Decentralized Scheduling of Discrete Production Systems with Limited Buffers

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von der KIT-Fakultät für Maschinenbau des
Karlsruher Instituts für Technologie (KIT)
angenommene

Dissertation

von

M.Sc. Olaf Zimmermann

aus Geseke

Tag der mündlichen Prüfung: 19.07.2022
Hauptreferent: Prof. Dr.-Ing. Kai Furmans
Korreferent: Prof. Dr. Horst Tempelmeier

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Fördertechnik und Logistiksysteme (IFL) des Karlsruher Instituts für Technologie. Ich möchte mich an dieser Stelle bei allen Personen bedanken, die zum erfolgreichen Gelingen dieser Arbeit beigetragen haben.

Ich danke meinem Doktorvater Prof. Dr.-Ing. Kai Furmans, da er durch die Übernahme der Betreuung sowie der Erstkorrektur diese Arbeit überhaupt erst ermöglicht hat. Weiterhin danke ich Prof. Dr. Horst Tempelmeier für die Übernahme des Korreferats sowie Prof. Dr.-Ing. Xu Cheng für die Übernahme des Prüfungsvorsitzes meiner Promotionsprüfung.

Ich möchte mich ebenfalls bei allen aktiven und ehemaligen Kollegen des IFL bedanken. Die gemeinsame Zeit am Institut hat mir sehr viel Freude bereitet und auch oft dazu beigetragen die Arbeit an der Disstertation in Vergessenheit geraten zu lassen.

Besonderer Dank gilt meiner Familie und meinen Freunden. Sie haben mich während der Dauer meiner Promotion nach Kräften unterstützt und stets ermutigt diesen Weg fortzusetzen. Insbesondere das Korrekturlesen und die fachlichen Diskussionen haben mir geholfen. Für diesen Rückhalt bin ich sehr dankbar. Ich danke meinen Eltern und meiner Schwester für die grenzenlose Unterstützung auf meinem bisherigen Lebensweg.

Geseke, Juli 2022 Olaf Zimmermann

Kurzfassung

Die Steuerung der Produktion ist eine der Kernaufgaben eines jeden produzierenden Unternehmens. Sie ist insbesondere wichtig, um auf die Anforderungen des Marktes und damit auf die Wünsche der Kunden reagieren zu können. Aktuelle Trends im Markt führen dabei zu einer hochindividualisierten Produktion bei gleichzeitiger Erhöhung der produzierten Stückzahlen. Eine Konsequenz daraus ist, dass Unternehmen über flexiblere und agilere Produktionssysteme verfügen müssen, um auf die sich ständig ändernden Kundenwünsche reagieren zu können. Da starre Fertigungslinien nicht mehr geeignet sind, werden zunehmend komplexere Strukturen wie die der Werkstattfertigung oder Matrixproduktion eingesetzt. Hierfür werden geeignete Steuerungsmethoden für die Produktion benötigt. Diese Arbeit beschäftigt sich mit eben jenen Steuerungsmethoden, genauer gesagt Methoden zur Planung von Produktionsaufträgen in diesen neuen Produktionssystemen.

Zur Steuerung eignen sich echtzeitfähige und autonome Entscheidungssysteme, mit denen die Steuerung der neuen Organisationsstruktur der Produktion angepasst ist. Agentenbasierte Systeme bieten genau diese Eigenschaften und erlauben es, komplexe Planungsaufgaben in kleinere Teilprobleme zu zerlegen, die schneller und genauer gelöst werden können. Sie erfordern die Verfügbarkeit von Daten in Echtzeit und eine schnelle Kommunikation zwischen den Agenten, was heute dank der vierten industriellen Revolution zur Verfügung steht. Demgegenüber steht der erhöhte Koordinierungsbedarf, der in diesen Systemen beherrscht werden muss. Das Ziel dieser Arbeit ist es, einen dezentralen Produktionsplanungs-Algorithmus zu entwickeln, der in einem Multi-Agenten-System implementiert ist. Er berücksichtigt begrenzte Verfügbarkeit von Pufferplätzen an jedem Arbeits-

platz, ein Thema, das in der Literatur wenig erforscht ist. Der Algorithmus ist in einer flexiblen Werkstattfertigung anwendbar und zeigt eine große Zeiteffizienz bei der Einplanung größerer Mengen von Aufträgen.

Um dieses Ziel zu erreichen, wird zunächst der Produktionsplanungs-Algorithmus ohne das Agentensystem entworfen. Er basiert auf der von Adams et al. (1988) veröffentlichten Shifting Bottleneck Heuristik. Da viele Änderungen notwendig sind, um die geforderten Eigenschaften berücksichtigen zu können, bleibt nur die grundlegende Vorgehensweise gleich, während alle Schritte der Heuristik von Grund auf neu modelliert werden. Anschließend wird ein Multi-Agenten-System entworfen, das die genannten Anforderungen abbildet und den Algorithmus zur Planung verwendet. In diesem System hat jeder Arbeitsplatz einen Arbeitsplatzagenten, der für die Planung und Steuerung seines zugeordneten Arbeitsplatzes zuständig ist, sowie einige zusätzliche Agenten für die Kommunikation, die Datenspeicherung und allgemeine Aufgaben. Der entworfene Algorithmus wird angepasst und in das Multi-Agenten-System implementiert. Da das System im praktischen Einsatz immer eine Lösung finden muss, stellen wir mögliche Fehlerfälle vor und wie mit ihnen umgegangen wird. Abschließend findet eine numerische Evaluierung mit zwei realen Produktionssystemen statt. Da sich diese Systeme in einem wichtigen Merkmal ähneln, werden weitere zufällig erzeugte Beispiele getestet und ausgewertet.

Abstract

Production control is one of the core tasks of any manufacturing company. It is crucial to respond to the market requirements and thus to the demands of the customers. Current trends in the market lead to highly individualized production with a simultaneous increase in the number of goods produced. One consequence of this is that companies must have more flexible and agile production systems to respond to constantly changing customer requirements. Since rigid production lines are no longer suitable, more complex structures such as job shops or matrix production increasingly see usage. They require suitable control methods for production. This thesis deals with those control methods, more precisely with planning production orders in these new production systems.

Real-time and autonomous decision systems are suitable for the control of the new organizational structure of production. Agent-based systems offer precisely these characteristics and break down complex planning tasks into smaller sub-problems that can be solved faster and more accurately. They require real-time data availability and fast communication between agents, which is available today thanks to the fourth industrial revolution. It contrasts with the increased coordination requirements that these systems must master. The goal of this work is to develop a decentralized production scheduling algorithm implemented in a multi-agent system. It considers the limited availability of buffer space at each workstation, a topic that is not well studied in the literature. The algorithm is applicable in a flexible job shop environment and shows excellent time efficiency in scheduling large quantities of jobs.

The production scheduling algorithm is first designed without the agent system to achieve this goal. It bases on the Shifting Bottleneck heuristic

published by Adams et al. (1988). Since many changes are necessary to accommodate the required characteristics, only the basic procedure remains the same, while all steps of the heuristic are modeled from scratch. Then, a multi-agent system is designed to represent the above requirements using the algorithm for scheduling. In this system, each workplace has a workplace agent responsible for scheduling and controlling its assigned workplace and additional agents doing the communication, data storage, and general tasks. The designed algorithm is adapted and implemented in the multi-agent system. Since the system always has to find a solution during practical use, we present possible failure cases and how to deal with them. Finally, a numerical evaluation with two real-world production systems takes place. Since these systems are similar in one crucial feature, further randomly generated examples are tested and evaluated.

Contents

Kurzfassung iii

Abstract v

1 Introduction 1

2 Basics of Scheduling 7
2.1 Introduction to Scheduling 7
2.1.1 Notation 8
2.1.2 Optimization Criteria 9
2.1.3 Shop Models 10
2.2 The Shifting Bottleneck Procedure 15
2.3 Extensions of the Shifting Bottleneck Procedure 18
2.3.1 Changes to the Single Workplace Subproblem 19
2.3.2 Changes to the General Procedure 24

3 Multi-Agent Systems for Production Planning 29
3.1 Characteristics of Agents 29
3.2 Multi-agent Systems 31
3.2.1 Interactions between Agents 31
3.2.2 Coordination and Planning 34
3.3 Existing Multi-agent Systems in the Literature 36
3.3.1 Applications of the SB Heuristic in Agent Systems 37
3.3.2 General Agent Systems for Production Planning 39

4 Scope of this Thesis 49
4.1 Conclusion to the Literature Overview and Research Gap 49
4.2 Problem Description 51
4.3 Research Questions 54

5 Scheduling Algorithm 57
5.1 Guidelines for the Algorithm 59

5.2 Preparations before Scheduling 61
5.3 Create a Local Sequence 63
5.3.1 Update the Key Variables of Operations 64
5.3.2 Determine the Next Operation to be Scheduled 67
5.4 Identification of the Bottleneck 76
5.5 Re-optimization 77
5.6 Finish the Scheduling Process 82

6 Decentralized Scheduling System 83
6.1 Multi-Agent System 83
6.2 The Decentralized Scheduling Algorithm 87
6.2.1 Preparations before Scheduling 90
6.2.2 Create a Local Sequence 92
6.2.3 Identify the Bottleneck 93
6.2.4 Re-optimization 94
6.3 Termination of the Decentralized Algorithm 106
6.3.1 Deadlock during Execution 107
6.3.2 Livelock during the Execution 108
6.3.3 Deadlock during Planning 110
6.3.4 Livelock during Planning 111
6.3.5 Additional Solution to Termination Problems 116

7 Numerical Evaluation of the Multi-agent System 119
7.1 Setup of Experiments 119
7.1.1 Input Parameter of Experiments 120
7.1.2 Implementation of the FIFO Dispatching Rule 121
7.1.3 Process of Experiments 122
7.2 First Application Example 123
7.3 Second Application Example 129
7.4 More Complex Production Systems 133

8 Conclusion 137
8.1 Summary 137
8.2 Outlook 140

Notation 143

References 145

List of Figures 157

List of Tables 159

A Data for Application Example One 163

B Data for Application Example Two 167

C Data for Randomized Experiments 175

1 Introduction

Requirements for production systems have always depended on the market and therefore on the needs of the customers. Companies need to react to the latest trends in society to stay competitive. Change in the market also offers opportunities for the companies, for example, to influence the trends or to open new business segments. In the last decades, a general trend went from requiring pure mass production towards a more individualized production while simultaneously increasing the number of goods produced. Furthermore, globalization added the need to offer various product variations for different regions of the world. Figure 1.1 shows a time-line of these trends in the production area of the last 150 years.

Other trends, which require adaptation of production systems, are shorter innovation and product life cycles, more competition on global markets, and higher customer quality demands. These trends lead to a need for more flexibility and agility in production systems (Leitão 2009). To achieve the desired flexibility, new design methods and tools for production are necessary. "In the future, long-established paradigms of production will still have to continue to change in order to meet the demand for even more individuality, customer-specific product variants and shortest delivery times within the meaning of the term production on demand" (Matt et al. 2015). One of these paradigms can be the organizational structure of the production. It is no longer suitable to build inflexible production lines because installing new lines or changing existing ones requires a high amount of work, know-how, and money. A new and more modular organization of the production is required to handle the current trends, based on a job shop or matrix production. They allow producing various products and product families in the

same production system. However, this only works with a suitable control and planning system for a much more complex production.

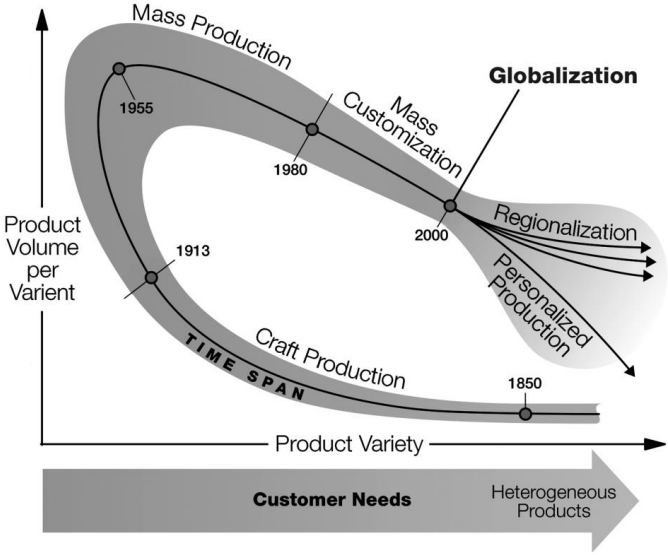

Figure 1.1: Overview of trends in production systems (Koren 2010, p.34)

Currently, we are in the fourth industrial revolution, also called Industry 4.0. Real-time and autonomous control of manufacturing processes becomes more and more relevant (Grundstein et al. 2017). It is characterized by "connected and communicating systems through the newest internet technology" (Roth 2016, p. 5). These features enable us to combine the required modular organizational structure of a production system with a decentralized and modular software structure controlling it. Agent-based systems are a possible solution for the previously mentioned more complex planning requirements, enabling decentralized, autonomous real-time planning decisions. Multi-agent systems allow to break down highly complex problems into smaller subproblems, making them much more manageable. The system agents coordinate with each other their local decisions and solve the original global problem together. This approach is known to offer much more flexibility than traditional hierarchical systems

with central planning units (J. Zhang et al. (2019)). Advantages of decentralized planning include faster computing of results, better flexibility to dynamic surroundings, robust result quality under changing external circumstances, and better security if parts of the system fail. On the other hand, it needs extra measures to make sure the system is working continuously. These include ensuring that the data is consistent between the elements and that the agents can exchange it in real-time with a reliable communication structure between the different parts of the production system. Furthermore, it needs to include additional rules if a communication is not successful or data is missing.

Parente et al. (2020) write that current decentralized multi-agent systems are limited to selfish decisions and local optima and that the challenge is to gather available data to approach globally optimal decisions. This thesis tries to improve upon that point by introducing a new multi-agent system for scheduling, in which the agents consider the other agents in their local decisions to obtain a better global solution. This work aims to develop a **decentralized scheduling algorithm**, which incorporates **limited available buffer space** at each workplace. It schedules a **flexible job shop** production and scales to large production systems without losing much **efficiency**. The solution is based on a multi-agent system and can react to any change of the production system it represents at the latest at the start of the following scheduling. The idea behind this thesis originated with an existing modular production system in mind. The goal was to create a suitable planning algorithm to schedule it. The usual approach in practice is to use an algorithm for scheduling and then decide how much buffer space is required in the production afterward, making limited buffers in production systems not often researched in the literature. In our case, since the factory for the application of this algorithm already existed, there were already clearly defined maximum buffer slots available. Therefore, the algorithm presented in this works needs to be able to handle this restriction. Nevertheless, we designed the presented algorithm to be applicable in any given discrete production system with the same features. It was a requirement that the algorithm should be as deterministic as possible, meaning that the rules and

decisions taken by it based on the current situation are comprehensible to a human, and nothing happens seemingly by chance. This requirement excluded several meta-heuristics, like, for example, a genetic algorithm or artificial intelligence methods from the start. Therefore, this work's algorithm is based on the **Shifting Bottleneck heuristic** by Adams et al. (1988) since previous publications have proven that it gives good results, and its structure makes it perfectly adaptable to a decentralized system.

Structure of the Thesis

The thesis starts by giving an overview of the two topics combined in this work. First, scheduling is introduced in **chapter 2**, followed by introducing multi-agent systems in **chapter 3**. Both chapters include a short introduction to their topic and an overview of the later chapters' methods. They also contain a literature review of the relevant publications. These include extensions of the method used for scheduling and existing appliances of multi-agent systems for production planning. **Chapter 4** continues by giving a presentation of the scheduling problem underlying this thesis. It combines the problem and the literature overview of the previous chapters, identifies the research gap, and develops the research questions concerning the problem.

The following two chapters are the central part of the thesis. **Chapter 5** presents an algorithm to solve the scheduling problem based on the Shifting Bottleneck heuristic presented in chapter 2. Since not much more than the basic idea is kept from the original method, it presents all the algorithm steps in detail. The new algorithm is an iterative procedure consisting of four steps. First, it describes the general procedure, and afterward, all of the steps in detail. **Chapter 6** applies the developed algorithm in a multi-agent system. It starts with describing the multi-agent system structure and a description of all elements present in it. Furthermore, it describes the necessary changes and additions to make the developed algorithm work in a decentralized system. Since termination is an essential topic in a decentralized system, we finish by describing the features and safety func-

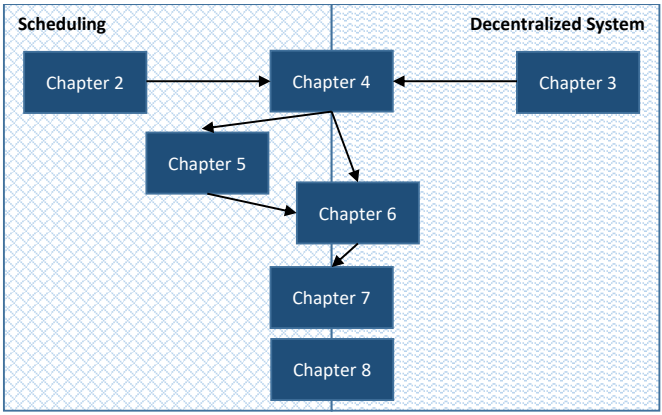

Figure 1.2: Organization of the thesis

tions present in the agent system to guarantee always finding a solution to a scheduling call.

Afterward, chapter 7 presents a numerical evaluation of the algorithm compared to a FIFO (First-In-First-Out) solution. After describing the experiments' setup, we apply the algorithm in the production system of two companies. Since these real-world examples have a common characteristic, we did further tests on randomized systems without that limitation. Finally, the **chapter 8** concludes this thesis. It presents the achieved results and answers to the research questions and presents an overview of how the work can continue in the future.

2 Basics of Scheduling

As mentioned in the introduction, the thesis combines two research fields, scheduling, and multi-agent systems, into one production planning system. In this chapter, we start presenting the first area, the scheduling of production systems. After giving a short general introduction to scheduling, including basic notations and definitions, it presents the Shifting Bottleneck heuristic used in this thesis's model in detail. Finally, a presentation and discussion of the extensions and appliances of the literature method conclude this chapter.

2.1 Introduction to Scheduling

According to Graves (1981), scheduling is the allocation of available production resources over time to best satisfy some set of optimization criteria. In the classical application of production scheduling, these resources are automated machines or manual workplaces used to produce jobs. Jobs are generally composed of operations, which the given workplaces process, needing a specific processing time to do so. A schedule then is an allocation of operations and time slots in which suitable workplaces do the processing. So-called Gantt-Charts can depict schedules, which makes them easy to read and understand. Figure 2.1 shows an exemplary schedule, once job-based and once machine-based. Both show a schedule, including four jobs and three workplaces.

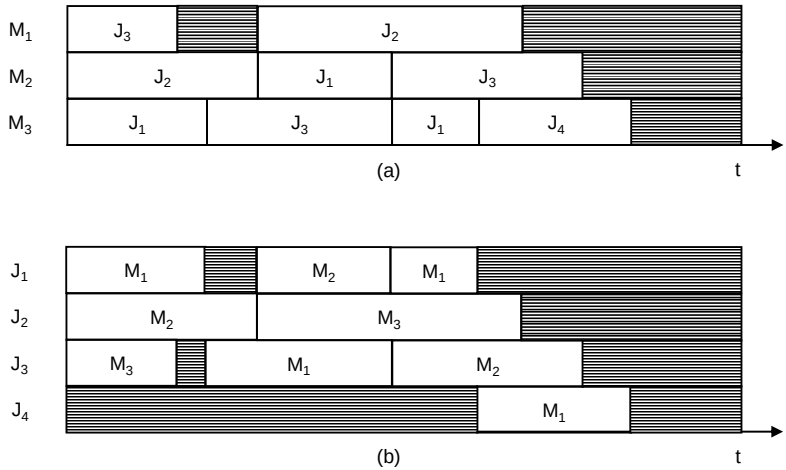

Figure 2.1: Gantt-Chart of a schedule, once machine-based (a), once job-based (b) (Brucker 2007, p. 2)

2.1.1 Notation

The number of jobs and workplaces has to be finite in every scheduling problem (Pinedo 2012, p. 13). A job is denoted by n and a workplace by m. N and M stand for the set of all jobs and workplaces. Subscripts are given as j for jobs and i for workplaces. A pair (i, j) describes the processing of job j on workplace i and is also called an operation o. Operations of a job number from 1 to k. When the last operation of a job finishes, the job itself is considered finished. In the basic case, the numbering means that operations process in that order, and the sequence is fixed. From these basic definitions, some more definitions regarding jobs are possible. The operation o always has a processing time t_p, representing the time needed to complete the processing of job j on workplace i. A job may have a release date r_j if it is not available at time 0. The release time represents the earliest time the job can start processing its first operation. Jobs may have a due date d_j. It stands for an externally given date when the job has to be completed, not to be late. Both due dates and release dates as defined here are for the whole

production system. They also exist locally at a workplace using the same notation, where they mean the earliest time an operation can start or has to finish at that workplace in order to comply with the global value.

2.1.2 Optimization Criteria

The optimization criterion defines the goal of the scheduling process that the scheduling algorithm tries to achieve. It has the form of an objective function, which is always formulated so that it is to be minimized in scheduling problems. A due date does not exist for every scheduling problem. In case jobs have due dates, the objective function bases on these. If scheduling is done without due dates, other criteria, for example, the completion time of jobs, are chosen. Many optimization criteria can apply to a scheduling problem. Here, we present only a few of the more common ones in the following table. Definitions of the criteria stem from Pinedo (2012, p. 18-19).

Name	Variable	Description
Makespan	C_{max}	Optimizes the completion time of the last job in the schedule.
Maximum lateness	L_{max}	Optimizes the worst violation of a due date for all scheduled jobs.
Total completion time	$\sum C_j$	Optimizes the sum of all completion times, possibly weighted.
Total tardiness	$\sum T_j$	Optimizes the sum of all due date violations, possibly weighted.
Number of tardy jobs	$\sum U_j$	Optimizes the number of jobs which violate their due date, possibly weighted.

Table 2.1: List of relevant optimization criteria (taken from Pinedo (2012))

There are also more complex objective functions, which combine several of the above goals or extend them. For this thesis, however, the above-mentioned objective functions are sufficient as we will only use them.

2.1.3 Shop Models

From the given notation, fundamental scheduling problems, also called shop models, can be derived. Here we will present the three standard models, called flow shop, job shop, and open shop, and some of the extensions made to them in the literature. The way the operations within a job are sequenced defines the three models. All of them assume that precisely one workplace can process a given operation. For a more detailed description of each of the three basic shop models, we refer the reader to Blazewicz et al. (2019), who explain the models in detail.

Flow Shop

The flow shop is the most simple of the three models. Here the flow of material is always the same. The flow direction is the same for every job that needs to be scheduled and is known beforehand. The only possible exception is the skipping of a workplace; going backward in the flow is not allowed, as shown in figure 2.2 showing an exemplary flow shop.

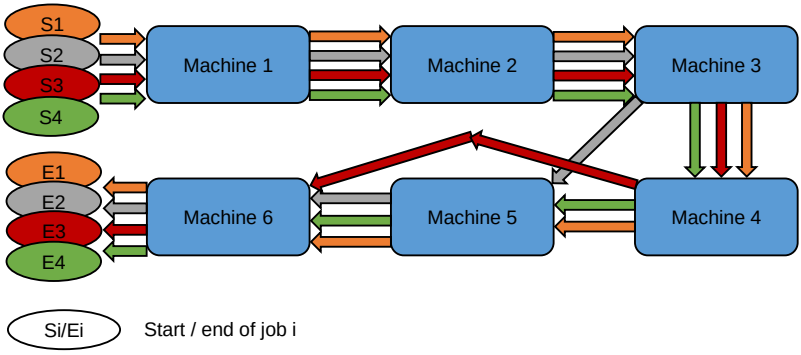

Figure 2.2: Exemplary flow shop

Job Shop

The assumptions in a job shop are less restrictive than in the flow shop. Here, an external source still gives the sequence of operations in a job, but various jobs can have different sequences. Thus, the order of workplace visits is still known before scheduling. However, material flow through the system differs because every job has its own route. Figure 2.3 shows an exemplary job shop.

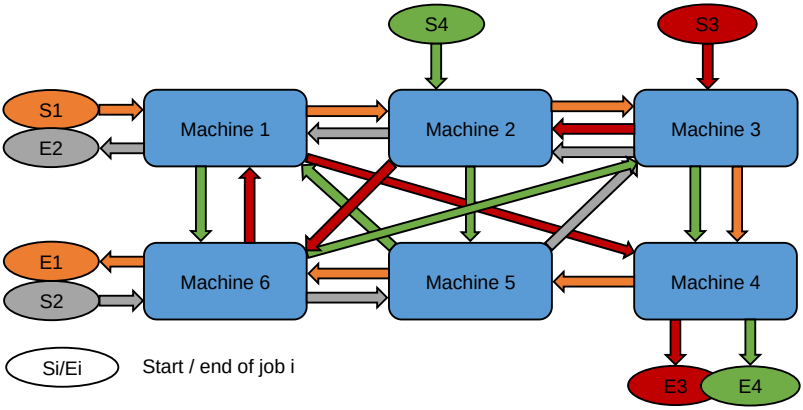

Figure 2.3: Exemplary job shop

Open Shop

Different from the other two models, the open shop relaxes the property that the sequence of operations is predetermined. In an open shop operations can instead be completed in any order. Since there are no precedence relations between a job's operations, workplace visits of two jobs from various products differ, like in the job shop. In contrast, the order of workplace visits also differs between identical jobs. The open shop allows the processing of all operations of the job in any order. Figure 2.4 shows an example where one job is processed in five different ways.

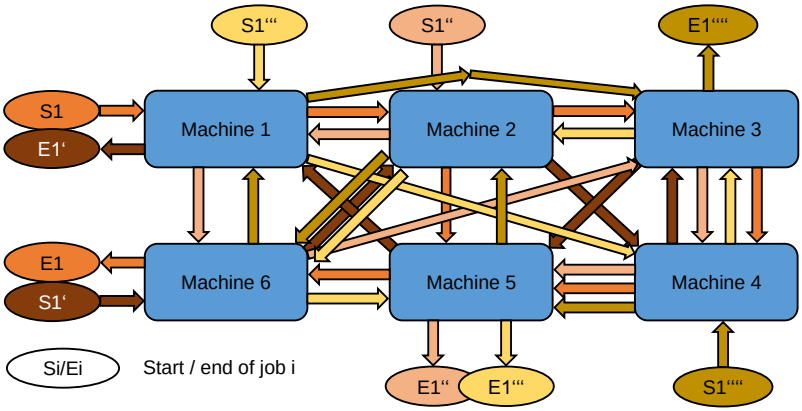

Figure 2.4: Exemplary open shop

More Advanced Shop Models

Numerous possible extensions to the three presented basic models are possible. Since the basic models are very restrictive, many additional features were developed to make the models better applicable in practice. In this section, we present some of them relevant to the model developed in this thesis. The presented models do not have a unique name in most cases. However, when authors mention more advanced shop models, they are sometimes named by their new features or complex or practical job shops. Table 2.2 gives an overview with a detailed description following afterward.

Name	Description
Flexible shop	Multiple machines offer the same process
Partial shop	Mix between a job shop and open shop
Transport	Includes transportation times
Setup	Includes setup times
Buffers	Slots for waiting jobs are limited
Stochasticity	Times in the model are stochastic

Table 2.2: List of advanced shop models

The first extension is the so-called flexible shop (for reference see Pinedo (2012, p. 15)). It is possible to make in any of the three basic models. In the three basic models, every workplace is unique and only available once. In the flexible shop, there may be more than one workplace processing a given operation. This situation leads to the problem that the scheduler does not know before scheduling which workplace will process which operation of a job. The flexible shop allows identical parallel workplaces in its easiest form, which have the same processing time for each operation. A more complex version allows non-identical processing times for the same operation at different workplaces. An even more complex variation is allowing one workplace to process more than one type of operation. As a practical example, we could have a system where workplace one is a sawing machine, but machine two can saw and drill. The scheduling problem is now more complicated than before because all parallel workplaces share the same subset of operations from a given set of jobs in the flexible shop. With the multi-purpose workplace extension (see Brucker (2007, p. 293ff.)), workplaces only share some of the operations, or they share them with different subsets of workplaces. In the case of these features, the scheduling problem splits into two parts. Then the first step only determines which workplace processes which operation. Then in a second step, the regular scheduling would take place. It is also possible to do both steps together which will most likely lead to better results as it offers an algorithm more optimization potential.

If some of the operations in a job have a fixed sequence but others do not, the resulting model is a mix between a job shop and an open shop (see Brucker (2007, p. 226)). This shop type has different names in the literature; we will refer to it as a partial shop, as for example in Amin-Naseri and Afshari (2012). Another feature regarding the sequence of operations in a job is to include recirculation or re-entrance of jobs. Then, one job can visit a workplace or do an operation more than once during its process sequence.

The only time given in the basic models is the processing time of a job. There can be additional steps in the processing of a job that require time usage. They could, for example, be transport times between workplaces or

setup times on a given workplace. Setups are mostly modeled sequence-dependent, meaning that the specific time needed for setup depends on the current and the next job to be processed. Furthermore, setup can be done parallel to processing or only in between the processing of two jobs. Both setup and transport could also be resource-limited so that only a fixed number of transport vehicles or a given number of tools is available.

Another important point is the inclusion of waiting areas at or between the workplaces. The basic models assume that there is an infinite amount of space for waiting available at every workplace. If the modeled system has limited buffer space, every job has to be at a waiting spot or in transport at any given time between the processing of two operations. A general assumption is that a job uses input buffer slots before processing an operation on a workplace starts and output buffer slots after processing. A buffer slot may also be used for both, depending on the model. A buffer slot may only be used by the workplace which it is allocated to and not by any other workplace. There can also be buffer slots allocated to no workplace, which are usable by every workplace. Limited waiting space can lead to blocking (see Pinedo (2012, p. 17) where a job cannot leave the current workplace since the next workplace in the process step has no space in its buffer to accommodate the job. The current workplace is then blocked if it cannot continue processing because all buffer slots are full. Limited waiting space can also lead to deadlocks where two workplaces want to transport two jobs between each other, but both cannot start because all buffer slots are full. What a deadlock exactly is will be described in chapter 3.

As the last extension, every aspect of a scheduling model is either deterministic or stochastic (see Pinedo (2012, p. 245ff.)). In deterministic models, every event or time is known beforehand, while in stochastic models, events or times occur according to a probability distribution. That includes any time used in the model, which can be either faster or slower than the planned average, and workplaces' downtime in which they are not available for processing.

Later in this work, in chapter 4, these extensions will be used again. There, the scheduling model underlying this thesis will be defined.

2.2 The Shifting Bottleneck Procedure

The algorithm presented in this work bases upon the Shifting Bottleneck procedure (SB) published by Adams et al. (1988). A numerical example of the procedure can be found in Pinedo (2012, p. 196ff.). The procedure applies in the standard job shop scheduling model. It follows an iterative heuristic approach, which is why we also call it SB heuristic, where the main idea is to identify a bottleneck in each iteration. In the following iteration, all other machines then take into account the local schedule of the bottleneck. This section describes the procedure as Adams et al. (1988) released initially. Table 2.3 summarises all variables used in the description of the procedure.

Variable	Description
N	Set of all jobs to be scheduled
M	Set of all machines in the job shop
O	All operations of jobs N
p_i	Deterministic processing time of operation i
t_i	Start time of operation i
r_i	Earliest time operation i can be started
f_i	Latest time operation i should be finished to not increase the objective function value

Table 2.3: List of variables used in the Shifting Bottleneck procedure

A set of $N = \{0, 1, ..., n\}$ jobs needs to be scheduled on M machines. The jobs are described by a number of process steps. Each process step is assigned to a machine and the order in which the process steps of a job have to be completed is known. The set O_n contains all process steps of job n, also known as operations, and the set O contains all operations to be planned. O_m contains all operations which are completed on machine m. Each machine can only process one operation at a time. The time needed for processing p_i is known and fixed and cannot be interrupted. In the optimization problem, t_i represents the start time of operation o_i. The goal is to create a schedule that minimizes the makespan given by the objective function

which minimizes the latest start time of any operation. The formalization of this problem looks as follows:

$$\begin{array}{lll} \min & \max_{i \in O} t_i & \\ \text{s.t.} & t_j - t_i \geq p_i, & j > i, \forall (i,j) \in O_n, n \in N \\ & t_i \geq 0, & \forall i \in O \\ & t_j - t_i \geq p_i \vee t_i - t_j \geq p_j, & \forall (i,j) \in O_m, m \in M \end{array} \tag{2.1}$$

The three constraints in this problem ensure a job shop. The first states that the difference in start times of two operations of the same job must be larger or equal to the earlier operation's processing time. The second states that start times can only be in the future, while the last one forbids that two operations can be processed simultaneously on a machine. A solution is achieved by deploying the following iterative approach, where M_0 contains all machines that are already sequenced (M_0 is empty at the start):

Step 1: Update the release times r_i and the due dates f_i of every operation.

Step 2: Create a sequence on all machines $M \backslash M_0$ and identify a bottleneck machine m among them. Set $M_0 = M_0 + m$.

Step 3: Re-optimize the sequence of each critical machine M_0 while keeping the other sequences fixed. Then, if $M = M_0$ stop, otherwise go to step one again.

In the following, each of the three steps is described in detail.

Updating the Release Times and Due Dates

At the start of the procedure, the initial makespan is equal to the longest job's total processing time in N. In the iterative solution procedure, each operation i is assigned a release time r_i, which describes the time it can be processed the earliest. This time calculates by summing up the processing times of all previous operations. If any previous operation is already scheduled, r_i calculates by taking the start time of the closest predecessor already scheduled and summing the processing times from there. The due

date f_i is the latest time the operation has to finish not to increase the complete schedule's makespan, which can be calculated similarly to the release time by taking the current makespan and subtracting the preceding processing times.

Identifying the Bottleneck

Before identifying the bottleneck machine in step 2, the local scheduling problem is solved for every machine using the algorithm of Carlier (1982), which will be explained subsequently. At every point during the creation of the local schedule, a priority dispatching rule is applied. The rule takes all operations available to be scheduled at time t (all operations with $r_i \leq t$) and chooses the one with the earliest due date f_i. In case of a tie, it takes the one with the longest processing time on the current machine. If there is no operation available at time t, but there are still operations to be scheduled, t is set to the minimum r of all operations to be scheduled. In the following formalized form of this algorithm, U is the set of operations already scheduled and $\overline{U}$ the set of all unscheduled operations:

Step 1: Set $t = \min_{i \in I} r_i$; $U = \emptyset$.

Step 2: At time t, from amongst the ready operations $i (r_i \leq t)$ of $\overline{U}$, schedule the operation j with the smallest f_i.

Step 3: Set $U = U \cup \{j\}, t = \max(t + d_j, \min_{i \in \overline{U}} r_i), \overline{U} = \overline{U} \backslash \{j\}$. If $\overline{U}$ is empty, the algorithm is finished. Otherwise, go to step two.

After scheduling all machines with this algorithm, the bottleneck has to be identified. For this, the makespan of each machine is calculated. Since aiming for the shortest makespan is equal to minimizing the maximum lateness of any job on the given machine regarding f_i, each operation's end time in the schedule is compared to its due date f_i. The maximum on a given machine equals the makespan delay of that machine. The bottleneck then is the machine with the highest makespan delay of all machines. In case of a tie, one can choose any of the machines at random.

Re-optimizing the Sequences

At the end of each iteration, the local re-optimization uses the same algorithm as the local sequencing. The local sequencing repeats for every machine that has already been the bottleneck of an iteration. If there was an improvement, the old schedule is replaced. Otherwise, no action is taken. The re-optimization cycle finishes once every bottleneck machine has been rescheduled. If there was an improvement during the current cycle, another one starts. Adams et al. (1988) however arbitrarily limit the number of cycles to three up to the point where M_0 contains all machines. Then, in the last iteration, the cycle only stops if there was no improvement for an entire cycle. During the re-optimization step, machines are rescheduled in order of their makespan value, starting with the highest—their order updates after each cycle.

Adams et al. (1988) propose an additional measure before starting the next iteration, which is only mentioned here for the sake of completeness of the method since the algorithm presented in this work does not use it. They found it helpful to repeat the procedure after removing some non-critical machines with a maximum of $\sqrt{|M_0|}$ machines removed. They define a machine as non-critical if none of the operations done on this machine is on the schedule's critical path. If there are more non-critical machines than the maximum allowed number, those with the lowest makespan are removed. After the re-optimization cycle, the machines are added one by one until M_0 contains all machines again. Then the next iteration starts.

2.3 Extensions of the Shifting Bottleneck Procedure

Since Adams et al. (1988) published the Shifting Bottleneck heuristic more than 30 years ago, numerous extensions to it have been made. Here, we want to give a short overview of the most relevant ones. The advanced scheduling models presented in chapter 2.1.3 are the basis for most exten-

sions. The extensions to the Shifting Bottleneck procedure divide into two groups. The first group extends or changes only the local workplace scheduling subproblem, while the second changes the procedure itself. The following two sub-chapters are not supposed to give the reader a complete overview of all the publications related to the Shifting Bottleneck procedure. Instead, we focus on those most relevant to our work. The publications not mentioned are applications of the procedure without adding new features or replacing large parts of the procedure with different algorithms, whereas we are only interested in publications that remain somewhat close to the original procedure. The results from this section will be discussed later in chapter 4.

2.3.1 Changes to the Single Workplace Subproblem

Changes to the local sequencing are as common as changes to the procedure itself. They are because many of the advanced shop models presented in chapter 2.1.3 can be implemented into the SB without changes to the procedure itself and only require more advanced local sequencing algorithms. An overview of all publications presented in this section can be found in table 2.4 and 2.5. Both tables list the same publications but separate after specific characteristics. The columns of the tables stand for the type of change or additional characteristics added in the publication.

The goal of the original Shifting Bottleneck procedure is the minimization of the makespan. Since jobs in a real-world production system often have delivery or due dates, much research has been done to incorporate a different optimization function within the same procedure. Pinedo and Singer (1999) change the objective function to minimize the Total Weighted Tardiness of the jobs. Mason et al. (2002) present a solution with the goal of total weighted tardiness minimization as well. They use the heuristic in their complex job shop application, which extends the job shop model with batch workplaces, setup times, parallel workplaces, and re-entrant process flows. In Mason and Oey (2003) the published heuristic is slightly changed

Publication	Additional/Changed features						
	Different objective function	New local sequencing approach	Transport time	Dependent Jobs	Setup time
Ovacik and Uzsoy (1992)	X	X			X
Dauzère-Pérès and Lasserre (1993)				X	
Balas et al. (1995)				X	
Dauzère-Pérès (1995)				X	
Uzsoy (1995)	X				
Holtsclaw and Uzsoy (1996)		X			
Ivens and Lambrecht (1996)	X		X		
Ramudhin and Marier (1996)					
Yoo and Martin-Vega (1997)		X			
Schutten (1998)	X		X		X
Sun and Noble (1999)	X	X			X
Mason et al. (2002)	X				X
Wenqi and Aihua (2004)		X			
Sourirajan and Uzsoy (2007)	X			X	
Balas et al. (2008)	X	X			X
Topaloglu and Kilincli (2009)	X				
Bilyk and Mönch (2012)	X	X			
Braune et al. (2012)	X	X			
Scholz-Reiter et al. (2013)	X	X			
Braune and Zäpfel (2016)	X	X			
Cayo and Onal (2020)	X	X			X

Table 2.4: Overview of the changes to the single workplace sub problem and their contents (part 1)

Publication	Additional/Changed features					
	Re-Entrance	Parallel workplaces	Batch workplaces	Work-in-Process	Shop types	
Ovacik and Uzsoy (1992)						
Dauzère-Pérès and Lasserre (1993)						
Balas et al. (1995)						
Dauzère-Pérès (1995)						
Uzsoy (1995)			X			
Holtsclaw and Uzsoy (1996)						
Ivens and Lambrecht (1996)		X		X	X	
Ramudhin and Marier (1996)					X	
Yoo and Martin-Vega (1997)						
Schutten (1998)		X	X		X	
Sun and Noble (1999)					X	
Mason et al. (2002)	X	X	X			
Wenqi and Aihua (2004)						
Sourirajan and Uzsoy (2007)		X	X			
Balas et al. (2008)					X	
Topaloglu and Kilincli (2009)	X					
Bilyk and Mönch (2012)						
Braune et al. (2012)					X	
Scholz-Reiter et al. (2013)						
Braune and Zäpfel (2016)						
Cayo and Onal (2020)		X	X			

Table 2.5: Overview of the changes to the single workplace sub problem and their contents (part 2)

since batch processing and delayed precedence constraints sometimes lead to an infeasible schedule.

Dauzère-Pérès and Lasserre (1993) update the local sequencing with a new heuristic method which updates the release times of all unplanned operations each time an operation is planned during the local sequencing. The update is done because there may be dependent jobs at the same workplace, which could lead to the situation that the delay of one operation influences another operation's release time at the same workplace. This is, for example, the case if the same job visits a workplace more than once. Balas et al. (1995) solve the same problem as Dauzère-Pérès and Lasserre (1993), but instead of a heuristic, they choose an exact approach for the problem with a branch-and-bound method, which Dauzère-Pérès (1995) also uses. Braune and Zäpfel (2016) propose new heuristics for the Total Weighted Tardiness criterion with the specific aim of solving larger problem instances. Before, in Braune et al. (2012), they proposed an exact algorithm for the single workplace subproblem solution with the same objective. Yoo and Martin-Vega (1997) used a new subproblem procedure for the number of tardy jobs objective function.

Ivens and Lambrecht (1996) extend the procedure with many characteristics that apply in real-world scheduling problems. These include transport times between workplaces, setup times, parallel workplaces, and planning not only in empty but also in production systems with work-in-process. They also include batch processing and the assembly and split of jobs during production. Schutten (1998) follows a similar approach and extends the procedure with more practical features. He takes setup times, parallel workplaces, transport times, orders with more than one job, additionally needed resources for processing, downtimes, and the assembly or split of jobs into account. Ramudhin and Marier (1996) take the different shop types into account, applying the SB to open shops, partial shops, and assembly shops. Sun and Noble (1999) update the single workplace problem with a heuristic for sequence-dependent setup times. Wenqi and Aihua (2004) propose a new single workplace algorithm, which they call Schrage algorithm with a disturbance. They deal with the particular problem of prioritizing be-

tween two jobs where job one has the earlier release date, but job two has the earlier due date. Thus, they reduce the due date of jobs that release in the future by multiplying the time between the end of the current schedule and the job's release with a coefficient and then subtracting the result from the original due date. With this change, they can obtain better results than with the original scheduling algorithm. Sourirajan and Uzsoy (2007) use a revised SB procedure. The revised procedure schedules single workplaces, parallel workplaces, and batch processes. They integrate the idea of Balas et al. (1995) and extend the batch processing algorithm of Uzsoy (1995) for parallel workplaces. Topaloglu and Kilincli (2009) propose a new single workplace method to schedule a re-entrant job shop with the Total Weighted Tardiness goal.

In addition to extending the local sequencing problem to include complex characteristics, some research was done to replace the single workplace scheduling with different approaches. Ovacik and Uzsoy (1992) apply the SB heuristic for semiconductor testing operations. They decompose the system into work centers scheduled with the Jackson heuristic and an extended local search. They specialize in testing work centers that also include setup times. In Holtsclaw and Uzsoy (1996) again, two different local subproblem solution procedures are tested, one providing an optimal, one a heuristic solution. Balas et al. (2008) extend their algorithm from 1995 with sequence-dependent setup, deadlines, and precedence constraints. They use an adapted Travelling Salesman Problem for the single workplace subproblem, which can be solved efficiently. Scholz-Reiter et al. (2013) replace the subproblem solution and re-optimization step with a variable neighborhood search. Bilyk and Mönch (2012) also use a variable neighborhood search but implement it in parallel to reduce computation time. One last relevant use of the SB heuristic comes from Cayo and Onal (2020). They use the heuristic to schedule a group of work centers but use different local algorithms to schedule each of them. The applied algorithms mainly include different dispatching rules but also insertion algorithms.

2.3.2 Changes to the General Procedure

In contrast to chapter 2.3.1, this section's publications alter the SB procedure or use it differently. That includes changing the bottleneck selection criterion, exchanging parts of the procedure for new methods – most often the re-optimization step –, introducing new features to the heuristic, or taking measures to reduce problem complexity. An overview of all referenced publications and their topic can be found in table 2.6.

Holtsclaw and Uzsoy (1996) test different bottleneck selection criteria including random, most workload, maximum lateness from two different local solution procedures, and a pre-emptive earliest due date approach. Aytug et al. (2003) work on the same topic. They include static measures, namely the Total Machine Load, the Average Remaining Operations to Completion, the Average Remaining Processing Time, and dynamic measures. For the dynamic criterion, they calculate an infeasibility profile for every workplace. The profile includes the number of jobs processed at a workplace at any given time. If the number is greater than one, the schedule is infeasible. From this infeasibility, they develop three criticality measures: The Maximum Infeasibility, Total Infeasibility, and Average Infeasibility of a workplace. Using these measures can reduce computing time by up to 20% compared to the static measures. Mönch and Zimmermann (2007) use simulation to test different workplace criticality measures for which they combine different measures into a single weighted sum where a simulation then optimizes the weights. Pfund et al. (2008) use a multi-criteria approach for the SB, which they based on Mason et al. (2002). They put together the objectives makespan, cycle time, and total weighted tardiness into a desirability function. It is then used on the single workplace subproblem and the workplace criticality level. They can show that the combined approach manages to minimize all three objectives simultaneously.

Lawrence and Sewell (1997) include uncertain processing times into the SB heuristic. To do so, they create an initial schedule with the expected value of the processing time and then update the schedule after each successful processing step according to the difference between expected and actual

Modified element	Additional/Changed features					
	Bottleneck Selection	Stochasticity	Runtime improvement	Reoptimization	Transportation constraints	Limited Buffers
Holtsclaw and Uzsoy (1996)	X					
Lawrence and Sewell (1997)		X				
Ovacik and Uzsoy (1997)			X			
Yoo and Martin-Vega (1997)			X			
Mehta and Uzsoy (1998)		X				
Pezzella and Merelli (2000)				X		
Cheng et al. (2001)	X					
Singer (2001)			X			
Mason et al. (2002)			X			
Aytug et al. (2003)	X					
Mason et al. (2004)		X				
Chen et al. (2006)			X			
Upasani et al. (2006)			X			
Yeung and Mason (2006)				X		
Mönch and Zimmermann (2007)	X					
Sourirajan and Uzsoy (2007)			X			
Pfund et al. (2008)	X					
Bülbül (2011)				X		
Driessel and Mönch (2012)					X	
Liu and Kozan (2012)				X		
Q. Zhang et al. (2014)					X	X

Table 2.6: Overview of publications to the general procedure and their focus

processing time. Mehta and Uzsoy (1998) use the procedure as the first step for their heuristics to include workplace breakdowns into the scheduling process. They create an initial solution with the SB and then apply their heuristic to create idle time in the schedules to even out potential breakdowns. Mason et al. (2004) consider general deviations from the created schedule. They tested three different rescheduling strategies to handle the deviations from the created schedule. The first, Right-Shift Rescheduling, delays the whole schedule by the duration of the interruption. Fixed-Sequence Rescheduling does not push all operations back by the delay duration but instead uses existing gaps in the original schedule to even out the delay. The last strategy is Total Rescheduling, which starts the scheduling process again with all remaining operations. They confirm that Total Rescheduling leads to the best results at the expense of computing time, whereas the other two strategies are faster but lead to 10-30% worse results.

Driessel and Mönch (2012) use the SB for complex job shops and add transportation constraints to it. They first calculate a production schedule and afterward compute if a capacity-limited transportation system can handle all required transports. Their goal is to determine the required amount of transport vehicles for the given production schedule. Q. Zhang et al. (2014) also solve the scheduling problem with the SB heuristic and then add a second heuristic to schedule transportation tasks afterward. If the heuristic finds no feasible transportation schedule, the scheduling starts again. They also allow the storage of orders between processing steps. Three storage types are considered: No wait, no storage, or storage allowed. In the first one, jobs are not allowed to wait between two operations, whereas in the second, jobs can wait but only at workplaces since there are no storage places. If storage places exist and storage is allowed, jobs may only wait up to a given maximum time between two processing steps. In the case of storage allowed, its capacity at workplaces is always infinite in this publication.

Pezzella and Merelli (2000) extend the SB with a Tabu Search. They use the SB solution as initial input for a Tabu Search to improve the results. Yeung and Mason (2006) change the re-optimization step. Instead of re-optimizing every time, they use a real option analysis to value the option of

re-optimizing and only do it if their model gives a promising result. They can reduce the computation time for the Total Weighted Tardiness objective while simultaneously getting good results. Bülbül (2011) changes the re-optimization procedure to a Tabu Search for the total weighted tardiness objective. Liu and Kozan (2012) solve a parallel workplace job shop scheduling problem with a hybrid approach between the SB heuristic and a Tabu Search. They also present a topological sequencing algorithm that effectively calculates the release and due dates for the single workplace subproblems. The hybrid approach is using a Tabu Search for the re-optimization step in the SB procedure.

Yoo and Martin-Vega (1997) apply the same decomposition technique for semiconductor production, which Mason et al. (2002) also uses. Unlike them, however, in this publication, the goal criterion is the number of tardy jobs. A so-called General Algorithm Framework does the scheduling of the decomposed work centers, which schedules a set of single workplace problems. A publication entirely focused on decomposition strategies was published by Ovacik and Uzsoy (1997). The ideas were later used again by Sourirajan and Uzsoy (2007). They use a revised procedure with a rolling horizon, where jobs group according to their availability date and only one group is scheduled at a time. Cheng et al. (2001) use the SB to schedule a parallel workplace flow shop. The bottleneck is not identified on a single workplace basis, but instead, a stage of the flow shop becomes the bottleneck. To achieve better results, they also propose a new parallel workplace scheduling heuristic for minimizing the maximum lateness.
Singer (2001) published a time horizon decomposition approach. It splits operations into groups planned for a certain time window, thus reducing the problem's complexity. Also, it allows some overlapping of time windows to improve the results. Upasani et al. (2006) reduce the problem size of the decomposition approach by partitioning the work centers into two groups, one of high utilization and one of low utilization. The SB is only done on the high utilization work centers, whereas the low utilization work centers bundle into a single infinite capacity workplace scheduled with a dispatching rule. Chen et al. (2006) publish an approach for the scheduling of job

shops with non-identical parallel workplaces. To increase computational efficiency, they propose a new heuristic that omits the bottleneck selection part. They schedule all work centers at once with a dispatching rule and then directly start the re-optimization procedure with all work centers. They also include a procedure to determine the number of workplaces needed in each work center.

3 Multi-Agent Systems for Production Planning

There is a trend towards modeling more and more complex systems in production and logistics with decentralized solutions as they present a way to deal with the flexible and modular structure of these systems (Leitão et al. 2015). This chapter is supposed to introduce a variant of these systems, called multi-agent systems, focusing on its use for production planning. In a decentralized system, multiple (sometimes identical) independent elements work together to achieve the system's goal. The elements are usually called "agents". Chapter 3.1 presents a definition, and some essential characteristics of agents. When more than one agent is present in a system, the system is called a multi-agent system. Multi-agent systems and some implications derived from them are outlined in chapter 3.2. Furthermore, we want to show some existing agent solutions for production scheduling in chapter 3.3.

3.1 Characteristics of Agents

As used in this thesis, the term "agent" is the short form of "software-agent". Ferber defines the term agent (Ferber 1999, p. 9-10). He writes that an agent is a physical or virtual entity which is capable of acting in an environment and can communicate with other agents. It is autonomous and tries to achieve its goals. For this, it watches its surroundings. This definition is extensive because it includes every control system and even humans in it. Ferber presents more specific definitions following the general one.

However, for this work, a definition by Wooldridge fits: "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives" (Wooldridge 2006, p. 15). This definition focuses on software agents and leaves out all physical agents, which fits agents' usage in this work. The autonomous actions mentioned in the definition typically involve the agent taking input from the given environment and transform it into an output, which is given back to the environment. With this transformation, the agent tries to fulfill its goal best.

There are several different categorizations of agents concerning their abilities and features from various authors, which extends this work's purpose. One differentiation made is between a reactive and a deliberate agent. A reactive agent only reacts to the environment without representing its state towards the environment (Ferber 1994). On the other hand, deliberate agents have explicit goals and plans, which fits the description from Wooldridge. They are also able to store information and save a representation of the environment in a knowledge base. Actions from deliberate agents are based on a planning process for realizing the given goals (Büttner 2011, p. 44-45). An exemplary early approach for a deliberate agent is the BDI-model of Bratman (Bratman 1987). There are also combinations of the two types, which combine the reactive agent's fast answers with the slower but more in-depth responses from the deliberate agent.

Apart from this definition, the term 'Intelligent agent' is also often used. According to Wooldridge (2006, p. 23), a reactive agent is not intelligent. Instead, the agent also has to take the initiative itself to reach its goal and has to be able to interact with other agents. This extends the deliberate agent with the ability to communicate and interact between agents. He states that it is hard to effectively balance between the blind execution of tasks (similar to the reactive agent) and the reaction towards new situations by changing the goal or reaction pattern (p. 24). His argument about the interaction between agents leads directly towards multi-agent systems. There is a need for interaction between these agents if their decisions influence each other in any system with more than one agent making decisions.

3.2 Multi-agent Systems

A multi-agent system consists of at least two agents who interact to reach their own or shared goals (Jung 2016). In theory, a multi-agent system can consist of an unlimited number of agents. The goal of using multiple agents is to solve problems that a single agent would not be able to solve (Monostori et al. 2006). Figure 3.1 shows an exemplary structure of a multi-agent system. Each agent in this system has its own goal. If two agents have the same goal, they have to cooperate to reach it. If their goals are different, they are automatically rivals if they cannot reach their goals simultaneously. Each agent has its area of influence in the environment, which does or does not overlap with that of other agents. As one can see in figure 3.1, agents may be in a group. Between the various groups or also within a group, there may be a hierarchy of the agents. A hierarchy indicates some control of one group of agents over another group. This, however, is not depicted in the figure. According to Lima et al. (2006), apart from a hierarchical relation, two other relations between agents are possible. Agents can either be autonomous, which means that they can talk to each other directly or build a federation where the communication takes place via a mediator architecture. Of course, different relation types can also mix in a single system.

3.2.1 Interactions between Agents

Interactions mostly take part between the agents in a group, but it does not have to be limited to that. Agents of different groups or even whole groups can also interact with each other. The interaction between the agents can include but is not limited to cooperation, coordination and negotiation (Wooldridge 2006, p. 3). For agents to be able to communicate, they need to talk in the same language. There are numerous communication protocols described in the literature, which will not be presented in detail here, as we developed the communication of the presented system without using an existing protocol. This will be presented in chapter 6. In addition to a common language that defines the structure and content of messages agents send and receive, agents also need a way to exchange messages.

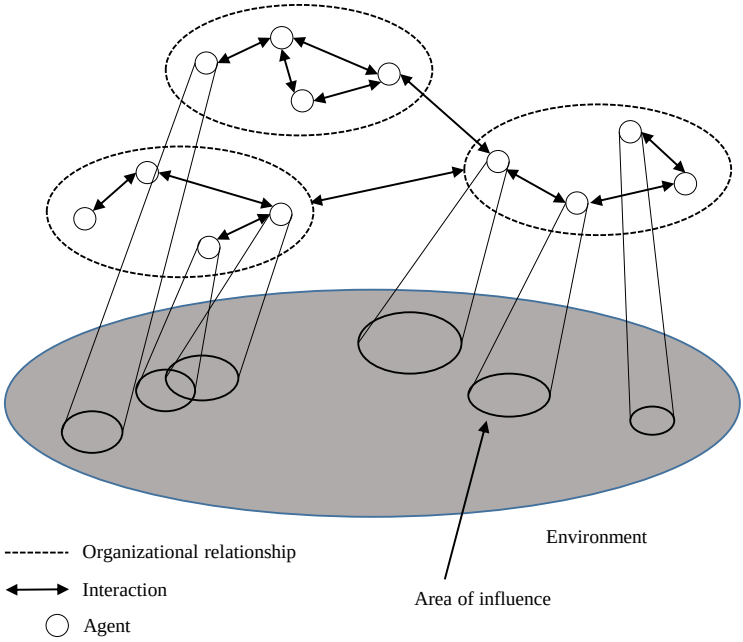

Figure 3.1: Exemplary multi-agent system (Wooldridge 2006)

This problem does not exist in centralized systems since all information is present at the only agent's location (in our case, the scheduler or scheduling agent). Measures to share data and information between the agents must exist if this is not the case. It is not only crucial that adequate data or knowledge is present in decentralized systems since it can be done without problems, as we will see. If many agents are to execute local algorithms, but the agents' results are dependent on each other, the system must guarantee that the independent agents achieve fitting results. This will be discussed further in the chapter 3.2.2 about coordination. For sharing information, two different ways are common in multi-agent systems, shown in figure 3.2.

The first is the blackboard approach (shown on the left), a decentralized system in which the data storage is centralized. Mandal et al. (2010) de-

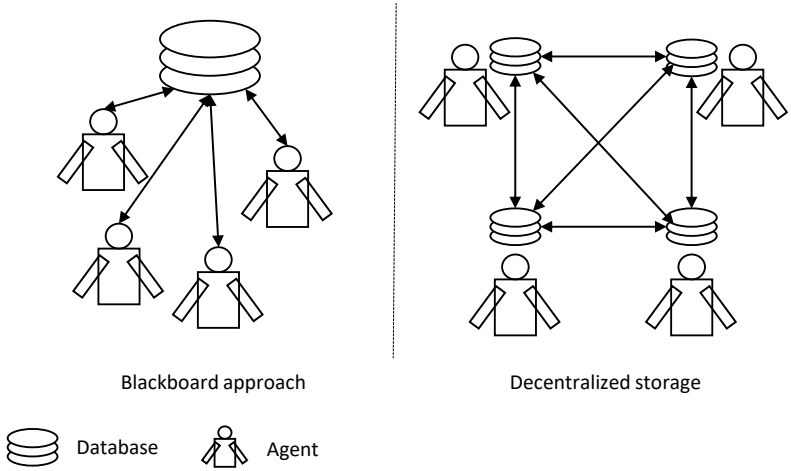

Figure 3.2: Examples for data storage

scribe it in more detail. A communication artifact or blackboard serves as a shared central information pool in which agents can store data and messages. All agents can then retrieve information from the blackboard. It makes the coordination between agents easier without taking away the agents' independent decision-making (see also Lee and Kim (2008)). The second way (right side of the figure) works without such a tool. It stores data directly in each agent. That creates a need for a feature to exchange new information between the agents to keep all databases in synchronization and to ensure that agents only get the information relevant to them. For this, event systems were developed. In them, each message (called "event") created by an agent contains a topic. Agents can subscribe to topics and so only receive the messages relevant to them. An event-managing agent is responsible for receiving and distributing the messages without processing them. This type of event processing is called a publish-subscribe system. A detailed introduction to these systems can be found in Tarkoma (2012). One can also use it in combination with the blackboard approach where messages can be used, for example, to tell other agents to start or stop working.

An essential point for the interaction is that agents have to understand the messages sent between them. If the agent system is developed newly from the start, this is not a problem. However, if agents from various projects or developers interact, a common language or ontology is necessary. An example of a formalized system is the contract net protocol by Davis and Smith (1983) which resembles an auction. In this thesis, we created the interaction between the agents without considering any existing systems. Therefore, we will not describe any protocol in detail here.

3.2.2 Coordination and Planning

A reason for using the described communication among the agents is the coordination between them. Coordination is the problem of managing all inter-dependencies between the agents (Wooldridge 2006, p. 200). There are several approaches to ensure proper coordination between agents. They depend on the fact whether the agents in the systems cooperate or compete towards their goals. Cooperating agents work together to reach a common goal. Competing agents, however, follow different goals and only work together if needed. As this thesis's topic requires the agents' cooperation, we will focus on this aspect in the following. Cooperation can be achieved in two different ways. By dividing the goal into several sub-goals where a better result for a sub-goal leads to a better result for the overall goal, agents automatically cooperate even if they act selfishly. This can be reversed in that the different sub-goals compete with each other, whereby the better fulfillment of one target can lead to a worse fulfillment of the overall target. The second approach is to divide the possible actions, which then distribute among the agents. If all of the agents have the same goal, they need to work together to achieve it. This way, joint intentions of the agents achieve the coordination (p. 204). Systems can also use a coordination agent to monitor the coordination or to make decisions if the agents cannot settle on a solution. This approach is called Cooperative Multi-Agent Planning as stated by Torreño et al. (2017).

Without considering hierarchies, there are three possibilities for planning in a multi-agent system. The descriptions and evaluations stem from Wooldridge (2006, p. 218-219). Ferber (1999) had previously described almost the same possibilities more in-depth. The simplest way, because it eliminates all necessary coordination, is to plan centrally and only execute the plan with several agents. Here, a central planning agent creates a plan for one or several executing agents while the planning agent views the whole system. This automatically introduces a hierarchy since the executing agents will use every plan given to them by the planning agent. As a second step, the planning could also be done decentralized. To keep it easier at first, several agents contribute to one plan where they are specialists for certain parts of the plan. This can be extended to the most difficult possibility, to create individual plans decentralized. As already mentioned, coordination so that all plans fit together is especially important now. That is also what makes this step the most complicated. The agents must correct their local results if they notice that the global solution becomes infeasible, all while still trying to get the best possible (or at least a sufficiently good) global result.

When conflicts arise in a multi-agent system, either between cooperating agents, which cannot decide on the same result or between rival agents, who want to use the same resource, a fast mechanism for solving the conflict is required. For solving conflict situations, various negotiation mechanisms are used. They include auctions, one-to-one negotiation, bargaining and argumentation-based negotiation (Monostori et al. 2006). Again, this thesis will use a newly designed system, which does not completely fit any of the categories mentioned here, as we found none of the existing system fits our needs perfectly.

Apart from conflicts about the planning result, there can also be conflicts within the organizational structure. Those lead to deadlocks or livelocks of the multi-agent system. Tanenbaum and Bos (2015, p. 439) give a formal description of deadlocks: "A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause". An example of this would be two people standing across each other

in a very narrow hallway. Both want to continue onwards but cannot do so since the other person is in the way. They are therefore blocking each other and causing a deadlock. It is either possible to recover from an existing deadlock or to avoid one occurring in the first place. An agent system needs to guarantee one of the two possibilities. Otherwise, it can lead to situations where the system fails, and manual intervention becomes necessary. There are two more related concepts. The first is the livelock, which we already mentioned. Explaining it with our hallway example again, now the two persons have enough space to go past each other. However, as they stand across each other, one moves to the right, while the other moves to the left simultaneously. They are now standing across each other again, and therefore reverse their actions. This creates a cycle in which both are still acting, but a solution to the problem is never acquired. The last point related to these two concepts is the starvation of a process. Starvation means that a process is available to be worked, but this never happens since higher prioritized processes always cut in front of it. For an agent system, this could be a communication request from one agent to another, which is never answered since the requested agent deems other work more important. Then, the requesting agent is waiting for an answer endlessly and is starved. We will also look at this topic and preventing it when discussing deadlocks and livelocks in chapter 6.3.

3.3 Existing Multi-agent Systems in the Literature

Having now presented an introduction to multi-agent systems, in conclusion, several existing multi-agent systems for production planning are presented. They divide into two categories—one for systems connected to the SB heuristic and one for those not. The most relevant parts for understanding the architecture of a multi-agent system are the types of agents present in the system and which of those agents are intelligent and which are not. A further topic of interest is the communication between the agents in the

given system. Here, we will not explain the presented systems in full detail. Some of them are very similar so that only the differences will be highlighted. For others, we mention only relevant or essential aspects of the publication. Furthermore, since there are numerous multi-agent systems published in the literature, this section focuses on publications that also contain the scheduling aspect in the multi-agent system, which is the point of interest of this thesis.

3.3.1 Applications of the SB Heuristic in Agent Systems

The first to use the SB heuristic in agent systems were Brandimarte et al. (2000). They use a two-level architecture consisting of local schedulers for the workstations and a material coordinator above them. Figure 3.3 depicts the schema. They assume that every job has a given release and due date. The coordinator determines a time window for each operation from the job time window and sends it to the respective local scheduler. Each time window's start and end times serve as the release and due date for that operation on the workstation. The local scheduler then tries to create an optimal schedule from the given time windows. The scheduling finishes if the time windows are feasible for each local scheduler. If not, the coordinator updates the time windows, and the process continues iteratively until it finds a solution. This process is decentralized, and the local scheduling agents plus the material coordinator exchange the time windows and solutions through messages. The result of this algorithm is not a plan that local work centers have to adhere to but instead time windows for operations, which were proven feasible during the planning step. If a local work center cannot hold the given due date during the processing of an operation, it exchanges messages with the following work center agent to see if the delay can be buffered there.

Mönch and Drießel (2005) use an approach very close to the original SB heuristic but introduce a second level of decomposition to reduce problem complexity. They decompose the job shop into several work areas. The

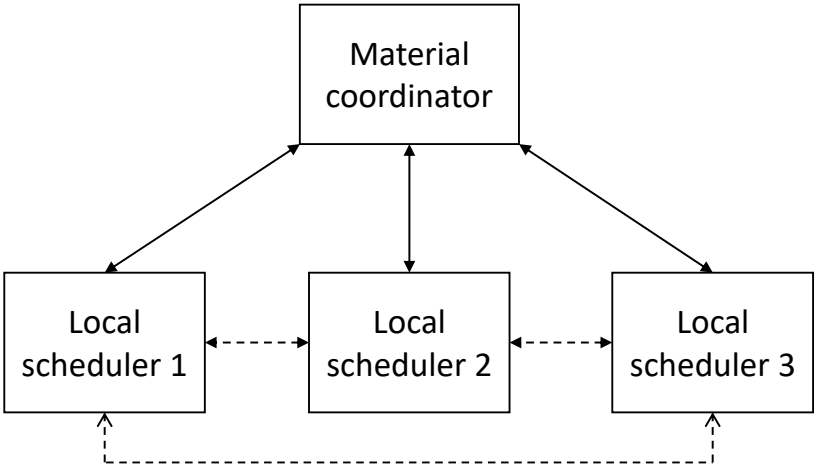

Figure 3.3: Two level architecture (Brandimarte et al. 2000)

SB heuristic is then applied on two levels. They use it to create a schedule among the work areas, where each work area uses the SB heuristic again to create a local schedule. From the local schedules, a bottleneck work area is determined. The release times and due dates of all other work areas then update with the determined bottleneck area's start and release times. From there on, the standard process of doing the local scheduling for all remaining work areas applies. This repeats until no more work areas remain to be scheduled. In this publication, there is no intention to use the system in a decentralized way. The authors call it "distributed" instead, which relates to the fact that the original bigger problem is decomposed into several more minor problems at different job shop areas while still, a central processing unit does the scheduling. This procedure was made faster by Drießel et al. (2010) where they state that the work area subproblems are independent of each other and can therefore be solved in parallel. The same group of authors also published their agent-based system for semiconductor scheduling. It was first presented in Mönch et al. 2003 and is called FABMAS. This system transports the hierarchical structure into a FAB (semiconductor fabrication plant) agent, work area agent, and work center agent. The idea is

extender further in Mönch and Stehli (2006) and Mönch et al. (2006). They operate their system with a centralized FAB scheduling agent, however.

Gavareshki and Zarandi (2011) use an agent system to improve upon a solution found by the SB heuristic. For this, they develop their search method based on a neighborhood search. Each machine gets one search agent that tries to improve the local solution towards a global optimization criterion. Iima et al. (1999) do not directly cite the SB heuristic. However, their method is very similar to a decentralized implementation of the reoptimization procedure in the heuristic. They generate an initial schedule according to the Earliest Due Date dispatching rule. Afterward, a random agent - each machine has an agent in this system - proposes a new solution. The proposal consists of choosing a random operation on the machine and producing it earlier. The agent proposes the new solution to all other agents, which can accept or refuse the new solution. If the new solution is better on a local agent's objective, the agent accepts it. The current best solution is updated in case of all agents accepting. This procedure repeats a given number of times.

3.3.2 General Agent Systems for Production Planning

Since usage of the SB heuristic in agent systems is somewhat limited, we extend our overview of agent systems for production planning to general agent systems to gather additional information about how other publications do decentralized production planning. First, we describe a few older systems and their basic ideas. Afterward, in more detail, more recent and more relevant publications are presented. An overview of the publications mentioned in this section and their characteristics can be found in table 3.1. We classified the publications according to whether the agent system is hierarchical or non-hierarchical and if the decision-making is centralized or decentralized. A separate category was included for auction-based systems, as most publications use auctions to reach decisions.

Sikora and Shaw (1997) focus on the communication between planning agents. In their system, a production line requires scheduling, but also, the

decision about lot sizes has to be made beforehand. In the first step, two agents, one for sequencing, one for lot-sizing, work together to create a line schedule. This is then extended by applications to a system with multiple lines and re-scheduling practices if changes to the system happen during a planned time horizon. This approach can be seen as one of the first publications implementing agents for production planning. It only contains scheduling agents and leaves out agents to control the system's workflow, which later publications also include. Another example focusing on communication is the publication by Krothapalli and Deshmukh (1999). They design three negotiation protocols that agents can use to create a schedule and then compare them to a hierarchical model. The first model used is based solely on the processing time of the tasks where the job chooses the machine with the shortest processing time. The second is a currency scheme, where jobs enter the system with some arbitrary currency. Machines calculate a price for processing a job task based on the expected completion time and their recent bids' success. The job then evaluates the prices and chooses one within the limits of its money and due date. The last model is similar to the second and additionally allows the pre-emption of other tasks depending on the required due dates. Sousa and Ramos (1999) use negotiation with a focus on problems of indecision and conflict avoidance. Indecision occurs when a resource makes a bid for a production task and is not informed about the bid's acceptance or decline before bidding for a second task, which it could produce in the same time slot. This problem is avoided by only allowing tasks to start a negotiation with resources currently not in another negotiation process.

A multi-agent system, called HOLOS, was published by Rabelo et al. (1999). It consists of a scheduling supervisor, enterprise agents assigned to every resource, so-called distribution centers (groups of enterprise agents) whose responsibilities contain choosing the best agent for an order. Lastly, the consortium consists of the enterprise agents chosen to process an order. The scheduling works with a negotiation process on an auction basis, which Rabelo and Camarinha-Matos (1994) describe in detail. Another early example for a negotiation on auction-basis is from Schoop et al. (2001). How-

Publication	Auction	Central decision making	Decentralized decision making	Hierarchical	Non-Hierarchical
Rabelo and Camarinha-Matos (1994)	X		X		X
Sikora and Shaw (1997)		X			X
Krothapalli and Deshmukh (1999)	X		X		X
Rabelo et al. (1999)	X		X		X
Sousa and Ramos (1999)	X		X	X	
Bussmann and Schild (2001)	X		X		X
Schoop et al. (2001)	X		X		X
Frey et al. (2003)	X		X		X
Leitao and Restivo (2008)	X	X	X	X	
Wang and Lin (2009)	X	X		X	
Sudo et al. (2010)	X		X		X
Renna (2011)	X	X	X	X	
Giordani et al. (2013)	X	X			X
Y. Zhang et al. (2014)		X		X	
Wang et al. (2016)	X		X		X
Klein et al. (2018)	X		X		X
Guizzi et al. (2019)	X		X		X
Maoudj et al. (2019)			X		X
Mayer et al. (2019)			X		X
Egger et al. (2020)			X		X
Groß et al. (2020)		X		X	

Table 3.1: Overview of publications of agent systems for production planning and their classification

ever, the publication does not state how offers and decisions are calculated. Bussmann and Schild (2001) present a more complex auction system. Their system consists of three agent types, one for the machines, one for the jobs, and a third for the transportation between machines. When an operation of a job finishes or a job enters the system, the job agent starts an auction. All machines configured to perform the task then make a bid based on their current workload and the number of tasks the machine can do. After collecting all bids, the job agent decides which machine to choose. If no bid arrives, it restarts the auction. One interesting point for practical application is that the machine agents manage their virtual input and output buffers. The input buffer contains all the machine's open tasks, while the output contains all jobs that finished processing and have not yet been assigned to a follow-up machine. The critical point for the management of the buffers is that they are limited in size. Therefore, machines can only make a bid for a job if they have a spot open in their virtual input buffer. This leads to the fact that deadlocks may occur during the scheduling. The implications from it for the scheduling process will be discussed later in chapter 4. Another aspect making this paper very interesting despite its age is that other publications related to it (Bussmann and Schild 2000, Sundermeyer and Bussmann 2001, and Schild and Bussmann 2007) describe this system in use in an automobile factory in Germany, the only publication of those mentioned here to do so.

Frey et al. (2003) introduce two different multi-agent systems. Both consist of machine and order agents. In the first system, orders start the scheduling process. They call each machine agent, asking for a proposal. Machine agents can refuse to either answer (decline the proposal), make a bid, or ask the order agent for a new proposal since the current one could not be understood. After collecting all answers, the order agent accepts one machine proposal and refuses all the others. The second system works exactly opposite to the first. In it, the machine agents ask for proposals after finishing the processing of a task. Then, the order agents decide whether they want to send a proposal to the machine agent or not. Afterward, they use the same procedure as in the first case. The results of both systems were

compared with a classical optimization problem. They conclude that the multi-agent system is better if the underlying production system becomes more complex and inhomogeneous and in cases of re-scheduling. Another example of an auction-based negotiation comes from Sudo et al. (2010). In this publication, again, job agents start a negotiation that machine agents respond to with a bid. Renna (2011) presents an architecture for scheduling manufacturing cells consisting of several machines. In his system, there are part agents, manufacturing cell agents, and machine agents. The scheduling process starts with the part agent determining which cell it has to visit next. Then it sends a request to the corresponding machine cell agent. The cell agent then asks all of its machine agents to send key performance figures regarding the part's processing. After obtaining all answers, the cell agent decides which machine will process the part and communicate it to the machine and the part. The performance figures considered are the time needed until the part's processing can start, the percentage of the machine's downtime, and the average deviation from the expected processing time. These three are weighted and calculated into a normalized index. Giordani et al. (2013) use an iterative procedure to deal with the problems of conflicts among the agents. A central coordinator agent starts all auctions and can change prices to improve result quality after getting all bids from the agents. This ensures that the coordinator always considers all options before reaching a decision and awarding tasks.

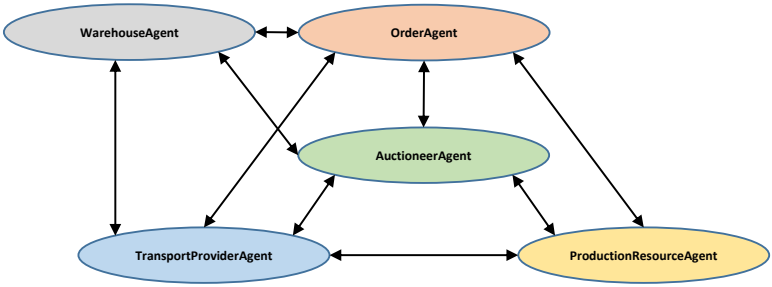

Figure 3.4: Agent system of Klein et al. (Klein et al. 2018)

Using negotiation for scheduling in multi-agent systems is not only done in older publications. Klein et al. (2018) use an approach in which an order agent tries to schedule its way through a production system by setting up auctions with the help of an auctioneer agent for each of its production steps. Bidders consist of two kinds of agents. Transport agents compete to get the transport order between production steps, and resource agents compete to get production orders. Both groups of bidding agents try to maximize their profit; therefore, they want to get as many orders as possible allocated to them. After all of the bids have arrived, the auction agent relays the best of them to the order agent. For transport, the order agent simply chooses the best offer. For production, a second step is implemented in which the order agent directly communicates with the best bidders. The offers are criticized, and resource agents get the possibility to improve their offer. This iterative procedure continues until none of the offers can be negotiated anymore. Afterward, the order agent chooses the best offer or cancels the negotiation. Guizzi et al. (2019) extend the well-known Contract Net Protocol for their interaction. This system consists of resource and job agents. Resource agents start the negotiation process. All jobs available for processing on that machine then respond with a bid containing their required processing time. The machine evaluates the bids according to the total processing time of a job, the job's residual processing time, and those two values only on the machine in question. Afterward, the machine decides to process one of the jobs. This is equivalent to a complex dispatching rule that is implemented decentralized.

The most recognized publication in this section, by the number of citations, is from Wang et al. (2016). The publication focuses on agents relevant to the scheduling process and leaves out all other agents present in the system. There are four agents for scheduling: Job agents (PA), machine agents (MA), supplementary agents (SA), and transport agents (CA). The supplementary agents are responsible for handling buffer spaces in the production. After a task finishes or a job enters the system, it starts an auction for the next task with all machine and supplementary agents. All receiving agents then determine if they are capable of doing the task and are idle. If both ap-

ply, they bid for the task. If an agent is capable but not idle, it informs the job agent about this fact. Otherwise, it ignores the message. The job agent then decides what to do next. If it receives no answer, there likely is a malfunction in the system. In case of only busy messages, the job agent waits and starts the negotiation again after a time. If there are bids, it will decide for one of them according to a set of priority rules. Before awarding the task to the chosen agent, it starts a negotiation with the transport agents. Here, however, multiple transportation agents can form a group to finish the task together if no one can do the transport by itself. After the transport negotiation finishes, both transport and production or buffer task are awarded to an agent.

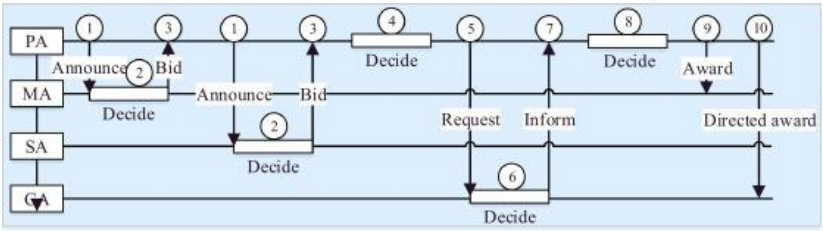

Figure 3.5: Negotiation procedure (Wang et al. 2016)

Other than in the systems presented before, there is no mention of limited buffers in this case. Since the authors write about dealing with deadlocks, buffer slots must, however, be limited in some way. Dealing with deadlocks is also an essential aspect of the model developed for this thesis; therefore, we take a closer look at this publication's deadlock avoidance strategies. The authors present four strategies to avoid deadlocks. The first is for a particular case of a production system in which multi-purpose machines exist. In this case, they try to limit the number of multi-purpose machines by restricting the use of those machines for tasks that could also be done on other machines. This procedure leaves a maximum of one machine, which shares the capability to do tasks with other machines. It considerably reduces the number of loops in the product flows and consequently the chance for deadlocks to occur. As a second step, the number of visits

on the multi-purpose machine and the machines with the same operation type is equally balanced. Strategy three reserves buffer spots for the critical machines identified after steps one and two. According to the authors, if there are enough buffer spots available, deadlocks can no longer occur after these three steps. If this is not the case, a protocol is applied, prioritizing jobs in a loop over jobs not in a loop. These strategies work well, according to the authors. However, they cannot eliminate deadlocks completely and only reduce the chance of a deadlock occurring.

So far, all of the presented systems have used some variation of an auction-based scheduling system. Some publications do not use an auction to schedule production tasks, although their number is smaller than auction-based publications. Wang and Lin (2009) describe a system called agent-based agile manufacturing planning and control system (AMPCS) in detail. The system includes variations of resource, job, and scheduling agents. Additionally, there are specialized data and monitoring agents present. The scheduling is done in multiple steps. A central scheduling agent first creates a production schedule based on demand, which contains the products to be manufactured. Then, the schedule is evaluated in a simulation and, if successful, saved. Afterward, with information from the resource agents, the scheduling agent creates an operation schedule. Here, a small part of an auction-based system remains since the resource agents submit their information by making a bid towards the scheduling agent. The simulation again evaluates the final plan before the production starts. Resource agents may also reject the final plan if they cannot comply with it. This triggers a re-scheduling from the scheduling agent. Another approach with a centralized scheduler is chosen by Y. Zhang et al. (2014). A Capability Evaluation Agent evaluates the orders coming into the system and assigns the tasks to the best-suited machine. Machines have their agents that supply the Capability Evaluation Agent and a Process Monitor Agent with real-time information. The Process Monitor Agent tracks the machine agents' status and calls for a re-scheduling if disturbances or interruptions occur. Finally, the Scheduling Agent schedules all tasks on the assigned machines considering the given real-time information. Groß et al. (2020) focus on the si-

multaneous scheduling of production and transports in remanufacturing. A central scheduling agent in a multi-agent system creates schedules for workplaces and AGVs.

The three presented non-auction-based publications while being multi-agent systems, had a centralized scheduling agent. The last part of this section presents publications that use decentralized non-auction scheduling in at least some parts of the process. Leitao and Restivo (2008) propose a hierarchical holon system. A supervisor does the scheduling, and operational holons execute the production plans. When disturbances such as breakdowns or general deviations from the plan occur, the operational holons gain more autonomy to re-schedule as they see fit. The system then dynamically changes to a decentralized structure wherein the holons communicate and negotiate how to best react to the disturbance.

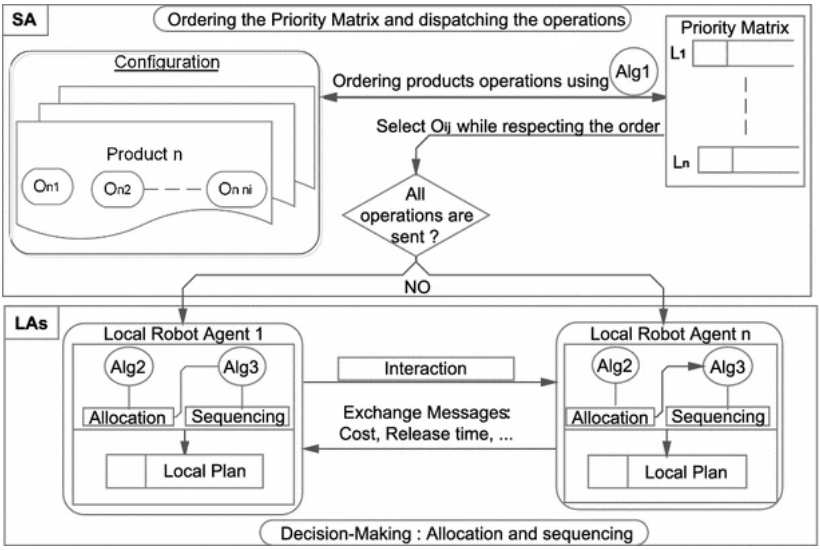

Figure 3.6: Scheduling process of Maoudj et al. (Maoudj et al. 2019)

Another example of a system in this category is the publication of Maoudj et al. (2019). They use robotic assembly cells where each cell has a local agent, which includes a scheduling module. Each agent only solves a part of the

scheduling problem without having a global view. Before each agent schedules its operations, a global scheduling agent assigns the operations to a cell. For this, operations are prioritized, and then each prioritization level is dispatched to a cell. Afterward, the cell scheduling agent assigns each operation to a robot within the cell, and finally, the operations on each robot are sequenced. Scheduling in each step is done according to time-based dispatching rules. Agents exchange information about operations so that local release times can be updated when necessary. Mayer et al. (2019) publish a decentralized system where the job agent decides its route through the production system by deciding which workstations to visit. The workstation itself regularly updates its schedule with an optimization model according to the jobs currently wanting to visit it. A vehicle agent handles requests for transports from the workstation agent, which assigns the transports to AGVs based on a different optimization model. In the publication of Egger et al. (2020), a job arriving into the system sends a request to the agent processing its last process step. That agent then asks the precessing step to make a statement about when they can finish processing these jobs. This continues until the first process of the process sequence. Each agent who has received the answers to its requests then plans its start and end time until the final step finishes the planning.

4 Scope of this Thesis

After providing the basics of the two main topics and an overview of the relevant works published in the areas, this chapter finalizes the introduction. It starts with concluding the literature research by summarizing it and identifying the research gap. A definition of the problem that the algorithm in this work solves and its assumptions are developed. This problem is then split up into research questions to structure the work on it.

4.1 Conclusion to the Literature Overview and Research Gap

The publication dates of the extensions to the SB heuristic indicate that it has fallen out of researchers' focus over time. Publications added many characteristics in the first years after the initial release, but there are relatively few publications during the last years. To find out why is not easy, but it may have to do with more powerful computers and the availability of other heuristics, enabling to schedule systems without decomposing them. It has not completely gone off the radar since there are still publications using it, mostly combining it with some artificial intelligence technique. It has been regularly used only in the area of semiconductor scheduling. Since semiconductor manufacturing is generally considered one of the most complex production processes, this shows that the method is still relevant. This is especially true in decentralized systems since the heuristic can easily be converted into a decentralized algorithm. However, its use in decentralized systems so far is also minimal. It is either extended with entirely different algorithms (such as a Tabu or Neighborhood search) or used

at several points in a distributed system but calculated centrally for each of them. Although researchers have made many extensions already to the SB heuristic, the literature overview has shown that only one of the mentioned publications research available buffer space in the production system (Q. Zhang et al. (2014)). And it even only mentions that their production system contains buffer slots, not that they are limited.

The argument about missing research about limited buffer slots in scheduling extends beyond the SB heuristic. Even generally, limited buffer space is a topic not widely researched in scheduling problems. In Fleischmann et al. (2021), we searched for publications focusing on mathematical formulations of the scheduling problem, including limited buffers, not only looking for the SB heuristic or decentralized systems. We found some but none of those matched the formulation of the presented problem. We refer to Brucker (2012) for a general overview of the limited buffer topic. However, none of the cases presented match this thesis's problem, as we will see soon. There are also publications concerning the so-called "blocking job shop", for example, in Lange and Werner (2019), which are equivalent to a system having zero buffer slots, but they only work with precisely zero buffer slots at every workplace. In conclusion, this topic is not only of interest for this specific application of the SB heuristic, but it has not been researched thoroughly in general.

There are numerous multi-agent systems for scheduling production systems, which do not use the SB heuristic. To find them, our search string consisted of three groups. The first group contained the words "Agent-based", "Distributed", "Decentralized", and "Multi-Agent". The second group contained the words "Production" and "Manufacturing". The last group consisted of "Scheduling", "Planning", and "Control". From these groups, all possible combinations (in total 24) were used in the Google Scholar search engine. A large part of the resulting publications focuses on auction mechanisms to create a production plan. Auctions can have the disadvantage that they base on relatively short-term planning, where they plan each step only when the previous step finishes. Since they are easy to implement and give good results if deadlocks are not a problem, publica-

tions not using auctions are relatively scarce (excluding meta-heuristics). We found only four publications matching our criteria and not using auctions where the scheduling is done decentralized. In three of these systems, scheduling is done hierarchically. Only one publication presents a decentralized, non-hierarchical agent-based scheduling system (Egger et al. 2020). Their system, however, also plans short-term, as agents negotiate about one job at a time. Among the auction-based publications, two are dealing with limited available buffer space in the production. However, both show rule-based strategies to avoid deadlocks but cannot guarantee the absence of deadlocks. It can be concluded from the overview that no decentralized deterministic scheduling algorithm taking into account limited buffers without hierarchies exists yet.

4.2 Problem Description

This work aims to develop a decentralized scheduling algorithm that incorporates limited available buffer space at each workplace. We made two significant design decisions before starting the project. The first one was that it had to be a decentralized system. The second one, and more relevant to the scheduling aspect, that the resulting algorithm should be deterministic in every aspect. After some research, we chose the SB heuristic as a base since it can be easily used in a decentralized way and showed remarkable results in the publications using it.

The underlying system is based on the job shop problem presented in chapter 2.1.3 extended by a few advanced features. The most important is that the job shop model assumes an unlimited available buffer space at each workplace in the production. In this work, the buffer space at each workplace is strictly limited. There is also no other buffer space than those at the workplaces. This creates a need to limit the number of orders released into the production system to prevent unnecessary waiting time, blockages, or even deadlocks. We define that a job can only enter or leave a workplace's processing area via a buffer slot. Therefore, there always needs to be a free buffer slot once an operation finishes for the job to leave the processing

area, and a free buffer slot on the following workplace in the process sequence when it is transported there. This definition of buffer slots makes a deadlock occur if two (or more) workplaces need to send jobs to each other in a circle but cannot do so because input buffer slots at each affected workplace are full.

We also changed the objective function of the algorithm. The original SB heuristic uses the maximum makespan criterion. Instead, here we use the average makespan of all jobs as our objective function. It is equivalent to the "Total Completion Time" presented in chapter 2.1.2. Table 4.1 can explain the reason for this change. It is supposed to show the finish times of eight orders after finishing the scheduling. The number of workplaces in the system is irrelevant for this example. With the original objective function, both plans would have the same value, which is 40 if we assume planning to happen at time zero. From a one-time optimization standpoint, that statement is correct. Taking a look at a scheduling algorithm's practical applications leads to a separate evaluation of the two plans. In the real world, not only one scheduling run is performed. Instead, scheduling repeats again and again. Therefore, more regular completion of orders makes more sense because some are already finished at the next scheduling point, which is achieved by each order having an impact on the objective function. Another even more critical argument for the changed objective function is that the more regular finishing of jobs leads to an overall reduced number of jobs in the system. Then, buffer usage is lower, and the algorithm gains more flexibility to improve the result.

Alternative	End times
Plan 1	17,26,33,34,36,38,39,40
Plan 2	5,10,15,20,25,30,35,40

Table 4.1: Job finishing times of two plans

Buffer slots incur a delay; a minimum time a job needs to spend in a buffer slot before it can leave again. Slots are also strictly divided into input and output buffer slots, meaning that input buffer slots only store jobs that the

workplace has yet to process, and output buffer slots only for those that it has already finished. For simplification, especially in the agent system, we assume that the division of buffer slots into input and output buffer is strictly given and cannot be changed. This work assumes that one of them is an output buffer, and the rest are input buffer slots for any given number of buffer slots. Furthermore, more than one workplace may offer an operation, this is the flexible shop extension from 2.1.3. Workplaces offering the same type of operation are called parallel workplaces throughout this work. We allow processing times to differ on workplaces offering the same operation. Each workplace can only offer one type of operation. Setup and transportation times are assumed to be always zero. The workplace following in the process sequence always carries out transports. The ownership of the job, therefore, switches once the job leaves the processing area. The production system does not have to be empty when planning starts. Instead, there can be any number of work-in-progress jobs currently being processed at a workplace or waiting in any buffer slot. Workplaces are always assumed to have 100% uptime, meaning there are no breakdowns or interruptions of a process. Furthermore, the algorithm cannot stop an operation in progress to start a different operation. We formulated this problem, including more than the mentioned extensions, as a mathematical optimization problem in Fleischmann et al. (2021).

4.3 Research Questions

After defining the research problem, we can specify the questions to solve it. The research of this thesis divides into four parts. The first area deals with the scheduling aspect of the problem. Here, we make the necessary extensions and changes to the SB heuristic integrated into the basic algorithm of Adams et al. (1988). Especially the limitation of available buffer slots needs significant changes in the logic of the algorithm. Therefore, the research question that will be answered in chapter 5 is the following:

> **1st research question: How can a Shifting Bottleneck procedure schedule the problem presented in chapter 4.2?**

The result of the first question is a scheduling algorithm that can handle the required characteristics but still executes centrally. The second research area builds on the algorithm and converts it into a decentralized multi-agent system. For this, some parts of the algorithm need to be adapted to continue to work. Furthermore, all parts of the multi-agent system, for example, data storage or communication between agents, have to be newly developed. Chapters 6.1 and 6.2 try to answer the following question:

> **2nd research question: How can a decentralized multi-agent system executing the algorithm of research question one be modeled and implemented?**

Finally, more critical than pure numerical quality is the reliability of the algorithm. For this, the algorithm needs to avoid deadlocks, livelocks, and starvation, which were described in chapter 3.2.2. There, the three problems were described referring to an agent system, which leads to the third research question. With regard to our problem description, ideally, we can prove that the algorithm will always create a schedule in any given situation.

3rd research question: Does the decentralized multi-agent system guarantee the creation of a production plan for every possible case of valid input data?

However, the same problems can also occur during the execution of a scheduling plan. Therefore, we also need to guarantee that the scheduling result is free of deadlocks, livelocks, and starvation. Whether that is possible, is the aim of the fourth and final research question. Since the third and fourth question are closely related, both will be answered in chapter 6.3.

4th research question: Is the production plan created by the decentralized multi-agent system always free of deadlocks, livelocks, and starvation?

5 Scheduling Algorithm

This chapter presents the algorithm to solve the problem presented in chapter 4.2 and therefore answers the first research question. For this, first, the general procedure of the modified SB heuristic is presented. It follows the original publication of Adams et al. (1988) in the order of the general steps with some changes and additions. Before all five steps of the algorithm are described in detail afterward, a short section describes the rules and ideas guiding the algorithm in general. They will be relevant for every step of the algorithm. The problem's objective function has also changed slightly as described in chapter 4.2. Chapter 5.4 will explain it in detail, where a bottleneck is identified, and the objective function is applied. As in the original procedure, M describes the set of all workplaces, and M_0 the set of workplaces already identified as a bottleneck. The general procedure of the algorithm is as follows, also pictured in figure 5.1:

> Step 1: Update all necessary information before scheduling.
>
> Step 2: Create a local sequence on all workplaces $M \backslash M_0$.
>
> Step 3: Identify a bottleneck workplace m among them. Set $M_0 = M_0 + \{m\}$.
>
> Step 4: Re-optimize the sequence of each critical workplace M_0 while keeping the other sequences fixed. Then, if $M = M_0$ go to step five, otherwise go to step two again.
>
> Step 5: Finish the scheduling by creating a production and a transport plan for each workplace from the scheduling result.

Step one of the original procedure is now included in the local sequencing since it has to be done workplace-specific. Instead, a new first step to update data, which only needs updating once per scheduling, is introduced.

Which data exactly needs to be gathered is described in chapter 5.2. Step two has not changed on the iteration level. We only divided it into two separate steps, presented in chapter 5.3 and 5.4. However, many changes were made within the local sequencing since it needs to incorporate the additional features required to solve the problem. The re-optimization (step four) is different in one crucial aspect in comparison to the original procedure. The step was repeated until no change in all workplaces' local plans occurs but only until a maximum of three repetitions finished. We removed the limit of three repetitions. Now, every time the re-optimization starts, it will continue until a full repetition of re-optimizations without any change completes.
We also have to include an additional feature in the re-optimization since limited buffers can lead to infeasible plans. We introduce a repair method for the case of creating an infeasible plan during the re-optimization. This step and the changes are described in chapter 5.5.

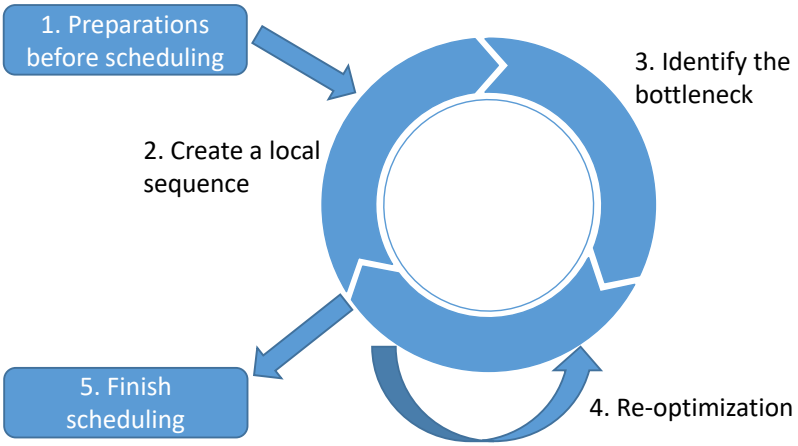

Figure 5.1: Flow chart of the scheduling algorithm

The original procedure uses the algorithm from Carlier to sequence bottleneck and non-bottleneck workplaces (steps 3 and 4 of the new procedure).

Here, we use different versions of a newly developed algorithm at various points during the scheduling process. Until a workplace becomes a bottleneck in an iteration, we use the basic version. It is described in chapter 5.3; we call it our local sequencing algorithm. Afterward, local sequencing is not called any more for that workplace unless it is part of a parallel workplace group. The differences between parallel and non-parallel workplaces will be explained in chapter 5.4. Once a workplace was added to the pool of bottleneck workplaces, the second version of the algorithm applies for the re-optimization. If an infeasible plan is detected during the re-optimization, a third version of the algorithm helps to repair the plan. Versions two and three are the same as the base version for large parts; chapter 5.5 explains the differences. After describing the algorithm's general guidelines, we can continue to give a detailed explanation of all steps in chronological order, starting with the preparations before scheduling.

5.1 Guidelines for the Algorithm

The main addition compared to the original SB heuristic is the addition of limited buffer space at each workplace. This brings up the question of how and if the local sequencing on each workplace needs to change because of this. If buffer space is unlimited, it is generally preferable that every workplace works as fast as it can. The reason for this assessment is that it minimizes the possibility that any succeeding workplaces become idle by offering them as many work-in-process as possible. Reducing idle time of workplaces is in turn a way to achieve a better global plan, as a plan with no idle time on any workplace can not be improved further concerning the makespan. The algorithm of Carlier, therefore, only uses the due dates of operations to prioritize between them, while the prioritized individual operation is always produced as early as possible. With limited buffer slots, this rule no longer applies. While producing as fast as possible on a workplace is still optimal for the global result, an algorithm also has to consider the fact that it may overfill the buffer slots of a succeeding workplace, blocking itself in the process, since finished operation can no longer be trans-

ported to the following workplace. In the worst case, this creates a deadlock in the production system. This introduces a need for the workplaces to only process operations as fast as the system can handle. As we will see, the algorithm presented here does not schedule every operation as early as possible. Instead, if the bottleneck is downstream from the current workplace, it tries to produce the operation after just-in-time, reducing required buffer time and, therefore, slots. For an upstream bottleneck, the algorithm still tries to produce as fast as possible. To achieve this behavior, the algorithm contains several additional prioritization criteria, which will be presented in detail later in this chapter. For each of them, a pairwise comparison with multiple conditions is done between the current best and a candidate operation. If all conditions are fulfilled, the algorithm determines the candidate as the more important operation. If not, then the current best operation stays the most important. Sometimes a formula in the text of chapter 5.3 contains more than one criterion. They were grouped together in the description of the algorithm since they are closely related. If that is the case, an operation only needs to fulfil one of the criteria to be deemed as more important. After the algorithm has compared all operations for each criterion, the most important operation is selected as next in the local sequence. Its start time is then set according to the rules. The details of this process and the criteria are described in section 5.3.

With just this guideline, the local sequencing can create infeasible plans. Infeasible in this case means that the plan would require more buffer slots for a production plan than there are available in the production system. When that case occurs during the scheduling, a method to repair the plan is needed. We extended the algorithm with a method to repair a plan, which we call a negotiation. How the negotiations function is described in detail in section 5.5 but we want to give a short overview here to describe the motivation behind them. The easiest way to repair a plan is to introduce a time delay. The operation which does not have a buffer slot can be delayed until a slot is available again. Also, the whole plan after that operation (timewise) is delayed by the same amount. This solution will always create a feasible plan with the cost of a worse objective function value. Therefore,

the algorithm first tries to find a different sequence of operations, in which the amount of buffer slots is sufficient. Only if it can not find one, a delay is included. That this delay will always work can be shown with a relatively simple example of an extreme case. If it were needed, one could introduce so many delays within the production plan that only one job is processed in the production system at a time, meaning that the next job is only allowed to start processing once the previous job has been finished. In that case, infeasibilities are not possible anymore.

The algorithm described in this chapter was designed with the intention of applying it in a decentralized scheduling system. Therefore, some of the solutions or explanations of this chapter may seem unnecessary complex for a centrally executed algorithm. However, they were chosen since they help to make the transition to a decentralized system as smooth as possible. Without this background some of the features could be reduced in complexity in case the user plans a centralized usage for which the above-mentioned negotiations are an example. For the following functional description of the algorithm, it helps to remember the general remarks of this section to understand the design decisions made for the algorithm.

5.2 Preparations before Scheduling

Since the algorithm does not need to assume an empty system at the point of planning, the first step is to gather the current system state before scheduling can start. The algorithm starts with a list of all operations needing processing for every workplace. Whenever a job enters the system, its operations get added to the workplace's lists. The workplaces delete an operation once it finishes processing. Therefore, the combination of all lists represents the amount of work that needs doing in the production system. It does, however, not correctly represent the operations that require scheduling. The algorithm needs to remove two kinds of operations before local sequencing can start since the workplace will not process them. The first kind are operations that are currently being processed on a workplace, if there are any. These operations have already been planned in a previous

call of the algorithm and cannot change any more since we do not allow the interruption of a process. We also decided that the algorithm will not plan the operations of all jobs already waiting in an input buffer slot. They already must have a planned start time for processing from previous scheduling, which the algorithm will not change. However, if there was any delay during the previous plan's execution, the starting times of jobs waiting in the input buffer have to be adjusted to account for the delay, if needed. The input buffering times of these jobs are also lengthened for the same delay. Table 5.1 shows a situation with which we will explain how the algorithm proceeds in this case.

Operation	Planned start time	Process time	Actual start time
Op. 1	0	10	0
Op. 2	10	5	12
Op. 3	15	8	-
Op. 4	27	4	-

Table 5.1: Example situation for a delay in the plan on a single workplace

The workplace in question had a local plan of four operations after the last scheduling. The first operation started and finished on time. The second operation had a delay of two time units. Operations three and four have not yet started, so the current time must be somewhere between time units 12 and 17. Assuming that operations three and four are already waiting in the input buffer, the algorithm needs to check their times. It corrects the start time from operation three since it can start at time 17 at the earliest. There was a gap of four time units before operation four so that one does not need a change in times. Then, the algorithm removes the old buffer slot reservation of operation three and creates a new one from the current time up to time 17. Now the algorithm will account for the delay in the current scheduling. A delay in one workplace might also lead to necessary changes in a preceding workplace. This only happens if the input buffer of the delaying workplace is currently full. In that case, a planned transport might have to be carried out later, therefore inducing a delay in the output buffer of the preceding workplace.

If an operation, processed at a workplace that is available multiple times in the production, is removed from the list of operations to be scheduled for one of the two reasons, it is removed at all parallel workplaces. After the list of operations for scheduling is now final, the algorithm deletes the old times for processing and buffering of all previously planned operations that will now be planned again (since they have not arrived at the workplace in question).

Because a workplace might have leftover work from the last plan, it cannot guarantee that the first operation can immediately start processing. Instead, the latest finish time of all operations, which were removed before on that workplace, is saved as the earliest time (t_{start}) an operation can start processing in the new local sequence. The same happens for the input buffer slots. The algorithm already knows if they are currently in use. It creates a buffer reservation from planning time to the time processing starts for operations waiting in the input buffer.

During the scheduling, the algorithm will need a list of all predecessors of a workplace. Since it is now clear, which operations it will plan, it can precompute this list. This is done for time-efficiency because the algorithm may use it many times later. It looks at every operation to be planned in a workplace to create the list. If the workplace does not process the first step in the process sequence, it saves all workplaces that offer the preceding operation in the list. Duplicates will only be listed once.
After these steps, the algorithm knows the current system state and has finished pre-computations. The scheduling process can begin. It starts in the same way as the original algorithm, with the local sequencing step at each workplace described in the next section.

5.3 Create a Local Sequence

The process of creating a local sequence, also called local sequencing throughout this thesis, describes the algorithm for scheduling the operations to be processed in a given workplace. The only input for it is the list

of operations resulting from the preparations. If this section mentions any operations, it refers only to those that the current workplace can process. If other workplaces can process the same operations as the workplace currently planning, the operations will split between the parallel workplaces during this step. The result of the local sequencing is a production and an input buffering plan for the given workplace. The algorithm of this section creates the local sequences for workplaces that are not yet a bottleneck. The described procedure considers the already fixed start times of all operations on workplaces identified as a bottleneck while creating them. Every time the sequencing starts, the algorithm makes an entirely new sequence of operations, and the old sequence from a previous local sequencing gets deleted. Due to an additional bottleneck, the plan from the last call is no longer valid.

The process starts by updating the key variables (chapter 5.3.1), which is done once at the start of sequencing. Afterward, it schedules one operation after another until there are none left. Chapter 5.3.2 describes the process of deciding which operation to schedule next and its start time. It consists of two separate steps, first selecting and timing an operation that is to be next in the sequence, then creating a buffer reservation for that operation.

5.3.1 Update the Key Variables of Operations

The local sequencing starts with an update of all operations with the currently available data from bottleneck workplaces. For each operation, key variables are updated, which will be used for the sequencing. Table 5.2 gives an overview of the variables, which will be described afterward.

From the variables in table 5.2, t_p, only the operation's processing time is known at this point. Every other variable has to be determined or calculated with information from other workplaces. The first is the variable *earlier*. It describes the relative position of the first fixed operation within the process sequence of this operation's job. If the first fixed operation's position in the process sequence is before the operation to be updated, *earlier* is true. If the position is after or there is no fixed operation yet, it is false. While looking

Name	Variable	Type
Process Time	t_p	Numerical value
Relative Bottleneck Position	*earlier*	Boolean variable
Job Started	*st*	Boolean variable
Earliest Available Time	t_{avail}	Numerical value
Earliest Possible Start Time	t_{earl}	Numerical value
Latest Finish Time	t_{late}	Numerical value
Optimal Start Time	t_{opt}	Numerical value

Table 5.2: Key variables used during the sequencing of operations

at the process sequence, *st* is also determined. It describes if the job of this operation has already started the first operation of the process sequence.

$$t_{avail,j} = \max_{i=1}^{j-1} t_{start,i} + \sum_{k=i+1}^{j-1} t_{p,k} \tag{5.1}$$

$$t_{late,j} = \min_{i=p+1}^{\infty} t_{start,i} - \sum_{k=p+1}^{i} t_{p,k} \tag{5.2}$$

Afterwards, t_{avail} and t_{late} are determined, which the formulas 5.1 and 5.2 describe. t_{avail} describes the earliest time the operation can begin processing. This time equals the earliest time the operation can arrive at the workplace plus the minimum input buffer delay. It is calculated by adding the process and minimum buffering times of all previous process steps in the sequence to the current time t. If there is any fixed step before or the job has already started, it takes the known end time of the last earlier fixed operation, and process and buffering times of the steps between are added. t_{late} is calculated in the same way, starting from the current makespan value of the job and subtracting the process and buffering times or taking the start time of the first fixed step occurring later in the process sequence and subtracting process and buffering times from there. The current makespan value of a job is equal to the earliest time it can finish. t_{avail} does not have to be equal to the earliest possible starting time, t_{earl}, since a job may arrive earlier at the workplace than the workplace can begin processing. There-

fore, t_{earl} is equal to the maximum of t_{avail} and t_{start}, which was determined during the preparations.

An optimal start time t_{opt} of the operation is calculated from these values. In a production system with unlimited buffers, it is always optimal to produce an operation as early as possible since it can be stored after processing until the next workplace needs it. If buffers are limited, as they are in this thesis, it is generally better to only produce an operation just in time for the next step of the process sequence to limit the buffer usage as much as possible to have a reserve for situations where buffering is not avoidable. Also, buffering increases the makespan of a job (though not always of the whole plan) and should only be done if necessary. Therefore, this algorithm sets t_{opt} to the following: If the operation has no fixed step before (*earlier* is false) and a fixed step afterward in the process sequence, it is optimal to start the order as late as possible (while causing no delay), which is equal to $t_{late} - t_p$. If fixed steps exist before and after this operation, the algorithm determines the first fixed operation. If it is after this operation, t_{opt} is as late as possible. In the two remaining cases, we try to produce the operation as fast as possible and, therefore, equal t_{earl}. We either know that there is a bottleneck before this operation and therefore want to produce it as fast as possible to limit buffer usage. Alternatively, there is no fixed operation at all in the process sequence, in which case the operation can be produced at any time between t_{earl} and t_{late}. We set the value to t_{earl} because we would rather have the algorithm produce it fast for a better scheduling result while producing it later is always possible. It may also be that the result of $t_{late} - t_p$ is smaller than the earliest start time, in which case the optimal start time is also t_{earl}. Table 5.3 summarizes this paragraph and the calculation of t_{opt}.

Fixed step before	Fixed step after	Result
false	true	$t_{late} - t_p$
false	false	t_{earl}
true	false	t_{earl}
true	true	Depends on first bottleneck

Table 5.3: Different cases for the optimal start time

After updating all operations, the algorithm classifies the operations into two groups. The first contains all operations which have a fixed step in their process sequence, and a second all orders without a single fixed step. We make this differentiation because orders which have a fixed step need to be planned as close as possible to their optimal start time, whereas the algorithm can schedule the other orders at any point between their earliest and latest start time. We will call the first group high-priority operations and the second group low-priority operations for the next steps.

5.3.2 Determine the Next Operation to be Scheduled

The process of creating a local sequence for a workplace is iterative. In every iteration, one operation is selected, scheduled at the end of the current plan according to its start time. Adding an operation to the plan before an already scheduled operation is not possible. An iteration consists of two connected steps. First, an operation is selected and its start time is determined. Afterward, a buffer reservation in accordance with the planned start time is created. At single workplaces, this process continues until all operations are scheduled. Parallel workplaces do this process simultaneously since they still have overlapping lists after the preparations step. Therefore, at a parallel workplace, only one operation is scheduled before stopping the process. Then, the algorithm determines on which parallel workplace the next operation will be scheduled. It always chooses the workplace with the earliest plan end time. In case of a tie, any workplace can be taken.

Select an operation

The decision which operation to schedule next is made over several criteria. For each criterion, a current best operation evaluates against the other operations. If a candidate fulfills the criterion, it replaces the current best operation. If the two compared operations are equivalent according to the criterion, the current best operation does not change. The algorithm needs to select one operation as the current best to start comparing operations. Since this only influences the case of a tie between two operations, which

indicates that it does not matter which operation to take, any operation can be chosen for this. We take the first alphabetical entry of the high-priority list as a starting operation. It is to note that only high-priority operations are available for selection for the following criteria. Lower priority operations will be checked afterward at the end of the selection process. Some of the criteria only apply if both operations were not planned during an earlier planning process or both were. Operations that were already planned before, but were not processed yet, are sometimes preferred over new operations to prohibit extensive backlogging of "bad" operations. "Bad" in this context means that an operation always is at the end of a schedule, as it does not fit well earlier. We will describe for each criterion if this rule applies. In the following, the index *best* indicates the current best operation, and the index *cand* the candidate operation, which evaluates against it.

$$t_{opt,cand} < t_{opt,best} \tag{5.3}$$

Equation 5.3 describes the primary condition after which the algorithm orders the operations for the sequencing. Generally, it chooses the operation with the earliest optimal start time. We take the operation with the earliest t_{opt} since it replaces the t_{late} of the original algorithm as described in chapter 5.3.1. If the same t_{opt} value occurs in multiple operations, a decision between them is made according to equation 5.4.

$$\text{Select} \begin{cases} \text{candidate} & t_{opt,cand} = t_{opt,best},\ t_{late,cand} < t_{late,best}, \\ & st_{opt} = \text{false},\ st_{cand} = \text{false} \\ \text{candidate} & t_{opt,cand} = t_{opt,best},\ t_{avail,cand} < t_{avail,best}, \\ & t_{late,cand} = t_{late,best} \text{ or } st_{opt} = \text{true or } st_{cand} = \text{true} \\ \text{current best} & \text{otherwise} \end{cases} \tag{5.4}$$

The original algorithm takes t_p as a tiebreaker. We already included the processing time in the calculations for t_{opt}. Therefore, the algorithm uses the due date t_{late} as a tiebreaker. It is, however, only applied if the jobs of both operations have not started yet. If at least one of the jobs has started or

the two compared operations have the same value for the tiebreaker, t_{avail} is the tiebreaker. It forces a FIFO ordering of the two operations. Should t_{avail} also be the same for the candidate and current best operation, the algorithm keeps the current best operation because it cannot decide which of the two operations it should process first.

t_{start} is the start time of an operation in the local sequence. After the first criterion, the algorithm plans to schedule the current best operation at its optimal starting time $t_{opt,best}$. If that time is earlier than the end time t of the current plan, it plans to schedule it at time t instead. Afterward, the selection process does not finish yet. In the following, the algorithm checks several additional unique cases, where it is better to diverge from the basic ordering. For this, it makes a case distinction. Case one applies if the planned start time is equal to t, case two if it is later than t, meaning there would be a gap in the plan if the selection process stops now.

$$\text{Select} \begin{cases} \text{candidate} & t_{earl,cand} \leq t, t_{late,cand} < t_{late,best}, \\ & t_{late,cand} < t + t_{p,cand} + t_{p,best} \\ \text{current best} & \text{otherwise} \end{cases} \tag{5.5}$$

First, the case of $t_{start,best}$ equal to t is presented. It implies that the current best operation either causes a delay or is processed just in time ($t_{start,best} + t_{p,best} \geq t_{late,best}$). The algorithm then checks if any candidate operation gets delayed because it will start after the current best operation. If one is found and has an earlier t_{late}, the algorithm switches the current best operation. The reason is that one of the two operations will be late. Therefore, the algorithm tries to minimize the combined lateness of the two operations. Equation 5.5 describes the conditions for the switch. Here, the algorithm only compares operations that both have not started. If at least one of them has started already, this condition gets ignored, and it keeps the basic ordering. Figure 5.2 describes an example of a situation in which this criterion is applied.

For simplicity's sake, we can assume that the two operations given in the diagram are the only two remaining to be scheduled. Both operations are

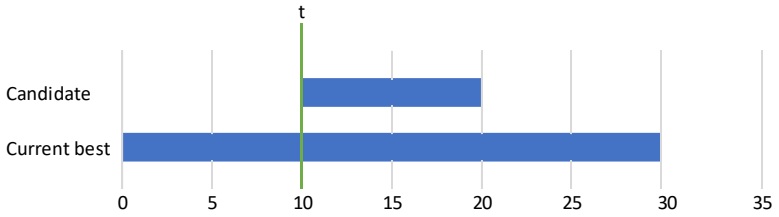

Figure 5.2: Example situation for condition 5.5

drawn at their optimal start time in the diagram. From that we can already gather that one will definitely be late as they must be processed in parallel to both be on time which is not possible. The time instant t depicts the current end time of the local sequence. Therefore, the operation marked as current best will be late, as it can start earliest 10 time units after its optimal start time. In this example, the primary criterion chooses the operation marked as current best to schedule since it has the earlier optimal start time. If the algorithm schedules this sequence, the candidate operation would be 30 time units late, as it would finish at time unit 50 instead of 20. The current best operation would be ten time units late, giving a total delay of 40 time units. If it switches the sequence, the candidate operation would be on time, and the current best operation would finish at time 50, making it 20 units late. That is a superior solution, so the algorithm switches the sequence.

At this point during the selection process, it might be that the current best operation finishes later than its $t_{late,best}$. At a later point, that is not possible anymore, since the second case of the case distinction describes that $t_{start,best} > t$. The criteria as mentioned earlier directly mirror that an operation finishing on-time starts at its $t_{opt,best}$. Therefore, the algorithm now checks if o_{best} finishes too late. If that is the case, it tries to find gaps earlier in the plan, which can be closed by shifting the start times of already scheduled operations forwards. Every operation with a start time later than its earliest start time can shift. With this, the algorithm tries to finish a few operations too early instead of finishing one too late. The maximum shift

time is $t_{start,best} + t_{p,best} - t_{late,best}$ as that is the delay of the current best operation. If the algorithm finds a possible shift smaller than that value, the sequence will still be shifted by that number since it improves the plan's objective function value by reducing the delay of the currently selected operation. After the possible shift, the first case finishes.

$$\text{Select} \begin{cases} \text{candidate} & t_{opt,cand} < t + t_{p,best}, t_{late,cand} < t_{late,best}, \\ & t_{earl,cand} \leq t - t_{p,cand} \\ \text{current best} & \text{otherwise} \end{cases} \tag{5.6}$$

As described, in the second case o_{best} is planned to finish exactly on time ($t_{end,best} = t_{late,best}$) and there is a gap to the previously scheduled operation. In this case, the algorithm checks two more criteria for a potential switch of the best operation. The first is for an operation supposed to start during the processing of o_{best} and finish earlier than it. The algorithm tries to put it into the gap left between o_{best} and the last scheduled operation. This criterion tries to produce an operation earlier than needed and buffer it for a time (if there are enough buffer spots available) rather than creating an unnecessary delay. Equation 5.6 shows the conditions under which this is possible.

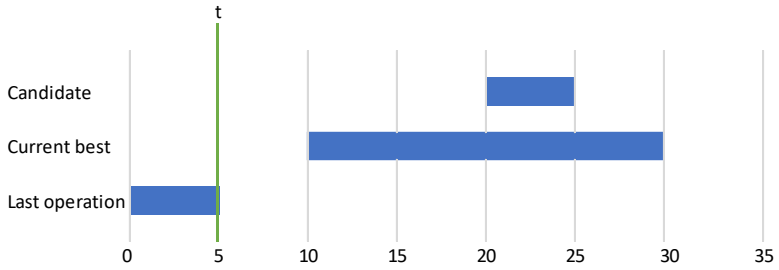

Figure 5.3: Example situation for condition 5.6

For an operation, which fulfills condition 5.6 but with a processing time too long to fit in the gap, the algorithm tries to shift already scheduled operations to create a large enough gap. If a large enough gap can be created or was there from the start, the candidate is taken as the next operation to

be scheduled, although it has a lower priority according to the basic ordering. All high-priority operations are considered for equation 5.6, no matter if their job has started or not. Figure 5.3 shows a representative situation for this condition. The sequence last operation → candidate → current best is superior to leaving a gap in the plan.

The second criterion in case of a gap in the plan is a variant of the first case. It applies if the gap was too small to fit the candidate, but the sequence would have less delay after switching the order of operations. It only considers candidate operations starting during the current best operation's processing and having a delay if produced afterward. Figure 5.4 is an example.

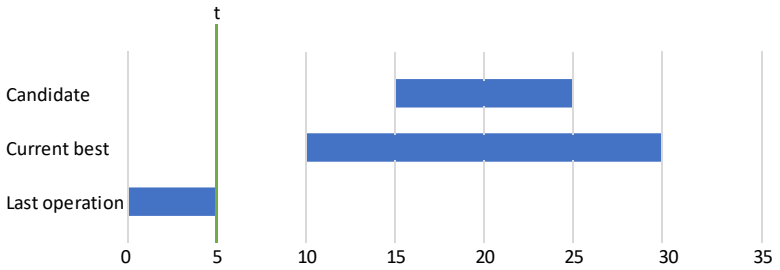

Figure 5.4: Example situation for condition 5.7

If one operation fits the criteria, the sequence current best → candidate is compared to the opposite sequence. Parallel workplaces are also taken into account because an operation not chosen at this workplace could be chosen as the following operation on a parallel workplace and therefore have less delay than it would have on the currently selecting workplace. The four delays of condition 5.7 are calculated by comparing the virtual end time of the two operations with their t_{late}. For parallel workplaces, we assume that the operation, which is later in the sequence on this workplace, would be produced next on the parallel workplace. If that creates less delay, it reduces the calculated lateness from this workplace. Assuming no parallel workplaces, for figure 5.4 this means that the candidate operation would

be chosen since the combined delay would be five in that case, compared to the 15 of the opposite sequence.

$$\text{Select} \begin{cases} \text{candidate} & t_{opt,cand} < t + t_{p,best}, \\ & t_{late,cand} < t_{best} + t_{p,best} + t_{p,cand}, \\ & delay_{best,first} + delay_{cand,second} \\ & > delay_{best,second} + delay_{cand,first} \\ \text{current best} & \text{otherwise} \end{cases} \tag{5.7}$$

If a gap in the plan before the current best operation still exists after condition 5.7, the list of the low-priority operations is checked. The goal is to find an operation fitting into the gap. Condition 5.8 depicts the conditions a low-priority operation must fulfill to be acceptable. It is similar to condition 5.6 so that the low-priority operation needs to fit in the gap without delaying the current best operation.

$$\text{Select} \begin{cases} \text{candidate} & t_{earl,cand} \leq t - t_{p,cand}, \\ & t - t_{previous} - t_{p,previous} \geq t_{p,cand} \\ \text{current best} & \text{otherwise} \end{cases} \tag{5.8}$$

If more than one operation fulfills this condition, condition 5.9 is the tiebreaker between them. Should the algorithm choose a low-priority operation as the current best operation, its start time is always the earliest time it can be processed, which equals the maximum of $t_{earl,cand}$ and t.

$$\text{Select} \begin{cases} \text{candidate} & t_{late,cand} < t_{late,best} \\ \text{current best} & \text{otherwise} \end{cases} \tag{5.9}$$

All conditions described up to this point only apply if at least one high-priority operation exists. If there are no high-priority operations to select, no operation has been selected as the current best so far. Only then, the low-priority list is searched for the best fitting operation. Like at the beginning of this process, any operation from the low-priority list needs to be

chosen as the base for the comparison. All other low-priority operations are compared according to condition 5.10. Here, the algorithm does not need to compare t_{opt}, since all low-priority jobs always start as early as possible, their t_{opt} is always equal to t_{earl}. Therefore, only t_{late} is relevant for sequencing, making this criterion the same as the one used in the original procedure.

$$\text{Select} \begin{cases} \text{candidate} & t_{earl,cand} \leq t, t_{late,cand} < t_{late,best} \\ \text{current best} & \text{otherwise} \end{cases} \tag{5.10}$$

At this point, the next operation to be scheduled and its start time are determined. Now the algorithm still needs to plan the buffer reservation for that operation in the next step.

Create a buffer reservation

To create a buffer reservation, a start time and an end time of the input buffering need to be defined. The end time is already known at this point since it is always equal to the start time of the processing t_{best}, leaving the determination of the start time of the input buffering, which will be used at the end of the scheduling process to create the workplaces' transport lists. Three cases have to be differentiated here:

1. Operation is waiting in the input buffer of the current workplace.
2. Operation is not at the current workplace, and ...
   a) ...the previous workplace is not a bottleneck.
   b) ...the previous workplace is a bottleneck.

First and most straightforward, the operation may be already in the input buffer at the scheduling time. Then, the start time is the current time t. If the job has not yet arrived at the workplace, the algorithm checks the preceding operation's planning status. If the workplace of the preceding operation is a bottleneck, all operations on it are considered planned. If it is not a bottleneck yet, the algorithm has no information about a possible fin-

ish time of the preceding operation and cannot plan the buffer reservation (case 2a). Therefore, it simply assumes that the preceding operation will be finished just in time and sets the start time of the input buffering as t_{best} minus the minimum buffer delay.

Only if the preceding operation's workplace is a bottleneck (case 2b), the algorithm can determine the time the job enters the input buffer. Here, the general goal is to create a reservation with as little time as possible in the buffer. Since we limit the number of output buffer slots to one, the algorithm knows when the job has to leave the preceding workplace at the latest, not to delay processing there. It is the time when the following operation on the preceding workplace will finish. The start time of buffering at the current workplace is then the minimum time the job needs to leave the preceding workplace and the time it has to arrive at the current workplace due to the minimum buffer time constraint.
A particular case remaining is that the preceding operation is currently processing or is already finished and waiting in the preceding workplace's output buffer. Then, the reservation is created in the same way as just described, as one can make the same case distinctions.

Now that the algorithm knows the start and end time of the operation's input buffering, it can create a reservation. For that, the algorithm checks if an input buffer slot is available during the required time frame. Since the algorithm saves all workplace reservations, it can compare the maximum number of jobs in the input buffer between the start and end time with the number of available buffer slots. If the maximum number is less than the number of available slots, the input buffer has a free slot during the required time. If there is space, the algorithm creates a reservation and finally adds the operation to the local sequence's end at its planned start time. Then it continues with selecting the next operation to be scheduled or, if both lists of operations are empty, goes to the next step. If no buffer slot is available, it means that the current plan is infeasible and needs repairing. For this, either the current best operation needs to start later when a slot is available, or the algorithm must create a new and different sequence. The second option is, however, only done during the re-optimization step. The algorithm

ignores infeasible plans at this step since its goal is to determine the next bottleneck, not creating feasible plans. Therefore, if no slot is available, the start time of the buffer reservation is set to the point in time when a slot becomes available. If that is earlier than the planned end time, the local sequence does not need to change. The operation's start time delays until it can be produced the earliest after the delay if it is later than t_{best}. However, this might increase the makespan value of the workplace unnecessary. That is a desired behavior since it increases the possibility that this workplace becomes a bottleneck and, therefore, that re-optimization resolves the problem. Afterward, the process continues with selecting the following operation if there are any left or the next step.

5.4 Identification of the Bottleneck

After the algorithm has created a local sequence on every non-bottleneck workplace, it calculates each workplace's lateness for the bottleneck criterion. This algorithm uses a slightly different objective than the original procedure. The goal still is the minimization of the makespan. The original procedure calculates a workplace's value as the maximum lateness of a single operation on that workplace compared to the current overall makespan. Instead, here, we calculate the value by taking the sum of all latenesses. We already explained the impact of this change and its reasoning in chapter 4.2. Equation 5.11 shows the formula, where t_{end} represents the end time of an operation on a workplace and U_m the set of all operations scheduled on a workplace.

$$O(m) = \sum_{i \in U} \max(t_{end,i} - t_{late,i}, 0) \tag{5.11}$$

After calculating this value for each workplace, the bottleneck can be identified. It is the workplace with the highest objective function value. Equation 5.12 shows the calculation.

$$\max_{m \in M \setminus M_0} O(m) \tag{5.12}$$

If the highest value occurs in multiple workplaces, any of them can become the bottleneck. Since this algorithm manages workplaces in alphabetical order, it chooses the one which comes first in the alphabet. The algorithm now fixes the local sequence of the chosen workplace. It means that during the step of updating data (5.3.1), planning workplaces recognize the start and end times of operations on the bottleneck workplace.

It is important to note that only one workplace is chosen as a bottleneck, no matter the circumstances. So, in the case of two or more parallel workplaces, they each become the bottleneck in a separate iteration. The reason behind this decision is found in the fact that the algorithm of this chapter was already planned as a decentralized system form the beginning. How this leads to the workplaces becoming bottleneck one at a time will be be described in the next chapter. Due to this decision, the strict division of workplaces, one group planning during the local sequencing, the other in the re-optimization step, does not apply to parallel workplaces. Following how the creation of a local sequence works on parallel workplaces, it is impossible to not do it on all of them at the same time. This also has a better optimization potential, for the operations can be exchanged between them fluidly during the whole planning and not only once at the start. Therefore, once one of the parallel workplaces is a bottleneck, and until all of them are, the algorithm will plan every member of the parallel workplace group in steps 2 and 4 of an iteration.

5.5 Re-optimization

The first time a bottleneck is identified, the algorithm skips this step. This step is supposed to make sure that all bottleneck workplaces' plans function together. Therefore, it would not give any different result doing the re-optimization on a single workplace as it will never have any changes only looking at itself. Therefore, after identifying the first bottleneck, the next

round of local sequencing can start immediately. If there are two or more bottleneck workplaces, the algorithm performs this step to make sure that the results of all bottlenecks are compatible.

New local sequences are created on all bottleneck workplaces (all workplaces in M_0) during the re-optimization. The order in which the workplaces are scheduled is the same in which they entered M_0. Generally, the local re-sequencing during this step is done precisely as on non-bottleneck workplaces, meaning that chapter 5.3.1 and 5.3.2 do not change for the re-optimization. There is, however, an additional step during the creation of buffer reservations, which the algorithm ignored before. If there is no input buffer slot available for the created buffer reservation, the current local sequence and, therefore, the whole schedule is infeasible. Before, during the local sequencing on non-bottleneck workplaces, the operation was delayed until the plan was feasible again. At this point, the algorithm will try to find a different plan, which is better than the current sequence. In this context, the new plan is better if it creates less delay than the old one, including the delay needed to make the sequence feasible. If the algorithm finds a new and feasible plan here, but it is worse than including a delay in the current plan, it is better to include the delay than switching to the new plan. We call the process of creating and evaluating the new plan negotiation and will describe it next.

Negotiation

A negotiation starts immediately if an infeasible plan is detected. The local re-sequencing pauses until the negotiation has finished and will continue afterward. Therefore, only operations already scheduled and the one that created the infeasibility are considered to be part of the negotiation. The remaining operations not yet scheduled during the current re-sequencing get ignored. As the first step, a new plan proposal is created on the workplace that detected the infeasible plan. Only one proposal is allowed for every infeasible situation. If the proposal is not successful, the operation is delayed in the same way as in chapter 5.3.2. Since the goal here is to create

an entirely new plan, t_{earl} and t_{late} of the operations can be ignored. The algorithm tries to create a new plan in two different ways. It first looks if there is a large enough gap in the current plan, into which the infeasible operation could fit. If there is one, the operation is inserted at that point, and the proposal is created. If there is no suitable gap, the algorithm creates a new plan from scratch, where it tries to achieve a more regular usage of buffer slots. This smoothing is done according to two criteria—the first criterion groups operations depending on which workplace processes the succeeding operation. The algorithm selects operations from each of them in turn. Operations with the same succeeding workplace are smoothed according to their processing time in this workplace taking fast and slow operations in turn. This process's goal is an average input buffer filling degree on the workplace that detected the infeasible plan.

The created proposal does not include the start and end times of operations at this point. Instead, it is only a sequence of operations. Since the infeasibility occurred in the current workplace's input buffer, the algorithm needs to first determine the start and end times on the preceding workplaces before evaluating the current workplace. Of course, those workplaces might need a result from their preceding workplaces in turn. This goes on until a workplace has no predecessors anymore or it only has predecessors that are already part of the chain. The second group of workplaces cannot be asked again because it would create a livelock where the algorithm switches between different workplaces, needing an answer of one before scheduling the other and vice versa. The answer to a proposal consists of an actual local sequence with start and end times and an evaluation of that plan compared to the last created local sequence. The same local sequencing algorithm of chapter 5.3 is used again, with a few changes explained at the end of this section. To summarize, the algorithm sets up a chain of queries to predecessors and then creates local plans according to their reverse order to evaluate the proposal.

After creating all local plans, the algorithm has a list of answers for the workplace, which created the negotiation. The evaluations state if the plan created in response to the proposal is better, equal, or worse than the cur-

rent local sequence. The algorithm calculates the new objective function value on all workplaces taking part in the negotiation. This value shows how much this workplace would worsen the global scheduling result. The relative change is then compared to the case of the delay of the infeasible operation. If each evaluation is at most as bad as the delay, the algorithm can finally create a new local plan for the workplace that started the negotiation. This step can be omitted if any evaluation is worse since the proposal will get denied in any case. Suppose the workplace starting the negotiation can create a feasible plan, all workplaces participating in the negotiation switch to the new local plan. If one of the participating workplaces would be worse with the new plan than with the delay, or the algorithm cannot create a plan according to the required new criteria at any workplace, the algorithm incorporates the delay since it is the better global solution. In this case, the algorithm checks the effects of the delay on every workplace, and local plans are delayed on preceding workplaces as well, if necessary. After finishing the negotiation, the algorithm continues to create the local sequence at the workplace it is currently planning. It does so by including the delay of the operation that caused the infeasibility or sequencing the subsequent operation at the end of the new plan. Figure 5.5 depicts this whole process.

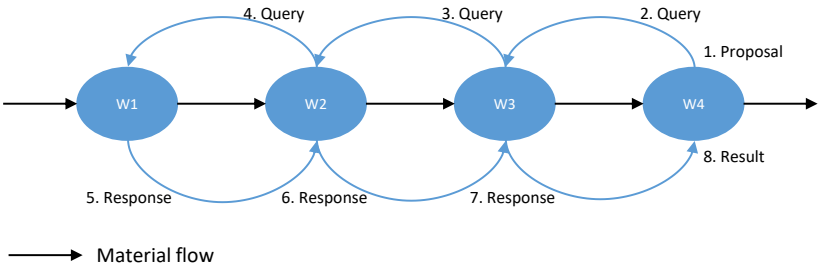

Figure 5.5: Exemplary negotiation procedure

In the example, the algorithm creates a negotiation proposal on workplace four. Since we have a flow shop, the algorithm asks each directly preceding workplace to create an answer to the proposal. It ends at workplace

one since it has no predecessors. The algorithm then creates responses in the opposite order. Once it created the response from workplace three, the algorithm can finalize the negotiation on workplace four.

As mentioned, the algorithm for local sequencing during negotiations works slightly differently on an administrative level from the previous cases, explained in the following. The conditions and criteria used to determine the next operation and its start time stay the same. There are two main differences. The first is that it does not take the data for updating all operations from the bottleneck workplaces but instead from the predecessors' results and the proposal's sequence. With this data, the update of operations is the same as described before in 5.3.1. The second difference is that the algorithm cannot start another negotiation during a currently running one. Instead, if the algorithm concludes that the proposal leads to a different infeasibility, local sequencing is stopped, and it declines the negotiation instantly. In this case, the local sequencing continues, including the delay, which it could not prevent.

Finish the re-optimization

The local re-sequencing, including negotiations, continues on all bottleneck workplaces until the algorithm schedules a complete repetition (a repetition consists of all workplaces in M_0) and every workplace has the same local sequence as in the repetition before. For this, the algorithm saves each workplace's local sequences at the end of each repetition and then compares the new sequences to the sequences of the previous repetition. In the first repetition of the re-optimization, the algorithm uses the last previous sequence before the re-optimization. The newly added bottleneck uses the plan just created during the local sequencing step. For the workplaces, which are bottlenecks from previous iterations, the latest sequence is from the re-optimization step of the last iteration. Once the algorithm detects no changes during a round, it goes to step 5 to finish the scheduling. Alternatively, it continues with step 2 if there are still non-bottleneck workplaces to be scheduled.

5.6 Finish the Scheduling Process

Once M equals M_0, and the last re-optimization step ended, the scheduling can finish. The results of steps one to four are two lists for each workplace. One is the workplace's production plan, containing the ID and the start time of each operation. It can be used in production immediately. The second list contains all buffer slot reservations on the workplace with their start and end times. Since the algorithm guarantees that at no point in time more buffer spots than available will be used simultaneously, this list is of no use for the production system directly. If the operating system also handles transportation within the production, it needs a list of all required transports between workplaces. For this, it can use the list of buffer reservations. Every start of an input buffer reservation amounts to a job arriving at the workplace. Each arriving job requires a transport ending then. As mentioned, we assume transports to happen instantaneously. Therefore, the end time of the output buffer on the predecessor workplace is equal to the start time of the input buffering on the successor workplace. The transport also happens at that time instant. In this last step of the scheduling process, the algorithm converts input buffer reservations into transportation orders for the operating system. With the scheduling process finished, this chapter has described an algorithm able to solve the scheduling problem presented in chapter 4.2. However, this algorithm is only able to function on a central planning unit having global information availability.

6 Decentralized Scheduling System

After developing the scheduling algorithm in chapter 5, this chapter describes its application in a decentralized system and answers the remaining three research questions. For this, we employ a multi-agent system. The chapter starts with the general structure of the multi-agent system and descriptions of all agents present in it in chapter 6.1. Afterward, chapter 6.2 describes the additions and changes to the scheduling algorithm to function in the developed multi-agent system to answer the second research question. In the end, chapter 6.3 deals with the third and fourth research question: Will the decentralized scheduling system always find an executable solution to any given scheduling problem?

6.1 Multi-Agent System

Figure 6.1 shows the multi-agent system of this thesis focusing on the workplace agents. The exemplary system consists of five workplaces, each accompanied by a workplace agent, pictured at the bottom. They are grouped (indicated by the square) because they connect to the other elements in the same way. The workplace agents also build smaller groups on their own during the scheduling process if needed, for example, when parallel workplaces need to create a local sequence. The exemplary system has two parallel workplaces M4-1 and M4-2, shown as a subgroup in the picture. However, also the agents of a parallel group communicate independently with their surroundings. At the top are the four elements, which only exist once

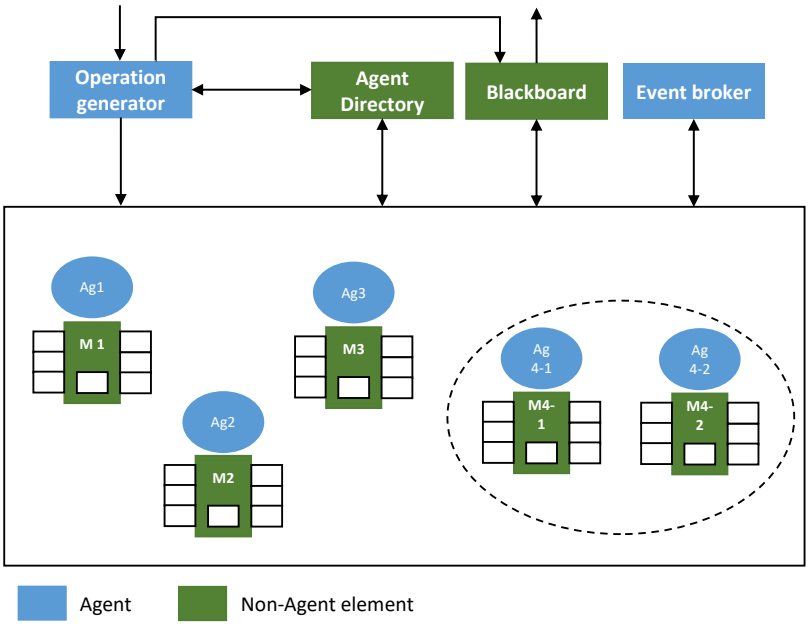

Figure 6.1: Multi-agent system for production planning

no matter how many workplaces are in the system. They are called central elements in the following.

Operation generator and agent directory

Production jobs can be pushed into the system via the operation generator by, for example, the existing production control system. The operation generator then transforms the jobs into the structure needed for the multi-agent system. This includes splitting the job into single operations and initializing the blackboard entries for the operations and the complete job. After initializing the operations, the operation generator informs the workplace agents about the work they have to do. Each agent only gets a list of the operations it can produce. The information required for the operation generator to do this is stored in the agent directory. When an agent

initializes, it signs up at the agent directory via an event containing the workplace's name and a list of operation types it can produce. The agent directory stores this information, retrieved by the workplace agents and the operation generator. If an agent leaves the system, it signs off with a different event.

Blackboard

The blackboard is a typical data storing interface, where each element can store and retrieve data as needed. Each agent has direct access to the blackboard. Due to the way the scheduling algorithm is designed, write and read accesses have a soft lock on them. Writing is locked because every single data point in the blackboard belongs to one specific agent. Only that agent will change the value of the data point. The responsible agent for an entry changes at synchronization points, so every agent always knows if it is responsible for an entry or not. Reading is safe because writing only happens at the synchronization points and all agents work with the data from the last synchronization point until they update it at the next one. The way reading and writing works in this system enables the possibility of decentralized data storage. In the decentralized database, instead of agents directly writing the new entry, an event with the entry and its new value would be created, which every other agent needs to process and update in their database. This is possible in the depicted system without any changes or further synchronization needed under the assumption that every write-event reaches every agent guaranteed. For this thesis, we chose not to implement the data storage decentralized for simplicity reasons and instead went with the blackboard approach. If an external system not part of the multi-agent system needs data from the production system, the external system also accesses the blackboard, for example, when the ERP-system needs finishing dates for the production jobs.

Event broker

The last of the central elements is the event broker. It is responsible for distributing the events between the agents. It functions in the classical publish-subscribe pattern. Filtering happens topic-based, and there are two topics in the system. One includes all events for the scheduling process, and the other includes the sign-in and sign-off events for the agent directory. Events always include the creating agent's name and any information needed by the receiving agents to process it. All workplace agents process every event for scheduling. It always starts with the receiving agent checking if the event is relevant for itself. If not, it skips further processing of the event. For the following description of the decentralized scheduling algorithm (chapter 6.2), we assume that all events always reach their destination. What happens if this is not the case will be discussed afterward in chapter 6.3.

Workplace agent

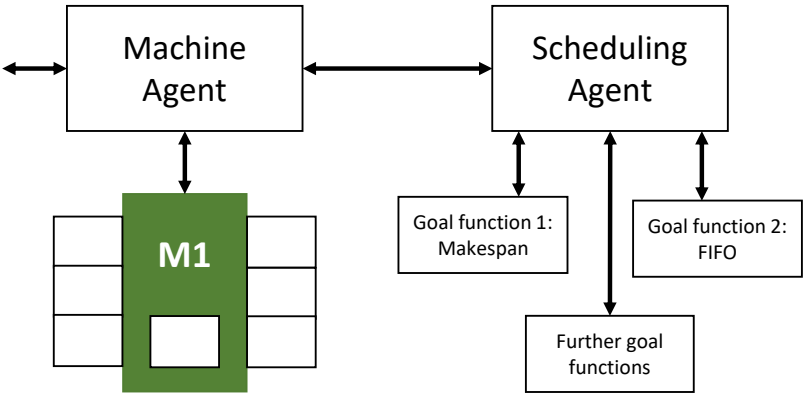

Figure 6.2: Structure of the workplace agent

Each workplace in the production system gets its workplace agent. If there are parallel workplaces, like M4-1 and M4-2 in the figure, they each have

their own agent. Figure 6.2 depicts the inner workings of the workplace agent. The workplace agent consists of two different agents internally, which function the same for every workplace. One is the machine agent responsible for handling the workplace and the only element to interact with it. Its tasks include handling jobs in the workplace, starting the processing of operations, and requesting transportation of jobs to its workplace.
The other one is the scheduling agent, which does all of the content described in chapter 5. The scheduling agent has no direct access to the workplace; the machine agent provides an interface to its scheduling agent containing all the necessary data. The scheduling agent's interface offers methods to add and remove operations, which the machine agent uses to add, when it gets new operations from the operation generator, and remove when an operation has finished processing. Once the scheduling finishes, the scheduling agent pushes the resulting plans into the machine agent, who then starts processing them. The scheduling agent has an inbuilt possibility to exchange the local sequencing algorithm according to the local objective function that measures the result. Examples include, also shown in the figure, the makespan, which is the content of this thesis, and a FIFO sorting, which we used for tests. It is possible to implement additional functions (e.g., a setup time minimization) without changing the agent's structure. The machine agent processes all external communication. This means that it first evaluates incoming events and either processes them or relays them to the scheduling agent. When the scheduling agent wants to publish an event, it creates the event and sends it to the machine agent, which sends the event to the event broker.

6.2 The Decentralized Scheduling Algorithm

Before we can discuss the changes and additions to the scheduling algorithm, we need a method to start the scheduling process in the multi-agent system. After the system finishes initializing, the first scheduling starts with

an event created by the operation generator. The operation generator is responsible for starting planning processes because it is the only element in the multi-agent system that knows how many jobs currently wait for production. Newly created jobs enter the system via the operation generator as described before. Every time an operation finishes, the machine agent checks if it was the last operation in the job's process sequence. If it was, an event is created, which informs the operation generator that a job has been entirely produced. From this information, triggers to create the start scheduling event can be derived:

- Based on the number of available jobs...
  - ...entering the system.
  - ...leaving the system.
- Schedule...
  - ...all available jobs.
  - ...only a fixed maximum number.

Within an empty production system, we can set the number of jobs to arrive in the system before the first scheduling starts. Afterward, we can set a different parameter that controls the interval after how many arriving jobs scheduling starts regularly. Alternatively, it can not plan according to the created jobs but try to keep the system's jobs close to a constant like in the CONWIP-principle. For this case, the operation generator compares the number of scheduled jobs to the number of finished jobs and creates an event to start scheduling if a certain amount of jobs has been finished. In both cases, the operation generator does not have to trigger the scheduling for all available jobs. An input parameter sets the maximum number of jobs that the system can schedule at the same time. If it is limited, the operation generator only takes the first jobs according to the time of their creation until it reaches the maximum. The list of jobs to be scheduled is sent to the workplace agents with the event that start the planning process. If there is no limit, the scheduling agents interpret it to schedule all available jobs.
As described before, once the machine agent has finished processing an operation, it removes the operation from the scheduling agent via the inter-

face between them. This, however, ignores the possibility of parallel agents, which would not know that the operation has finished. Therefore, while removing the operation from all planning lists, the scheduling agent checks if it has parallel agents. If it has, it creates an event containing the ID of the operation to remove (Parallel Remove Event). Upon receiving an event of this type, an agent checks if a parallel agent sent it and then also removes the operation from its planning lists if that is the case. The scheduling algorithm's general procedure and objective function are the same as described in chapter 5. Figure 6.3 shows them once again, including the added synchronization points:

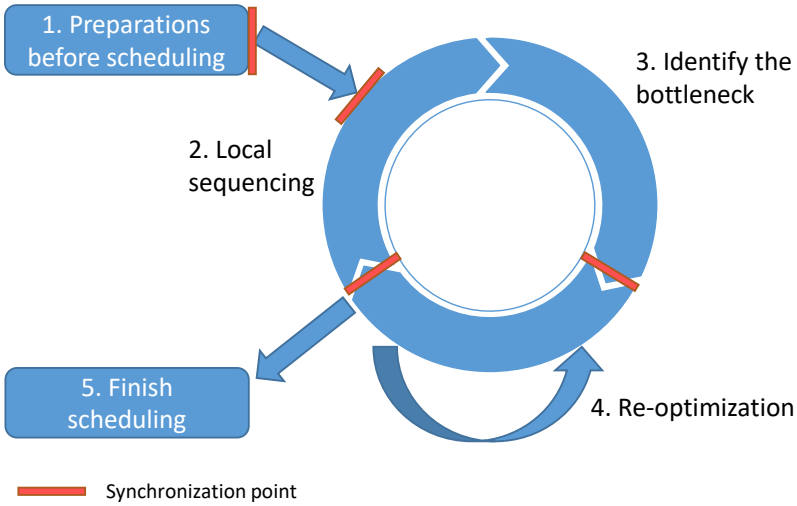

Figure 6.3: Flow chart of the scheduling algorithm including synchronization

In the following, we will mention the scheduling algorithm's steps again, focusing on the additions needed to make the algorithm work in the multi-agent system. This includes the explanation of all types of events and the different synchronization points. In general, a synchronization point describes a point during the scheduling process at which an agent will wait with continuing until all other agents are also at the synchronization point.

It is needed because it guarantees that the information needed for the next step is updated correctly. All synchronization points are implemented with the help of events. Once an agent reaches the point, it will send out an event. Most events are specific to a single synchronization point, but a few are used for more than one if agents need the same information during different steps of the scheduling process. One example of this is the event that the local sequencing has finished. It can be used during steps two and four of the scheduling process. When an agent receives an event related to a synchronization point, it saves the event and its information locally. Each time the agent processes an event of this type, it checks if all agents have arrived at the synchronization point. If that is the case, the scheduling process continues. If not, the agent waits for the next event of the same type. It is crucial to note that the agent will also update the data concerning itself the same way. When an agent sends out an event, it will immediately forget all information sent out and only learn it again once its own event is processed. This is necessary to guarantee the same order of steps and availability of information in all agents. Furthermore, the agent can tell if all agents have reached the synchronization point by comparing the number of received events with the number of agents taking part in the scheduling. This works because we assume that all events always reach their destination and agents never fail to function. Table 6.1 describes all events used for scheduling in the decentralized system. The events are listed chronologically in the order they are first used during a scheduling process.

6.2.1 Preparations before Scheduling

The scheduling agent first needs to get information about the other workplaces in the system. The information is present in the agent directory, and the agent updates it at the start of every scheduling process. An agent saves information about other workplaces in two separate lists. One contains all agents taking part in the scheduling process (equal to all agents registered in the agent directory), and a second list all agents parallel to itself. The agent saves all information needed about other agents in the two

Name	Description
Parallel Remove	Removes operations from parallel agents
Start Planning	Starts the planning process
Delayed Transport	Informs a predecessor agent that a transport will be delayed
Fixed Jobs	States that an agent has finished the first part of preparations
Preparations	States that the preparations before scheduling are finished
Parallel Planning	Sent out after a parallel agent has sequenced an operation
Result Local Plan	Distributes the result of the local sequencing
Plan Fixed	Sent out once a new bottleneck agent has fixed its local sequence
Re-Optimization	Distributes the result of a local re-sequencing
Iteration Finished	Information that an agent has finished the re-optimization step
Announcement	Announces a negotiation
Query	Contains a negotiation proposal
Response	Contains a response to a Query event
Commit	Informs about the result of a negotiation
Commit Response	States that an agent has realized the result of a negotiation
Parallel Restart	Tells a parallel agent to restart its local sequencing

Table 6.1: List of all events used for scheduling in the multi-agent system

lists' entries during the scheduling process. The remainder of this chapter will mention specifically which data it saves at the points the data is updated. There are two synchronization points during the preparation phase. The first one happens after the agents have determined which operations they do not need to plan again since they will remain the same from the last scheduling process (Fixed Jobs event). This information is relevant for parallel agents since they otherwise might plan operations that are already done on a different machine and is the only content of the first synchronization point. The second synchronization happens after agents finish

the preparations to start the local sequencing simultaneously (Preparations event). Furthermore, they exchange the information obtained during the preparation at the second synchronization point, including the lists of all predecessors, the points in time at which the agents can start producing new operations, and the start and end times of operations taken over from the last scheduling process for each agent.

One aspect of the preparations from chapter 5.2 included more than one workplace. If the delay at a workplace influences the preceding workplace, the preceding workplace agent needs to be informed. A workplace can calculate the required delay with the final schedules from the last planning process. Afterward, if a preceding agent needs to be informed, the Delayed Transport Event is used to inform the receiving agent about the delayed operation, and the new time its transport will start. The receiving agent then incorporates the delayed transport in its plans and adjusts them if needed.

6.2.2 Create a Local Sequence

Creating a sequence at a given workplace is done locally. Therefore, there is no need for any changes to the procedure as long as it is not at parallel workplaces. The necessary information to create a buffer reservation needs to be spread between the agents, however. For this, the agents always include their new local sequence in the events used to indicate that local sequencing has finished (Result Local Plan and Re-Optimization Event). Agents save these plans and use them to look up the end times of preceding operations when needed. This information is also present in the blackboard. It would, however, be harder to find the correct information there. The reason is that the blackboard is organized on a job basis and does not contain information about sequences on workplaces. So, the agent would need to build the sequence of its predecessor with the information from the blackboard first. Therefore, it is faster to exchange all local sequences between the agents.
Parallel agents need to build the aforementioned group to create a local sequence for parts of the scheduling process. Before building a group, each parallel agent can update the key variables locally. In the sequencing al-

gorithm, parallel workplaces always only plan one operation at a time before stopping again. Afterward, the parallel workplaces need to choose one that will plan the next operation, which they achieve with the help of an event (Parallel Planning Event). Every time an agent with parallel agents has planned an operation, it sends out an event consisting of the ID of the operation it just planned and its start and end time. Since all different local sequencing types use this event, it also includes an indicator in which step the agent currently is. In this case, the indicator states local sequencing. Only agents parallel to the creator of this event process it, while all others ignore it. They then update their list of parallel agents with the information about the newly planned operation and remove that operation from the list of operations that still need planning during their local sequencing. If the list of operations is empty after the removal, the agent finishes the local sequencing immediately. If there are still operations to be sequenced, the agent can now identify which parallel agent will plan next from the list of parallel agents. The identification works as described in chapter 5.3.2; if it is the agent in question, it continues the local sequencing. If it identifies another agent, the agent waits for the next event of this type. This situation works the same way before the first operation is sequenced. There, agents have information about the earliest time their parallel agents could start the first operation from the preparations before scheduling. The agent with the earliest possible start time starts local sequencing first.

6.2.3 Identify the Bottleneck

Identifying the bottleneck of the current iteration contains the next synchronization point in the multi-agent system. After the local sequencing finishes, an agent can locally calculate its objective function value. It then sends out this value in an event containing the agent's name and its value (Result Local Plan event). All agents save the value locally to be able to identify a bottleneck. The synchronization happens here because an agent who has finished sequencing needs to wait for all other agents' results. Once the event has arrived from all agents, it can identify a bottleneck. The agent that

identifies itself as the bottleneck then writes its local sequence's results into the blackboard. Afterward, the bottleneck agent sends out another event stating that the writing process is finished (Plan Fixed event). Upon receiving the event that a writing process has finished, the agents note which agent has become the new bottleneck. All agents that are bottleneck then start the re-optimization step.

6.2.4 Re-optimization

Re-optimization is the most complex step in the multi-agent system. For easier understanding, the content is split in two parts. The first describes the re-optimization step without an infeasible plan occurring, the second a negotiation in case of an infeasible plan.

Standard procedure

The first important task of the re-optimization procedure is to identify which agent is next to create a new local sequence. Before, we had the central scheduler, which would do the local planning in the order of workplaces becoming a bottleneck. Now, the workplaces' agents need to know for themselves if it is their turn or that of another agent. From the previous step of the procedure, an agent knows which agents are bottlenecks. The agents do not only note which agents have become a bottleneck; they also keep a separate agent list in which they add an agent once it has become a bottleneck. Since the list is empty at the start, it is automatically sorted by the order in which agents became bottlenecks. The agents then use a simple counting system to determine which agent must plan next. Assuming a situation in which the list of bottlenecks has five entries and the agent we look at is third in the list. Agents re-sequence according to the order they became a bottleneck; thus, this agent knows that it must plan once the counter reaches three. The next time it has to plan after a full round, equalling five plus three. Therefore, it would plan again once the counter hits eight. This procedure can continue indefinitely. An agent calculates this in the same way. If the current counter matches an expected number,

the local re-sequencing starts. If the agent does not need to plan at the current counter, it increases its counter and still checks if the agent matching the counter is parallel to itself. In that case, the local re-sequencing also starts. To be able to handle parallel agents, the counter mentioned above is also kept by non-bottleneck agents. If their parallel agents plan, they also do the re-optimization even if they are not a bottleneck. However, those agents do not save their results since they are not relevant for the re-optimization. If a bottleneck agent plans out of its turn because of a parallel agent, it saves the plan and uses the same result again once it is its turn to save computation time. This means that all agents of a parallel group do their re-optimization according to the position of the agent who first became a bottleneck within the group. Once the local re-sequencing ends, the agent or agents who planned update the blackboard with their new plans. Afterward, only the agent who matches the counter sends out an event that it has finished the local re-optimization (Re-Optimization Event). Parallel agents, which planned and updated the blackboard earlier than their turn, immediately send out the event once they notice it is their turn since they have already done the re-optimization.

Upon receiving a re-optimization event, an agent saves the information in it, consisting of the new local sequence, and a decision variable if there was a change in the local sequence. The new local plans of all agents are needed to create buffer reservations, as described earlier. Then, the agent determines again which agent is to re-sequence next. Once all bottleneck agents have created a new local sequence, the current repetition of the re-optimization completes. Each agent then checks if there was a change of a local plan for any agent. If that is the case, the next re-optimization repetition starts. If there is no change, each agent creates an event to inform all agents about that fact (Iteration Finished Event). The event indicates the completion of a whole iteration of the planning process and is the last synchronization point. Only bottleneck agents send out the event. All agents wait until they have received the event from every bottleneck agent to make sure the re-optimization has finished. Once that is the case, the non-bottleneck agents start with the local sequencing (chapter 5.3) again.

Agents who already are a bottleneck but have a parallel agent, which is not a bottleneck yet, must also do the local sequencing. Should there be no more non-bottleneck agents at this point, the complete planning process finishes. Finishing the scheduling works the same as in chapter 5 since it is done locally at each workplace.

Negotiation

At any point during the re-optimization step, an agent can detect an infeasible plan and start a negotiation to solve the problem. As we have seen in the previous section, this requires creating local sequences on multiple workplaces. Therefore, the multi-agent system needs a process to inform agents that they have to participate in a negotiation and a method to resolve negotiations. Since agents can work in parallel, the negotiation system is designed with that fact in mind. It can handle any number of negotiations starting in parallel by sorting the negotiation queries time-wise and only allowing one negotiation to start once the previous one has ended. The following explanations describe a negotiation in chronological order. Specific steps may not be happening in every negotiation, which is explained when it happens.

Creating a proposal

When an agent detects an infeasible plan and wants to start a negotiation, it sends out a negotiation announcement event (Announcement Event). This event does not include a proposed sequence or recipients yet; instead, it only serves as a reminder so that all agents know that there will be a negotiation. Upon receiving the announcement, every agent in the production system creates a local reminder about this negotiation without knowing if they will or will not participate in the actual negotiation. Furthermore, an agent always assumes that it currently takes part in its reminder list's oldest entry. This is necessary to ensure that every agent processes the negotiations in the same order and that no new negotiation is processed until the previous one has finished. Every time a negotiation finishes, the agents re-

move its entry in the reminder list and see if there is another negotiation waiting. If there is, the agents process the negotiation accordingly. Apart from creating the reminder, no further action is necessary for the negotiation announcement event. The only agent continuing to work on the announcement is the creator of it. When it is time to process its announcement (once it is the oldest entry in the reminder list), the agent first checks if the negotiation is still required. There might have been other negotiations in the meantime, resulting in a situation different from when the agent created the announcement. Table 6.2 lists the possible situations, which the following paragraph describes.

Case	Was there a negotiation?	Was it successful?	Did I take part in it?	Conclusion
#1	No	-	-	Continue negotiation
#2	Yes	Yes	Yes	Cancel negotiation
#3	Yes	Yes	No	Restart local re-sequencing
#4	Yes	No	Yes	Continue local re-sequencing
#5	Yes	No	No	Continue local re-sequencing

Table 6.2: Possible situations for the creator of a negotiation

In case that no negotiation happened between the announcement and the creator processing it (#1), the negotiation still needs to be done since the whole multi-agent system is still in the same state. If the announcing agent took part in a successful negotiation (#2), its negotiation is no longer needed. As a result of the successful negotiation, the creating agent has already switched to a different local sequence. Then, it cancels the negotiation by sending out an event (Commit Event), which signals a canceled negotiation. If the negotiation stops at this point, the local re-sequencing can continue directly. There, the agent only has to send out the event that it has finished the re-optimization repetition since it always creates a complete local sequence during a negotiation it did not start. If the agent, who

wanted to negotiate, is part of a parallel group, this process is slightly different. Instead of directly continuing the negotiation at the point it was stopped, the parallel agent starts from the beginning again. It also notifies the other parallel agents that the local re-sequencing is restarted (Parallel Restart Event). This step is not strictly necessary. However, it may improve the result's quality since parallel agents may distribute the operations between their workplaces again and may find a better distribution and therefore better local sequences than they had before.
If there was a successful negotiation and the agent did not participate (#3), the agent's situation has changed. Therefore, it must create a new local sequence from the beginning based on the new system state. If it can create a new sequence without an infeasibility, it cancels the negotiation. Otherwise, the negotiation will still take place. If there was any non-successful negotiation (#4 and #5), the agent first tries to continue the local re-sequencing at the same point it was stopped before to check if the infeasibility still exists. If it does not exist anymore, the negotiation can stop; otherwise, it continues. It is important to note that if the agent still wants to negotiate in these three situations, it does not need to send out a new announcement. Instead, it uses its old announcement and directly creates a proposal for the negotiation according to the rules described in chapter 5.5. If it would send out a new announcement, it might get stuck in a loop of announcements. This happens if the system state always changes between the announcement and the processing of the negotiation.

If the negotiation continues, the agent can create a proposal in the same way as in the central algorithm. Next, assuming a proposal was created, the agent determines all direct predecessors of the operations in the proposal. If one of the predecessors is part of a parallel group, it adds all agents in that group. This stays the same at any point during a negotiation: Every time a predecessor is part of a parallel group, all agents of the group are added to the negotiation. It might be in special cases that the list of recipients is empty. Then the negotiation is finished as not successful immediately instead of continuing with a query event. This happens if all preceding operations are fixed in time already and cannot be changed anymore (for ex-

ample, because they are currently being processed or taken over from the last plan). In that case, the only possible solution is to include the necessary delay into the local sequence. After the list of predecessors is determined and contains at least one agent, the agent sends out a query event for the proposal (Query event). It contains the plan, a list of recipients (the predecessors), and the agent's name.

Processing Queries

All agents in the recipient list of a query event process it directly because the query can only be sent out if the negotiation is on top of every agent's reminder list. A processing agent saves the creator of the event and the proposal. Afterward, there are several possible situations, which the agent has to differentiate:

1. Agent receives a query asking it to take part in the negotiation the first time and...
   - ...it can create a response directly.
   - ...it cannot create a response yet.
2. Agent receives a query asking it to take part in the negotiation a second or more time and...
   - ...has already created a response.
   - ...has not yet created a response.
3. Agent receives a query of a negotiation it is not part of, and the query does not mention the agent.

If it is the first time that the agent receives a query event for the current negotiation, it checks if it can create a response to the query. For this, it creates a list of its predecessors based on the operations in the proposal. It is not allowed to add all agents to this list, however. Instead, there are some limitations, which prevent query circles, in which two or more agents are querying each other and wait endlessly for a response. The agent cannot send a query to the requester of the negotiation, and all other agents already part of the negotiation, so it does not include them in the predecessor list.

With these means, a temporary topological sorting of agents arises during a negotiation. Agents sort by when they entered the negotiation, with the original creator on top of the chain. If the agent's list of predecessors is not empty, the agent sends out its query event to the agents in its list.

An example for the distribution of query events is shown in figure 6.4. Numbers on the arrows stand for the order in which queries are sent if agents were not working in parallel.

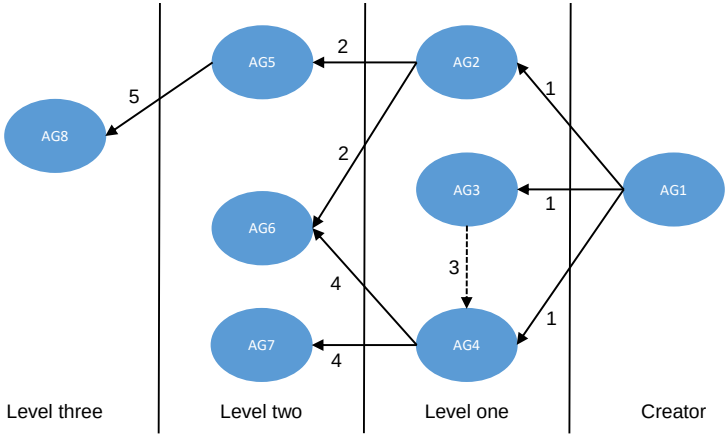

Figure 6.4: Example for the spontaneous sorting of agents during a negotiation

The figure describes a negotiation with eight participating agents. Agent 1 is the creator of the negotiation. It identified agents 2, 3, and 4 as its predecessors. Since these three agents enter the negotiation simultaneously, as part of the recipient list of the same query event, they get placed on the same level within the negotiation. Next, agent 2 identifies agents 5 and 6 as its predecessors. They are not yet part of the negotiation, so it can send a query to them. However, they are not on the same level anymore since they are now two queries away from the creator and therefore on level two.
Agent 3 only has agent 4 as a predecessor. However, agent 4 is already part of the negotiation and on the same level as agent 3. Therefore, agent 3 is not allowed to send a query to it. The situation is slightly different for agent 4. It wants to send a query to agent 6, which agent 2 already made part of the

negotiation. This time, agent 4 can send a query even though 6 is already in the negotiation since agent 6 is on a lower level. For agent 6, it makes no difference if it has to respond to only agent 2 or also agent 4. Agent 4 also sends a query to agent 7. It gets placed on the same level as 5 and 6 since it is the same number of queries away from the creator. Last, agent 5 sends a query to agent 8, which gets its separate level again.

Before an agent uses the list of recipients for its query event, it can do another step to improve the final topological sorting by considering agents not yet part of the negotiation. The agent checks if it would create a possible future query circle to send a query to another agent from the recipient list. Each predecessor is checked independently for this. It searches for predecessor circles with each agent's predecessors list created during the scheduling preparations. A predecessor circle is a chain of agents that would send query events to the following agent in the chain, ending at the current working agent again. This circle would already not be closed by the agents because of the prohibitions explained earlier. However, the agent can forecast that situation, and a decision, which agent is allowed to send a query, can be reached by ordering the agents in the possible circle according to their position in the full agent list. If an agent detects a potential negotiation circle now, it can include the agent in the recipients only if it is listed earlier in the full agent list. If it is behind, it removes the agent from the recipients, and the working agent will receive a query event from it at some later point. This addition helps with finishing the negotiation faster and also creates better results the closer the production system structure is to a line, in which always the agent later in line would ask the agent in front of it. This leads to situations in which agents with more predecessors are generally higher in the topological sorting than agents with fewer predecessors. Of course, in a system where every agent is a predecessor of every other agent, it helps less because there is no structure to identify and use.

There are two important points about this topological sorting of agents. First, it works in every case, not depending on the number of agents in the negotiation or the predecessor relations. Second, it works in parallel or non-parallel computing systems. It does not matter if all agents process the

events simultaneously or if only one agent works on each event at a time. Both cases lead to the same result because also in parallel systems only one negotiation is open at a time and the afore-mentioned process leads to the same result no matter the processing order of queries. The presented rules also guarantee that the negotiation process always finishes and never enters a deadlock or livelock.

If the predecessor list of an agent (other than the creator) is empty, it can create a response to the query. The agent creates a new local sequence according to the proposal. The process to create a new local sequence was described in detail in chapter 5.5. An answer to a query contains a list of all agents receiving the response (equal to all agents, which sent a query to this agent), the responding agent's name, the created local sequence, and an evaluation of the new local plan (Response Event). The agent can receive queries from multiple agents for the same negotiation, as shown in the example. If a second query for the same negotiation arrives, the agent reacts in two possible ways. If it has already created and sent out a response before, it directly takes that response and sends it to the querying agent. If it did not create a response yet, it only needs to save the name of the sender of the query. The process for creating a response has already started in this case. Once it finishes, the response only needs to be sent to an additional agent. Parallel agents use the same method here, which they also use during the other local sequences. They plan alternatively until all operations have been planned between them, and then each sends its response.

Lastly, specific agents not in the recipient list of a query event also need to process it. The negotiation's creator updates its list of all participating agents, which it will need at the end of the negotiation. Agents not (or not yet) participating in the negotiation also keep a list of all participants. They do so because they need to know which agents are already participating in order to determine their recipient list if they join the negotiation.

Processing Responses

When an agent receives a response event, it saves the event as part of the current negotiation. Once responses from all agents, which the agent sent a query event, have arrived, it can create its response. It does not matter how the responses arrived, as the topological sorting ensures that an agent creates responses at the correct time for the whole negotiation. If the agent processing the response event is not the negotiation creator, it creates a new local sequence. This is, however, only done if all responses received are positive. If one of the responses states that the proposed new plan is infeasible, creating a new local sequence is skipped, and it directly reports the infeasibility in the response. In either case, the agent creates its response event for all agents that require it. The evaluation included in the response is the worst evaluation between itself and all the responses it received. Therefore, in the end, the creator of the negotiation always sees the worst impact of the proposal.

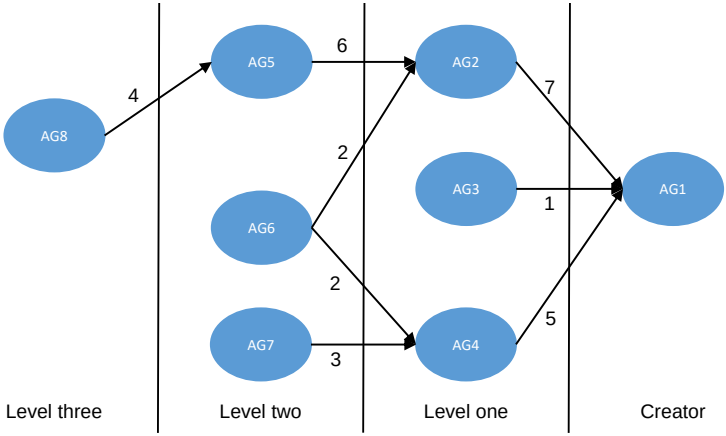

Figure 6.5: Order of the responses in the earlier example

Again, we present an exemplary situation in figure 6.5. It shows the same negotiation as before; this time, the arrows indicate the order of responses. The first response created comes from agent 3. It was not allowed to send another query earlier, so it can directly create a response. Next to create a

response is agent 6. It first sends its response to agent 2 and then directly afterward is queried by agent 4. Then, it sends the same result also to agent 4. Agent 7 can also directly respond to the query of agent 4. For the third level, agent 8 can respond to agent 5. Now, the responses of agents 6 and 7 have arrived at agent 4. So, agent 4 itself can create a response for agent 1. After agent 5 and then 2 have created their responses, the creator can decide about the negotiation result. As for the queries, this works independent of the order of processing responses (and queries). Since there is a defined sequence of when each agent can create its response, delays in the processing an event do not have an influence on the result of the negotiation.

Finishing the negotiation

Once the creator has received all responses it is waiting for, it can finish the negotiation. For this, it first evaluates all the responses it has received. If any evaluation among the responses is worse than the delay or infeasible, the negotiation was not successful. If the result is satisfactory and therefore better than having a delay, the creator does a new local sequence. If the creator itself detects no infeasibility in the local sequence, the whole negotiation was successful. In both cases (successful or not), the negotiation result spreads with a finishing event (Commit Event). It contains the result of the negotiation, all participants, and the duration of the delay in case of a negative result.

Agents who participated in the negotiation, but are not the negotiation's creator, now implement the negotiation result upon receiving the commit event. This can be done in parallel by all agents. If the negotiation was successful, the agent replaces its local sequence with the one created during the negotiation and updates the blackboard with the new information. Furthermore, it cancels its local re-sequencing for the current re-optimization repetition if it is not already finished at this point. Instead, it takes the plan created during the negotiation as the result of the current repetition. If there was a successful negotiation, another re-optimization repetition is needed since there were changes in multiple agents' local sequences.

If the negotiation result is negative (or non-successful), the agent checks if the resulting delay has an effect on its workplace. That is the case if the agent's workplace also processes the delayed operation's job. If the associated operation has not been planned during the current local re-sequencing, its latest finishing time is updated to include the delay, and it marks that the operation must not finish earlier than this time. If the local re-sequencing of the operation is finished, the plan is updated to accommodate the delay, which is written to the blackboard afterward.
In the case of a canceled negotiation, the agent needs to do nothing as it only happens when the agent creating the negotiation could independently solve the problem. One additional step for non-canceled negotiations is to confirm to the negotiation's creator that the necessary local changes have been implemented and written to the blackboard. This is done via one last event, not containing unique information (Commit Response Event).
After doing all necessary local steps, if an agent was part of the negotiation, all agents save the result of this negotiation in a list to use if they have requested or will request their own negotiation during the current re-optimization repetition. In this case, they require the information to determine if that negotiation is still needed, as described earlier. Finally, all agents except the creator remove the negotiation from their reminder list and see if another negotiation query is waiting to be processed.

Only the creator of the negotiation processes commit response events. It waits until it has gathered the confirmations of all participating agents. Then, it can do the same steps as the participating agents. The reason why that had to wait until now is that in the case of a non-canceled negotiation, the local re-sequencing continues at the point the infeasibility was detected. But to be able to do so, the agent needs the updated information from the blackboard. After receiving all confirmations, it can be sure that the blackboard is up-to-date again. The negotiation is completed with the creator removing its negotiation from the reminder list and continuing the local re-sequencing.

6.3 Termination of the Decentralized Algorithm

After having described the functionality of the multi-agent system, there are still two essential aspects left. The algorithm has to guarantee that it generates a solution for every possible input data case and that every solution is executable. Only with these two conditions fulfilled, the system can be used in a real-world production system. For the decentralized algorithm as described in this chapter so far, this would not always be the case. While it guarantees an executable plan, it is not able to find one for every possible set of input data. Therefore, this section gives an overview of additional measures to ensure that the algorithm terminates with an executable solution every time and also a reasoning of why every created plan is executable. For this, we present a detailed look at the possible cases in which the scheduling could fail either during planning or execution. They include a description of how the algorithm deals with them for each case. In general, the system tries to avoid deadlocks from the start instead of repairing them.

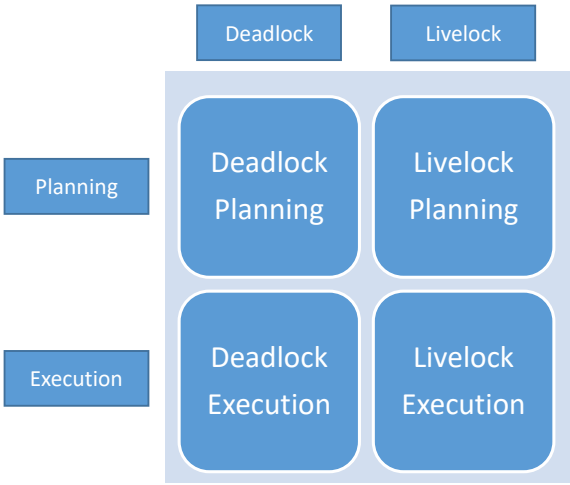

Figure 6.6: Possible termination problems of the algorithm

Two characteristics can identify the algorithm's possible failures regarding convergence and problem-free execution of the resulting plan. The first is the point in time at which the problem occurs. It can occur during the planning process described in this and the last section, or it can only happen during the plan's execution. The other characteristic is the type of problem. The algorithm cannot converge either because of a deadlock or because of a livelock. These alternatives and starvation as a special case of a livelock were already described in chapter 3. The two characteristics can be combined in any way, leading to the four possible ways the algorithm could not work correctly, shown in figure 6.6.

The remainder of this section explains the four possible problem scenarios and the algorithm's solutions to prevent them from happening in order of their difficulty, starting with the easiest. As we will see, it is possible for almost all of the scenarios to be solved within the algorithm itself. However, some scenarios remain where we could not guarantee a solution for every possible case within the algorithm. Therefore, an additional external solution is necessary, presented in the last sub-chapter.

6.3.1 Deadlock during Execution

The first case we want to look at is the example of a deadlock during this plan's execution. Figure 6.7 shows an example. In this situation, workplaces two and three currently work on an operation, and all of their six buffer slots are occupied by jobs. Now, one of the jobs at workplace two needs transportation to workplace three and vice versa. Assuming instantaneous transport and no other workplaces in the system, this would not be possible, and a deadlock has occurred.

Although this situation appears during the execution of a plan, the algorithm already prevents it during the planning phase. We defined that in a feasible plan, every job has to have a clearly defined place during the whole timespan the plan covers, can not be scheduled at two places at the same time, and each job that gets planned has to finish. Every operation within each job has to have a planned start and end time, and all buffering times

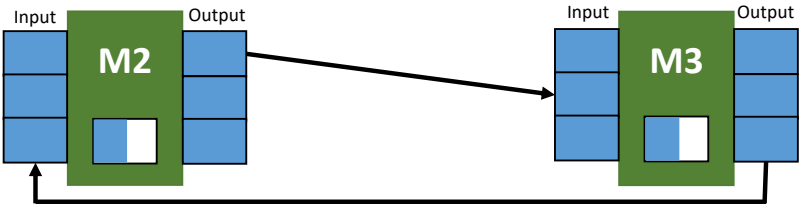

Figure 6.7: Example for a deadlock during the execution

scheduled. All scheduled times of a job must uphold the feasibility criteria. With these, a situation like in the example can never happen since the algorithm would detect the infeasibility during the planning process and use its method (the negotiation) to solve it. In case of a non-successful negotiation, the algorithm delays an operation and therefore puts a time gap in the plan of a workplace. With this measure, every plan with a deadlock can be made feasible. If nothing else works, the algorithm could create time gaps so large that only one job is in the production system at any given time. Then, a deadlock cannot occur. Of course, the objective function value also would be rather bad. Therefore, this measure creates gaps as small as possible each time it is used.

6.3.2 Livelock during the Execution

The problem of livelock during the execution can only appear in applications with multiple scheduling runs at different times. If the system plans only once, this problem can not occur since, according to the section above, the plan will be functional and finish every job. On closer look, not all types of livelocks can occur during the execution. Instead, only the problem of starvation can happen here. Several steps are needed to solve it. First, it can happen as a result of parallel workplaces offering to process the same operation. Figure 6.8 shows an example where two parallel workplaces, M4-1 and M4-2, repeatedly switch the planned processing of job J1 between them. In the first plan, the workplace one wants to process it. Then, a new plan is made, and the operation is now planned on workplace two and

therefore transported to it. Afterward, it could again happen that before workplace M4-2 processes it, a new plan is made, and the operation now has to go to workplace M4-1 again.

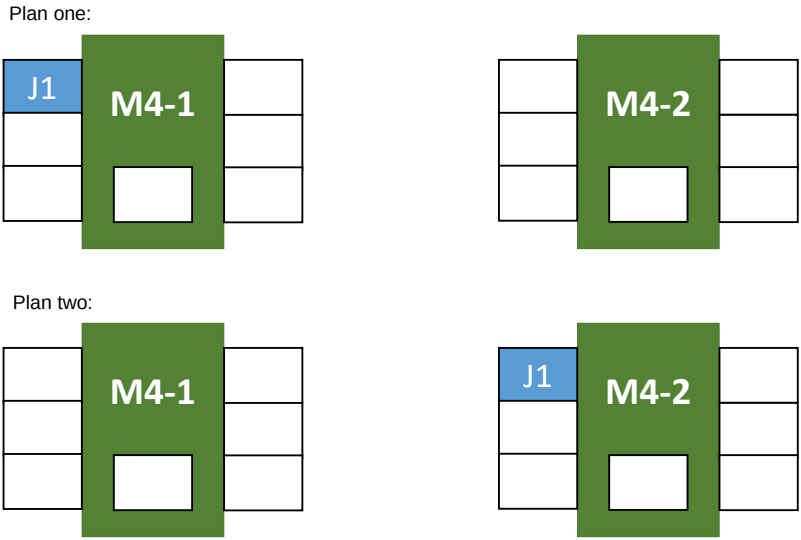

Figure 6.8: Example for a starvation during the execution

To prevent this situation from occurring, we restricted the agents in the operations they are allowed to plan during the preparation phase before the scheduling, which was mentioned in chapter 5.2. There we introduced the rule that a workplace must not plan an operation, which has already physically arrived at another workplace capable of processing it. This rule, however, does not entirely solve the problem. It could still be that the single workplace always plans the operation at the end of a local sequence and always has this one operation left from the old plan when it creates a new one. Then, the operation stays in the input buffer of that workplace for an indefinite time. For this situation, we introduced a second rule that operations belonging to a job, which has finished the preceding operation, are not scheduled again and keep their start time from the previous plan. These two rules, in combination, ensure that a job always gets finished once it

started but still leaves one last case of possible starvation. They do not prevent a job from never being started in the first place. This problem solves itself as long as the production system's utilization is below 100% and with an unlimited amount of time available. For practical applications, we want to prevent a large spread of individual jobs' makespans, if possible. Therefore, we can not rely on the fact that the problem will solve itself over time. We include the time between creating a new job and its planned starting time in the objective function. This makes longer waiting jobs prioritized over jobs that only arrived in the system a short time ago, preventing inconvenient jobs from being delayed indefinitely and starved.

6.3.3 Deadlock during Planning

Describing how to prevent a deadlock during the algorithm's run time or, even further, proving it, is not easily done. In the algorithm described in chapters 5 and 6, every step is clearly defined, and each agent always knows what to do next or for which other agents it has to wait. As long as all events always reach their destinations and no agent has a failure so that it does not respond anymore, the system will never enter a deadlock. For this algorithm's practical application, where the mentioned assumptions of chapter 4.2 are not given, we included several features to lead the system out of a deadlock. The first is a timeout of agents. Each agent has to send an event to the agent directory each minute that it is present and functional. If it fails to do so, the agent directory assumes that the agent is not part of the system anymore and cancels the current planning process in case one is running. Afterward, the planning process is started with the same parameters again. The same also happens if a new agent enters the system. Since agents update their view of the system at the start of the planning process, they also remove or add agents.
The problem of events not reaching agents is not so easily solved and is not implemented in the algorithm yet. An event never being created can only happen if the agent fails and that case is dealt with already. If the event reaches only some but not all needed agents, the situation can be resolved.

Agents already know which event from which agent they are waiting for to continue working. If it does not arrive within a predefined time, they could ask the other agents if they have received the event and can resend it. For this purpose, the agents could save a copy of all events between two synchronization points. If an event is created but does not reach a single agent, the event broker has failed, or there is a connection problem to the sending agent. Both cases should be solvable with the timeout feature, but we will have to analyze this in more detail once it is implemented. The features mentioned here are not needed to apply the algorithm in the scope of this thesis and are not part of the version of the algorithm used to create the numerical results. This question is therefore not answered in this thesis.

6.3.4 Livelock during Planning

A livelock during the algorithm occurs when the algorithm continues to plan indefinitely and never finds a solution. This can only happen during the algorithm's re-optimization step since the other steps have a clearly defined end and the general iterations a maximum number of steps. Starvation can not occur during the planning phase. The reason is similar to the deadlocks during planning described before. It simply does not happen that an event is not processed because the other agent has more important processes to work. All communication between agents is well defined and there are no livelocks within a single agent. This guarantees that an agents will process all events and, therefore, that agents waiting on another agent will receive an answer within reasonable time.

Livelocks divide into four different sub-cases again similar to the overall termination problems:

1. Feasible plans with the same objective function values
2. Infeasible plans with the same objective function values
3. Feasible plans with worse objective function values
4. Infeasible plans with worse objective function values

While describing the solutions for the cases, two other points have to be kept in mind, preventing more straightforward workarounds to the livelock situations. The first is that the livelock can affect any number of agents. Were it only a maximum known number of agents in a livelock, the resolution could be easier. The second point is that we also have to include systems where each agent can be the predecessor and successor of every other agent. Again, were the agents to be sorted (for example, in a line production system), solutions would be easier to obtain.

Feasible plans with the same objective function values

In this first case, the agents create multiple feasible plans, which also have the same objective function value. However, they cannot decide on which of those to take. The situation is equal to the livelock example from chapter 3.2.2, just with agents instead of people in a hallway. It happens because agents react to the local sequences of the other agents from the last re-optimization repetition. If two or more similarly react to each other, a loop can appear. For this, livelock prevention was implemented into the agents. Agents save the results of all their local sequences during the re-optimization in a hash value. After each local sequence, they check if they have created the same sequence at a prior point. They only search sequences starting from the second to last created, since if the last and the current plan are identical, the agent is not in a loop but instead signals that it is okay with the sequence and wants to finish the re-optimization. This only happens if there was no negotiation during the current repetition because we assume that the agents make progress while negotiations happen. If an agent detects itself being in a loop, it stops making new local sequences and instead always sends the last created one for all future repetitions. Which local sequence it sends does not matter here since they all result in the same objective function value. If multiple agents detect themselves being in a loop, only one is allowed to break it. Agents decide this by looking up if they are the first agent to become a bottleneck among those who want to break the loop. Only the highest ranked agent then actually does it. That agent also informs the other agents that it will now try to break

the loop with an event (Loop Interrupted Event). All agents then save the number of the repetition the event was sent in.
The sending of old local sequences leads to the other agents not changing their plan anymore as a reaction to the agent, which has now stopped making new sequences. It may, however, be that it is not enough that one agent stops making new local sequences. If more repetitions have passed than there are bottleneck agents after an attempted loop interruption, and the same problem still exists, the next agent stops making local sequences. This has to continue at maximum until the second to last agent in the loop stops making plans since one agent can never go into a loop on its own.

Infeasible plans with the same or worse objective function values

If an agent detects an infeasible plan, the negotiation process starts. If one or several agents do not correctly implement the negotiation results, a loop can appear. In that case, the agents try the same negotiation repeatedly, but they will never solve the problem. The negotiation process described in chapter 6.2.4 does not prevent an agent from simply reverting the changes it made in reaction to a negotiation during the next local re-sequencing. Figure 6.9 shows an example in which this problem could occur. In this situation, two workplaces (M1 and M2) currently have the local sequences on the top left. Assume that M2 now starts a negotiation, leading to a delay for $OP2$ of two time units (top right). In the next repetition, M2 correctly keeps the delay in its local sequence. M1, however, did not and reverts to the old start time (bottom left). This prompts another negotiation from M2 with the same result as in the previous repetition (bottom right). With the rules so far, this happens because M1 tries to optimize its local sequence which means producing $OP2$ as early as possible even if knows from the negotiation that M2 needs a delay.

To prevent this behavior, we implemented two additional functionalities, one for successful negotiations, one for non-successful ones. When a negotiation fails, the preceding agent delays an operation so that the successor has enough buffer space again. To prevent the agent from planning it

Current Situation		
	M1	M2
Operation	Start Time	Start Time
OP1	10	20
OP2	15	22

After Negotiation		
	M1	M2
Operation	Start Time	Start Time
OP1	10	20
OP2	**17**	**24**

Next Repetition		
	M1	M2
Operation	Start Time	Start Time
OP1	10	20
OP2	15	24

After Negotiation		
	M1	M2
Operation	Start Time	Start Time
OP1	10	20
OP2	**17**	24

Figure 6.9: Example for infeasible plans

earlier again in the following local re-sequencing, it saves the start time of the delayed operation. Before the next local re-sequencing, when updating the key variables, it makes sure that it does not sequence the operation that caused the infeasibility earlier than its start time after the negotiation. It does so by setting the t_{earl} variable of the operation to the maximum of the determined t_{earl} and the saved start time. This prevents the same problem from occurring again.

If there was a successful negotiation, the agent that started it skips the next local re-sequencing because it finished the last repetition with a working local sequence. But since not all agents might have participated in the negotiation, they need time to implement the changes into their sequences. Therefore, it is a benefit not to directly change the new plan again.

These two additional features solve this problem. With them as part of the algorithm, all negotiation results are correctly implemented. This type of livelock does not occur in the version of the algorithm presented in this thesis any longer. So far in this subsection, we assumed that the objective function value stays the same over multiple iterations. If the same problem occurs, but the objective function gets worse over time, it is actually be-

cause of the same reason. The negotiation process (and its included dealing with failed negotiations) is also not correctly implemented locally in the agents. It can lead to delay after delay without solving the problem. The above solutions, therefore, solve both types simultaneously. In contrast to the first livelock problem (feasible plans with the same objective function values), however, we can not proof that it will never happen in any random example as there may still be cases where the implementation does not help avoiding a problematic situation with infeasible plans which we might not even have seen yet. This type of problem did not occur in any of our uses of the algorithm for our tests, which is a good sign.

Feasible plans with worse objective function values

The final and most complex solution is the case of feasible plans, which get worse over time. It happens when an agent reacts to a delay in another agent's plan with a delay of one of its operations, and in turn, the other agent delays the same operation again. Figure 6.10 gives an example of this. It is a problem solely of the local sequencing, where the agents cannot find the appropriate solution for the current situation. In the example, on the left side, two exemplary local sequences are presented. Assume that in the next repetition, M2 delays the operation $OP2$ by two time units. The reason for this delay is not important here. Also assume that the following operation, $OP1$, is scheduled directly after $OP2$. In that case, M2 also delays it by two time units as there is no other option. Now, since M2 is planning first, it must have become a bottleneck earlier than M1. Thus, M1 tries to accommodate the change and delays both operations by two time units to continue delivering $OP2$ just-in-time and to minimize the amount of time $OP1$ needs to be buffered. This, however, does not change the situation for M2 at all as the relative difference of start times on the two workplaces has stayed the same. Therefore, in the following (not pictured) repetition, M2 would incorporate another delay of two time units, which M1 again would also include. This situation would then go on endlessly and create a livelock.

Current Situation			
M1		M2	
Op.	Start Time	Op.	Start Time
OP1	10	**OP2**	20
OP2	15	**OP1**	22

Next Repetition			
M1		M2	
Op.	Start Time	Op.	Start Time
OP1	**12**	OP2	**22**
OP2	**17**	OP1	**24**

Figure 6.10: Example for feasible plans with worse objective function values

The solution implemented for this problem was already included in the local sequencing description in chapter 5.3. Delaying the operations on M2 creates a time gap to the operations prior to $OP2$ in the local sequence, into which other operations can fit after a few repetitions. The agent makes use of the time gap, if this case occurs, by switching the sequence of operations if the time gap is big enough for the following operation to fit into it. In the case of the example, at some point, M2 would switch the order and plan $OP1$ first once the sum of the incremental delays becomes larger than the processing time of $OP1$. As in the section before, we cannot proof that this solution works guaranteed for every possible case as they can get arbitrarily complex in number of workplaces and operations included in the problem. In contrast to the previous problem, this one also still occurs irregularly during our testing with random production systems. In all cases presented in chapter 7, it did not occur, however.

6.3.5 Additional Solution to Termination Problems

Since we could not proof the algorithm's termination in every possible case, we had to implement a solution outside of the planning process. All cases described before in this sub-chapter only occur in the re-optimization part of the algorithm. Agents already track their local sequences for every repetition as part of the problem avoidance. This can be used to cancel the planning process if no result is found after the agents have non-successfully tried the implemented solutions for termination problems. For this, the operation generator saves all agents' local sequences and creates a hash value

of the current repetition. If the same result occurs more often than there are workplace agents in the system, the planning process stops. Since the non-creation of a plan is not an option, the planning process immediately restarts afterward. One small change is made before restarting because the whole planning process is deterministic, and just restarting it would lead to the same problem again. When restarting the planning process, the operation generator reduces the number of jobs planned by one. It has a list of all jobs sorted after their time of creation. It removes the newest job and starts the planning process again. This reduces problem complexity for the agents while only slightly reducing optimization potential. If still no plan is found, the number of jobs can be reduced by one again. In theory, the operation generator can do this until there is only a single job left. In that case, only one, and therefore optimal, solution exists, which will be found by the algorithm. For the removed jobs, the operation generator has two possibilities. The operation generator either includes them in the following scheduling process or adds them to the end of the schedule with the FIFO dispatching rule. This procedure always guarantees a planning result although with a reduced optimization potential. In our numerical experiments, following in the next chapter, the system never applied this method as it found a solution for all jobs in every experiment without this feature.

7 Numerical Evaluation of the Multi-agent System

After answering the research questions by presenting the complete multi-agent system, including the algorithm and all of its extensions, it also has to be shown how the algorithm performs numerically in practical applications. In this chapter, the algorithm is tested in three different settings. Before describing the settings and their results, an overview of the experimental setup is given in chapter 7.1. Afterward, the decentralized multi-agent scheduling algorithm is tested in two different practical applications, for which we used real data from the two companies. Finally, the system gets tested in settings consisting of randomized examples, whose input parameters represent characteristics not present in the application examples. This is done in chapter 7.4. We chose the FIFO dispatching rule as our basis for comparison mainly because the current production planning systems in both real-world application systems use it, and it gives good results in systems that plan without a due date as our examples do.

7.1 Setup of Experiments

Before we can discuss the experiments, we can state some points about the general procedure, relevant to all of them:

1. The required input parameters to set up an experiment.
2. Describing the FIFO algorithm used as a base for comparison.
3. The process of an experiment, including our stop criterion.

7.1.1 Input Parameter of Experiments

To be able to set up an experiment, all input parameters have to be defined beforehand. The possible input parameters for an experiment are the following:

- Number of workplaces
- Number of different job types/variants
- Number of different processes in the production system
- Assignment of workplaces and processes (if multiple workplaces offer the same process)
- Number of buffer spots at each workplace
- The process sequence for each variant
- The processing time for each operation on each workplace
- The number of pieces in a job

With these parameters given, the multi-agent system and the FIFO algorithm can begin scheduling. However, even with all of them defined, there are still many possible tests to do. In a production system with a given amount of variants and jobs to schedule, the number of possible scenarios equals to $|N|^{|V|}$. Exemplary, with ten variants and ten jobs, it would still be possible to create ten billion different scenarios for an experiment since one can pick this amount of different combinations of the ten variants to fill the ten jobs. After deciding on one of the scenarios to use, each of the combinations can still arrive at the system between a single one and 3.7 million different sequences $|S|$ depending on the order of creation of the ten jobs. This value is dependent on the number of different variants $|V_s|$ in the scenario, which can be between one and ten, and the number of times K_v a variant is present in the job batch of the experiment. Formula 7.1 calculates the possible number of job sequences given the number of variants, and their distribution within the job batch.

$$|S| = \frac{|N|!}{\prod_{i=1}^{|V_s|} K_i!} \tag{7.1}$$

Moreover, these exemplary numbers given here were only for a single production system. As one can see, the possible number of experiments for performance analysis of this multi-agent system is virtually unlimited. Therefore, it is practically impossible to explore even a single production system in all of its possibilities (statistical methods help here) or to create a meaningful coverage of the relevant different production systems. Weber et al. (2019) mention the same problem stating a need for standardization of job shop test data, which has not been done yet. Therefore, in the following sub-chapters, we focus on analyzing exemplary systems, primarily based on actual data, to show some key characteristics of the algorithm instead of choosing many random examples.

7.1.2 Implementation of the FIFO Dispatching Rule

Our software implements FIFO as a dispatching rule in the agents. Each agent only knows the operations it has to do immediately and tries to process them in the order in which it was notified of them. In the example of ten jobs, only ten operations are planned at the beginning of the scheduling. Once one of them finishes, the following workplace able to process it adds it at the end of its work queue. Parallel workplaces communicate so that operations get added to the workplace, which has less work left in its queue. Transports are always carried out as soon as possible. As long as a workplace has free spots in its input buffer, it tries to transport all jobs in the work queue to itself until they are at the workplace or the input buffer is full. The FIFO dispatching rule does not prevent deadlocks, and depending on the topology of the system, they may happen during our experiments. Since it would be biased to add a fixed time penalty for each deadlock, as we do not have data about how long it takes to repair, we decided to resolve all deadlocks instantaneously by switching two (or more) jobs creating a deadlock in the production system. The only thing tracked afterward is the number of times FIFO had to do this, which we will evaluate at the end of an experiment.

From this procedure, it directly follows that a complete plan of the operations is available at no point. Therefore, to obtain results for the FIFO algorithm, a simulation is used to simulate the jobs' production. Results can then be gathered after all jobs have finished in the simulation. In comparison to this, the algorithm of this thesis only needs to do its planning. Since we are using deterministic values for each step, the simulation does not need to carry out the created plan. Instead, it can assume that everything will happen as the algorithm has planned. We still use the same simulation for the multi-agent system, but only to set up the system and create the experiments in the same way. After the algorithm has finished planning, the simulation stops again. Another important point from the FIFO algorithm's description is that it depends on how the simulation creates the jobs. The result's quality can drastically change depending on generating a sequence of the jobs, either good or bad for FIFO. On the other hand, the multi-agent system is almost independent of this order. The creation sequence of the jobs, as long as the total jobs are equal, only influences tiebreaker situations during the local sequencing.

7.1.3 Process of Experiments

To obtain a fair base for comparison between FIFO and the algorithm, we decided to define that one experiment contains a fixed number of scheduled jobs with a fixed distribution of variants between them. This experiment is then done once for the deterministic algorithm and repeated with different job arrival sequences for FIFO until it fulfills the following statistical criterion.

We know that there is a maximum possible number of sequences. Therefore, we can use an estimator to estimate the statistical population based on our observed data. We do this with a student's t-test deciding if the observed average makespan represents the average of all possible makespans. We defined the makespan as the time between starting the first operation and the last operation's end time. We decided to choose the confidence interval's width based on the first FIFO simulation run. We said that the

final makespan result should deviate less than a fixed percentage (0.1% or 1%) of the first simulation run's makespan. As an example, a makespan of ten hours will have a confidence interval of plus and minus 36 or 360 seconds. Which value we take depends on the input values since some tests with a high makespan and large fluctuations would have a runtime of several months otherwise. After each finished simulation run, we can compare the number of finished runs with the number of required runs to achieve statistical significance. We use a statistical significance of 95% if not otherwise mentioned. If the number of required runs is smaller or equal to the number of finished runs, the experiment can stop.

Before starting the actual experiments, we decided to do one experiment to test our procedure. For this, we planned five jobs in the first application example and let the simulation run for all possible 120 different arrival sequences of those five jobs. After obtaining the result, we also used the iterative procedure, which stopped after 30 iterations. The confidence interval was plus or minus 50 seconds on an average makespan of roughly 4.3 hours. In the end, the average makespan of all possible runs resulted in 15,347 seconds, and the iterative procedure gave 15,374 as a result, which was within the defined confidence interval. For all this chapter's experiments, the simulation and algorithm ran on a single CPU core and were all done on the same system. Therefore, when mentioning run times, those do not include the speed-up potentially gained by parallel execution of the agents' computations.

7.2 First Application Example

The first application example stems from the production of an electronics component manufacturer from Germany. The part of the production we looked at is a final assembly, including testing and packaging of the finished goods. All required components for the products are already finished before this point. As we designed the algorithm with this production system in mind, we had to make no adjustments to the available data. The assembly contains fifteen manual and automatic steps on sixteen different

workplaces, meaning that there are two workplaces doing the same step. A total of eleven different variants are assembled here with similar but slightly different process sequences through the system. Process times for single pieces range from 20 seconds up to two minutes. Each variant has a fixed number of pieces in a job, differing between 15 and 30. Four of the variants make up for 80% of the total production volume, while the other seven variants are less frequently produced versions making up the remaining 20% of jobs. Each workplace in the system has six buffer spots, of which we use one as the output buffer and five as input buffers. Table 7.1 summarizes the input parameters; the detailed data, including all process sequences for this system, can be found in the appendix.

Workplaces	Variants	Process times	Pieces per job	Structure
16	11	19-120 sec	15-30	Job shop

Table 7.1: Input parameters for the first application example

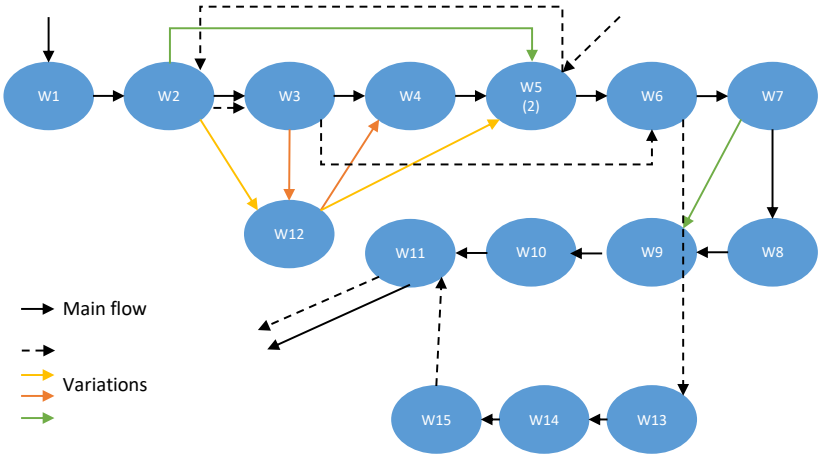

Figure 7.1: Material flow of the first example

Figure 7.1 shows the material flows in the system. There are two main flows, which belong to the two product families. If one takes the black line as a

basis, there is only one example of a backward flow in the second family with the dotted black line. The thick black line makes up about 60% of the production volume and the dotted black line another 25 %. Variations of the main flow also only happen alongside the thick black flow in the same general flow direction. Therefore, this system is close to a production line except for one flow starting at W5 before going to the line's start.

We decided to schedule two different job batch sizes for this system. The first contains 20 jobs, the second 50 jobs. It takes roughly eleven hours of real-time to finish the 20 orders, representing one planning run per shift if including some overlapping of planning runs. The bigger batch takes about 24 hours to finish, which equals one planning run per day if one assumes a three-shift model. In both batches, each variant is represented as closely as possible to its average overall production volume. The exact composition of variants within the batches can also be found in the appendix. Table 7.2 provides an overview of the results of this application example.

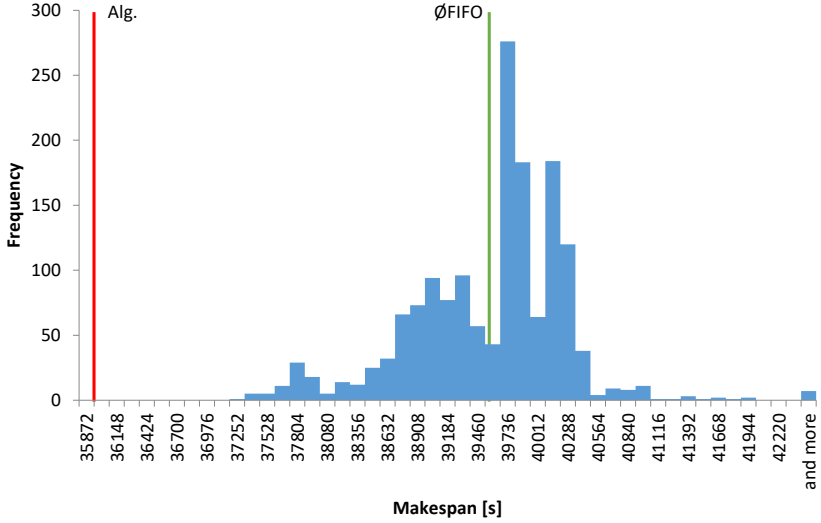

Figure 7.2: Makespan results of the first experiment

We start with the results of the smaller batch size. It took our simulation 1578 iterations to achieve statistical significance, which took about 24 hours in total. The average makespan was calculated with a 95% significance and a confidence interval of plus and minus 39 seconds. Figure 7.2 shows the results of the makespan distribution. The x-axis depicts the makespan results of all simulation runs, which we classified to improve readability. The y-axis shows the number of runs in each class. The fastest FIFO run took 37,114 seconds to complete. On the other side of the spectrum, the longest run took 50,690 seconds. This run seems to be an extreme outlier since the second slowest run only took 44,619 seconds. On average, it took 39,510 seconds to finish each of the FIFO runs, which the right line in the figure indicates. No deadlock occurred during any of the FIFO simulation runs. On the other hand, the algorithm, depicted by the left line, needed only 35,935 seconds to finish the 20 orders. Therefore, it saves 9.0% on the average FIFO result with a maximum saving of 29.1% (or 17.1% if we ignore the outlier) and a minimum improvement of at least 3.2%.

Experiment	Small	Large
Batch size	20	50
Simulation runs	1,578	392
Average FIFO [s]	39,510	91,423
Best FIFO [s]	31,114	88,540
Worst FIFO [s]	50,690	94,122
Result Alg. [s]	35,935	81,809
Difference	-9.0%	-11.3%
Deadlocks FIFO	0	40
Avg. deadlocks per run	0	0.1
Runtime Alg. [s]	5	25

Table 7.2: Results of the first application example

Results of the larger batch size show a similar picture. Here, the simulation needed 392 runs to achieve significance with a confidence interval of plus and minus 92 seconds. Figure 7.3 shows an average makespan of 91,423 seconds with a maximum of 94,122 and a minimum of 88,540 seconds. In

total, the spread of the FIFO results is much narrower than it was for the smaller batch size. Since the plan's absolute time is larger than in the first experiment, the absolute gap between FIFO and algorithm also becomes more significant as the algorithm's plan needed 81,809 seconds to finish. Percentage-wise this makes it faster by 11.3% on average, 7.6% at least, and 13.1% at maximum. With this batch size, the FIFO rule also produced some deadlocks, with 40 of them occurring over the 392 runs. As mentioned before, these did not influence the makespan result of the simulation. Therefore the difference between algorithm and FIFO would probably be more considerable in a real-world application.

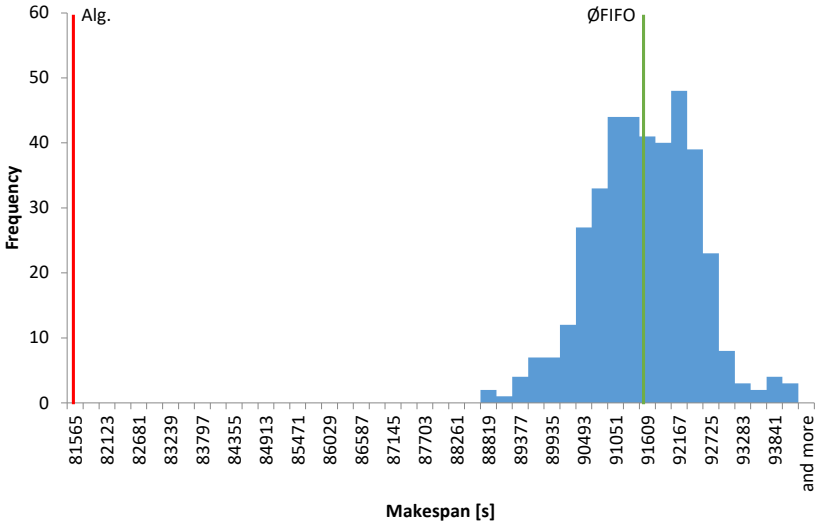

Figure 7.3: Makespan results of the second experiment

One promising result to note about the algorithm is the time it needs to create the plan. The plan for 20 jobs takes about five seconds to finish, and the plan for 50 jobs takes 25 seconds. As mentioned, these times resulted with the whole agent system running on a single core in the CPU. Therefore, none of the calculations were done in parallel, possibly further reducing the required computation time.

To obtain further information about the reasons behind the algorithm's better performance, we tested the following scenario. In a production system with unlimited buffers, performance gains can only be obtained by choosing a better sequence of operations at the workplaces and, therefore, in the whole system. If buffers are limited, the usage of the available buffer spots is another factor influencing performance. The FIFO rule does not limit its buffer usage and instead always fills them as much as possible, leading to congestion problems within the production system. To single out this fact, we also reran the same tests with unlimited buffer space. Table 7.3 gives the results. It shows the different minimum, maximum and average makespans for FIFO for the limited and unlimited buffer case.

Jobs	Category	Limited buffer	Unlimited buffer	Difference
	Minimum	37,114	36,902	-0.6%
20	Average	39,510	38,589	-2.3%
	Maximum	44,619	40,092	-10.1%
	Minimum	88,540	85,993	-2.9%
50	Average	91,423	87,466	-4.3%
	Maximum	94,122	89,040	-5.4%

Table 7.3: Comparison of the limited and unlimited system

From this table, it can be seen that the performance gains of the algorithm do not only come from the better usage of the buffer spots. FIFO with unlimited buffers had better results by 2.3 and 4.3% on average. That was, however, not enough to close the gap to the algorithm. Even if FIFO has unlimited buffer spots available, the algorithm still produces a better result than the best FIFO performance. The algorithm itself does not change results between the limited and unlimited case. After analyzing the algorithm's buffer usage in this example, it shows that it never uses more than two input buffer slots of a workplace at any given time. Therefore, the solution stays the same as long as two or more input buffer slots are available.

7.3 Second Application Example

The second application example also comes from a German company, this time from the healthcare business. Again, the data comes from the assembly area, including testing. The packaging is not a part of this process. In this case, we could not take the data without adjusting it to build this example. The main reason for this is that the company produced about 1,000 different product variants over one year in this production system. As the algorithm is suitable for short-term planning of a shift or a day at maximum, it made no sense to model all variants for our example as almost 30% occur only once a year. Only 164 of all variants are produced more than once a month. To build our example, we decided to model only the most produced variants. In the data, 28 variants occur at a minimum once a day on average. This system can produce a job batch of roughly 100 orders taking one day of real-time as the basis for a production plan again. If we take the production orders for the reduced number of variants, the job batch contains the rarest of the 28 variants exactly once. We, therefore, decided to only model these 28 variants, which make up roughly one-third of the total production volume.

The entire system consists of 62 different processes at 194 workplaces. Many of them are special processes for exotic variants, however. The 28 modeled variants only use 19 of those processes. We model them on 21 workplaces, meaning that the two most utilized processes are offered by two workplaces each. Processing times range between 1 and 240 seconds. We use a fixed number of pieces in a job for all variants since the difference in production volume between variants was already taken into account via the job batch's composition. As the whole system has unlimited buffers, we set the buffer size to be the same as in the first example with six per workplace. Table 7.4 summarizes the input parameters.

We do not present the production system's material flow in a graph here because the reduced system results in a flow shop. The total production system has a flow shop area at the beginning, followed by a job shop for the remaining processes. Since we reduced the number of variants, the whole

Workplaces	Variants	Process times	Pieces per job	Structure
21	28	1-240 sec	20	Flow shop

Table 7.4: Input parameters for the second application example

production system ends up as a flow shop. In the total system, after the flow shop part, most variants only have to do very few (0-4) additional steps on workplaces specialized for their product families. The job shop then only builds within the processes of the product families. As we removed the processes the 28 variants do not use, we removed many exotic variants of the product families and ended up with a flow shop. We compare three different job batch sizes this time. This time they are a bit larger with 50, 100, and 200 jobs for the experiments, as average job process times are faster in this example. Table 7.5 presents the numerical results. Figure 7.4 shows the result of the experiment with 50 jobs.

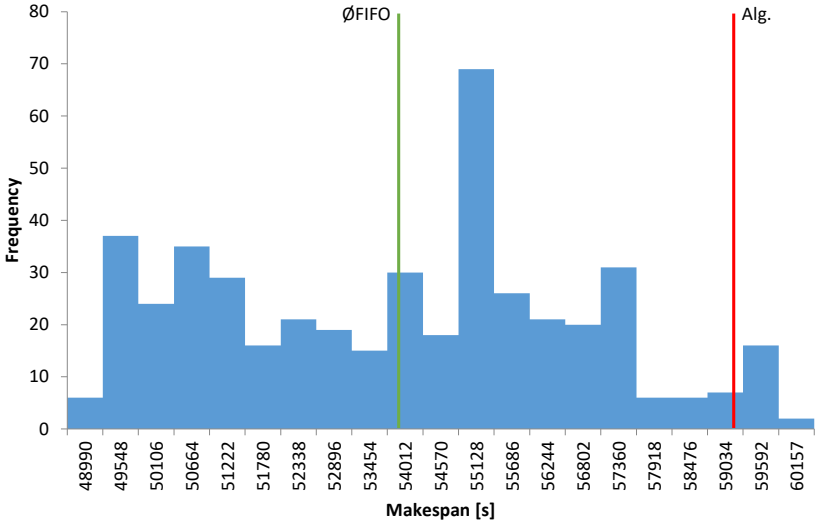

Figure 7.4: Makespan results of the third experiment

We increased the confidence interval to 0.5% of the first simulation run for this production system. As the diagram shows, the algorithm does not perform well at all in this example. It is 10.2% worse than the average FIFO result and only 1.9% better than the worst FIFO run.

Experiment	Small	Medium	Large
Batch size	50	100	200
Simulation runs	454	96	10
Average FIFO [s]	53,580	94,855	182,026
Best FIFO [s]	48,433	91,582	180,892
Worst FIFO [s]	60,157	100,576	183,668
Result Alg. [s]	59,025	97,611	194,199
Difference	+10.2%	+2.9%	+6.6%
Runtime Alg. [s]	12	33.5	140

Table 7.5: Results of the second application example

The same also shows for the larger job batch size depicted in figure 7.5. Results here are slightly better, with the algorithm being 2.9% worse on average and also 2.9% better than the worst case. This trend does not continue onwards, as a job batch size of 200 jobs leads to the algorithm being 6.6% worse on average. On the bright side, the algorithm's run times are even faster than they were in the first case. The algorithm needs 12 seconds to schedule the 50 jobs, 33.5 seconds to schedule 100 jobs, and 140s to schedule 200 jobs.

Every single job has to visit the bottleneck workplace in this example. As long as that workplace is 100% utilized, the total plan cannot be improved. The algorithm does not manage to achieve this. One reason could be that the bottleneck has too little influence on the planning result. This is a very general reason and is hard to prove or improve. A different argument for the worse results is much easier to explain. In a flow shop, deadlocks cannot occur. Therefore, it is always advantageous to have the buffers as full as possible to never arrive in a situation where a workplace has no work left because it has to wait on the preceding process. Precisely this is how FIFO works, and it shows in the results. The basic idea of the algorithm is to be care-

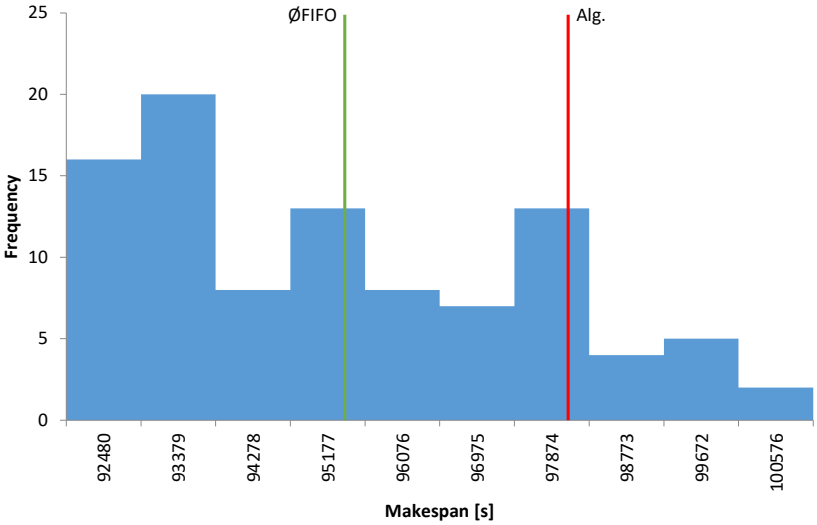

Figure 7.5: Makespan results of the fourth experiment

ful with buffer utilization to prevent deadlocks. That prevents it from being able to create good plans for flow shops. In this situation, the algorithm needs to be more aggressive with its buffer usage. It must, however, achieve a balance; if it uses buffers too aggressively, it will be bad for complex system structures. Then, the algorithm would run into the same problem as FIFO. As there is another point where the algorithm currently is not aggressive enough in its buffer usage, we will discuss this argument and possible solutions more in the conclusions. It is not worth using the algorithm in the reduced and simplified form of the example in its current form. More generalized, it is not worth using an algorithm this complicated to plan a flow shop. This fact also aligns with the discovery in the first example that the algorithm does not get better with unlimited buffers. If buffers are unlimited, the same logic as in flow shops applies, where one always wants as much work-in-process as possible to maximize throughput of the system.

7.4 More Complex Production Systems

It is now clear that the algorithm does not function well in flow shops. It has worked well in the first example, which is still very close to a flow shop with only one sequence not in the flow direction. Therefore, this section presents examples for five more complex production systems. Table 7.6 shows the values for each parameter, which were tested:

Workplaces	Variants	Process times	Pieces per job	Structure
20	10	1-25	1-20	Job shop

Table 7.6: Input parameters for the random experiments

The production systems all consist of 20 workplaces. These workplaces offer between 14 and 19 different processes. A maximum of three workplaces offer one process in the examples. Each system consists of 10 different variants, which consist of 7 to 19 operations. Which processes a variant needs to complete, and their order was chosen randomly from the available processes.

Experiment	1	2	3	4	5
Simulation runs	174	406	278	231	315
Average FIFO [s]	22,859	28,168	21,124	19,862	24,506
Best FIFO [s]	19,453	22,217	17,340	15,822	20,325
Worst FIFO [s]	26,629	35,558	27,856	24,515	31,026
Result Alg. [s]	21,806	21,095	25,321	20,654	22,817
Difference	-4.6%	-25.1%	+19.9%	+4.0%	-6.9%
Deadlocks FIFO	1,070	3,628	4,511	3,152	2,886
Avg. deadlocks per run	6.1	8.9	16.2	13.6	9.2
Unsolvable deadlocks	4	14	22	22	9
Runtime Alg. [s]	46	51	50	39	54

Table 7.7: Results of the random experiments

Figure 7.6 shows the material flow of random experiment two. Almost every possible flow is present in the system. Connections in the example are

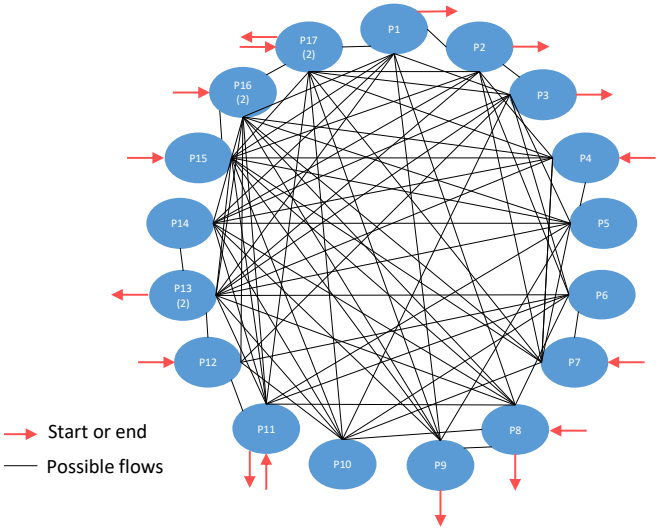

Figure 7.6: Exemplary random material flow

either one-way or go both ways. Processing times are random from 1 to 25 seconds per piece, while a job consists of 1 to 20 pieces. We decided to do the tests with the input parameters with a fixed buffer size of six available buffer slots. Every experiment schedules 50 jobs. The detailed data for each experiment is listed in the appendix. Table 7.7 shows the results of the experiments.

As one can see, the results were of mixed quality. They do not allow to make a general statement about the quality of the algorithm. FIFO produces an enormous amount of deadlocks in these complex systems ranging from 6 to 16 deadlocks per job batch on average. We also encountered deadlocks during the execution of FIFO, which we could not solve by switching the place of two jobs. In that case, we canceled the simulation run and started the next. It seems as though the algorithm can work fine in these random examples but cannot do so for every possible case. Runtimes still are excellent, planning jobs for several hours in under a minute.

Comparing the amount of time the algorithm loses in experiments three and four with the average deadlocks per run, FIFO would be faster if deadlocks take less than a minute (experiment four) or less than 4.5 minutes (experiment three) to solve. Therefore, one could argue that the algorithm's result is still acceptable in experiment four, as detecting and solving a deadlock in practice may take longer than a minute. The result of experiment three is, however, not acceptable under any circumstance. So far, we were not able to identify an apparent reason for this behavior. The buffer usage in these experiments shows the same behavior as in the first application example. Even with five input buffer slots available, the algorithm never uses them, giving another hint that the algorithm needs to be more aggressive in that area.

8 Conclusion

Although decentralized planning systems offer many advantages, not many of them are in use today. This thesis contributes by developing a multi-agent system with decentralized planning. At the end of it, we summarize the results of our research. Chapter 8.1 does so according to the research questions of chapter 4.3. An outlook with further extensions and improvements to the presented system is given in chapter 8.2.

8.1 Summary

This thesis aimed to develop a decentralized scheduling algorithm that incorporates limited available buffer space at each workplace. For this, we have extended the Shifting Bottleneck heuristic by Adams et al. (1988) to include the additional characteristics. It was implemented in a decentralized multi-agent system offering flexibility to changes of the underlying production system and good scalability options. The algorithm can be used to schedule a flexible job shop and has shown excellent efficiency. It enables companies to schedule their production system under the required demands from the market. The content of this work divided into four research questions:

> First: How can a Shifting Bottleneck procedure schedule the problem presented in chapter 4.2?
>
> Second: How can a decentralized multi-agent system executing the algorithm of question one be modeled and implemented?
>
> Third: Does the decentralized multi-agent system guarantee the creation of a production plan for every possible case of valid input data?

> Fourth: Is the production plan created by the decentralized multi-agent system always free of deadlocks, livelocks, and starvation?

To answer question one, we adapted the idea of the SB heuristic to the problem statement. We kept the general iterative procedure, adding only a step at the start and end of it. The necessary significant changes regarding the limited buffer space came from the fact that the procedure potentially creates infeasible plans now. Therefore, it was not possible to limit the algorithm in the time it spends. Instead, it has to make sure that it saves only feasible plans at the end of every iteration. Therefore, the procedures within every step had to change completely to model the new requirements. The additional step at the start of the scheduling allows us to plan in an empty production system and systems already in use. Creating a local sequence on a workplace was adapted to plan the workplace's buffer slots. It also needed several additional scheduling rules to improve the resulting quality under the new conditions. The objective function was slightly changed as we found out that using averages instead of a maximum makespan finishes jobs more regularly, which on the one hand reduces buffer usage as jobs leave the system faster and is also better suited for real-world applications. During the re-optimization, we added a method to solve infeasible plans should any be detected. This negotiation method was developed entirely from the start and can now solve the significant problems created by limited buffers.

Question two was referring to the implementation of the algorithm in a decentralized multi-agent system. For this, we designed a new agent system consisting of four agents being present once in the system and the same workplace agent for each workplace. The workplace agent got an inner structure, so that it consists of two agents, one for scheduling, one for the control of the workplace. Afterward, all steps of the algorithm were analyzed again, and changes to fit the agent system made where necessary. The communication among the agents works with the help of events and a publish-subscribe distribution pattern. For the negotiations to work in the decentralized system, a new pattern was developed. It can handle

any number of negotiations simultaneously and is functional no matter if agents are running on a single or multiple CPU cores.

The algorithm was not able to prevent all deadlocks, livelocks, and starvation itself without additional measures. Therefore, for research questions three and four, the different situations in which one of the three problems can occur were presented and categorized. Livelocks can be further categorized by making the resulting plan worse or not, as well as if only feasible, or also infeasible plans occur during the repetitions. For deadlocks, we can guarantee that the algorithm will produce a planning result that is deadlock-free and also that the decentralized multi-agent system itself will not enter a deadlock. To prevent starvation, we implemented several rules related to the prioritization of jobs to ensure production. According to the assumptions made for this work, the system can never suffer from starvation during execution or planning. Livelocks proved to be the most challenging point to solve. Here, we found ways to avoid typical livelocks in the agent system, and they did not occur anymore during our testing. It was, however, not possible to prove that they will never occur under any given circumstance. Therefore, an additional solution was developed, which stops a planning run, reduces complexity, and starts it again if the system is in a livelock. It guarantees a solution at the cost of optimization potential. In the end, the algorithm as presented in this thesis is able to guarantee that no deadlock, livelock, or starvation occurs during the planning phase or the execution of the plan.

Finally, after answering all research questions, a short numerical study was done. It focused on two real-world application examples. In one of them, the system performed well; in the other, however, not so much. We found reasons for this and will also present a possible solution approach in the following sub-chapter. An excellent point about the algorithm in its current state is the runtime. It can schedule even quite large systems with a high number of jobs in only a few minutes. This fact permits further potential for additional features, which also the following section will describe. As one real-world system is a flow shop and the other only had a single connection against the flow direction, we also tested five systems with random

material flows. In these systems, the algorithm shows mixed results. It prevented the enormous amount of deadlocks FIFO produces but was not able to provide better results consistently. This result shows that the numerous little rules the agent system uses for scheduling are all fine individually but do not work together well in every case. Therefore, more work is to be done on the algorithm to guarantee that it can extend the promising results to every experiment. So far, most of the work went into developing solutions for the characteristics of the system. Improving them is now an essential task for future work.

8.2 Outlook

As it is planned to use this thesis's multi-agent system in a real-world production system, it needs to deal with a few more characteristics. We have identified three of them, which are needed for the algorithm to plan application example one. So far, the algorithm can not merge or split jobs as part of the processing. It is, however, typical if components are produced in the same production area as the final product. The algorithm needs to plan the start time of operations according to finish times from another job. Another point is the inclusion of transport times. We assumed them to happen instantly, but that cannot be. Adding transport times between workplaces makes planning more manageable, as, during transport, a job occupies no buffer slot at any workplace. Therefore, one can view the transport system as an additional buffer, increasing the total number of buffer slots in the system. This in turn reduces the time jobs need to be buffered at the workplaces. Adding transport times will be done within the local sequencing. The last addition is the most complex one. Currently, the algorithm allows multiple workplaces to offer the same process. In reality, it is also the case that one workplace offers multiple processes. This problem can be solved with the help of our parallel groups. However, it might be that these groups get large, which would significantly increase planning complexity. We have not yet identified a final solution to include this feature. While

working on these points, the algorithm also needs to be generally improved regarding result quality.

During chapter 7.3, we talked about a shortcoming of the algorithm in its current version. It cannot plan flow shops well because it is too careful with its buffer usage and only produces operations just in time for the following process. The same happens in another situation, which is also very relevant for practical application. When multiple planning runs are done on the same system, it is advantageous to produce operations more early than needed. This is because the algorithm does not know which operations it has to plan during the next scheduling period. If a workplace has little work to do for a first period and therefore includes gaps but then has to do much work in the second plan, it would have been better to pre-produce during the first period. Currently, the algorithm always plans as if no more jobs will arrive in the future. Similar to the bad performance in the flow shop problem, this point is solved by not producing operations just-in-time but instead, use available buffer slots as much as possible. That is, however, easier described than integrated into the algorithm (we already tried it) since it could lead to many infeasibilities and the algorithm not able to solve them. It will be necessary to find a middle ground. This problem might even be the starting point of another work of this size since it can be generalized towards planning under uncertainty.

Apart from these features for practical applications, countless additional characteristics could be implemented, including all of the advanced models from chapter 2.1.3 for example. While working on the algorithm, we noted numerous smaller improvements, which we could not try out so far. The following list presents some of them:

- No distinction between input and output buffer slots. Instead, the algorithm assigns them dynamically.
- During the local sequencing, allow the algorithm not only to add operations at the end of the sequence.
- Do not set some operations from the previous plan as unchangeable. Instead, plan them as any other operation.

- Remove the distinction between high and low priority operations.
- If several workplaces have the same objective function value, use an additional criterion to decide for a bottleneck.
- Implement more ways to create proposals for negotiations and apply them specifically depending on the situation.
- Add a way to connect several multi-agent systems (e.g., factories at different locations or component manufacturing and assembly).

It will also be possible to implement runtime improvements by redesigning parts of the system's implementation to reduce the required computations. Apart from these small and medium-size additions to the system, the single most extensive additional feature will surely be including stochasticity in the multi-agent system. It can appear in processing and transportation times as well as in the downtime of workplaces. Then, new strategies for dealing with deviations from the plan have to be developed and implemented. As one can see, the work on this system can and will be continued for a long time still.

Notation

Basics

FIFO	First-In-First-Out	5
SB	Shifting Bottleneck	15
d_j	Due date of job j	8
k_j	Number of operations of job j	8
M	The set of all workplaces	8
m	A single workplace	8
M_0	All machines declared as bottleneck	16
N	The set of all jobs	8
n	A single job	8
$o_{i,j}$	Operation of job j on workplace i	8
r_j	Release date of job j	8
t_o	Start time of operation o	15
$t_{p,o}$	Processing time of operation o	8

Scheduling Algorithm

$delay_o$	Delay of operation o in case it is produced late	69
$earlier_o$	Relative bottleneck position of operation o	60
st_j	Job j has started or not	60
$O(m)$	Objective function value of workplace m	72

$t_{avail,o}$	Time operation o is available for processing 60
$t_{earl,o}$	Time operation o can start processing 60
$t_{end,o}$	End time of processing operation o 67
$t_{late,o}$	Time operation o has to finish processing 60
$t_{opt,o}$	Optimal start time of operation o 60
t_{start}	Earliest time a workplace can start processing..... 59
U_m	Set of all operations scheduled at workplace m 72

Numerical Evaluation

$\|K_v\|$	Number of times variant v is in the number of jobs of an experiment..... 116
$\|N\|$	Number of jobs in an experiment..... 116
$\|S\|$	Number of possible job sequences in an experiment 116
$\|V\|$	Number of variants in a production system 116
$\|V_s\|$	Number of jobs in an experiment..... 116

References

Adams, J., E. Balas, and D. Zawack. 1988. The shifting bottleneck procedure for job shop scheduling. *Management science* 34 (3): 391–401.

Amin-Naseri, M., and A. J. Afshari. 2012. A hybrid genetic algorithm for integrated process planning and scheduling problem with precedence constraints. *The International Journal of Advanced Manufacturing Technology* 59 (1-4): 273–287.

Aytug, H., K. Kempf, and R. Uzsoy. 2003. Measures of subproblem criticality in decomposition algorithms for shop scheduling. *International Journal of Production Research* 41 (5): 865–882.

Balas, E., J. K. Lenstra, and A. Vazacopoulos. 1995. The one-machine problem with delayed precedence constraints and its use in job shop scheduling. *Management Science* 41 (1): 94–109.

Balas, E., N. Simonetti, and A. Vazacopoulos. 2008. Job shop scheduling with setup times, deadlines and precedence constraints. *Journal of Scheduling* 11 (4): 253–262.

Bilyk, A., and L. Mönch. 2012. Variable neighborhood search-based subproblem solution procedures for a parallel shifting bottleneck heuristic for complex job shops. In *2012 IEEE International Conference on Automation Science and Engineering (CASE)*, 419–424. Seoul: IEEE.

Blazewicz, J., K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. 2019. *Handbook on scheduling: From theory to practice*. International Handbooks on Information Systems. Cham: Springer International Publishing.

Brandimarte, P., M. Rigodanza, and L. Roero. 2000. Conceptual modeling of an object-oriented scheduling architecture based on the shifting bottleneck procedure. *IIE Transactions* 32 (10): 921–929.

Bratman, M. 1987. *Intention, plans, and practical reason*. Vol. 10. Cambridge, MA: Harvard University Press.

Braune, R., and G. Zäpfel. 2016. Shifting bottleneck scheduling for total weighted tardiness minimization—a computational evaluation of subproblem and re-optimization heuristics. *Computers & Operations Research* 66:130–140.

Braune, R., G. Zäpfel, and M. Affenzeller. 2012. An exact approach for single machine subproblems in shifting bottleneck procedures for job shops with total weighted tardiness objective. *European Journal of Operational Research* 218 (1): 76–85.

Brucker, P. 2007. *Scheduling algorithms*. 5th edition. Berlin: Springer.

Brucker, P. 2012. *Complex scheduling*. Edited by Knust, S. Berlin Heidelberg: Springer.

Bülbül, K. 2011. A hybrid shifting bottleneck-tabu search heuristic for the job shop total weighted tardiness problem. *Computers & Operations Research* 38 (6): 967–983.

Bussmann, S., and K. Schild. 2000. Self-organizing manufacturing control: an industrial application of agent technology. In *Proceedings fourth international conference on multiagent systems*, 87–94. Boston: IEEE.

Bussmann, S., and K. Schild. 2001. An agent-based approach to the control of flexible production systems. In *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No. 01TH8597)*, 2:481–488. Antibes-Juan les Pins: IEEE.

Büttner, R. 2011. *Automatisierte Verhandlungen in Multi - Agenten - Systemen*. Wiesbaden: Gabler.

Carlier, J. 1982. The one-machine sequencing problem. *European Journal of Operational Research* 11 (1): 42–47.

Cayo, P., and S. Onal. 2020. A shifting bottleneck procedure with multiple objectives in a complex manufacturing environment. *Production Engineering* 14 (2): 177–190.

Chen, K.-P., M. S. Lee, P. S. Pulat, and S. A. Moses. 2006. The shifting bottleneck procedure for job-shops with parallel machines. *International Journal of Industrial and Systems Engineering* 1 (1-2): 244–262.

Cheng, J., Y. Karuno, and H. Kise. 2001. A shifting bottleneck approach for a parallel-machine flowshop scheduling problem. *Journal of the operations research society of Japan* 44 (2): 140–156.

Dauzère-Pérès, S. 1995. A procedure for the one-machine sequencing problem with dependent jobs. *European Journal of Operational Research* 81 (3): 579–589.

Dauzère-Pérès, S., and J.-B. Lasserre. 1993. A modified shifting bottleneck procedure for job-shop scheduling. *The International Journal of Production Research* 31 (4): 923–932.

Davis, R., and R. G. Smith. 1983. Negotiation as a metaphor for distributed problem solving. *Artificial intelligence* 20 (1): 63–109.

Drießel, R., U. Hönig, L. Mönch, and W. Schiffmann. 2010. A parallel shifting bottleneck heuristic for scheduling complex job shops: architecture and performance assessment. In *2010 IEEE International Conference on Automation Science and Engineering,* 81–86. Toronto: IEEE.

Driessel, R., and L. Mönch. 2012. An exploratory study of a decomposition heuristic for complex shop scheduling with transportation. In *2012 IEEE International Conference on Automation Science and Engineering,* 413–418. Seoul: IEEE.

Egger, G., D. Chaltsev, A. Giusti, and D. T. Matt. 2020. A deployment-friendly decentralized scheduling approach for cooperative multi-agent systems in production systems. *Procedia Manufacturing* 52:127–132.

Ferber, J. 1994. Simulating with reactive agents. In *Many agent simulation and artificial life,* edited by Hillebrand, E. and Stender, J., 8–30. Amsterdam: IOS Press.

Ferber, J. 1999. *Multi-agent systems: an introduction to distributed artificial intelligence*. Vol. 1. Reading: Addison-Wesley.

Fleischmann, J., O. Zimmermann, and K. Furmans. 2021. A mixed integer linear program for the job shop scheduling problem considering limited buffers, multi-purpose machines, transportation and setup times. *Submitted to Logistics Research*.

Frey, D., J. Nimis, H. Wörn, and P. Lockemann. 2003. Benchmarking and robust multi-agent-based production planning and control. *Engineering Applications of Artificial Intelligence* 16 (4): 307–320.

Gavareshki, K., and F. Zarandi. 2011. A multi agent system based on modified shifting bottleneck and search techniques for job shop scheduling problems. *AUT Journal of Modeling and Simulation* 43 (1): 7–15.

Giordani, S., M. Lujak, and F. Martinelli. 2013. A distributed multi-agent production planning and scheduling framework for mobile robots. *Computers & Industrial Engineering* 64 (1): 19–30.

Graves, S. C. 1981. A review of production scheduling. *Operations research* 29 (4): 646–675.

Groß, S., W. Gerke, and P. Plapper. 2020. Agent-based, hybrid control architecture for optimized and flexible production scheduling and control in remanufacturing. *Journal of Remanufacturing*.

Grundstein, S., M. Freitag, and B. Scholz-Reiter. 2017. A new method for autonomous control of complex job shops - Integrating order release, sequencing and capacity control to meet due dates. *Journal of manufacturing systems* 42:11–28.

Guizzi, G., R. Revetria, G. Vanacore, and S. Vespoli. 2019. On the open job-shop scheduling problem: a decentralized multi-agent approach for the manufacturing system performance optimization. *Procedia CIRP* 79:192–197.

Holtsclaw, H. H., and R. Uzsoy. 1996. Machine criticality measures and subproblem solution procedures in shifting bottleneck methods: a computational study. *Journal of the Operational Research Society* 47 (5): 666–667.

Iima, H., T. Hara, N. Ichimi, and N. Sannomiya. 1999. Autonomous decentralized scheduling algorithm for a job-shop scheduling problem with complicated constraints. In *Proceedings of Fourth International Symposium on Autonomous Decentralized Systems -Integration of Heterogeneous Systems-*, 366–369. Tokyo.

Ivens, P., and M. Lambrecht. 1996. Extending the shifting bottleneck procedure to real-life applications. *European Journal of Operational Research* 90 (2): 252–268.

Jung, H. 2016. *Konzept einer agentenbasierten Transportsteuerung: für komplexe, dynamische und multimodale Logistiknetzwerke*. Vol. 87. Karlsruhe: KIT Scientific Publishing.

Klein, M., A. Löcklin, N. Jazdi, and M. Weyrich. 2018. A negotiation based approach for agent based production scheduling. *Procedia Manufacturing* 17:334–341.

Koren, Y. 2010. *The global manufacturing revolution: product - process - business integration and reconfigurable systems*. Vol. 80. Wiley Series in Systems Engineering and Management. Oxford: Wiley-Blackwell.

Krothapalli, N. K. C., and A. V. Deshmukh. 1999. Design of negotiation protocols for multi-agent manufacturing systems. *International Journal of Production Research* 37 (7): 1601–1624.

Lange, J., and F. Werner. 2019. On neighborhood structures and repair techniques for blocking job shop scheduling problems. *Algorithms* 12 (11): 242.

Lawrence, S. R., and E. C. Sewell. 1997. Heuristic, optimal, static, and dynamic schedules when processing times are uncertain. *Journal of Operations Management* 15 (1): 71–82.

Lee, J.-H., and C.-O. Kim. 2008. Multi-agent systems applications in manufacturing systems and supply chain management: a review paper. *International Journal of Production Research* 46 (1): 233–265.

Leitao, P., and F. Restivo. 2008. A holonic approach to dynamic manufacturing scheduling. *Robotics and Computer-Integrated Manufacturing* 24 (5): 625–634.

Leitão, P. 2009. Agent-based distributed manufacturing control: a state-of-the-art survey. *Engineering Applications of Artificial Intelligence* 22 (7): 979–991.

Leitão, P., N. Rodrigues, J. Barbosa, C. Turrin, and A. Pagani. 2015. Intelligent products: the grace experience. *Control Engineering Practice* 42:95–105.

Lima, R. M., R. M. Sousa, and P. J. Martins. 2006. Distributed production planning and control agent-based system. *International Journal of Production Research* 44 (18-19): 3693–3709.

Liu, S. Q., and E. Kozan. 2012. A hybrid shifting bottleneck procedure algorithm for the parallel-machine job-shop scheduling problem. *Journal of the Operational Research Society* 63 (2): 168–182.

Mandal, S., X. Han, K. R. Pattipati, and D. L. Kleinman. 2010. Agent-based distributed framework for collaborative planning. In *2010 IEEE Aerospace Conference,* 1–11. Big Sky: IEEE.

Maoudj, A., B. Bouzouia, A. Hentout, A. Kouider, and R. Toumi. 2019. Distributed multi-agent scheduling and control system for robotic flexible assembly cells. *Journal of Intelligent Manufacturing* 30 (4): 1629–1644.

Mason, S. J., S. Jin, and C. M. Wessels. 2004. Rescheduling strategies for minimizing total weighted tardiness in complex job shops. *International Journal of Production Research* 42 (3): 613–628.

Mason, S. J., J. W. Fowler, and W. Matthew Carlyle. 2002. A modified shifting bottleneck heuristic for minimizing total weighted tardiness in complex job shops. *Journal of Scheduling* 5 (3): 247–262.

Mason, S. J., and K. Oey. 2003. Scheduling complex job shops using disjunctive graphs: a cycle elimination procedure. *International journal of production research* 41 (5): 981–994.

Matt, D. T., E. Rauch, and P. Dallasega. 2015. Trends towards distributed manufacturing systems and modern forms for their design. *Procedia CIRP* 33:185–190.

Mayer, S., N. Höhme, D. Gankin, and C. Endisch. 2019. Adaptive production control in a modular assembly system - towards an agent-based approach. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN),* 1:45–52. Helsinki: IEEE.

Mehta, S. V., and R. M. Uzsoy. 1998. Predictable scheduling of a job shop subject to breakdowns. *IEEE Transactions on Robotics and Automation* 14 (3): 365–378.

Mönch, L., M. Stehli, J. Zimmermann, and I. Habenicht. 2006. The FABMAS multi-agent-system prototype for production control of water fabs: design, implementation and performance assessment. *Production Planning & Control* 17 (7): 701–716.

Mönch, L., and R. Drießel. 2005. A distributed shifting bottleneck heuristic for complex job shops. *Computers & Industrial Engineering* 49 (3): 363–380.

Mönch, L., and M. Stehli. 2006. ManufAg: a multi-agent-system framework for production control of complex manufacturing systems. *Information Systems and e-Business Management* 4 (2): 159–185.

Mönch, L., M. Stehli, and J. Zimmermann. 2003. FABMAS: An agent-based system for production control of semiconductor manufacturing processes. In *International Conference on Industrial Applications of Holonic and Multi-Agent Systems,* 258–267. Prague: Springer.

Mönch, L., and J. Zimmermann. 2007. Simulation-based assessment of machine criticality measures for a shifting bottleneck scheduling approach in complex manufacturing systems. *Computers in Industry* 58 (7): 644–655.

Monostori, L., J. Váncza, and S. Kumara. 2006. Agent-based systems for manufacturing. *CIRP annals* 55 (2): 697–720.

Ovacik, I. M., and R. Uzsoy. 1992. A shifting bottleneck algorithm for scheduling semiconductor testing operations. *Journal of Electronics Manufacturing* 2 (03): 119–134.

Ovacik, I. M., and R. Uzsoy. 1997. *Decomposition methods for complex factory scheduling problems*. New York: Springer Science+Business Media.

Parente, M., G. Figueira, P. Amorim, and A. Marques. 2020. Production scheduling in the context of industry 4.0: review and trends. *International Journal of Production Research* 58 (17): 5401–5431.

Pezzella, F., and E. Merelli. 2000. A tabu search method guided by shifting bottleneck for the job shop scheduling problem. *European Journal of Operational Research* 120 (2): 297–310.

Pfund, M. E., H. Balasubramanian, J. W. Fowler, S. J. Mason, and O. Rose. 2008. A multi-criteria approach for scheduling semiconductor wafer fabrication facilities. *Journal of Scheduling* 11 (1): 29–47.

Pinedo, M. 2012. *Scheduling*. New York: Springer Science+Business Media.

Pinedo, M., and M. Singer. 1999. A shifting bottleneck heuristic for minimizing the total weighted tardiness in a job shop. *Naval Research Logistics (NRL)* 46 (1): 1–17.

Rabelo, R. J., and L. Camarinha-Matos. 1994. Negotiation in multi-agent based dynamic scheduling. *Robotics and computer-integrated manufacturing* 11 (4): 303–309.

Rabelo, R. J., L. M. Camarinha-Matos, and H. Afsarmanesh. 1999. Multi-agent-based agile scheduling. *Robotics and Autonomous Systems* 27 (1-2): 15–28.

Ramudhin, A., and P. Marier. 1996. The generalized shifting bottleneck procedure. *European Journal of Operational Research* 93 (1): 34–48.

Renna, P. 2011. Multi-agent based scheduling in manufacturing cells in a dynamic environment. *International Journal of Production Research* 49 (5): 1285–1301.

Roth, A. 2016. *Einführung und Umsetzung von Industrie 4.0: Grundlagen, Vorgehensmodell und Use Cases aus der Praxis.* Berlin Heidelberg: Springer-Verlag.

Schild, K., and S. Bussmann. 2007. Self-organization in manufacturing operations. *Communications of the ACM* 50 (12): 74–79.

Scholz-Reiter, B., T. Hildebrandt, and Y. Tan. 2013. Effective and efficient scheduling of dynamic job shops—combining the shifting bottleneck procedure with variable neighbourhood search. *CIRP Annals* 62 (1): 423–426.

Schoop, R., R. Neubert, and A. W. Colombo. 2001. A multiagent-based distributed control platform for industrial flexible production systems. In *IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society,* 1:279–284. Denver: IEEE.

Schutten, J. M. 1998. Practical job shop scheduling. *Annals of Operations Research* 83:161–178.

Sikora, R., and M. J. Shaw. 1997. Coordination mechanisms for multi-agent manufacturing systems: applications to integrated manufacturing scheduling. *IEEE Transactions on Engineering Management* 44 (2): 175–187.

Singer, M. 2001. Decomposition methods for large job shops. *Computers & Operations Research* 28 (3): 193–207.

Sourirajan, K., and R. Uzsoy. 2007. Hybrid decomposition heuristics for solving large-scale scheduling problems in semiconductor wafer fabrication. *Journal of Scheduling* 10 (1): 41–65.

Sousa, P., and C. Ramos. 1999. A distributed architecture and negotiation protocol for scheduling in manufacturing systems. *Computers in industry* 38 (2): 103–113.

Sudo, Y., N. Sakao, and M. Matsuda. 2010. An agent behavior technique in an autonomous decentralized manufacturing system. *Journal of Advanced Mechanical Design, Systems, and Manufacturing* 4 (3): 673–682.

Sun, X., and J. S. Noble. 1999. An approach to job shop scheduling with sequence-dependent setups. *Journal of Manufacturing Systems* 18 (6): 416–430.

Sundermeyer, K., and S. Bussmann. 2001. Einfuehrung der Agententechnologie in einem produzierenden Unternehmen - Ein Erfahrungsbericht. *Wirtschaftsinformatik* 43 (2): 135–142.

Tanenbaum, A. S., and H. Bos. 2015. *Modern operating systems.* Boston, Mass.: Pearson.

Tarkoma, S. 2012. *Publish/subscribe systems: design and principles.* Hoboken, NJ: John Wiley & Sons.

Topaloglu, S., and G. Kilincli. 2009. A modified shifting bottleneck heuristic for the reentrant job shop scheduling problem with makespan minimization. *The International Journal of Advanced Manufacturing Technology* 44 (7-8): 781–794.

Torreño, A., E. Onaindia, A. Komenda, and M. Štolba. 2017. Cooperative multi-agent planning: a survey. *ACM Computing Surveys (CSUR)* 50 (6): 1–32.

Upasani, A. A., R. Uzsoy, and K. Sourirajan. 2006. A problem reduction approach for scheduling semiconductor wafer fabrication facilities. *IEEE Transactions on semiconductor manufacturing* 19 (2): 216–225.

Uzsoy, R. 1995. Scheduling batch processing machines with incompatible job families. *International Journal of Production Research* 33 (10): 2685–2708.

Wang, L.-C., and S.-K. Lin. 2009. A multi-agent based agile manufacturing planning and control system. *Computers & Industrial Engineering* 57 (2): 620–640.

Wang, S., J. Wan, D. Zhang, D. Li, and C. Zhang. 2016. Towards smart factory for industry 4.0: a self-organized multi-agent system with big data based feedback and coordination. *Computer Networks* 101:158–168.

Weber, E., A. Tiefenbacher, and N. Gronau. 2019. Need for standardization and systematization of test data for job-shop scheduling. *Data* 4 (1): 32.

Wenqi, H., and Y. Aihua. 2004. An improved shifting bottleneck procedure for the job shop scheduling problem. *Computers & Operations Research* 31 (12): 2093–2110.

Wooldridge, M. 2006. *An introduction to multiagent systems*. Hoboken, NJ: John Wiley & Sons.

Yeung, T. G., and S. J. Mason. 2006. Using real options analysis to value reoptimization options in the shifting bottleneck heuristic. *Naval Research Logistics (NRL)* 53 (4): 285–297.

Yoo, W.-S., and L. A. Martin-Vega. 1997. A decomposition methodology for scheduling semiconductor test operations for number of tardy job measures. *Journal of Electronics Manufacturing* 7 (01): 51–61.

Zhang, J., G. Ding, Y. Zou, S. Qin, and J. Fu. 2019. Review of job shop scheduling research and its new perspectives under industry 4.0. *Journal of Intelligent Manufacturing* 30 (4): 1809–1830.

Zhang, Q., H. Manier, and M.-A. Manier. 2014. A modified shifting bottleneck heuristic and disjunctive graph for job shop scheduling problems with transportation constraints. *International Journal of Production Research* 52 (4): 985–1002.

Zhang, Y., G. Q. Huang, S. Sun, and T. Yang. 2014. Multi-agent based real-time production scheduling method for radio frequency identification enabled ubiquitous shopfloor environment. *Computers & Industrial Engineering* 76:89–97.

List of Figures

1.1 Overview of trends in production systems 2
1.2 Organization of the thesis 5

2.1 Gantt-Charts of a schedule, once machine-based (a), once job-based (b) 8
2.2 Exemplary flow shop 10
2.3 Exemplary job shop 11
2.4 Exemplary open shop 12

3.1 Exemplary multi-agent system 32
3.2 Examples for data storage 33
3.3 Two level architecture of Brandimarte et al. 38
3.4 Agent system of Klein et al 43
3.5 Negotiation procedure of Wang et al. 45
3.6 Scheduling process of Maoudj et al 47

5.1 Flow chart of the scheduling algorithm 58
5.2 Example situation for condition 5.5 70
5.3 Example situation for condition 5.6 71
5.4 Example situation for condition 5.7 72
5.5 Exemplary negotiation procedure 80

6.1 Multi-agent system for production planning 84
6.2 Structure of the workplace agent 86
6.3 Flow chart of the scheduling algorithm including synchronization 89
6.4 Example for the spontaneous sorting of agents during a negotiation 100

6.5 Order of the responses in the earlier example 103
6.6 Possible termination problems of the algorithm 106
6.7 Example for a deadlock during the execution 108
6.8 Example for a starvation during the execution 109
6.9 Example for infeasible plans 114
6.10 Example for feasible plans with worse objective function values . 116

7.1 Material flow of the first example 124
7.2 Makespan results of the first experiment 125
7.3 Makespan results of the second experiment 127
7.4 Makespan results of the third experiment 130
7.5 Makespan results of the fourth experiment 132
7.6 Exemplary random material flow 134

List of Tables

2.1 List of relevant optimization criteria (taken from Pinedo (2012)) . 9
2.2 List of advanced shop models 12
2.3 List of variables used in the Shifting Bottleneck procedure 15
2.4 Overview of the changes to the single workplace sub problem and their contents (part 1) 20
2.5 Overview of the changes to the single workplace sub problem and their contents (part 2) 21
2.6 Overview of publications to the general procedure and their focus 25

3.1 Overview of publications of agent systems for production planning and their classification 41

4.1 Job finishing times of two plans 52

5.1 Example situation for a delay in the plan on a single workplace . . 62
5.2 Key variables used during the sequencing of operations 65
5.3 Different cases for the optimal start time 66

6.1 List of all events used for scheduling in the multi-agent system . . 91
6.2 Possible situations for the creator of a negotiation 97

7.1 Input parameters for the first application example 124
7.2 Results of the first application example 126
7.3 Comparison of the limited and unlimited system 128
7.4 Input parameters for the second application example 130
7.5 Results of the second application example 131
7.6 Input parameters for the random experiments 133

7.7 Results of the random experiments 133

8.1 List of workplaces and their processes in application example one 163
8.2 List of variants and their sequences in application example one . 163
8.3 List of process times per piece in application example one 164
8.4 Orders of batch size 20 in application example one 165
8.5 Orders of batch size 50 in application example one 165

8.1 List of workplaces and their processes in application example two 167
8.2 List of variants and their sequences in application example two . 168
8.3 Orders of batch size 50 in application example two 168
8.4 List of process times per piece in application example two (part 1) 169
8.5 List of process times per piece in application example two (part 2) 170
8.6 Orders of batch size 100 in application example two 171
8.7 Orders of batch size 200 in application example two (part 1) . . . 172
8.8 Orders of batch size 200 in application example two (part 2) . . . 173

8.1 List of workplaces and their processes in random example one . . 175
8.2 List of variants and their sequences in random example one . . . 176
8.3 Orders in random example one 176
8.4 List of process times per piece in random example one 177
8.5 List of workplaces and their processes in random example two . . 178
8.6 List of variants and their sequences in random example two . . . 178
8.7 List of process times per piece in random example two 179
8.8 Orders in random example two 180
8.9 List of workplaces and their processes in random example three . 180
8.10 List of variants and their sequences in random example three . . 181
8.11 Orders in random example three 181
8.12 List of process times per piece in random example three 182
8.13 List of workplaces and their processes in random example four . 183
8.14 List of variants and their sequences in random example four . . . 183
8.15 List of process times per piece in random example four 184
8.16 Orders in random example four 185
8.17 List of workplaces and their processes in random example five . . 185

8.18 List of variants and their sequences in random example five . . . 186
8.19 Orders in random example five 186
8.20 List of process times per piece in random example five 187

A Data for Application Example One

Workplace	Process	Workplace	Process	Workplace	Process
W1	P1	W6(2)	P6	W11	P11
W2	P2	W7	P7	W12	P12
W3	P3	W8	P8	W13	P13
W4	P4	W9	P9	W14	P14
W5	P5	W10	P10	W15	P15
W6	P6				

Table 8.1: List of workplaces and their processes in application example one

Variant	Process sequence
V1	P1 P2 P3 P5 P6 P7 P8 P9 P10 P11 P12
V2	P1 P2 P3 P5 P6 P7 P8 P9 P10 P11 P12
V3	P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12
V4	P1 P2 P6 P7 P8 P10 P11 P12
V5	P1 P2 P4 P6 P7 P8 P9 P10 P11 P12
V6	P1 P2 P3 P5 P6 P7 P8 P10 P11 P12
V7	P1 P2 P3 P5 P6 P7 P8 P10 P11 P12
V8	P6 P2 P3 P7 P13 P14 P15 P12
V9	P6 P2 P3 P7 P13 P14 P15 P12
V10	P6 P2 P3 P7 P13 P14 P15 P12
V11	P6 P2 P3 P7 P13 P14 P15 P12

Table 8.2: List of variants and their sequences in application example one

W-P	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11
W1-P1	19	25	19	23	20	19	19				
W2-P2	48	51	50	35	48	50	50	35	35	35	35
W3-P3	18	16	14			14	14	25	18	26	26
W4-P4			20		21						
W5-P5	24	22	18			20	20				
W6-P6	35	40	70	30	35	30	30	100	120	120	120
W6(2)-P6	35	40	70	30	35	30	30	100	120	120	120
W7-P7	27	50	49	25	27	30	30	16	16	16	16
W8-P8	16	24	22	11	16	22	22				
W9-P9	27	27	26		27						
W10-P10	26	26	26	26	26	25	25				
W11-P11	60	62	63	60	60	60	60				
W12-P12	32	32	30	34	32	30	30	34	34	34	34
W13-P13								36	36	36	36
W14-P14								66.5	66.5	76	76
W15-P15								83	83	83	83

Table 8.3: List of process times per piece in application example one

Variant	#	Variant	#	Variant	#	Variant	#
V1	30	V1	30	V2	30	V4	25
V1	30	V2	30	V3	25	V8	25
V1	30	V2	30	V3	25	V8	25
V1	30	V2	30	V3	25	V8	25
V1	30	V2	30	V4	25	V8	25

Table 8.4: Orders of batch size 20 in application example one

Variant	#	Variant	#	Variant	#	Variant	#
V1	30	V1	30	V2	30	V8	25
V1	30	V2	30	V2	30	V8	25
V1	30	V2	30	V3	25	V8	25
V1	30	V2	30	V3	25	V8	25
V1	30	V2	30	V3	25	V9	25
V1	30	V2	30	V3	25	V9	25
V1	30	V2	30	V3	25	V9	25
V1	30	V2	30	V4	25	V9	25
V1	30	V2	30	V4	25	V9	25
V1	30	V2	30	V5	30	V10	25
V1	30	V2	30	V6	25	V11	15
V1	30	V2	30	V7	30		
V1	30	V2	30	V8	25		

Table 8.5: Orders of batch size 50 in application example one

B Data for Application Example Two

Workplace	Process	Workplace	Process	Workplace	Process
W1	P1	W8	P8	W15	P14
W2	P2	W9	P9	W16	P15
W3	P3	W10	P10	W17	P16
W4	P4	W11	P11	W18	P17
W5	P5	W12	P11	W19	P17
W6	P6	W13	P12	W20	P18
W7	P7	W14	P13	W21	P19

Table 8.1: List of workplaces and their processes in application example two

Variant	Process sequence	Variant	Process sequence
V1	P1 P2 P3 P4 P7 P11	V15	P1 P2 P3 P4 P11 P10
V2	P1 P2 P3 P4 P10 P15	V16	P1 P2 P3 P4 P15
V3	P1 P2 P3 P4 P7	V17	P1 P2 P3 P4 P7
V4	P1 P2 P3 P4 P5 P14	V18	P1 P2 P3 P4 P7
V5	P1 P2 P3 P4 P12 P17	V19	P1 P2 P3 P4 P5 P14
V6	P1 P2 P3 P4 P5 P14	V20	P1 P2 P3 P4 P14 P11 P6 P7
V7	P1 P2 P3 P4 P5	V21	P1 P13 P7
V8	P1 P2 P3 P4 P7	V22	P1 P2 P3 P4 P5 P14
V9	P1 P2 P3 P4 P7	V23	P1 P11 P9
V10	P1 P2 P3 P4 P11 P17	V24	P1 P13 P14 P7
V11	P1 P2 P3 P4 P19 P6	V25	P1 P2 P3 P4 P14 P5
V12	P1 P2 P3 P4 P11 P18	V26	P1 P2 P3 P4 P14 P10
V13	P1 P13 P14 P16 P6	V27	P1 P2 P3 P4 P14 P8
V14	P1 P2 P3 P4 P11 P18	V28	P1 P2 P3 P4 P14 P17

Table 8.2: List of variants and their sequences in application example two

Variant	#	Variant	#	Variant	#	Variant	#
V1	20	V3	20	V8	20	V18	20
V1	20	V3	20	V8	20	V19	20
V1	20	V4	20	V9	20	V20	20
V1	20	V4	20	V9	20	V21	20
V1	20	V4	20	V10	20	V22	20
V1	20	V4	20	V10	20	V23	20
V2	20	V5	20	V11	20	V24	20
V2	20	V5	20	V12	20	V25	20
V2	20	V5	20	V13	20	V26	20
V2	20	V6	20	V14	20	V27	20
V2	20	V6	20	V15	20	V28	20
V3	20	V7	20	V16	20		
V3	20	V7	20	V17	20		

Table 8.3: Orders of batch size 50 in application example two

W-P	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14
W1-P1	13	13	13	21	13	21	13	13	13	13	13	13	13	13
W2-P2	3	8	3	110	113	148	52	6	3	304	3	3		4
W3-P3	8	8	8	14	17	18	18	8	8	18	16	16		15
W4-P4	1	5	1	13	15	15	25	1	1	29	12	1		1
W5-P5			30		42	1								
W6-P6											18			
W7-P7	18		18					12	12					
W8-P8														
W9-P9													47	
W10-P10		16												
W11-P11	103									240		24		24
W12-P11	103									240		24		24
W13-P12					107									
W14-P13													31	
W15-P14				42		33							28	
W16-P15		15												
W17-P16													20	
W18-P17					240					188				
W19-P17					240					188				
W20-P18												84		84
W21-P19											1			

Table 8.4: List of process times per piece in application example two (part 1)

W-P	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V25	V26	V27	V28
W1-P1	13	13	13	13	13	21	13	13	13	13	13	13	13	13
W2-P2	32	87	11	11	8	35		95			38	23	8	118
W3-P3	8	16	8	8	11	8		11			11	17	12	29
W4-P4	12	29	1	1	15	1		15			1	20	1	23
W5-P5				58			37			33				
W6-P6						9								
W7-P7			19	15		20	31			23				
W8-P8													49	
W9-P9									52					
W10-P10	40											52		
W11-P11	42					39			83					
W12-P11	42					39			83					
W13-P12														
W14-P13							42			18				
W15-P14					43	6		42		46	45	12	17	208
W16-P15		138												
W17-P16														
W18-P17														327
W19-P17														327
W20-P18														
W21-P19														

Table 8.5: List of process times per piece in application example two (part 2)

Variant	#	Variant	#	Variant	#	Variant	#
V1	20	V3	20	V6	20	V14	20
V1	20	V3	20	V6	20	V14	20
V1	20	V3	20	V7	20	V15	20
V1	20	V3	20	V7	20	V15	20
V1	20	V3	20	V7	20	V16	20
V1	20	V3	20	V7	20	V16	20
V1	20	V3	20	V8	20	V17	20
V1	20	V4	20	V8	20	V17	20
V1	20	V4	20	V8	20	V18	20
V1	20	V4	20	V8	20	V18	20
V1	20	V4	20	V9	20	V19	20
V1	20	V4	20	V9	20	V19	20
V2	20	V4	20	V9	20	V20	20
V2	20	V4	20	V10	20	V20	20
V2	20	V4	20	V10	20	V21	20
V2	20	V4	20	V10	20	V21	20
V2	20	V5	20	V11	20	V22	20
V2	20	V5	20	V11	20	V22	20
V2	20	V5	20	V11	20	V23	20
V2	20	V5	20	V12	20	V23	20
V2	20	V5	20	V12	20	V24	20
V2	20	V5	20	V12	20	V25	20
V2	20	V5	20	V13	20	V26	20
V3	20	V6	20	V13	20	V27	20
V3	20	V6	20	V13	20	V28	20

Table 8.6: Orders of batch size 100 in application example two

Variant	#	Variant	#	Variant	#	Variant	#
V1	20	V3	20	V6	20	V14	20
V1	20	V3	20	V6	20	V14	20
V1	20	V3	20	V6	20	V14	20
V1	20	V3	20	V6	20	V14	20
V1	20	V3	20	V7	20	V15	20
V1	20	V3	20	V7	20	V15	20
V1	20	V3	20	V7	20	V15	20
V1	20	V3	20	V7	20	V15	20
V1	20	V3	20	V7	20	V16	20
V1	20	V3	20	V7	20	V16	20
V1	20	V3	20	V7	20	V16	20
V1	20	V3	20	V7	20	V16	20
V1	20	V3	20	V8	20	V17	20
V1	20	V3	20	V8	20	V17	20
V1	20	V4	20	V8	20	V17	20
V1	20	V4	20	V8	20	V17	20
V1	20	V4	20	V8	20	V18	20
V1	20	V4	20	V8	20	V18	20
V1	20	V4	20	V8	20	V18	20
V1	20	V4	20	V8	20	V18	20
V1	20	V4	20	V9	20	V19	20
V1	20	V4	20	V9	20	V19	20
V1	20	V4	20	V9	20	V19	20
V1	20	V4	20	V9	20	V19	20
V2	20	V4	20	V9	20	V20	20
V2	20	V4	20	V9	20	V20	20
V2	20	V4	20	V10	20	V20	20
V2	20	V4	20	V10	20	V20	20
V2	20	V4	20	V10	20	V21	20
V2	20	V4	20	V10	20	V21	20
V2	20	V4	20	V10	20	V21	20
V2	20	V4	20	V10	20	V21	20
V2	20	V5	20	V11	20	V22	20
V2	20	V5	20	V11	20	V22	20
V2	20	V5	20	V11	20	V22	20

Table 8.7: Orders of batch size 200 in application example two (part 1)

Variant	#	Variant	#	Variant	#	Variant	#
V2	20	V5	20	V11	20	V22	20
V2	20	V5	20	V11	20	V23	20
V2	20	V5	20	V11	20	V23	20
V2	20	V5	20	V12	20	V23	20
V2	20	V5	20	V12	20	V23	20
V2	20	V5	20	V12	20	V24	20
V2	20	V5	20	V12	20	V24	20
V2	20	V5	20	V12	20	V25	20
V2	20	V5	20	V12	20	V25	20
V2	20	V5	20	V13	20	V26	20
V2	20	V5	20	V13	20	V26	20
V3	20	V6	20	V13	20	V27	20
V3	20	V6	20	V13	20	V27	20
V3	20	V6	20	V13	20	V28	20
V3	20	V6	20	V13	20	V28	20

Table 8.8: Orders of batch size 200 in application example two (part 2)

C Data for Randomized Experiments

Random Example One

Workplace	Process	Workplace	Process	Workplace	Process
W1	P1	W8	P8	W15	P15
W2	P2	W9	P9	W16	P16
W3	P3	W10	P10	W17	P17
W4	P4	W11	P11	W18	P18
W5	P5	W12	P12	W19	P7
W6	P6	W13	P13	W20	P3
W7	P7	W14	P14		

Table 8.1: List of workplaces and their processes in random example one

Variant	Process sequence
V1	P8 P2 P5 P11 P9 P10 P13 P18 P6 P15 P12 P16 P3
V2	P9 P13 P7 P3 P17 P2 P8 P16 P4 P18
V3	P3 P15 P7 P18 P2 P11 P1 P14 P6 P8 P17
V4	P12 P11 P10 P9 P13 P5 P2 P17 P15 P16 P14 P6 P18 P8 P7
V5	P14 P1 P9 P3 P2 P10 P6 P8 P13
V6	P15 P7 P10 P8 P3 P18 P14 P16 P9 P17
V7	P4 P17 P18 P14 P10 P5 P2 P7 P13
V8	P12 P16 P10 P8 P9 P3 P7 P1 P5 P4 P11 P17 P18 P14 P15 P6 P13
V9	P8 P7 P18 P5 P15 P16 P3 P2 P17 P10 P16 P12 P1 P14 P13
V10	P6 P13 P9 P15 P2 P12 P18 P17 P11 P10

Table 8.2: List of variants and their sequences in random example one

Variant	#	Variant	#	Variant	#	Variant	#
V1	17	V2	6	V6	7	V10	4
V7	20	V1	6	V8	5	V6	10
V1	15	V10	1	V2	20	V8	1
V9	15	V9	13	V7	3	V4	1
V10	14	V8	13	V3	3	V1	17
V2	8	V4	19	V5	6	V8	16
V7	7	V9	11	V7	15	V1	19
V2	18	V1	1	V9	3	V10	7
V5	10	V10	16	V1	5	V8	16
V3	6	V8	10	V9	5	V2	17
V4	15	V7	1	V8	2	V7	7
V1	18	V4	18	V8	3		
V1	1	V9	17	V10	16		

Table 8.3: Orders in random example one

W-P	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
W1-P1			11		24			10	5	
W2-P2	16	2	6	15	18		21		6	1
W3-P3	3	4	7		15	23		19	15	
W4-P4		7					12	2		
W5-P5	23			7			6	9	3	
W6-P6	15		4	21	15			16	10	15
W7-P7		21	10	22		25	4	11	19	
W8-P8	9	16	3	7	20	7		15	8	
W9-P9	1	23		22	1	13		21		1
W10-P10	9			14	11	16	18	20	1	21
W11-P11	23		16	16				19		18
W12-P12	10			7				4	5	15
W13-P13	5	2		16	23		14	9	1	4
W14-P14			4	25	24	22	8	14	3	
W15-P15	5		22	22		24		20	3	7
W16-P16	7	11		23		17		8	22	
W17-P17		13	24	20		24	12	21	1	7
W18-P18	22	5	16	20		14	4	15	22	12
W19-P7		5	22	1		22	4	17	7	
W20-P3	5	13	15		20	1		25	23	

Table 8.4: List of process times per piece in random example one

Random Example Two

Workplace	Process	Workplace	Process	Workplace	Process
W1	P1	W8	P8	W15	P15
W2	P2	W9	P9	W16	P16
W3	P3	W10	P10	W17	P17
W4	P4	W11	P11	W18	P17
W5	P5	W12	P12	W19	P16
W6	P6	W13	P13	W20	P13
W7	P7	W14	P14		

Table 8.5: List of workplaces and their processes in random example two

Variant	Process sequence
V1	P11 P12 P16 P15 P2 P13 P3 P14 P17 P5 P7 P6 P9 P10 P1 P8
V2	P17 P10 P8 P14 P2 P4 P7 P16 P9 P1 P13
V3	P12 P6 P10 P7 P5 P16 P9 P17 P2 P13 P11 P15 P4 P14 P3 P1
V4	P15 P7 P4 P16 P14 P9 P1 P17 P11 P12 P6 P13 P3 P2
V5	P8 P16 P13 P1 P2 P7 P4 P10 P15 P5 P3 P14 P11
V6	P16 P1 P2 P6 P11 P12 P13 P5 P3 P10 P15 P9
V7	P7 P5 P14 P1 P10 P8 P9 P13
V8	P7 P4 P5 P9 P17 P3 P6 P8 P13 P14 P10 P12 P11
V9	P11 P8 P15 P4 P13 P1 P9 P14 P3 P12 P6 P7 P16 P17
V10	P4 P17 P7 P5 P11 P16 P15 P13 P10 P6 P3

Table 8.6: List of variants and their sequences in random example two

W-P	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
W1-P1	7	11	3	7	9	25	22		16	
W2-P2	12	1	3	6	17	11				
W3-P3	7		12	20	2	10		5	15	1
W4-P4		10	24	20	3			17	2	19
W5-P5	9		19		2	1	7	15		7
W6-P6	8		18	6		21		12	12	5
W7-P7	25	20	24	6	24		20	25	18	4
W8-P8	7	12			19		20	9	19	
W9-P9	18	25	25	17		11	25	14	22	
W10-P10	12	13	19		22	13	10	21		23
W11-P11	8		13	18	16	13		21	21	18
W12-P12	1		12	25		3		1	19	
W13-P13	12	13	22	19	8	1	20	4	19	16
W14-P14	8	1	8	23	17		14	11	24	
W15-P15	16		2	13	10	4	21		8	21
W16-P16	21	17	8	14	16	23			13	13
W17-P17	11	6	16	2	23			3	4	9
W18-P18	19	1	13	1	6			2	24	10
W19-P7	17	7	13	18	9	19			10	14
W20-P3	21	10	24	19	23	4	6	19	25	3

Table 8.7: List of process times per piece in random example two

Variant	#	Variant	#	Variant	#	Variant	#
V9	14	V2	20	V7	3	V3	6
V2	2	V6	7	V2	4	V4	8
V3	4	V1	17	V3	13	V1	2
V7	3	V4	9	V1	13	V5	11
V8	9	V2	20	V2	4	V1	20
V6	11	V3	2	V7	2	V5	12
V9	7	V3	7	V1	14	V6	8
V5	6	V10	9	V1	13	V9	3
V3	11	V3	14	V5	1	V3	17
V5	10	V4	15	V10	17	V7	6
V10	1	V6	16	V3	4	V3	6
V9	9	V8	11	V9	7		
V7	14	V5	3	V2	19		

Table 8.8: Orders in random example two

Random Example Three

Workplace	Process	Workplace	Process	Workplace	Process
W1	P1	W8	P8	W15	P15
W2	P2	W9	P9	W16	P16
W3	P3	W10	P10	W17	P11
W4	P4	W11	P11	W18	P12
W5	P5	W12	P12	W19	P1
W6	P6	W13	P13	W20	P6
W7	P7	W14	P14		

Table 8.9: List of workplaces and their processes in random example three

Variant	Process sequence
V1	P5 P11 P2 P1 P13 P7 P9 P15 P8 P4 P12 P6 P10 P14 P16
V2	P2 P10 P4 P11 P8 P16 P14 P13 P3 P5 P15 P7
V3	P16 P9 P1 P14 P7 P4 P10 P8 P11 P5 P3 P6 P15 P2
V4	P4 P7 P10 P12 P9 P6 P5 P14 P2 P15 P1 P11
V5	P13 P2 P15 P8 P3 P16 P6 P5 P14 P10 P11
V6	P16 P11 P10 P6 P15 P2 P5 P13 P12 P14 P8 P7 P4 P9 P1
V7	P13 P14 P8 P16 P15 P2 P12 P10 P4 P11 P1
V8	P1 P15 P9 P6 P16 P8 P2 P14 P4 P5 P3 P10 P7
V9	P4 P7 P11 P15 P13 P2 P6 P5 P16 P14 P3 P1
V10	P13 P14 P16 P5 P11 P7 P10 P9 P15 P8 P12 P6 P2

Table 8.10: List of variants and their sequences in random example three

Variant	#	Variant	#	Variant	#	Variant	#
V2	14	V2	17	V3	20	V1	19
V8	18	V2	9	V4	9	V1	4
V1	3	V6	12	V1	4	V7	15
V8	17	V5	13	V10	1	V10	4
V10	9	V8	19	V1	7	V3	16
V1	13	V5	1	V6	15	V2	3
V1	16	V9	4	V8	15	V9	1
V7	10	V7	13	V7	20	V4	18
V10	4	V1	16	V9	14	V4	5
V7	6	V7	16	V8	8	V9	19
V10	7	V3	17	V8	14	V6	3
V7	13	V10	17	V3	12		
V5	9	V4	2	V10	1		

Table 8.11: Orders in random example three

W-P	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
W1-P1	6		21	5		25	2	9	18	
W2-P2	8	10	19	11	11	15	18	5	5	3
W3-P3		17	11		17			3	3	
W4-P4	6	6	22	23		18	21	4	23	
W5-P5	22	9	19	9	12	1		7	14	19
W6-P6	9		20	9	10	24		14	19	13
W7-P7	16	14	18	9		2		5	19	24
W8-P8	15	8	16		8	23	24	12		12
W9-P9	5		2	9		20		10		17
W10-P10	15	4	7	6	17	23	1	16		16
W11-P11	24	9	10	19	12	11	13		19	21
W12-P12	3			14		14	5			22
W13-P13	14	19			10	4	21		13	20
W14-P14	25	5	1	9	4	18	8	14	7	12
W15-P15	2	11	1	18	13	23	14	23	8	22
W16-P16	21	15	5		17	7	10	15	23	12
W17-P17	16	5	10	16	9	25	1		2	11
W18-P18	10			16		14	22			8
W19-P7	19		10	2		6	5	3	4	
W20-P3	11		8	18	23	1		5	17	5

Table 8.12: List of process times per piece in random example three

Random Example Four

Workplace	Process	Workplace	Process	Workplace	Process
W1	P1	W8	P8	W15	P3
W2	P2	W9	P9	W16	P5
W3	P3	W10	P10	W17	P8
W4	P4	W11	P11	W18	P7
W5	P5	W12	P12	W19	P5
W6	P6	W13	P13	W20	P12
W7	P7	W14	P14		

Table 8.13: List of workplaces and their processes in random example four

Variant	Process sequence
V1	P5 P13 P1 P14 P12 P2 P7 P11 P9
V2	P3 P11 P7 P10 P1 P5 P9
V3	P1 P13 P6 P4 P8 P7 P3 P2 P10 P12 P11 D14 D9
V4	P11 P7 P9 P4 P8 P3 P12 P6 P10 P13 P2 P5
V5	P7 P11 P8 P3 P13 P10 P5 P6 P2 P14
V6	P4 P1 P9 P5 P7 P2 P10 P8 P11
V7	P9 P2 P6 P4 P8 P11 P5 P7 P14 P1 P13 P10 P12
V8	P7 P4 P12 P5 P1 P13 P11 P3 P6 P10 P14 P9 P2
V9	P10 P7 P6 P4 P5 P3 P12 P2 P9 P1 P8 P14
V10	P13 P3 P9 P11 P7 P1 P6

Table 8.14: List of variants and their sequences in random example four

W-P	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
W1-P1	24	21	19			5	18	7	5	14
W2-P2	12		4	17	11	5	3	5	19	
W3-P3		19	7	2	5			5	12	20
W4-P4			15	1		6	13	15	19	
W5-P5	3	2		16	7	1	6	18	20	
W6-P6			13	17	9		18	8	12	15
W7-P7	1	4	21	23	8	9	12	3	8	25
W8-P8			14	20	9	2	16		5	
W9-P9	20	5	23	8		20	14	14	13	16
W10-P10		7	24	3	16	11	25	10	9	
W11-P11	6	25	7	5	5	18	11	2		10
W12-P12	15		15	12			7	18	17	
W13-P13	14		16	21	11		17	21		13
W14-P14	4		21		16		1	4	10	
W15-P3		21	10	15	20			22	12	7
W16-P5	8	15		7	19	18	2	8	1	
W17-P8			7	25	23	2	7		16	
W18-P7	13	21	13	1	12	21	3	6	20	5
W19-P5	20	16		7	10	23	23	1	2	
W20-P12	18		19	17			8	17	2	

Table 8.15: List of process times per piece in random example four

Variant	#	Variant	#	Variant	#	Variant	#
V1	14	V10	10	V1	14	V6	16
V9	6	V8	9	V1	1	V9	12
V9	9	V2	4	V7	6	V2	1
V7	19	V3	19	V7	15	V5	19
V1	11	V4	9	V4	8	V3	12
V8	7	V4	3	V3	4	V1	2
V3	5	V8	19	V3	7	V1	8
V7	3	V1	15	V1	3	V10	19
V6	20	V5	5	V6	2	V2	15
V6	4	V5	16	V3	7	V9	12
V10	1	V3	18	V8	7	V8	2
V5	13	V9	17	V7	13		
V4	12	V2	9	V6	13		

Table 8.16: Orders in random example four

Random Example Five

Workplace	Process	Workplace	Process	Workplace	Process
W1	P1	W8	P8	W15	P15
W2	P2	W9	P9	W16	P16
W3	P3	W10	P10	W17	P17
W4	P4	W11	P11	W18	P18
W5	P5	W12	P12	W19	P19
W6	P6	W13	P13	W20	P8
W7	P7	W14	P14		

Table 8.17: List of workplaces and their processes in random example five

Variant	Process sequence
V1	P10 P13 P11 P16 P17 P15 P4 P1 P18 P8
V2	P8 P19 P18 P16 P10 P2 P13 P4 P9 P12 P7 P6 P15
V3	P5 P6 P8 P19 P18 P14 P1 P17 P9 P2 P7 P4
V4	P4 P1 P13 P6 P19 P8 P10 P2 P11 P16 P12 P5 P7
V5	P6 P9 P17 P5 P13 P10 P14 P3 P4 P16 P19 P15 P1 P2 P18
V6	P12 P10 P1 P17 P14 P7 P2 P11 P19 P15 P9 P18 P6 P4 P13 P16 P5
V7	P4 P18 P15 P17 P3 P19 P16 P14 P2 P1 P9 P10 P12 P7 P8
V8	P8 P11 P4 P13 P17 P10 P16 P1 P14
V9	P6 P16 P15 P10 P19 P8 P14 P13 P4 P12 P18 P11 P2 P1 P5 P3 P3 P17
V10	P19 P14 P3 P1 P16 P12 P8 P11 P15 P17 P9 P18 P7 P5 P10 P2

Table 8.18: List of variants and their sequences in random example five

Variant	#	Variant	#	Variant	#	Variant	#
V1	6	V10	2	V3	12	V2	10
V9	18	V2	6	V6	8	V3	5
V8	1	V4	9	V5	17	V9	18
V4	11	V8	7	V2	17	V8	20
V10	2	V8	9	V5	20	V7	8
V7	7	V6	1	V4	2	V2	12
V8	19	V10	12	V8	4	V7	9
V10	6	V9	14	V2	11	V6	15
V4	10	V4	18	V2	15	V5	15
V2	12	V6	16	V3	1	V4	7
V1	2	V8	20	V5	1	V10	2
V1	4	V1	12	V3	3		
V8	4	V8	5	V2	9		

Table 8.19: Orders in random example five

W-P	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
W1-P1	8		17	11	15	20	19	14	18	25
W2-P2		23	6	25	6	23	4		8	13
W3-P3					15		19		2	22
W4-P4	22	12	23	20	21	16	18	25	11	
W5-P5			17	23	11	15			11	2
W6-P6		1	1	17	18	14			4	
W7-P7		18	9	3		5	9			5
W8-P8	17	11	16	10			16	11	3	4
W9-P9		19	1		22	1	17		2	11
W10-P10	12	10		8	21	16	25	21	11	22
W11-P11	18			7		24		23	19	14
W12-P12		4		8		21	2		14	14
W13-P13	19	1		10	24	13		13	3	
W14-P14			25		8	4	7	2		13
W15-P15	5	4			22	9	6		14	22
W16-P16	24	1		17	8	1	4	22	19	18
W17-P17	17		16		18	13	20	12	25	1
W18-P18	15	1	23		12	7	3		6	16
W19-P19		19	22	1	8	14	17		19	20
W20-P8	9	14	6	20			13	11	19	2

Table 8.20: List of process times per piece in random example five