described above to take into account the blocked task and the busy agent in our schedule.

4.8 Runtime and Memory Analysis

4.8.1 Pre-Processor

The Pre-Processor has to iterate over all temporal constraints to do the merging, and must produce new lists of tasks and constraints, so its runtime and space requirements are $O(|W| + |D|)$.

4.8.2 Car-Level Scheduler

The number of times the ScheduleLoop procedure is called is upper bounded by the number of events processed. In both cases, the amount of AE events is $O(|W_c|)$ and the amount of DE events is $O(|D_c|)$. The work in each loop is $O(\log n + \log m)$, where n is the maximum size of AT and m is the maximum size of the Event Queue. This is because both AT and the Event Queue are implemented as a heaps (such that the head has the highest heuristic value or the earliest event, respectively), so that the add and remove operations are $O(\log n)$ and $O(\log m)$ respectively. The number of available tasks is upper bounded by the total number of tasks, and the size of the Event Queue is upper bounded by the total number of events, so the total runtime of a call to Schedule or Reschedule is

$$O((|W_c| + |D_c|) \log(|W_c| + |D_c|))$$

In terms of memory requirements, AA is $O(n_c)$, AT is $O(|W|)$, RT, CT, and the

Event Queue are $O(|W| + |D|)$, and O_c, M_t, and M_A are $O(|W|)$. Thus, the space requirement for a call to SCHEDULE or RESCHEDULE is $O(|W| + |D|)$.

4.8.3 Band-Level Scheduler and Simulation

The SIMULATE algorithm calls Car-Level Scheduler's SCHEDULE once for every car, and the total runtime of that is

$$O(|C|(|W_c| + |D_c|) \log(|W_c| + |D_c|))$$

The number of times SIMULATE loops is upper-bounded by the number of events processed. There are $O(|C|)$ AE events, since they are generated by cars finishing, and the number of agents per task is upper-bounded by a constant. There are $O(|C|)$ EBE events, since there is only one for each car. There are $O(|C| + |E|)$ TE and LBE events, since there is at most one stale one for every error and one valid one for every car. There are $O(|E|)$ ESE and EEE events because there is only one per error. There are $O(|E|)$ PE and LPE events in the common case because the number of parking checks and relocations caused by each error is upper-bounded by a constant on average.

In each loop, the algorithm removes an event from the Queue, which is $O(\log q)$, where q is the size of the Queue. Then it processes the event. Then it assigns agents to cars. Each time an agent is assigned to a car, the runtime is $O(\log q)$, since events are added to the queue. The total number of times an agent is assigned is $O(|C|)$, since the number of agents assigned to each car is upper-bounded by a constant.

We will now analyze the runtime for processing each event type. AE and EBE events are $O(1)$, since they only involve one operation of adding an element to an unsorted collection. TE events are $O(1)$ because they only involve appending to the log. LBE events are $O(\log q)$, since they involve adding a number of events to the Queue, upper-bounded by a constant. ESE events add events to the queue and call the Car-Level Scheduler, so their runtime is $O((|W_c| + |D_c|) \log(|W_c| + |D_c|) + \log q)$. EEE events are $O(\log q)$ because they add an event to the Queue. PE events iterate

over all accessible parking spots, and then add some events to the queue, or, in the rare case when no parking can be found for an error requiring immediate parking, iterate over all cars. If we assume the number of parking spots accessible from any one location is upper-bounded by a constant, then the runtime for the common case of PE events is $O(\log q)$, and the runtime for the at most $O(|E|)$ instances of a car being added to FC is $O(|C| + \log q)$. LPE events similarly are $O(\log q)$ in the common case and $O(|C| + \log q)$ when a car is removed from FC.

Now we multiply the number of occurrences of each event type with its runtime to get the total contribution. We get $O(|C|)$ for AE events, $O(|C|)$ for EBE events, $O(|C| + |E|)$ for TE events, $O((|C| + |E|) \log q)$ for LBE events, $O(|E|((|W_c| + |D_c|) \log(|W_c| + |D_c|) + \log q))$ for ESE events, $O(|E| \log q)$ for EEE events, and $O(|E|(|C| + \log q))$ for PE and LPE events. Thus, the total runtime of SIMULATE is

$$O((|C| + |E|)((|W_c| + |D_c|) \log(|W_c| + |D_c|) + \log q) + |E|(|C| + \log q))$$

where $q = O(|C| + |E|)$ and $|W_c|$ and $|D_c|$ are the maximum number of tasks and temporal constraints for one car, respectively.

The space requirement is $O(|C| + |E|)$ because none of the data structures used are larger than that.

4.8.4 Overall System

To get the total runtime of the system, we add that of the Pre-Processor and that of SIMULATE to get:

$$O(|W| + |D| + (|C| + |E|)((|W_c| + |D_c|) \log(|W_c| + |D_c|) + \log q) + |E|(|C| + \log q))$$

To get the total space requirement of the system, we add that of the Pre-Processor and that of SIMULATE to get:

$$O(|W| + |D| + |C| + |E|)$$

Chapter 5

Comparison of Automotive Assembly Layouts

In this chapter, we will develop a method to compare the Conventional Automotive Assembly Layout, described in Chapter 3, and the Flexible Automotive Assembly Layout, described in Chapter 4, in terms of their throughput and response to errors. We will make use of the simulations we developed for the two layouts in the previous chapters. Our goal will be to make the comparison as fair as possible and to draw useful conclusions about the merits of the two layouts, which will help determine whether development of technology to enable the Flexible Layout could be worthwhile.

5.1 Layouts

Before we describe our comparison method, let us first review the two layouts we will be comparing. Both are divided into linear segments called bands. Each band has different agents, resources, and tools available, and performs a different class of tasks accordingly. The setup of each band is where the two layouts differ, which is why each of our simulations of the two layouts models a single band.

5.1.1 Conventional Automotive Assembly Layout

In the Conventional Automotive Assembly Layout, the band is comprised of a moving belt. All cars are placed on the belt and move together. During normal operation, the belt moves forward at a constant speed. As the cars move, they pass without stopping by successive stations, in each of which specific tasks are performed. The tasks are performed by agents agents situated in each station by the side of the belt. When an error occurs, preventing the agents from completing their tasks on a car before it leaves the station limits, the belt stops. The belt remains stationary until the error is resolved. As a result, an error that occurs on a single car also delays every other car in the band.

Recall from Section 3.3 the inputs to our simulation of the Conventional Layout:

- the set C of cars
- the set A of agents
- the number n_s of stations
- the cycle time t_c
- the set E of errors that will occur

5.1.2 Flexible Automotive Assembly Layout

In the Flexible Automotive Assembly Layout, the cars are placed on mobile platforms that carry all the required resources for the tasks in the band. Agents get on platforms at the beginning of the band and stay on them until they reach the end, performing all the tasks in the band for the car on the platform. During normal operation, the platforms move through the band at a constant speed. When an error occurs, the affected car's platform can move to a parking spot at the side, thus allowing other platforms to overtake it. The main motivation for this layout is that an error only delays the affected car.

Recall from Section 4.3 the inputs to our simulation of the Flexible Layout:

- the set C of cars
- the speed v of the cars
- the number a of agents in the band
- the time r it takes agents to move back to the start of the band
- the length x_{max} of the line
- the mapping $f: x \rightarrow P$ from location along the line to a set of accessible parking spots
- the set E of errors that will occur

5.2 Comparison Method

In this section, we will describe our method for comparing the two layouts by going over how we selected each of the inputs to our simulations.

5.2.1 Car Tasks and Temporal Constraints

Each car that is passed as an input to either of the two simulations has an associated set of tasks that are performed on it in the band, as well as a set of temporal constraints between its tasks. The cars produced in the factory we are modeling are highly customizable, so, for each one, a different set of tasks is performed. To generate cars with realistic sets of tasks and constraints, we used a blackbox tool, provided by the sponsor, that takes into account order frequencies for different optional features to generate cars with the corresponding tasks. The tool takes 8 minutes to generate one car in a virtual machine on a commercial 2.3GHz Intel Core i5 processor with 8GB of RAM. Because the tool is so slow, generating each car every time one is needed for a simulation was not a viable option. Instead, we created a pool of 1385 cars and randomly selected our cars from that set. We ran each simulation where our C was sequence of 700 cars, which is the number produced daily in the factory our

models are based on. For each variation of our parameters, we ran simulations with the same 30 sequences of 700 cars on both layouts to make the results as comparable as possible.

5.2.2 Agents and Stations

For the Conventional Layout, the agents that perform each task and subsequently the stations at which those tasks are performed are specified by the same blackbox tool that generates the tasks and the temporal constraints. The set A of all the agents specified and the count of distinct stations n_s give us the inputs to our Conventional Layout Simulation.

For the Flexible Layout, we used the same number of agents a as those used in the Conventional Layout to make the comparison fair. This is not necessarily the number of agents that maximizes throughput in this layout[1].

5.2.3 Cycle Time and Car Speed

In the factory our model is based on, the cycle time t_c is 100 seconds and the length of each station is 6.65 meters. We use the same cycle time in our Conventional Layout simulation. We can divide the station length by the cycle time to get the speed of the cars in the Conventional Layout, and we use the same speed[2] v in the Flexible Layout.

5.2.4 Comparing the Effect of Errors

It is not immediately obvious how to set up the error input E to the simulations in order to perform a fair comparison between the two layouts. We could measure the

[1]To maximize throughput in this layout, enough agents would have to be allocated such that a car would never have to wait for agents. Specifically, if a new car arrives at the band every t_c and the agents take t_a to finish the tasks on the car and return to the start, the number of agents for optimal throughput assuming no errors is t_a/t_c.

[2]They did not necessarily have to be the same. In both layouts, the speed of the cars needs to be slow enough for the agents to be able to perform tasks on them. The difference is that, in the Conventional Layout, the agents are stationary, while, in the Flexible Layout, the agents move along with the car. Because of this the speed limit of the Flexible Layout is higher than that of the Conventional Layout.

performance of the two models in response to the same errors at the same times on the same sequence of cars. Although this seems like a fair comparison, it is somewhat misleading. The cause of the errors is not considered in our model, but we know that each error is associated with a certain car, station, agent, or task. The two layouts schedule the execution of the same tasks on the same car at different times, by different agents, and at different locations. Thus, two errors of the same duration occurring at the same time in the two models would not necessarily be the reflection of the same real-life error, nor would they have similar effects on the two layouts. We could separate the errors based on their cause and apply them to the two models in appropriately similar ways. For example, a task-related error would occur on the same task in the two models, even if those tasks are scheduled at different times. There are two reasons why we don't take this approach. First, we don't have information about the cause of different errors. We have information about the frequency and duration of different error groups, but we don't know the cause of each group, as that is proprietary information of the sponsor. Second, our models for both layouts would need to change to allow us to specify the time occurrences of different errors based on different aspects of the state of the simulation.

Our solution to this problem is to not try and compare the effect of "similar" errors, and instead run enough simulations of errors randomly drawn from a probability distribution on the two models that we can draw general conclusions about the behavior of the two models. This is one reason why we run simulations with the same parameters on 30 different sequences of cars. We vary the frequency of errors to evaluate their effect on the two models. Specifically we simulate $0\phi, 0.25\phi, 0.5\phi, \phi, 2\phi, 4\phi, 8\phi$, where ϕ is the average frequency of errors per car observed in the factory our models are based on. We use errors per car rather than errors per car per unit time because that is the metric we have data for. Using the errors per car per time metric would have favored the Flexible Layout, as the higher latency[3] of each individual car in the Conventional Layout would give it more time to experience an error.

[3]The Flexible Layout has lower latency for individual cars, since the spacial constraints of the tasks have been lifted, resulting in a less constrained problem that can be scheduled within a lower makespan.

While the error frequency determines the number of errors in E, the specific errors used are randomly generated. We generate them using a different tool provided by the sponsor. The tool generates errors whose duration and group are drawn from probability distributions based on observations of errors in a real factory. The errors we used were drawn uniformly from a pool of 1194 errors generated by the tool.

5.2.5 Band Length

The only parameter that we did not align in the two models was the length of the band. The length of the band in the Conventional Layout is given by the number of stations[4] multiplied by the length of each station[5]. It would be wasteful to set the length of the band for the Flexible Layout to be the same as that of the Conventional Layout. Due to the lack of spacial constraints, cars in the Flexible Layout finish their tasks in a shorter amount of time than those in the Conventional Layout. If the two bands had the same length, agents in the Flexible Layout would remain idle for a long time after they finished the tasks on their car, waiting for it to reach the end of the band. If we instead set the length of the band x_{max} such that the time to traverse it is slightly more than the typical amount of time an error-free car in the Flexible Layout takes to complete its tasks, then the agents will return to the beginning of the band to work on the next car quicker, which means that we achieve better agent utilization.

We ran some simulations of the Flexible Layout with different lengths to determine a good length to use in the comparison. Figure 5-1 shows, for four different band lengths, the agents' total idle time between the times at which they finished the tasks on a car and the time at which the car left the band. Note that these results are specific to the band whose data we were using and not universal. Our simulations determined that most cars finished their tasks after traveling somewhere between 56 and 60 meters in the band. As a result, most cars had to park in the 56 meter band to avoid reaching the end without finishing their tasks. This created higher contention

[4] 14 stations, based on the real factory band we got the task data from
[5] 6.65 meters, based on real factory measurements

for the parking spots and even resulted in some instances of cars reaching the end without being able to park, and stopping the entire band behind them in order to finish their tasks. Figure 5-2 shows that such stoppages were only observed for the 56 meter band. Due to these additional stoppages, we ruled out the 56 meter band. Of the remaining lengths we simulated, the 60 meter band exhibited the lowest agent idle time, so we used that in our comparison.

5.2.6 Flexible Layout Band Shape

There are two other inputs we have not yet addressed. The first one is the time r it takes agents to move back to the beginning of the band. The second one is the mapping $f: x \rightarrow P$ from location along the line to a set of accessible parking spots. Both depend on the shape of the band. The shape we decided to use, shown in Figure 5-3, has several desirable traits. The line along which the cars move is wrapped around a linear arrangement of parking spots, such that each parking spot is accessible from either side of the line. The number of parking spots depends on the length of the line. In the case of Figure 5-3, the line of parking includes 5 spots because that is how many fit in it. The band length we simulated for happened to also fit 5 parking spots. We also added two additional parking spots that are accessible from the final section of the band (P6 and P7 in Figure 5-3). As stated in Chapter 4, if a car can't find parking at its current location, it stays in the line until it reaches an area where different parking is available. This is problematic for cars at the final section of the band, so we added the extra spots to make it unlikely that they exit the band before finding parking. The end of the band is close to the start of the band, which makes r low, which results in better agent utilization. The area requirement for the layout is relatively small, as its ratio of parking space to total space is approximately 1/3. We considered trying layouts that featured more parking, but this layout provided enough parking for the amounts of errors we tested for, so layouts with more parking would be wasting space. It may seem as though the layout is too long and narrow, which would make it hard to fit into a closed space, but that is not necessarily the case. The line of parking spots does not have to be straight. We could have the line of parking

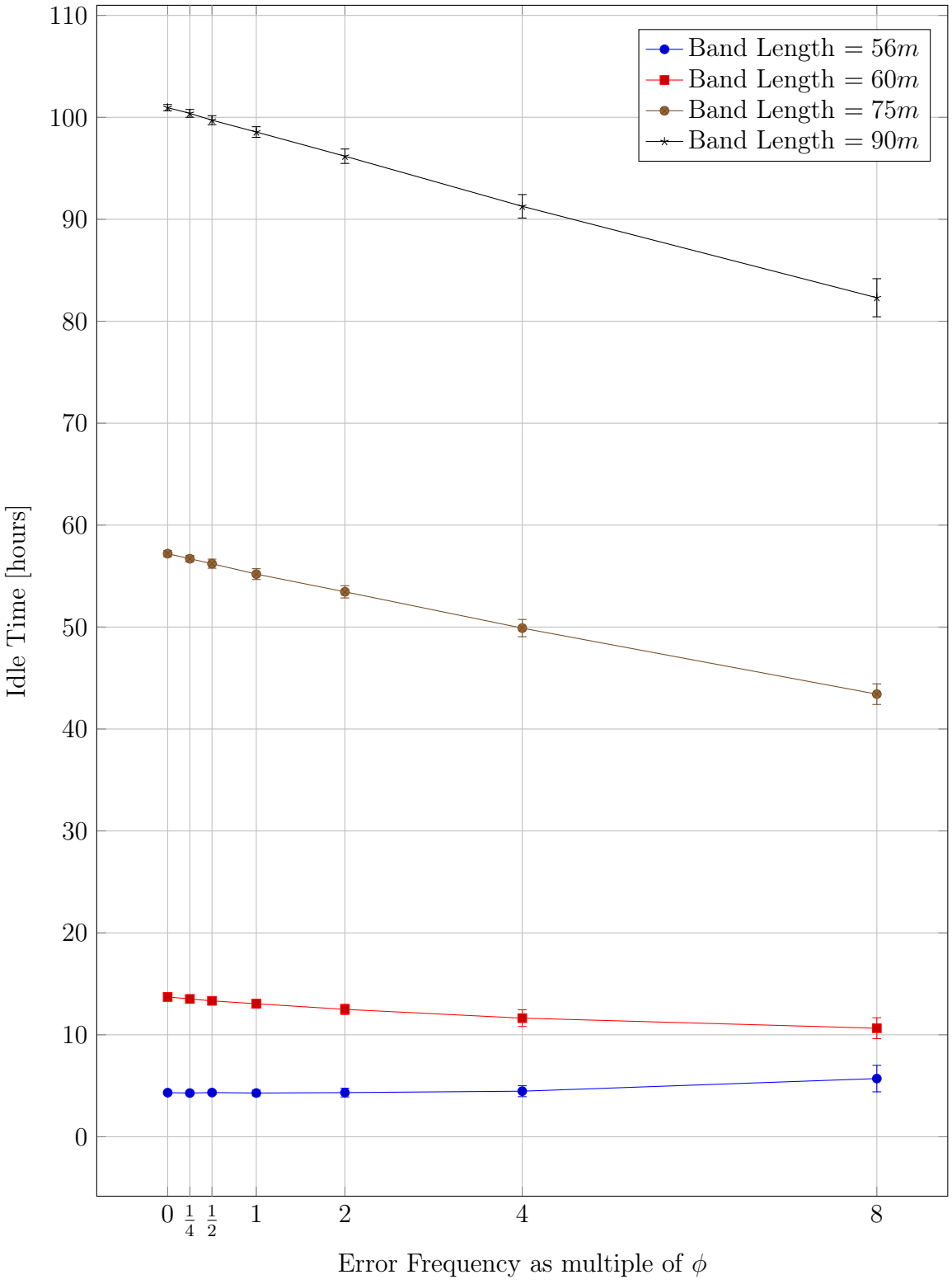

Figure 5-1: Idle time in Flexible Layout for different band lengths

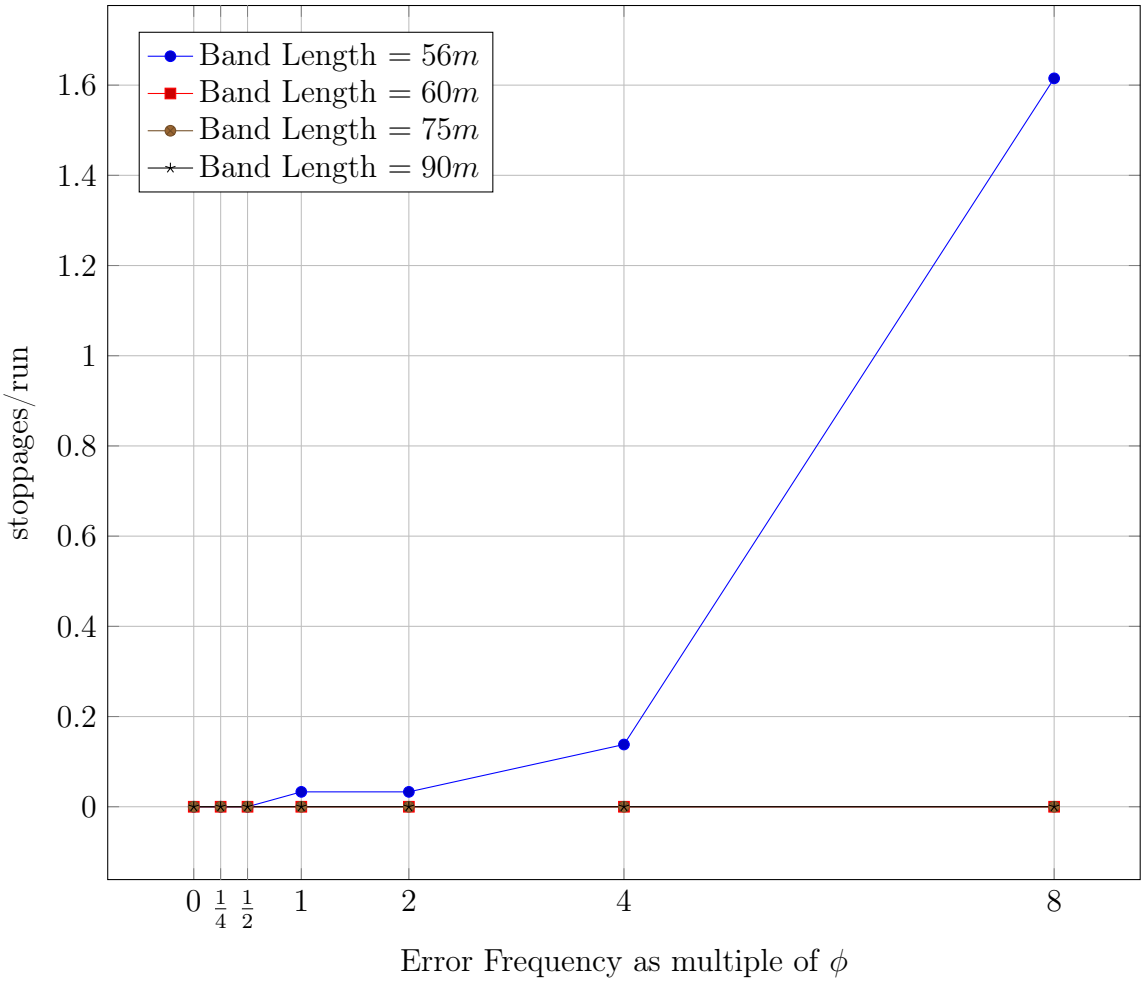

Figure 5-2: Full-Band stoppages per 700 car run in Flexible Layout for different band lengths. Full-Band stoppages occur when a car reaches the end of the band without having finished its tasks or resolved its errors. The 60m, 75m, and 90m lines are all constant at 0 and overlapping.

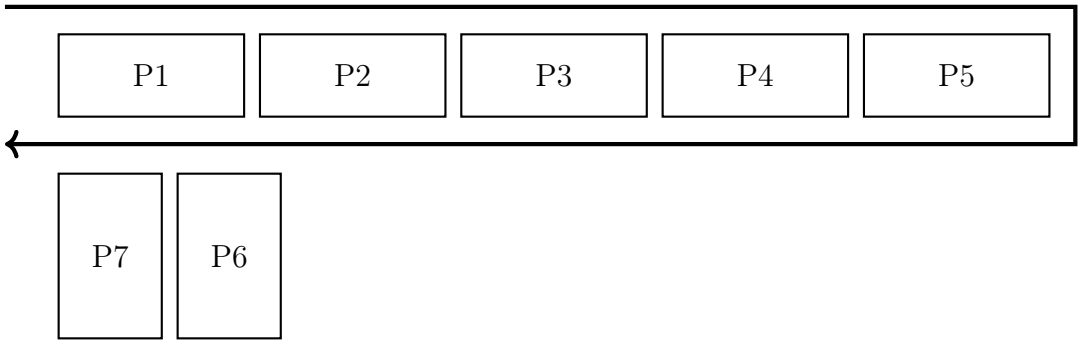

Figure 5-3: The line along which the cars move wraps around the line of parking spots. Additional parking spots P6 and P7 are accessible from the final section of the band.

turn and fold on itself, and, as long as the line of cars continues to wrap around it, the same parking spot mapping holds[6]. By folding the line of parking spots, we can fit this layout into a variety of differently shaped areas. As long as there is one row of parking spots between two parallel and approximately equally long segments of the line, the layout is correctly modeled by our simulation. An example of the line of parking spots folding is shown in Figure 5-4.

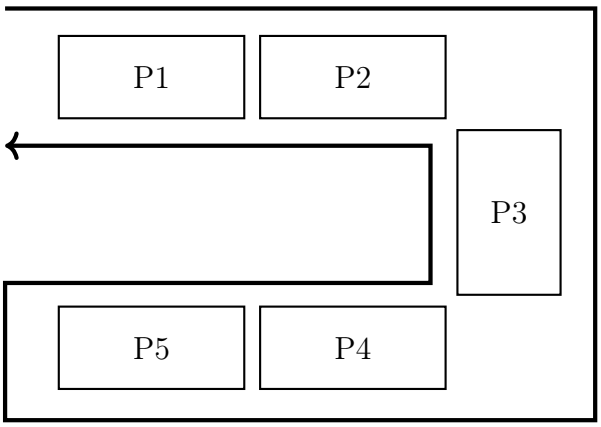

Figure 5-4: The line of parking spots folds to fit into a shorter but wider space.

5.2.7 Space Comparison

Now that we have specified the shape of the Flexible Layout, we can compare it to the Conventional Layout in terms of the area they occupy. It is reasonable to assume that the line along which the cars move in the Flexible Layout will be wider than that of the Conventional Layout, since it needs to fit platforms that carry not only the car, but also the agents working on it and the tools and resources required for its tasks. Let the ratio of the Flexible Layout line's width to the Conventional Layout line's width be λ. Also let the Conventional Layout line's width be w meters. We assume that the parking spot width is also λw, since parking spots also must have enough space for the mobile platforms. The area covered by the band we simulate in the Conventional Layout is given by the number of stations multiplied by the station

[6]This ignores the difference between the length of the outside and inside turning radius, but is a good approximation, especially if the line of parking spots turns left and right approximately the same number of times.

length multiplied by the width, which is $14 \times 6.65 \times w = 93.1w$ meters squared. Based on the shape we used for the Flexible Layout, the line along which cars move is 60m, and the line of parking is approximately half of that, estimated here as 30m long. We also add in two additional parking spots, each with size $6.65\lambda w$ meters squared. Thus, according to our assumptions, the area of the band we simulate in the Flexible Layout is $103.3\lambda w$ meters squared.

5.2.8 Performance Metric

We evaluate the performance of each layout under a given error frequency by measuring the amount of simulated time it needs to finish a sequence of 700 cars. Note that the time we are referring to is how long it takes the cars to pass through the band in the simulation, not how long our simulation takes to run. One might argue that measuring throughput of the two layouts in steady state, meaning when all agents are occupied, would yield a better metric of performance. We observe that our metric is a good approximation of steady-state throughput because the time taken by one car to move through the band is small compared to the total time all 700 cars take.

5.3 Results

The simulations were implemented in Java and took two hours to run on two cores of a commercial 2.2GHz Intel Core i7 processor with 16GB of RAM. Figure 5-5 shows the data points for all of our simulations. Figure 5-6 and Table 5.1 show the mean and standard deviation of the time in hours to complete 700 cars for each error frequency for the two layouts. The Error Delay columns in Table 5.1 show the cumulative delay added by errors to each model as a percentage of the 0ϕ average time. The Average Speedup column in Table 5.1 shows the speedup of the Flexible Layout as a percentage of the Conventional Layout time at each error frequency.

In the case of no errors, the Flexible Layout finished in 17.1% less time than the Conventional Layout. This can be explained by the following observations. In the Conventional Layout, each task must be executed at a particular station. On

Error Frequency (multiples of ϕ)	Conventional			Flexible			Average Speedup
	Mean (hours)	Error Delay	Standard Deviation (hours)	Mean (hours)	Error Delay	Standard Deviation (hours)	
0	24.01	0.0%	0.22	19.91	0.0%	0.01	17.1%
0.25	25.26	5.2%	0.52	19.99	0.4%	0.06	20.9%
0.5	26.38	9.9%	0.71	20.07	0.8%	0.07	23.9%
1	28.66	19.4%	0.85	20.23	1.6%	0.10	29.4%
2	33.01	37.5%	1.49	20.63	3.6%	0.14	37.5%
4	40.25	67.7%	1.83	21.41	7.6%	0.25	46.8%
8	53.06	121.0%	1.75	23.23	16.7%	0.26	56.2%

Table 5.1: Simulation Result Comparison

the other hand, the tasks in the Flexible Layout can be executed at any location of the band. Thus, a car in the Flexible Layout will not have to wait to reach the appropriate station to execute certain tasks, when it would have to do so in the Conventional Layout. Simply stated, the scheduling problem for the Conventional Layout is more constrained, so the solution found will usually be slower than that of the Flexible Layout. The only case in which the two layouts would result in solutions of equal duration is if the time the tasks in each station took to complete was exactly equal to the time spent at each station, which would mean that the additional spatial constraint would not affect the problem's solution.

The advantage of the Flexible Layout becomes even greater in the presence of errors. Recall that an error that occurs on a car in the Conventional Layout will cause the entire band to stop moving, while a car experiencing an error in the Flexible Layout can park and allow other cars to overtake it. As a result, an error in the Conventional Layout delays all cars in the band, while an error in the Flexible Layout delays only the affected car. Because of this, the Conventional Layout discussed needs 19.38% more time to finish in the case of 1ϕ (average) error frequency compared to no errors, while the Flexible Layout is barely affected at 1ϕ error frequency. Overall, in the case of 1ϕ error frequency, the Flexible Layout finished in 29.4% less time than the Conventional Layout. With more errors, the difference in the performance of the two layouts grows further, with a trend that is approximately linear. In the extreme

case of 8ϕ error frequency, the Flexible Layout finished in 56.3% less time than the Conventional Layout.

5.4 Conclusions

Our comparison showed that the Flexible Layout produces better throughput than the Conventional Layout and the resulting schedule is more robust to errors. In order to decide whether implementing the Flexible Layout is worthwhile, these benefits must be weighed against the cost of the investment. The Flexible Layout requires mobile platforms that can carry a car, the agents working on it, and the necessary tools and resources for all the car's tasks in the band. It also requires agents able to perform all the tasks a car needs done in a band rather than just all the tasks a car needs done in a station. Improving the throughput of a single band by switching it to the Flexible Layout will only improve the factory's overall throughput if that band was a bottleneck. If it was not, it will continue to get blocked or starved because of errors in other bands. To gain the full benefit of the Flexible Layout, it must be applied to enough bands such that the bottleneck becomes a band in the Flexible Layout. A future cost-benefit analysis could use the findings of this chapter to determine whether the productivity improvement warrants such a large-scale change to existing factories.

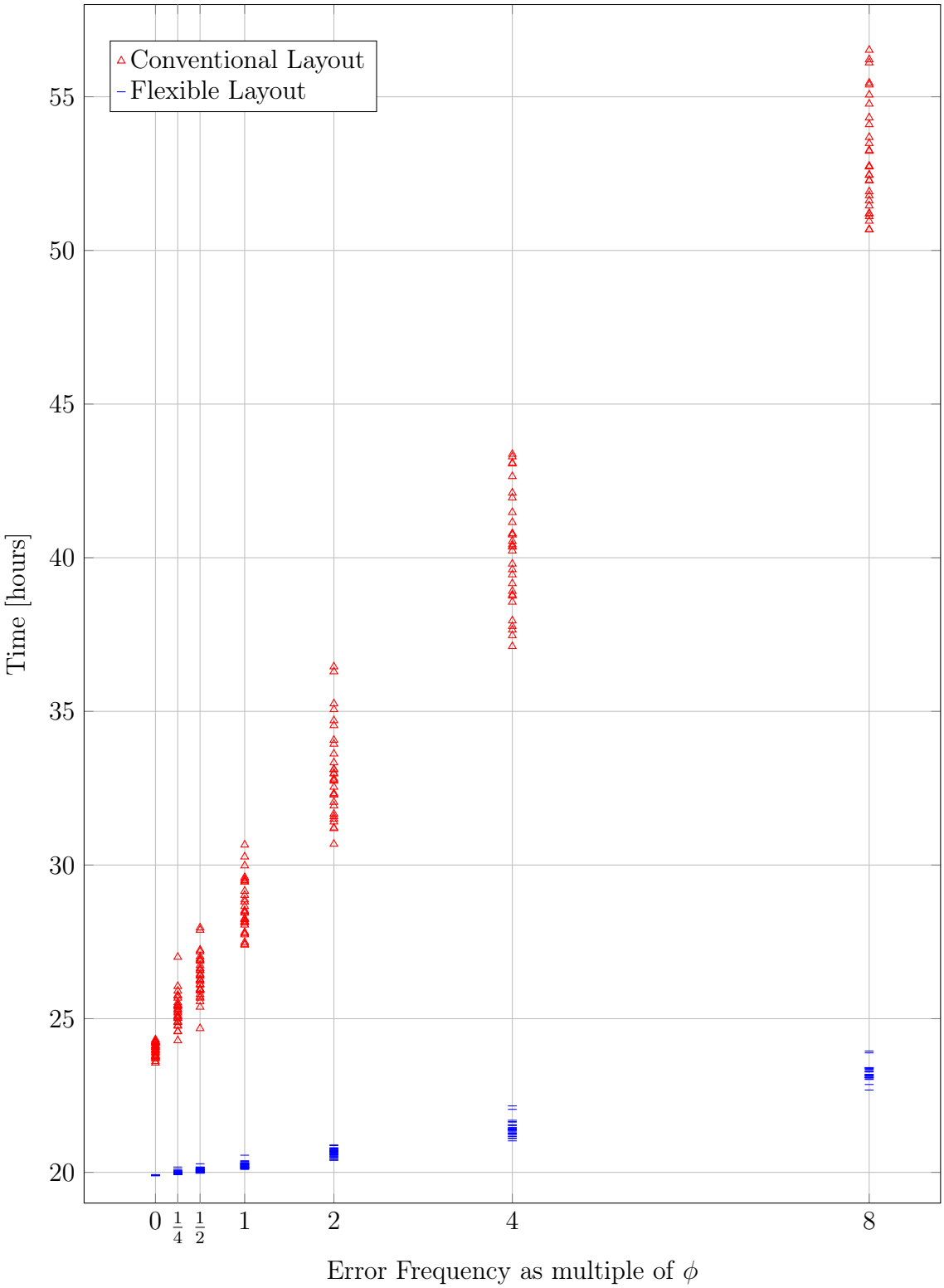

Figure 5-5: Time to finish 700 cars

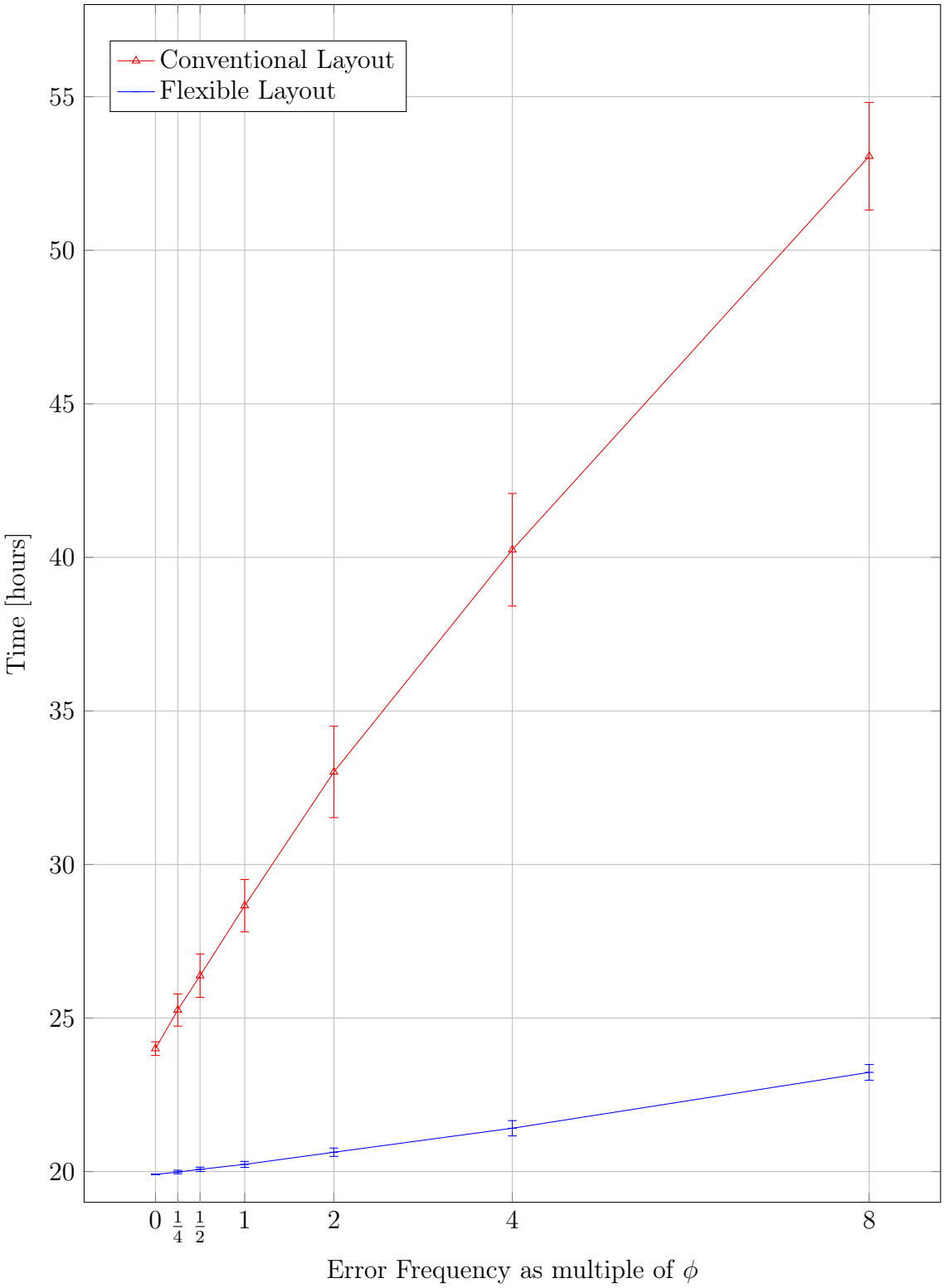

Figure 5-6: Simulation result means and standard deviations

Chapter 6

Conclusion

6.1 Review

For automobile companies, increasing the throughput of their assembly lines can prove very profitable, as it allows them to meet demand with lesser resources. The current assembly process is very efficient under normal operation, but suffers greatly when unpredictable disturbances, such as equipment failure or logistics errors, occur. Companies are searching for a layout that can lower the cost of errors, while retaining the same or better efficiency.

In this thesis, we developed simulations for two assembly layouts and used those simulations to evaluate the layouts. The first is the Conventional Layout, in which cars are placed on a conveyor belt that moves at a constant speed. Agents on both sides of the belt perform a small variety of tasks on the cars passing in front of them. The main downside of this layout is the fact that when an error occurs, the entire line stops moving until the error is resolved. The second is the Flexible Layout, in which cars along with the agents working on them are placed on mobile platforms and move together along the line. In this layout, agents are required to be able to perform a larger variety of tasks, as opposed to the Conventional Layout, in which they are fixed to a specific station in the line, performing similar tasks for a large number of cars. The main advantage is that in the case of an error, the affected car can move to the side and allow the other cars to overtake it, thus limiting the effect

of the error to one car, instead of the entire line.

We developed Discrete Event Simulations for both layouts. We applied approximations for some types of temporal constraints to computationally simplify our scheduling algorithms. We developed mathematical formulations of the problems that our systems solved. We explained the architecture of the simulation and schedulers for each layout and analyzed the runtime and memory usage.

We explained the setup of our comparison, which we tried to make as fair as possible. We reported our results, which show that the Flexible Layout was able to finish the same number of cars 17.1% faster in the case of no errors and 29.4% faster in the case of an average number of errors, according to observations of real factories. We provided an explanation of why these simulation results matched our expectations.

6.2 Future Work

Our simulations show that the Flexible Layout can provide better throughput and robustness to errors. It does, however, require significant investment. It requires mobile platforms capable of fitting a car, the agents working on it, and the tools and resources required for all the tasks in one band. The factory's logistics will need to be modified, as materials will need to be available at the beginning of the band instead of at a station. Agents will be required to perform a larger variety of tasks, since a small set of them will be performing all the tasks for a car in one band. This is in contrast to the Conventional Layout, in which agents are fixed to a station and perform a small variety of tasks on all the cars that pass through the station. A cost-benefit analysis taking all the aspects of a potential investment into account, along with the results of this thesis, could determine whether a switch to the Flexible Layout could be profitable.

Our simulations modeled a single band within the factory. An increase in the throughput of one band from switching to the Flexible Layout will only affect overall throughput significantly if that band was the bottleneck or experienced a particularly

large amount of errors. Bands have buffer zones between them in order to contain starving and blocking effects to the band in which the error that caused them occurred. However, long-lasting errors can cause the buffers to fill or empty, thus spreading the effects to surrounding bands. A future project could study the behavior of multiple-band segments of these layouts, and compare them with each other, or even with hybrid layouts in which some bands are under one layout, while others are under the other. Such a study could give insight into the larger-scale effects that a change from Conventional to Flexible Layout could have on the assembly line throughput.

Bibliography

[1] Matthew C. Gombolay, Ronald J. Wilcox, and Julie A. Shah, "Fast Scheduling of Multi-Robot Teams with Temporospatial Constraints," *Proc. of the Robotics: Science and Systems Conference*, Berlin, Germany, June 2013.

[2] George Chryssolouris, "Manufacturing Systems: Theory and Practice," Springer-Verlag, 2006.

[3] Hoda ElMaraghy and Waguih ElMaraghy, "Learning Integrated Product and Manufacturing Systems," *Procedia CIRP*, vol. 32, pp. 19–24, 2015.

[4] Dimitris Mourtzis, Nikolaos Papakostas, Dimitris Mavrikios, Sotiris Makris, and Kosmas Alexopoulos, "The role of simulation in digital manufacturing: applications and outlook," *International Journal of Computer Integrated Manufacturing*, vol. 28, no. 1, pp. 3–24, 2015.

[5] Wolfgang Kuehn, "Digital factory — Integration of simulation enhancing the product and production process towards operative control and optimisation," *International Journal of Simulation — Systems, Science & Technology*, vol. 7, no. 7, pp. 27–39, 2006.

[6] D.J. Murray-Smith, "Continuous System Simulation," Springer-Verlag, 1995.

[7] Alessandra Caggiano, Fabrizia Caiazzo, and Roberto Teti, "Digital factory approach for flexible and efficient manufacturing systems in the aerospace industry," *Procedia CIRP*, vol. 37, pp. 122–127, 2015.

[8] Steffen Bangsow, "Manufacturing Simulatio with Plant Simulation and SimTalk," Springer-Verlag, 2010.

[9] Siemens PLM Software, "Mahindra Vehicles: Mahindra rises with digitally planned new vehicle manufacturing facility," *Customer Case Studies*, [cited July 31, 2016] http://www.plm.automation.siemens.com/CaseStudyWeb/dispatch/viewResource.html?resourceId=30584, 2015.

[10] Zhu Jie, "Tecnomatix software and its application in BIW welding," *Electric Welding Machine*, vol. 43, no. 2, pp. 16–19, 2013.

[11] Marek Kliment, Peter Trebuňa, Miriam Pekarčíkova, Radko Popovič, and Jaromír Markovič, "Use of simulation in optimization of the production process and their car doors assembly," *Applied Mechanics and Materials*, vol. 816, no. 7, pp. 555–561, 2015.

[12] George Michalos, Platon Sipsas, Sotiris Makris, George Chryssolouris, "Decision making logic for flexible assembly lines reconfiguration," *Robotics and Computer-Integrated Manufacturing*, vol. 37, no. 1, pp. 233–250, 2016.

[13] Luis Pinto Ferreira, Enrique Ares Gómez, Gustavo C. Peláez Lourido, José Diéguez Quintas, and Benny Tjahjono, "Analysis and optimisation of a network of closed-loop automobile assembly line using simulation," *The International Journal of Advanced Manufacturing Technology*, vol. 59, pp. 351–366, 2012.

[14] Iiro Harjunkoski, Christos T. Maravelias, Peter Bongers, Pedro M. Castro, Sebastian Engell, Ignacio E. Grossmann, John Hooker, Carlos Méndez, Guido Sand, John Wassick, "Scope for industrial applications of production scheduling models and solution methods," *Computers and Chemical Engineering*, vol. 62, pp. 161–193, 2014.

[15] Michał Mazur and Antoni Niederliński, "A Two-stage approach for an optimum solution of the car assembly scheduling problem Part 1. Problem statement,

solution outline and tutorial example," *Archives of Control Sciences*, vol. 25 (LXI), no. 3, pp. 355–365, April 2015.

[16] Michał Mazur and Antoni Niederliński, "A Two-stage approach for an optimum solution of the car assembly scheduling problem Part 2. CLP solution and real-world example," *Archives of Control Sciences*, vol. 25 (LXI), no. 3, pp. 367–375, April 2015.

[17] Henry Ford, in collaboration with Samuel Crowther, "My Life and Work," Doubleday, Page & Co, 1923.

[18] Yuval Cohen, "Assembly line segmentation: determining the number of stations per section," *Journal of Manufacturing Technology Management*, vol. 24, no. 3, pp. 397–412, 2013.

[19] Jörn-Henrik Thun, Robert P. Marble, and Victor Silveira-Camargos, "Conceptual framework and empirical results of the risk and potential of Just in Sequence - A Study of the German Automotive Industry," *Journal of Operations and Logistics*, vol. 1, no. 2, pp. 1–13, 2007.

[20] Stephan M. Wagner, and Victor Silveira-Camargos, "Decision Model for the Application of Just-in-Sequence," *International Journal of Production Research*, vol. 49, no. 19, pp. 5713–5736, October 2011.

[21] Rina Dechter, Itay Meiri, and Judea Pearl, "Temporal constraint networks," *Artificial Intelligence*, vol. 49, pp. 61–95, 1991.

[22] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384–393, June 1975.

[23] Stewart Robinson, "Simulation: The Practice of Model Development and Use," Wiley, 2004.