FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

# Social Value Orientation Agents in Production Scheduling

## Master's Thesis

in partial fulfilment of the requirements for
the degree of *Master of Science (M.Sc.)*
in Praktische Informatik

submitted by
Jennifer Hanna

First examiner: Dr. Kai Sauerwald
Artificial Intelligence Group

Advisor: Dipl. Ing. Christopher Mühlich
Fraunhofer Institut für Produktionsanlagen und Konstruktionstechnik (IPK)

# Statement

I declare that I have written the master´s thesis independently and without unauthorized use of third parties. I have only used the indicated resources and I have clearly marked the passages taken verbatim or in the sense of these resources as such. The assurance of independent work also applies to any drawings, sketches or graphical representations. The work has not previously been submitted in the same or similar form to the same or another examination authority and has not been published. By submitting the electronic version of the final version of the master´s thesis, I acknowledge that it will be checked by a plagiarism detection service to check for plagiarism and that it will be stored exclusively for examination purposes.

I explicitly agree to have this thesis published on the webpage of the artificial intelligence group and endorse its public availability.

Software created for this work has been made available as open source; a corresponding link to the sources is included in this work. The same applies to any research data.

. . . . . .

(Place, Date) (Signature)

## Zusammenfassung

In Produktionssystemen bestehen in der Regel mehrere konkurrierende Ziele nebeneinander, wie die Minimierung der Durchlaufzeit und die Ausbalancierung der Maschinenauslastung. Das kombinatorische Problem der Zuordnung von Auftragsschritten zu Maschinen unter Beachtung der Optimierungsziele wird *Job Shop Scheduling Problem* genannt. Hier wird die Erweiterung des Problems um alternative Verarbeitungsrouten, das *Flexible Job Shop Scheduling Problem* (FJSP), betrachtet.

Verschiedene Verfahren, zum Beispiel auf der Basis von Heuristiken, sind in der Lage, Lösungen für das FJSP zu finden, doch dynamische Produktionsbedingungen stellen dabei durch die Notwendigkeit häufiger Neuberechnungen ein substantielles Problem dar. Eine Aufteilung des Gesamtproblems in Subprobleme durch Modellierung als Multi-Agentensystem verspricht eine höhere Flexibilität, doch die lokale Sicht der Agenten führt häufig zu suboptimalen Lösungen.

In dieser Arbeit wurde eine Methode entwickelt, die lokale Sicht von Agenten zu erweitern und damit ihre Kooperationsfähigkeit zu erhöhen, indem sie die Perspektive anderer Agenten in ihre Entscheidungen miteinbeziehen. Dazu wurde das soziologische Konzept der *Social Value Orientation* als variables Maß zur Abwägung zwischen individuellen und kollektiven Interessen herangezogen. Eine situationsabhängig variable Gewichtung erlaubt Agenten, sich flexibel an eine sich ändernde Umgebung anzupassen.

Ein entsprechendes Multi-Agentensystem wurde zusammen mit zwei Referenzalgorithmen in einer diskreten Simulationsumgebung implementiert, und unter Verwendung unterschiedlicher *Utility*-Funktionen getestet. Das System war in der Lage, große Probleminstanzen in einem Bruchteil der Zeit zu lösen, die von der exakten Referenzmethode benötigt wurde, mit Abweichungen in der Durchlaufzeit von 10 % unter statischen Bedingungen bis zu 3.3 %, wenn sowohl Aufträge als auch Maschinen dynamisch modelliert waren. Die Machinenauslastung wich stärker von der Referenzmethode ab, da durch die variable Gewichtung die Optimierung der Durchlaufzeit unter höherer Last bevorzugt wird.

Das entwickelte Verfahren zur Lösung von FJSP stellt somit ein vielversprechende Heuristik zum Einsatz unter hochdynamischen Produktionsbedingungen dar, die gute Lösungen in sehr kurzer Zeit liefern kann.

## Abstract

Scheduling in production systems aims at optimizing a number of competing goals, like processing time and workload balance. The combinatorial problem of assigning processing steps to machines in an optimized way is known as the job shop problem. Here, the extension of the problem by alternative processing routes, the flexible job shop scheduling problem (FJSP), is considered.

Multiple algorithms successfully tackle the problem, e.g. by applying heuristics, but a major issue arises when the environment changes dynamically, making frequent re-scheduling necessary. The division into sub-problems by use of multi-agent systems provides a more agile framework, but when scheduling decisions of autonomous agents are based on a local view of the search space, the resulting schedule tends to deviate from the overall optimum.

In this thesis, a method was developed that aims at countering the narrow view of autonomous agents in distributed scheduling systems by enabling them to consider the perspective of other agents, and thereby fostering cooperation. The method is based on the sociological concept of social value orientation, a measure for the individual balance between collective interest and self-interest. The measure is applied in a situation-dependent manner, allowing agents to flexibly adapt to environmental changes.

A multi-agent scheduling system based on social value orientation was implemented in a discrete event simulator along with two reference algorithms, and several utility functions were tested for effectiveness. The system was able to solve large benchmarking problems in a fraction of the time needed by the exact reference, with average makespan errors ranging between 10 % under static conditions and 3.3 % when both jobs and machines were modeled dynamically. Workload balancing deviated more severely from the reference, due to a flexible weighting approach that emphasizes makespan optimization under increased loads.

Taken together, the developed scheduling method was shown to be a promising heuristic for production scheduling in highly dynamic conditions, that can deliver good solutions in a very short time-frame.

# Contents

# List of Figures

# 1 Introduction

Industrial production is under immense pressure to compete on the global market, which makes efficiency more important than ever to stay competitive. Operations research (OR) is a research branch dedicated to optimizing production processes, which range from strategic planning to shop floor scheduling. Despite decades of extensive research, the field is far from exhausted. As production environments are continuously in flux, optimization strategies have to keep up and adapt.

Production environments are trending towards high flexibility, with diversified product portfolios and quickly fluctuating conditions. For example, addressing individual customer needs leads to more variable batch sizes of a wider range of products as compared to the early days of operations research. Stock-keeping is often reduced to lower warehousing costs, which in turn calls for high adaptability, e.g. when material deliveries are delayed [1]–[3].

While many good solutions exist for efficient production scheduling under static conditions, dynamically changing environments are often neglected or inadequately addressed. This thesis proposes a novel scheduling approach for shop floors that is suited for dynamic conditions. The intrinsic agility of multi-agent systems is leveraged to provide flexibility, and agents are equipped with basic perspective-taking abilities to foster cooperation. The latter is realized by utilizing the concept of social value orientation (SVO) to flexibly weight individual concerns of agents against each other. In order to test the algorithm, a simulation environment was developed that allows evaluation under static and two types of dynamic conditions, and can be easily expanded in the future.

The following section introduces the fundamentals of production scheduling before research questions and goals are addressed in Section 3. Section 4 provides general insight into the employed methods and frameworks, while Section 5 covers modeling and implementation of the proposed scheduling system. The evaluation is presented in Section 6, and conclusions are discussed in Section 7.

# 2 Fundamentals

Most products are manufactured in a multi-step process, which can be described as one job consisting of a sequence of operations. The operations are typically performed on different machines, while each single machine can be involved in the processing of multiple jobs. The combinatorial problem of sequencing operations on machines in the most efficient way is called the Job Shop Scheduling problem (JSP) [4], [5]. As there are usually several machines in modern shop floors capable of performing a particular operation, the optimization problem is complicated by the need to assign operations to specific machines, or in other words, to optimize the routing of a job. We focus on this extension of the JSP, which is called the Flexible Job Shop Scheduling Problem (FJSP) [6].

Flexible job shop problems can be formalized as a set of machines and a set of jobs, where each job consists of a sequence of operations. For each operation, a subset of machines is eligible, and processing times can differ on the machines. The following assumptions are typically made for a FJSP [7]:

- All machines are available from the beginning.
- All jobs are available from the beginning.
- No interruption is possible once an operation is started (non-preemptiveness).
- Machines can only process one operation at a time.
- Operations cannot start until their predecessor in the job's operation sequence is finished.

The time span from the beginning until the last operation is finished is called makespan. It is usually the main optimization target, as speed is a key factor in efficiency. Idle times of machines are detrimental to makespan, but they are often unavoidable, when the available operations do not perfectly match the machines' capacities. Balancing the machines' workload is another commonly used optimization goal, which is especially relevant when manual labor is required to operate machinery.

## 2.1 Scheduling Algorithms

Due to the combinatorial nature of the JSP, complexity rises rapidly with increasing numbers of jobs and machines. Alternative machines in the FJSP add to the complexity, although the routing problem can in principle be treated separately [6].

The complexity becomes evident when modeling the JSP as a disjunctive graph [8], with nodes representing the operations, directed conjunctive edges depicting consecutive operations of a job, and disjunctive edges representing the exclusion of simultaneous operations on a machine [9]. Figure 1 illustrates the principle on a small instance of a FJSP with three jobs and three machines. Feasible machine schedules can then be derived by constructing directed acyclic graphs containing the machine sequence paths [10]. Alternatively, the FJSP can be modeled as a Petri Net [11], where nodes represent locations, or a Markov chain of scheduling decisions [12].

Considering competing goals further complicates schedule optimization. Here, we focus on the goals of minimizing makespan and balancing the workload of machines. The general FJSP problem was shown to be NP-hard, but solutions for its instances can at least be approximated using various techniques [7], [14], [15]. The general method categories and some representative examples are introduced in the following paragraphs.

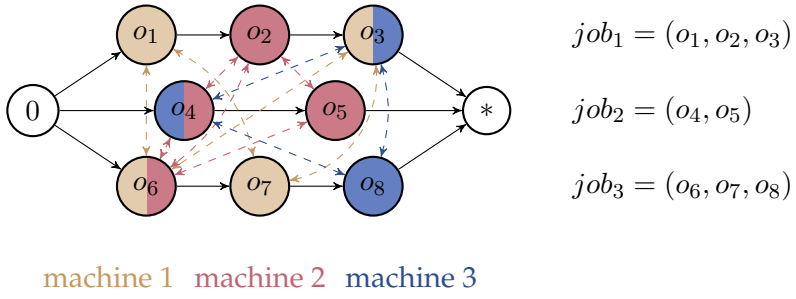



Figure 1: Disjunctive graph model of a FJSP with three jobs and three machines, adapted from [10] and [13]. Node (0) represents a virtual start node and node (*) the end. Jobs are depicted as a single row of operation nodes, connected by conjunctive edges (solid). Nodes are colored according to the eligible machines, with correspondingly colored disjunctive edges (dashed) depicting mutual exclusion of simultaneous operations of different jobs on the same machine.

### Exact Methods

For FJSP instances limited to a small number of machines and jobs, it is possible to calculate the optimal solution within reasonable time. Mixed integer linear programming (MILP) is commonly applied as an exact method, either by using generic solvers or within dedicated algorithms. Decision variables include the assignment of operations to machines, the relative sequencing of operations, and the concrete start time of operations [7]. A frequently used MILP-based technique is branch-and-bound, in which the search space is explored iteratively by branching to new solutions and applying upper and lower bounds to avoid the unnecessary exploration of infeasible branches [16], [17].

Constraint programming (CP) takes a slightly different approach for reducing the search space, namely by iteratively propagating changes applied to one variable to all related variable nodes [18], [19]. This technique is also known as finite domain propagation solving [20]. As especially problems with multiple related variables can benefit from that, CP has been found to be applicable for larger instances of the FJSP than MILP [7], [17].

A popular implementation of CP is the CP-SAT solver [21], which is part of Google's OR-TOOLS Suite. It combines CP with Boolean satisfiability (SAT) solving in a technique called Lazy Clause Generation. Constraints are expressed as sets of Boolean expressions, and domain changes of variables trigger the generation of new Boolean clauses that explain the domain change. An implication graph is created that is used to detect conflicts. Boolean variables in the implication graph get a temporary priority boost in the further search process, thereby allowing the search to focus on the harder parts of the problem [20]. In this thesis, the CP-SAT solver is used as an upper bound reference for schedule quality.

### Approximation Methods

In realistically sized JSP instances, determining the optimal solution by exact methods is often too time-consuming, thus settling for an approximation becomes necessary. Approximation methods can be classified according to the general approach as construction or improvement methods [22], or by distinguishing between simpler heuristics and more sophisticated, but often slower metaheuristics [7].

**Heuristics** Heuristics based on domain knowledge and / or neighborhood topology can be used to guide the search process and reach reasonable schedules in a short time frame [7], [23].

Beam search is an example for a heuristic modification of an exact method. It is based on the branch-and-bound technique and relies both on domain knowledge and on neighborhood topology. Beam search is a constructive approach, where the current decision branches are ranked based on a heuristic, and the search tree is narrowed down to the amount of branches specified by the beam width. Filtered beam search further refines the technique by combining local priority filtering with global cost evaluation, and can also be adapted to the FJSP [23], [24].

The most simple heuristic is the use of priority dispatching rules. Jackson's rule is a well-known example: Jobs are prioritized according to their remaining processing time, with the largest value getting the highest priority. This simple rule can already yield good results in some problem instances [25]. Priority rules are also a constructive approach and have the advantage that they can be easily applied dynamically (see Subsection 2.2). On the other hand, they rely completely on the quality of the injected knowledge [23], and no single rule has been found to yield consistently good results across multiple problem instances [26], [27]. Still, combinations of different rules can facilitate integrating competing optimization goals, and further refinement by metaheuristics or learning-based approaches can be used to improve solution quality [7], [26]. Here, we will apply a simple first-come-first-serve priority rule as the basis for scheduling decisions and as a lower bound reference for schedule quality.

**Metaheuristics** Metaheuristic approaches provide a way to improve the quality of a heuristic by introducing mechanisms to overcome the entrapment in local minima that greedy algorithms face [28]. Mechanisms from outside of the problem domain, e.g. physical or biological systems, are often leveraged to guide the process. While metaheuristics lack intrinsic means to evaluate the quality of the solution, their benchmark results are in many cases superior to more simple heuristics. Metaristics can be classified by the underlying principle as either trajectory- or population-based approaches [7].

Trajectory-based approaches closely resemble neighborhood search heuristics by relying on the topology of the solution space, but provide a less greedy way to navigate it, thus allowing the algorithm to escape local minima. Tabu-search is a well-

known example of a trajectory based approach, which iterates tabu restrictions to constrain the search and aspiration criteria to relax it [28]. Another common approach is simulated annealing, which is inspired by stochastic thermodynamic processes. Starting from an initial solution, cycles of slow constriction (cooling down) and relaxation (heating up) are performed until convergence is reached [12], [29].

Among the many population based approaches, genetic algorithms and particle swarm optimization are commonly used for the FJSP. The general idea is to start with a set of initial solutions, represented by the population, that are modified to further explore the solution space, keeping only the most promising solutions. Again, a stochastic component in the exploration helps to overcome local minima better than greedy improvement algorithms, that strictly adhere to the properties of the best solutions at a time.

Many combinations of different heuristic and metaheuristic techniques are found in the literature that are able to reach better solutions than a single method [7].

**Learning-based Approaches** Further improvement of initial solutions can be achieved by learning methods. Most importantly, reinforcement learning has gained attention in recent years to improve schedule optimality, especially when dynamically generated schedules are concerned. It draws on the principle of reinforcement well-known from the field of behavior research [30] by rewarding beneficial actions as a function of the system state. In an offline training phase, different states are explored, and beneficial actions for each state are learned by rewarding actions that improve the overall outcome or lead to states with an improved expected outcome. In this manner, system states are mapped to a set of actions that work well under the given circumstances. Furthermore, continuously updating weights during production can allow the system to adapt to formerly unexplored conditions [27], [31], [32].

**Summary** Scheduling algorithms can be categorized as exact, heuristic, metaheuristic, and learning-based approaches. While exact algorithms can guarantee optimality, their runtime can be excessive. Simple heuristics shine in dynamic environments by providing fast solutions, but deliver mixed schedule quality. Metaheuristics and learning approaches try to overcome limitations of heuristics, with the latter being especially suited for dynamic environments.

In this thesis, a first-come-first-serve heuristic will serve as the base scheduling algorithm for the FJSP and as a lower bound reference for schedule quality. Results of the CP-SAT solver will serve as an upper bound reference.

## 2.2 Scheduling Dynamics

The original job shop problem is defined as static, with predefined jobs that have to be arranged in a schedule [6]. Under static conditions, optimizing the production process in its entirety yields significantly better results than on-the-fly scheduling

by taking into account future consequences of scheduling decisions. With the advances in metaheuristic approaches, near-optimal global solutions can be achieved even for large problems, but computation times are often considerable. This makes complete rescheduling infeasible in case of disruptions that inevitably occur in real-life production systems, e.g. machine failures or delay in material delivery. Additionally, there is an increasing trend to more agile production schemes, with less stock-keeping and planning ahead, but instead dynamically incoming jobs and job cancellations, that calls for highly adaptive scheduling schemes [3], [33].

In this thesis, a scheduling approach suited for dynamic environments with incoming jobs and machine breakdowns is developed. Approaches to deal with dynamic environments can be categorized as predictive-reactive, completely reactive, and robust pro-active [34], which are discussed in the following paragraphs.

**Predictive-Reactive Scheduling** Predictive-reactive methods start with a precalculated (predictive) schedule and react to changes by event-driven and / or periodic rescheduling. Rescheduling methods include simple right-shifting of operations, insertions, complex partial rescheduling, and complete regeneration of the schedule [33]. To avoid introducing instabilities during disruption-free periods, hybrid scheduling systems have been proposed, which switch back and forth between a predictive and reactive state. This approach could potentially combine the best of two worlds, but the decision about when to switch between predictive and reactive scheduling is an additional challenge that is still not completely resolved [35].

**Completely reactive** In dynamic production environments, a predictive-reactive approach can result in extensive rescheduling activities and schedule nervousness [33]. In that case, a completely reactive approach may be favorable, where the schedule is replaced by ad-hoc push or pull decisions. Those decisions can be based on simple dispatching rules, e.g. choosing the machine with the lowest workload (push) or the job with the earliest due date (pull), or more complex combinations. To enhance the quality of dynamic scheduling, a lot of research went into evaluating the effectiveness of scheduling rules for different optimization goals and determining influencing factors in the environment [33].

**Robust pro-active Scheduling** Robust-pro-active scheduling aims for generating schedules that are robust against future disruptions and stochastic uncertainties, e.g. in production times [34]. This can be done by inserting idle time into the schedule that can absorb deviations, and thereby reduce nervousness of the system due to frequent rescheduling events, but the performance of such a system is dependent on the accuracy of predicted disruptions [33], [36], [37].

**Reinforcement Learning** Many recent publications on scheduling techniques for dynamic environments lean on reinforcement learning to adapt to the current system status. The scheduling process is modeled as a Markov decision process, where

the choice of action is only dependent on the current state. The action space is usually defined as a set of single or combined dispatching rules, and the most appropriate action for each state is learned during an offline training phase [32], [38]. While it seems intriguing to let the system learn by itself which rules to apply in which situation, the results are highly dependent on the choice of policies for the action space and of meaningful state features for the state space. As the convergence of reinforcement learning algorithms, particularly when based on Q-learning, relies on repeatedly visiting all states [39], extensive training periods would be necessary if the state space was made up of many low-level features. A suitable selection of high-level features is therefore crucial to limit training times and allow for generalization outside of the training scenarios. In addition, the scheduling performance of the system is limited by the quality of its action space, and both issues are still being actively investigated [40]–[42].

**Summary** Frequent dynamic events pose a significant challenge in production systems by invalidating precalculated schedules and prohibiting the repeated use of time-consuming scheduling algorithms. Approaches to that problem can be categorized as completely reactive, predictive-reactive and robust pro-active. Reinforcement learning can be employed to improve schedule quality by identifying suitable strategies for different system states.

Here, we consider a FJSP with dynamically incoming jobs and machine breakdowns. A predictive-reactive scheduling scheme is developed, with an event-driven rescheduling strategy.

## 2.3 Scheduling System Architecture

When optimizing the schedule for a small static problem, centralized scheduling is the natural choice to ensure global optimization, but the computational effort can get out of hand for large problem instances. A decentralized architecture can break the problem down into smaller parts that can be managed more easily. To this end, the shop floor can be modeled as a distributed system of software agents that make individual scheduling decisions on their behalf [43], [44]. In this thesis, a decentralized multi-agent architecture will be adopted.

**Multi-Agent Systems** Multi-agent systems are concurrent software systems, in which encapsulated entities, called agents, perform actions based on their perception of the environment, as schematically shown in Figure 2. Their complexity depends largely on the application they are designed for and can range from very simple reactive entities to agents with advanced reasoning skills and high adaptability [45], [46]. A fundamental architecture for designing practical reasoning agents is the Belief-Desire-Intention (BDI) model, where beliefs represent the knowledge an agent has gained about the environment, desire stands for its long-term goal, and intentions for the shorter-term strategies to approach said goal [47].

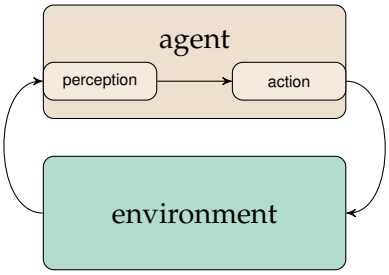



Figure 2: Abstraction of a reactive agent, adapted from [46].

The interplay between agents often leads to behavioral patterns that cannot be attributed to single agents. This phenomenon is known as emergent behavior, and can be seen as the result of combining simple rules of single agents to more complex results [48].

Software agents in a scheduling system typically represent jobs and machines and are modeled as self-interested entities. Each of them can only observe part of the environment, which bears the risk of convergence to local minima [49]. They are competing for resources: Job agents compete for machine availability, and machine agents compete for job assignments. Without a central coordinating instance, arising conflicts have to be resolved by the involved agents themselves by using their communication abilities.

**Conflict Resolution** Market-based approaches are widely applied to resolve conflicts in decentralized scheduling systems. Auctions or similar bidding systems, that are mostly based on the Contract Net protocol [50], rely on a repeated back-and-forth of messages between parties to settle on a decision [43]. For example, in a more recent publication, Madureira et al. proposed a Contract-Net-based negotiation mechanism, where machine agents trade operations with other machine agents based on their idle time [51]. As the bidding process can go on between the agents for multiple rounds, such an approach is prone to get lengthy, which is a serious disadvantage for real-time applications.

This thesis aims to develop a simplified approach to conflict resolution in multi-agent scheduling systems by equipping agents with the ability to evaluate the impact of their decisions on other agents. By weighting the benefits and losses against each other internally, agents honor each other's interests, while communication overhead is kept to a minimum.

In summary, decentralized architectures can render large scheduling problems more manageable by breaking them down into sub-problems. To that end, shop floors can be modeled as multi-agent systems consisting of job and machine agents, that negotiate operation assignments among themselves.

Cooperative conflict resolution in multi-agent scheduling systems is a key part of this thesis. In the context of production scheduling, cooperation can be described as the ability of working together to collectively approach a common optimization goal. Instead of utilizing a bidding system, we enable agents to respect the individual goals of other agents by employing the concept of social value orientation, which is explained in more detail in the following section.

## 2.4 Social Value Orientation

The concept of social value orientation (SVO) originates from the field of social sciences, where it is used to describe the degree to which an individual weights its own benefit against the benefit of the group. SVO types can be depicted as an angle between the axes of self-interest and group-interest in a coordinate system, where they form a unit circle. The relevant states for cooperative behavior are located in the upper right quadrant, in the interval between pure altruistic to solely self-interested attitudes [52] (see Figure 3).

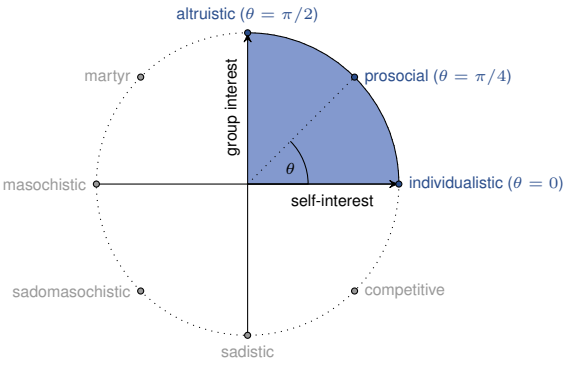



Figure 3: Unit circle of social value orientation (SVO), adapted from [53]. Self-interest and group interest are located on orthogonal axes. The weight that an individual applies to its own interest versus the interest of the group determines the angle of its SVO.

The concept of SVO has been successfully applied in scenarios of autonomous vehicles with an emphasis on navigating a mixed human-machine environment [53]–[55]. As a fundamental contribution, Buckman et al. introduced a utility function based on a vehicle's SVO angle $\theta$ that integrates the individual reward $R_i$ with the group reward $R_j$:

$$u_i = R_i \cdot cos\theta_i + R_j \cdot sin\theta_i \tag{1}$$

The factor by which a vehicle weights its own benefit follows a cosine curve from one at the pure individualistic SVO down to zero for the altruistic SVO, while the competitor's benefit is weighted according to the corresponding sine function. A coordinating instance can then perform pair-wise joint optimization of two vehicles' utilities, e.g. when they are competing for priority at an intersection [53].

In summary, social value orientation was introduced as a measure for altruistic versus self-interested attitudes of individuals, but can also be applied to model cooperative behavior in software agents. Here, we utilize it as a tool for conflict resolution between agents in a scheduling system, e.g. when two job agents compete for machine resources, and in balancing workload between machines.

### 2.5 Summary

The problem under consideration is a flexible job shop with dynamically incoming jobs and machine breakdowns. Two competing optimization goals are taken into account: Minimizing the makespan and balancing machine workloads.

A predictive-reactive scheduling approach with an event-driven rescheduling strategy is employed. To enable fast scheduling decisions, the problem is broken down into subproblems within a decentralized multi-agent scheduling system, and heuristic priority rules serve as a basis for the agent's decisions. As a novel contribution, the concept of social value orientation is employed to guide the agents' decisions towards the overall optimization goals. A first-come-first-serve heuristic and a CP-SAT solver algorithm are used as references for evaluation.

The following section delineates the research questions and goals.

## 3 Research Questions

Centralized scheduling suffers from low agility when rescheduling becomes necessary, e.g. in case of machine breakdowns or when jobs arrive dynamically. Despite the intrinsically distributed approach, most multi-agent scheduling schemes nonetheless employ a central entity for coordination and to enforce the pursuit of a globally optimized schedule. Such a central agent is at risk of becoming a bottleneck and thwarting the agility advantage of distributed scheduling. On the other hand, genuinely decentralized scheduling algorithms often fail to reach the same benchmark scores as centralized or hybrid approaches in terms of schedule optimality. It has been postulated that this is rooted in the agents' narrow view of their environment, and therefore an inability to consider the larger picture [49].

Broadening the view of individual agents, while omitting irrelevant information and thereby reducing computational effort, could counter this limitation of decentralized scheduling while preserving its advantages. To this end, an adaptation of the SVO concept to production systems is suggested, where agents are equipped with the ability to consider the perspective of competing agents. This leads to the first research questions (RQ):

RQ1. How can the SVO concept be employed to enhance cooperation in a multi-agent scheduling system?

RQ2. How does SVO affect solution quality in comparison to a basic FCFS system, and to results of an exact method?

Employing social value orientation is expected to add computational load and therefore extend the runtime for scheduling compared to the base algorithm. This is addressed by the third research question:

RQ3. How is the algorithm's runtime affected by integrating SVO?

As one of the optimization goals is related to machine utilization, while the other is not, we are interested in comparing solution quality under different shop workloads:

RQ4. Does the shop workload level influence solution quality?

Aside from incoming jobs, machine breakdowns are a common dynamic event in actual production systems. Therefore, a final research question addresses the scheduling system's robustness to machine breakdowns:

RQ5. Can the scheduling system handle machine breakdowns effectively?

To answer these questions, the following research goals are devised:

RG1. Implement a simulation environment for a multi-agent scheduling system.

RG2. Implement multi-agent FCFS scheduling.

RG3. Implement multi-agent SVO scheduling.

RG4. Implement an adapter between CP-SAT solver and simulation environment.

RG5. Evaluate performance of SVO scheduling under static conditions.

RG6. Evaluate performance under the condition of dynamic job arrivals.

RG7. Evaluate the influence of varying shop workloads on the optimization goals.

RG8. Evaluate performance under machine breakdowns.

The following section goes into details of the employed methods and third party libraries that were used.

# 4 Methods

Discrete event simulators are indispensable tools for modeling operations research problems and developing solutions. An abundance of options is available, both as commercial applications and as open source frameworks in multiple programming languages [56]. For our purpose, we opted for an open source solution versatile enough to implement reactive agents, and fast enough for reasonable computation times.

The following subsections go into the specifics of the employed simulation framework, the reference algorithm CP-SAT, and the benchmark data used to create concrete problem instances.

## 4.1 Simulation Framework

Simulation environments were implemented in Kotlin using the discrete event simulation engine KALASIM [57]. It is based on the Python framework SALABIM, but offers better performance, since Python is slower due to being an interpreter language, and cannot handle concurrency very well. Kotlin on the other hand facilitates concurrency handling by extending the base language Java with the coroutine concept, thus providing an excellent basis for implementing autonomous agents.

The framework KALASIM allows to define simulation components with independent action processes and states, as well as events the components can respond to. A component can be scheduled for activation directly by other components or indirectly within event callbacks of the environment, either for the next available time or for a specific time in the future. The simulation engine manages a queue of components that are scheduled for activation, and offers a priority flag system for components that are scheduled for the same time. When activated, a component's process function is executed until it reaches its end or executes an deactivation statement. A component can be deactivated either for a specific time, until another component enters a a specific state, or until further notice.

## 4.2 CP-SAT Solver

The CP-SAT Solver was used as an upper bound reference for schedule quality. It is part of Google's OR-TOOLS suite, an open source software for combinatorial optimization, targeted at operations research (OR) applications. CP-SAT was briefly explained in Subsection 2.1 as a commonly used representative of exact methods to solve scheduling problems. The solver itself is implemented in C++, but also provides APIs for Python and Java.

An existing Python wrapper for solving FJSPs was provided by Fraunhofer IPK. The algorithm uses CP-SAT's Python API to encode the benchmark problem as variables, constraints, and objectives, and calls the CP-SAT solver to generate a solution. The implementation was ported to Kotlin, adapted to our multi-agent structure, and extended by the capability to handle dynamic environments.

## 4.3 Benchmark Data

The properties of job and machine agents were defined using benchmark problems from the literature [6], [10], [58]–[62]. The data sets were partly provided as a collection by Mastrolilli [63], and complemented by Fraunhofer IPK.

The benchmark sets provide a wide range of FJSP examples with different properties that are summarized in Table 1. The number of problem instances in a set is given by # inst, and instance sizes can be gauged by the average number of jobs ($\overline{\#j}$), machines ($\overline{\#m}$), and operations ($\overline{\#o}$). The difficulty to solve an instance to optimality increases with its size and flexibility. $\beta$ is given as a flexibility measure, defined as the fraction of machines an operation can be processed on (Equation 2).

Table 1: Benchmark set properties. # inst: number of instances, $\overline{\#j}$: average number of jobs, $\overline{\#m}$: average number of machines, $\overline{\#o}$: average number of operations, $\overline{\beta}$: average $\beta$-flexibility, $\overline{dv}$: average duration variety.

| set | | # inst | $\overline{\#j}$ | $\overline{\#m}$ | $\overline{\#o}$ | $\overline{\beta}$ | $\overline{dv}$ | |
|---|---|---|---|---|---|---|---|---|
| Barnes | | 21 | 13.33 | 13.67 | 158.33 | 0.089 | 0.422 | [59] |
| Behnke | | 60 | 45.00 | 40.00 | 225.00 | 0.316 | 0.018 | [62] |
| Brandimarte | | 10 | 15.50 | 8.70 | 141.40 | 0.345 | 0.036 | [6] |
| Dauzere | | 18 | 15.00 | 7.67 | 292.00 | 0.330 | 0.169 | [64] |
| Fattahi | | 20 | 5.35 | 5.10 | 17.40 | 0.517 | 0.744 | [61] |
| Hurink | vdata | 66 | 14.76 | 8.85 | 133.39 | 0.476 | 0.159 | [58] |
| | edata | 66 | 14.76 | 8.85 | 133.39 | 0.151 | 0.477 | [58] |
| | rdata | 66 | 14.76 | 8.85 | 133.39 | 0.258 | 0.279 | [58] |
| | sdata | 66 | 14.76 | 8.85 | 133.39 | 0.131 | 0.550 | [58] |
| Kacem | | 4 | 9.75 | 8.00 | 31.75 | 1.000 | 0.093 | [60] |

The distribution of processing times $pt$ among eligible machines for an operation is captured in the duration variety $dv$ (Equation 3).

$$\beta = \frac{avg(\#m_{eligible})}{\#m_{total}} \tag{2}$$

$$dv = \frac{\#pt_{distinct}}{\sum_{ops} \#m_{eligible}} \tag{3}$$

Both $\beta$ and $dv$ were suggested by Hutter [65] as measures for defining FJSP complexity.

### 4.4 Summary

The framework KALASIM was chosen as a discrete event simulation engine to model the environment and implement multi-agent production scheduling. An existing CP-SAT solver implementation in Python served as a template for an adaptation to the specific needs in multi-agent scheduling with KALASIM. Benchmarking data from the literature encompassing a wide range of problem instances was used to evaluate the algorithms.

## 5 Modeling and Implementation

The complete implementation is publicly available under `https://github.com/ComfyChair/KalasimFJSP`. Figure 4 visualizes the package structure, which reflects the overall architecture of the scheduling system.

The model package encompasses classes representing the problem domain, like an abstract `Job` and `Machine` class. The main classes within the environment package are the `FJSPEnvironment` class, which serves as the static environment, and

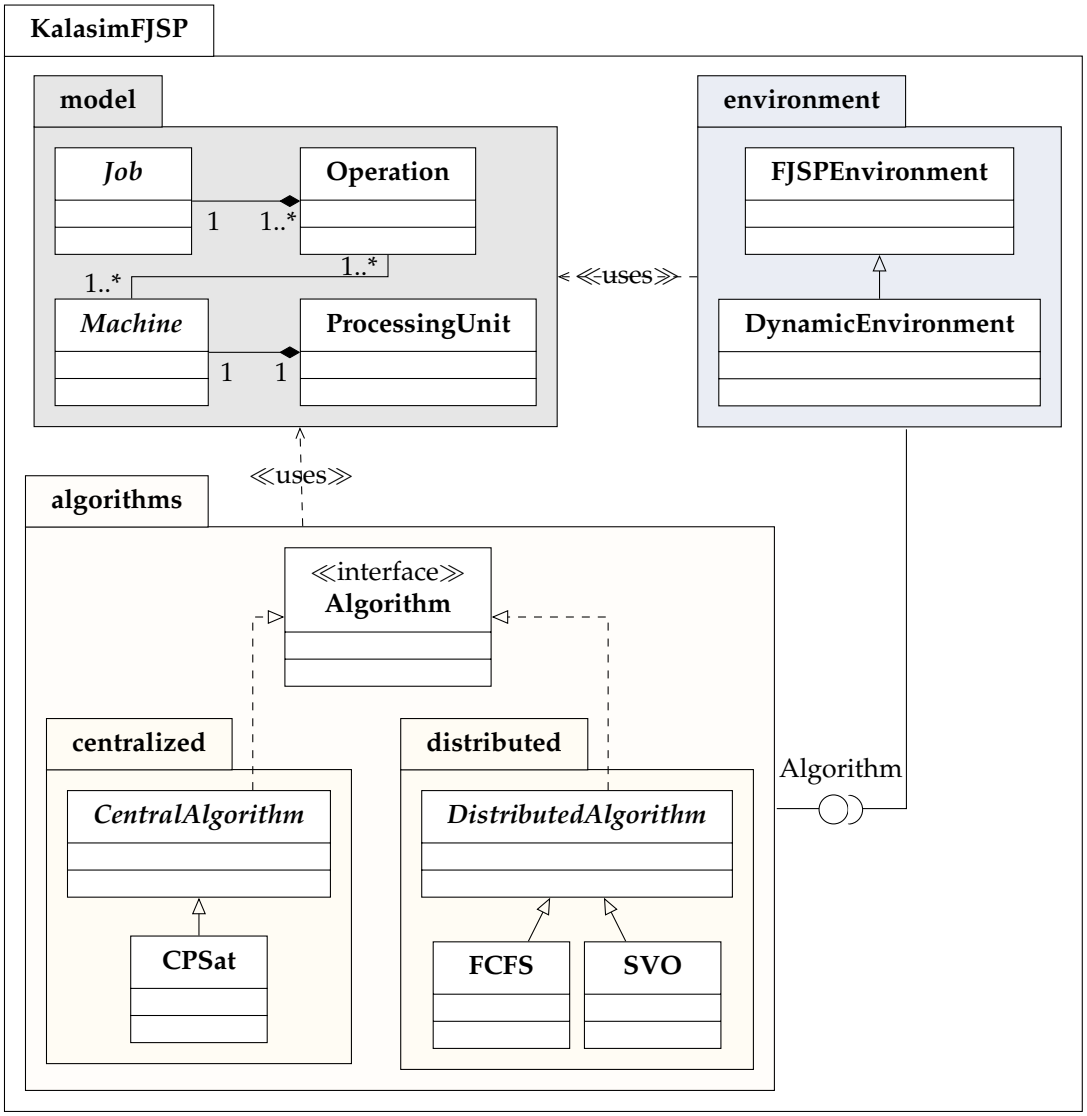



Figure 4: Package structure as an overview over the system architecture. Algorithms use the base classes defined in the model package. The simulation environment is agnostic of the concrete algorithm, but uses the Algorithm interface.

the derived `DynamicEnvironment` class, which can be used for both dynamic job arrival scenarios as well as to simulate machine breakdowns, depending on the `type` flag used for initialization. The algorithms package contains classes necessary for solving the FJSP within the environment. The `Algorithm` interface is implemented by the abstract classes `CentralAlgorithm` and `DistributedAlgorithm`, which are located in separate sub-packages. The concrete algorithms `FCFS`, `SVO`,

and `CPSat` inherit from their respective abstract superclass. As the environment is agnostic to the concrete algorithm implementation, the software can be easily extended by new algorithms. The distributed algorithms serve as factories to provide concrete job and machine agents to the environment, while the centralized algorithm `CPSat` also implements a solve method and a central agent as an adapter to the environment.

The following subsections go into further details of the implementation.

## 5.1 Simulation Environment

To approach Research Goal RG1 of implementing a simulation environment for a flexible job shop with dynamically arriving jobs and machine breakdowns, we started by implementing a static system, and expanded it to the dynamic scenarios later on. The shop floor was modeled as a multi-agent system with job agents and machine agents that are directly negotiating scheduling decisions.

Job and machine agents, operations and processing units of machines were implemented as separate components in the discrete event simulation engine KALASIM. Details about job and machine agent design are given in Subsection 5.2. Operation and processing unit components serve to simulate the actual processing of operations in dynamic environments and are explained in the following paragraph.

**Dynamic Processing Simulation** Operations were implemented as mainly passive components with the states UNSCHEDULED, SCHEDULED, PROCESSING and DONE. They are activated only once in their life cycle to start processing, wait for the duration specified by their machine assignment, and permanently deactivate themselves when reaching state DONE.

Processing units each belong to a machine agent and start in an IDLE state. Machine agents regularly inform their processing unit about upcoming operations or breakdown periods in the internal schedule, which schedules the processing unit for activation at the corresponding start time with low priority. This priority setting ensures that machine and job agents can safely conclude schedule negotiations, and that any prior processing or breakdown periods are finalized.

Figure 5 illustrates the interplay of processing units and operations to simulate processing in dynamic environments. Upon activation, a processing unit switches to PROCESSING or BREAKDOWN state. In case of a breakdown, the unit deactivates itself for the breakdown duration. In case of processing, it switches the state of the corresponding operation component to PROCESSING, activates it, and waits for its state change to DONE. Afterwards, the processing unit requests the next upcoming schedule slot from its machine agent, adapts its state accordingly, and schedules its own activation for the corresponding start time. When no further operations or breakdowns are scheduled, it deactivates itself until further notice.

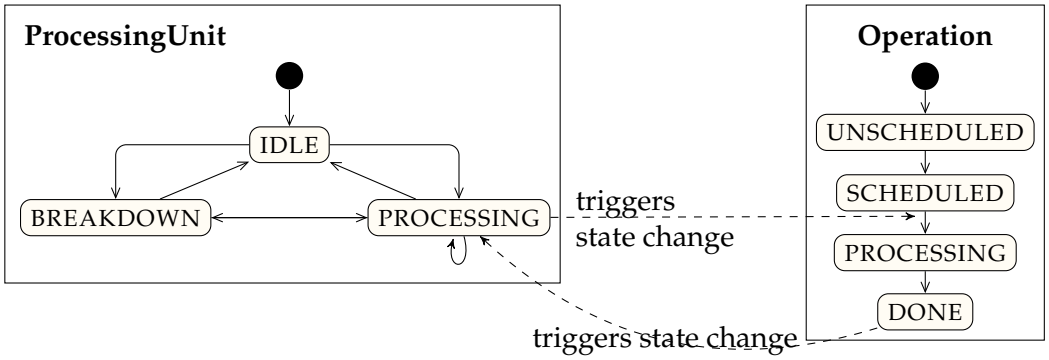



Figure 5: State diagram of the processing units and operations. Upon entering PROCESSING state, the ProcessingUnit triggers a state transition of the Operation to PROCESSING and waits until the Operation reaches state DONE. The following state transition depends on the machine schedule.

**Dynamic Job Injection** Dynamic job arrival scenarios were derived from static benchmark data by duplicating the set of jobs. One set was released into the environment immediately, and a pre-schedule was generated by running a static simulation, assigning all operations to a specific machine. The sum of the processing time for all operations as assigned in the static run was defined as 100 % workload.

For the dynamic part, a `JobInjector` was implemented, which injects the duplicated set of jobs one by one in a randomized order. The time point for injection was determined by simulating processing as described in the preceding paragraph, and monitoring the remaining workload at each discrete timestep of the simulation. A new job from the duplicated set was injected when the workload dropped below a predefined threshold. Thresholds were chosen according to values suggested by Dar-El and Wysk [66] for high (85%), medium (77%) and low (70%) workloads of a job shop.

The procedure for determining injection times is shown in Algorithm 1. As processing times for operations differ between eligible machines, the workload for an operation is not a static value. It is initially estimated by calculating the median processing times for all possible machine assignments, and gets updated to the actual processing time when assigned to a specific machine.

**Dynamic Machine Breakdowns** The KALASIM framework provides methods for a variety of distribution functions, which were used to randomize start times and durations of breakdown periods. Algorithm 2 depicts the main process of the resulting `BreakdownGenerator` class. Each machine is initially assigned a future breakdown, which is updated with a new future breakdown when the simulation reaches that time point.

Breakdown durations are assumed to be predictable, meaning that machine agents can continue scheduling operations for subsequent time periods, and are random-

**Algorithm 1** JobInjector Process

**Precondition:** $initialSet$ of operations is assigned
**Postcondition:** all jobs from the $injectionSet$ are injected

```
function PROCESS
    allJobs ← initialSet
    waitFor ← target · Σ_allJobs Σ_ops workload        ▷ target: workload fraction
    while injectionSet ≠ ∅ do
        workloadEstimate ← Σ_allJobs Σ_ops remainingWorkload
        while workloadEstimate > waitFor do
            wait for 1 simulation tick
            workloadEstimate ← Σ_allJobs Σ_ops remainingWorkload
        newJob ← draw random job from injectionSet
        allJobs ← allJobs ∪ {newJob}
```

**Algorithm 2** BreakdownGenerator Process

**Precondition:** $nextBreakdown$ is initialized as a mapping from each machine to a future `Breakdown` with time and duration drawn from the distribution functions

```
function PROCESS
    machine, breakdown ← entry in nextBreakdown with min(breakdown.start)
    wait until breakdown.start
    initiate breakdown in machine
    nextBreakdown[machine] ← new random Breakdown
```

ized along a uniform distribution of 10 to 30 min as suggested by Lv et al [67]. Breakdown occurrence is randomized with exponential distribution as suggested by He and Sun [68]. The mean distance between breakdown events was set to approximate one event per machine within one run if the schedule is tightly packed. This was calculated from the minimal makespan that could be achieved in case of perfect packing according to Equation 4, by dividing the sum of minimal processing times for all operations by the number of machines.

$$meanDistance = \frac{\sum_{ops} \min(processingTime)}{numMachines} \tag{4}$$

## 5.2 Agent Design

Design decisions for distributed agents apply to both the `FCFS` and the `SVO` algorithm and therefore address both Research Goal RG2 and RG3. Job and machine agents are modeled as independent simulation entities (components) in the framework KALASIM. This allows us to dissect the FJSP into separate assignment and sequencing problems by letting job agents decide about assignments, while machine

agents handle operation sequencing. Benchmarking data is used to instantiate the components: Operations are initialized with a mapping of eligible machines to processing times and associated with a specific sequence position within their jobs.

**Class Diagrams** In order to separate the domain model from the concrete solution algorithms, abstract `Job` and `Machine` classes were designed as shown in Figure 6.

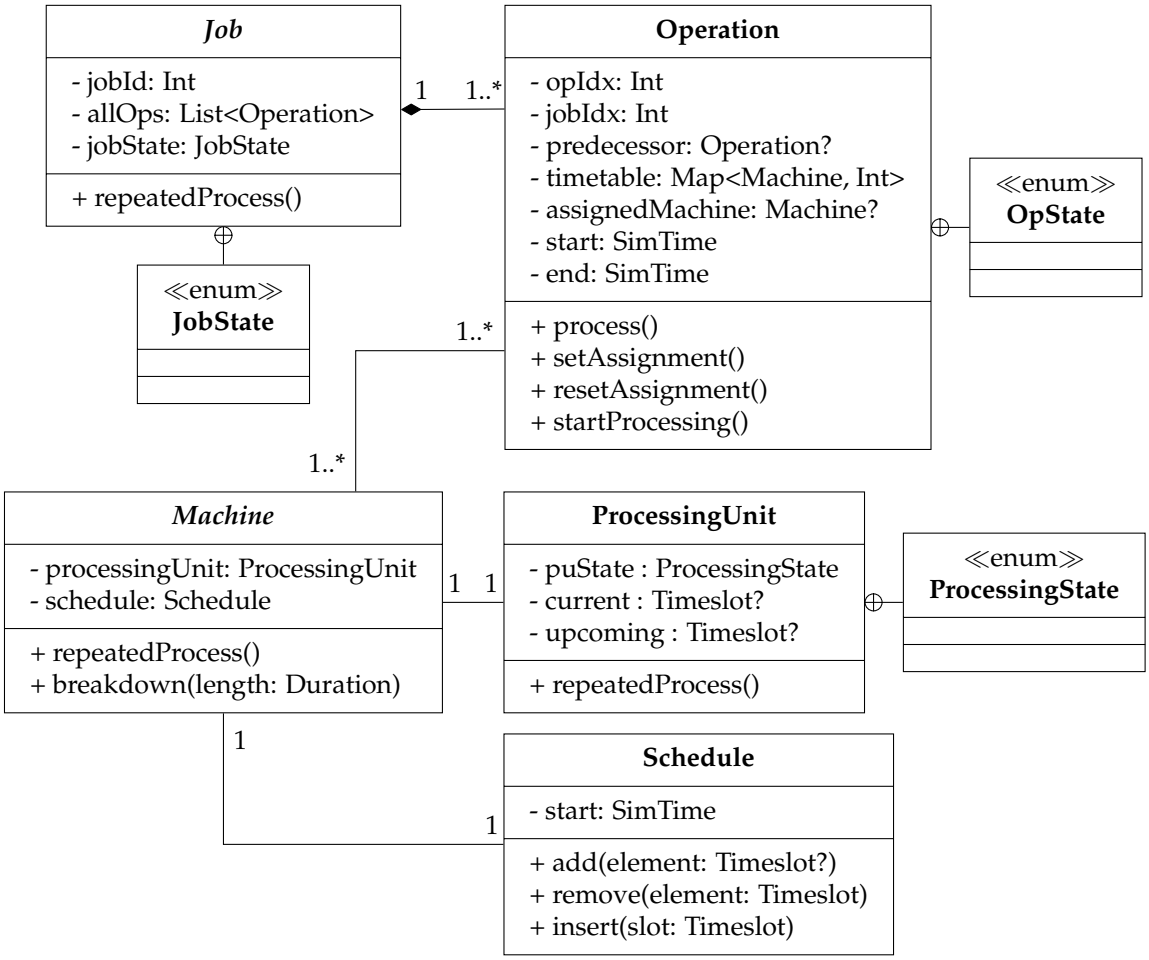



Figure 6: Domain model as a class diagram.

The instantiation process is thereby separate from the concrete algorithm. The simulation environment parses the benchmark data and instantiates `Operation` objects. Those are passed on to the algorithms, which act as agent factories by providing concrete job and machine agents to the environment.

Figure 7 illustrates the inheritance relations and internal structure of job agents as a class diagram. Properties inherent to the problem domain, like the operation sequence, are located in the abstract `Job` class. The abstract `DistributedJob` class

encapsulates general properties and methods for communication tasks that are independent of the algorithm. `FcfsJob` then implements the specific logic of actions depending on its state and received messages. Since no specific logic needed to be implemented for job agents in the `SVO` algorithm, `FcfsJob` could be reused there. The centralized reference algorithm `CPSat` provides a passive job agent to the environment for compatibility reasons.

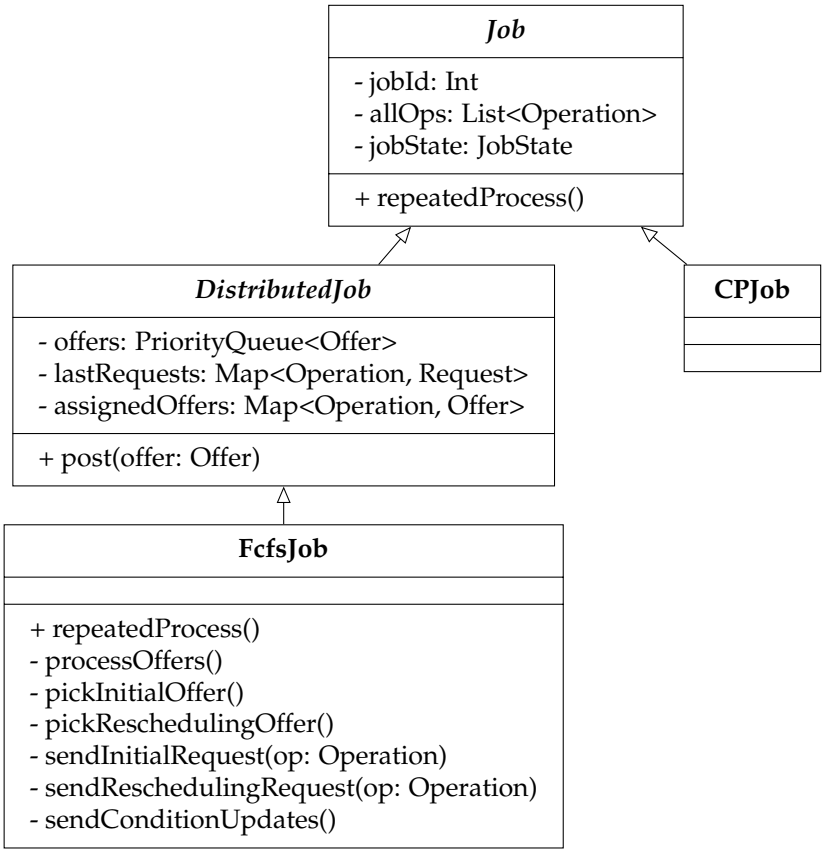



Figure 7: Structure of job agents as a class diagram.

Likewise, the inheritance relations and internal structure of machine agents are displayed in Figure 8. The abstract `Machine` class provides the general structure with references to `Schedule` and `ProcessingUnit` objects, and functions related to the dynamic environment, like the `breakdown()` method. The `Schedule` object plays an essential role, as it manages time slots for scheduled operations and breakdown periods, and enforces that they do not overlap. The abstract `DistributedMachine` class encapsulates properties and functions for state and communication handling that are only necessary in distributed algorithms. The concrete `FcfsMachine` class implements the scheduling logic based on the first-come-first-serve paradigm. Its subclass `SVOMachine` reuses part of its code, but applies SVO principles for optimizing its schedule in its process definition. Ma-

chine agents of the reference algorithm `CPSat` implement compatibility methods for operation assignment and the additional method `minimizeStarts()` to refine the solution.

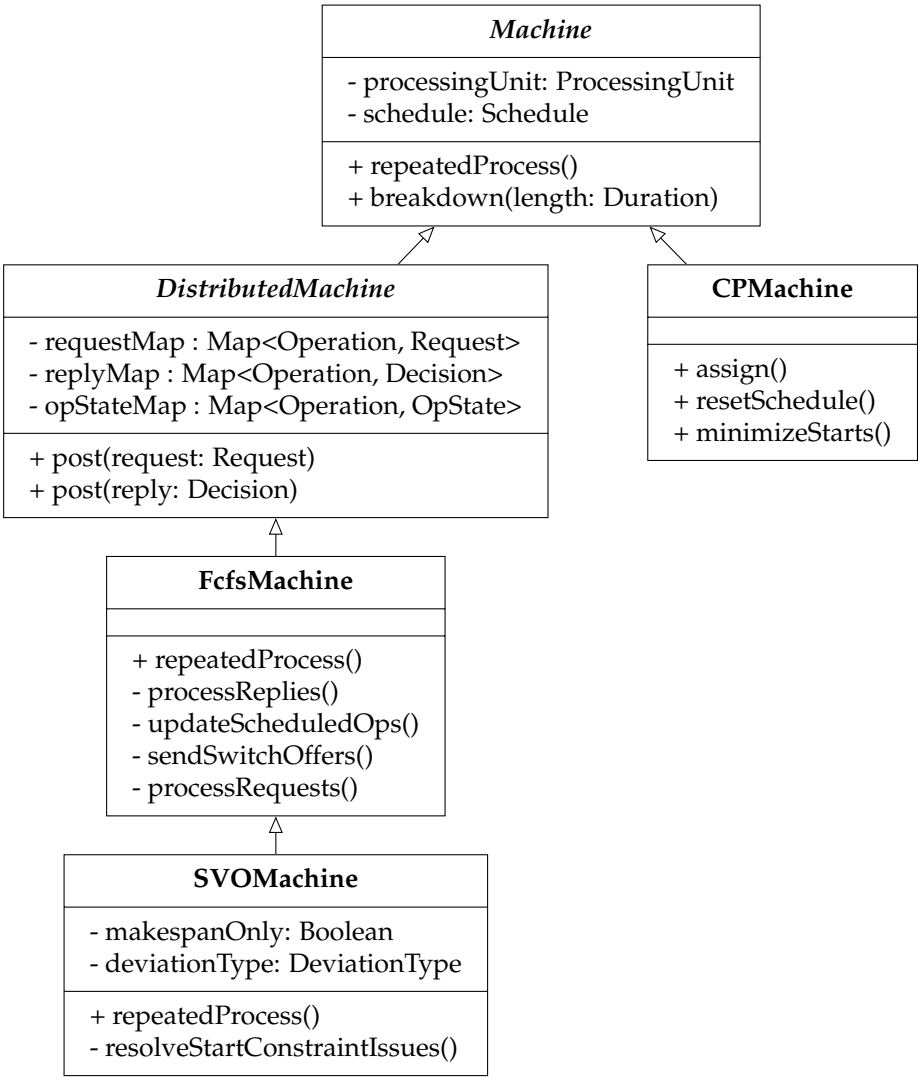



Figure 8: Structure of machine agents as a class diagram.

**Goals** The optimization objectives of minimizing makespan and balancing workload both refer to the complete shop and are not directly accessible as a target function on the single agent level. By designing agent goals with equivalent concepts, the collective goals should be approached as an emergent behavior of the multi-agent system.

Makespan refers to the time needed to complete all jobs and corresponds to the maximum completion time of all jobs [7]. Individual completion times can therefore be considered equivalent on the level of single jobs. On the level of single scheduling decisions, job agents are assigned with the goal to reduce the end time of individual operation assignments in order to minimize their completion time.

To model workload balancing on the agent level, we assign machines the goal to maximize their individual utilization rate. With social value orientation in place, machine workloads should balance out across the shop. Note that due to the absence of SVO, our `FCFS` base algorithm does not implement any machine goals, but only pursues the makespan objective.

**Communication** In order to jointly achieve a goal, like creating a consistent schedule, agents need to communicate. We simulate communication by posting messages to the receiver's message queue and utilizing the event handling system of the simulation framework to schedule the receiver component for activation. Message classes were designed for specific purposes of requesting (`Request`, job agent), offering (`Offer`, machine agent), and accepting or declining offers (`Decision`, job agent), as shown in Figure 9.

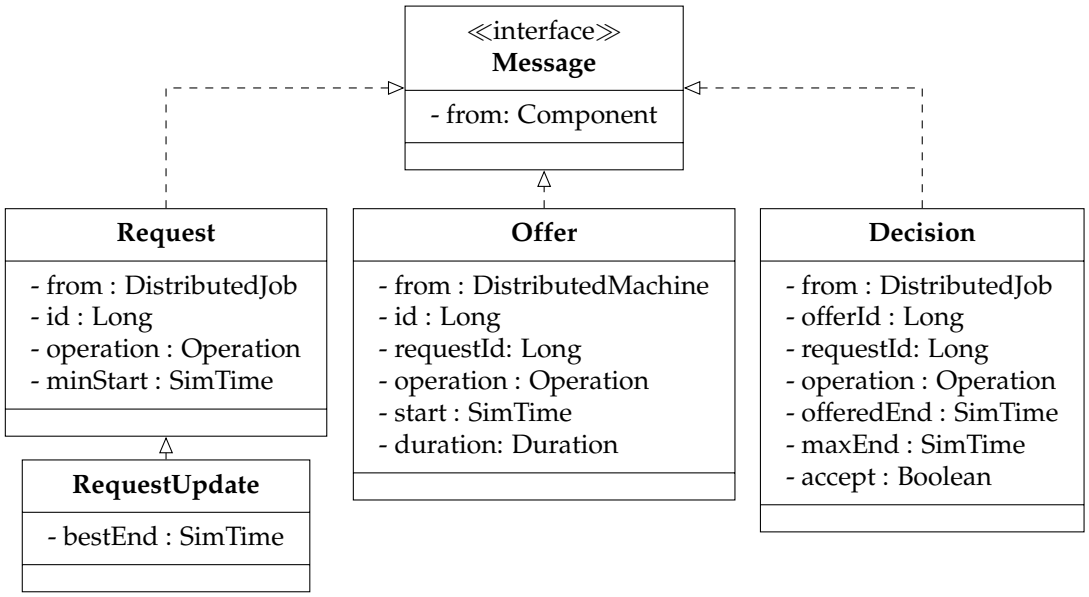



Figure 9: Class diagram of messages for communication between agents.

**Actions** When activated by messages from other agents, or by environmental events, agents perform actions defined in their `repeatedProcess()` method. The information transferred by the events and messages is their source for perceiving changes in the environment, and is thus shaping their beliefs. Within the discrete

event simulation environment, the pattern of components entering and leaving the active state is crucial for preventing both an early abortion of the simulation before reaching a solution, as well as endless activation cycles.

Figure 10 shows the state diagram of a `DistributedJob` agent. When a job agent is activated by an event, it first assesses the processing state of its operations. If processing of all operations is finished, the agent deacitvates itself permanently. Otherwise, it processes received messages in the order of its operation sequence and initiates scheduling or re-scheduling of operations if necessary.

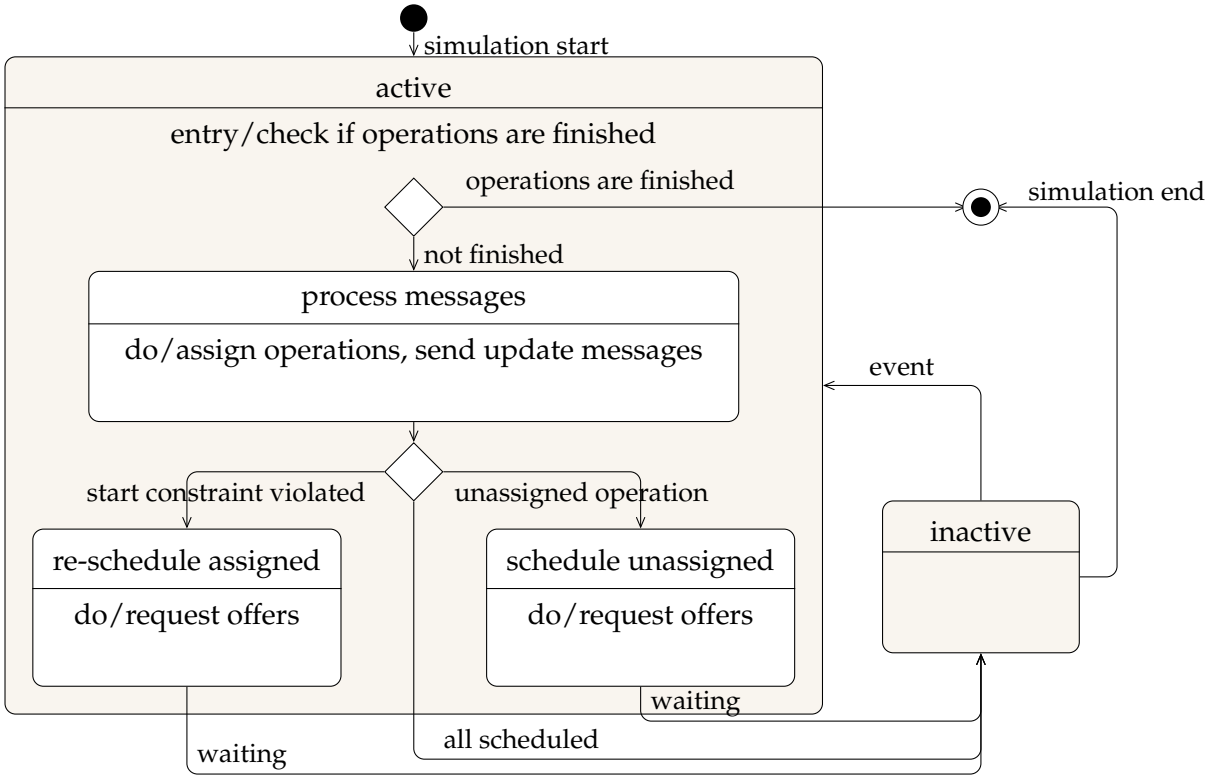



Figure 10: State diagram of job agents.

The knowledge of a job agent is limited to the scheduling state of its operations and the offers it received from machine agents. Scheduling requests for an operation are sent out when its predecessors are scheduled without start constraint violations, i.e. no operation is scheduled for an earlier time than its predecessor end. A `DistributedJob` always chooses the offer with the earliest end time, while honoring start constraints. It updates machine agents about changes in start constraints and current end time of an operation in `sendConditionUpdates()`, to prevent that machine agents propose invalid or worse time slots when sending updated offers. When start constraint violations occur, job agents adapt their behavior by postponing further message processing. In order to resolve the violation, the agent sends out re-scheduling requests and enters an inactive state while waiting for re-

sponses. If no start constraint violation is detected, the agent requests scheduling of the next unscheduled operation in its operation sequence, if applicable. The agent then returns to an inactive state, waiting for responses to scheduling requests or for other activating events.

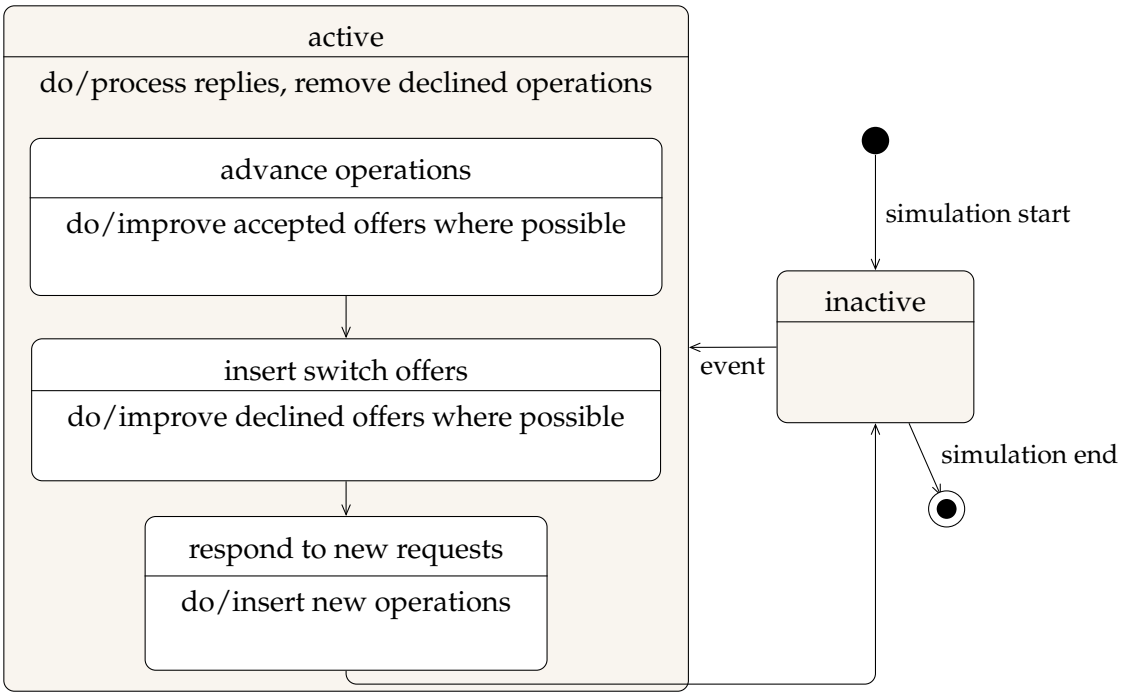



Figure 11: State diagram of `FCFSMachine` agents.

The actions of `FCFSMachine` agents are visualized as a state diagram in Figure 11. Upon activation, machine agents process replies to previously made offers and remove operations from the internal schedule if the offer was declined. Next, all accepted offers are checked for potential advancements, as the removal of declined offers might have opened up better schedule positions. Then, all previously declined offers are checked for the possibility to improve them. Finally, new scheduling requests are processed by inserting the corresponding operations into the internal schedule and sending offer messages to the job agents.

For `SVOMachine` agents, the process had to be modified (Figure 12). First of all, start constraint violations can occur when predecessor operations are pushed back, and this is resolved immediately after processing `Decision` and `RequestUpdate` messages. Secondly, pushing back scheduled operations can lead to instabilities in the schedule, in the sense that operations repeatedly switch places in subsequent activation periods. To stabilize machine schedules, the step for advancing operations was moved to the end of the action process. Instead of only advancing confirmed operations, all scheduled time-slots are updated in an iterative process, sorted by descending deviation of their current best end from the ideal value. Only after no further beneficial moves are found, offers are sent out for all schedule changes.

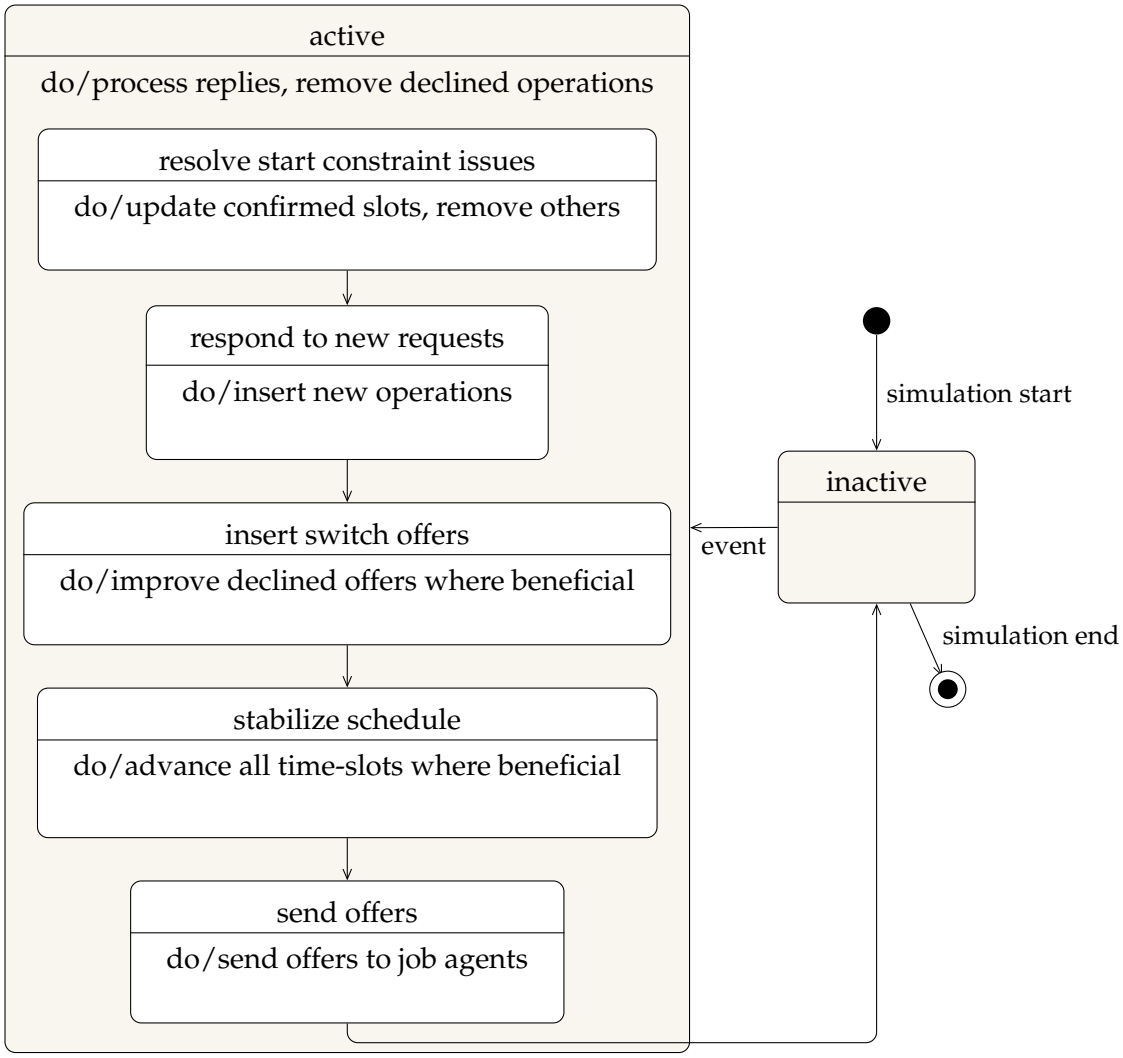



Figure 12: State diagram of `SVOMachine` agents.

`DistributedMachine` agents maintain knowledge about each operation they have received requests for. When generating new or updated offers, `FcfsMachine` agents employ simple insertion and left-shifting into gaps. `SVOMachine` agents use utility functions to choose the insertion slot with the greatest overall benefit, which is explained in more detail in the following subsections.

## 5.3 Negotiation Algorithm

The negotiation protocol between agents can be described as a variant of the Contract Net protocol [50]. Job agents communicate scheduling requests, and all machines eligible for processing an operation are required to respond with an offer. Job

agents reply to offers by sending an accept or decline message. Further messages are exchanged if rescheduling becomes necessary, either from the machine's or from the job agent's end, or if the machine agent can improve an offer due to an opening in its schedule. Thus, message exchange for each scheduling decision is kept to a minimum, reducing communication overhead compared to market-based approaches with multiple bidding rounds.

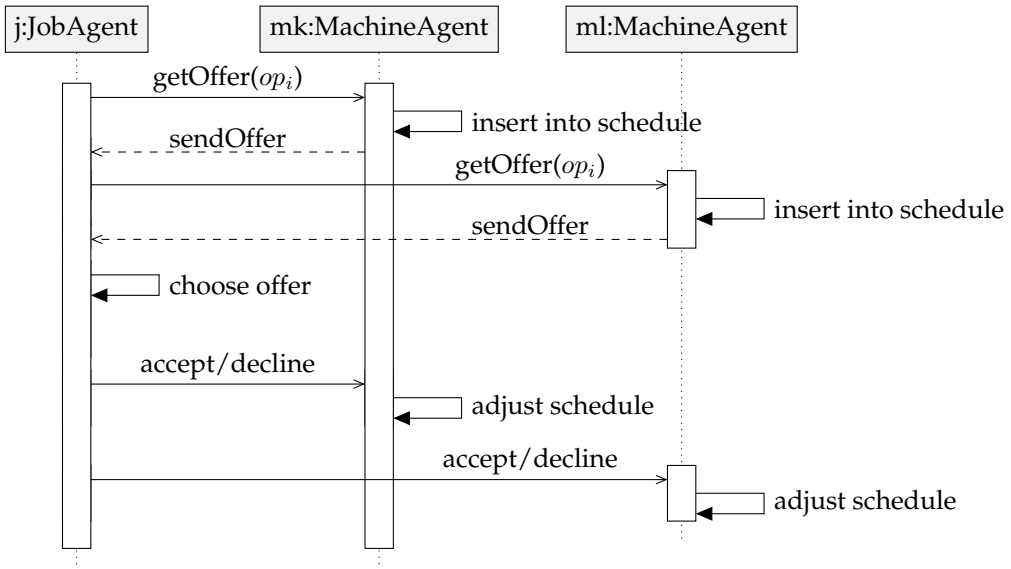



Figure 13: Sequence diagram of scheduling decisions. Job agents request scheduling from all eligible machines. Machine agents insert the operation into their schedule and respond with an `Offer`. Job agents choose the best offer and send `Decision` messages to the machine agents. Machines that received a decline message adjust their schedule by removing the blocked time slot from their schedule and check for possible improvements of other offers.

Scheduling is initiated by job agents for each of their operations consecutively, until all of them are scheduled. As the negotiation algorithm is constituted by the interplay of job and machine agents, it is visualized as an interaction diagram in Figure 13. Job agents are responsible for assigning operations to machines, while machine agents decide about sequencing operations within their schedule. Details of the process from the view point of its participants are explained in the following paragraphs.

**Job Agents** Algorithm 3 depicts the scheduling process from the view point of the initiating job agent. A scheduling request is sent out to all eligible machines for the first unscheduled operation in the operation sequence (line 4 – 5). Job agents accept the offer with the earliest end, and decline all others (line 8 – 9). The end

time of the accepted offer restricts the start time of the following operation, which is scheduled next. This process is the same for the `FCFS` base algorithm and the `SVO` algorithm.

**Algorithm 3** Operation Scheduling by Job Agents

**Precondition:** job agent $j$ with operation sequence $O_j =< o_1..o_n >$ has scheduled all operations preceding $o_x$

**Precondition:** eligible machines are known to the job agent
**Postcondition:** operation $o_x$ is scheduled on an eligible machine

```
1: function SCHEDULE(o_x)
2:     startConstraint ← if x = 1 now else end_{o_{x-1}}
3:     M_x ← eligible machines for o_x
4:     for m ∈ M_x do
5:         send scheduling request to machine agent m with startConstraint
6:     collect responses in offers (non-blocking)
7:     if activated and offers contains valid offers then
8:         acceptedOffer ← o ∈ offers with min(offer_end)
9:         send Decision in reply to all o ∈ offers
10:        if size(M_x) > size(offers) then ▷ also inform other agents about new best end
11:            send RequestUpdate to all agents in M_x
```

We originally started with a version of the algorithm were job agents wait for offers from all eligible machines. This led to complications when postponements came into play, e.g. in SVO based scheduling or with machine breakdowns, where start constraint changes can invalidate offers before they are processed. To simplify matters, we decided to let job agents pick the best valid offer, even if some machines did not reply yet (line 7). Although this increases message exchange in case that better offers are received later on, the system performance was barely impacted. As an additional benefit, unresponsive machines cannot stall the scheduling process, although such real-time communication issues were not included in our simulation.

**FCFS Machine Agents** Algorithm 4 depicts the scheduling process from the view point of `FCFSMachine` agents. They handle scheduling requests by inserting operations into their schedule on a preliminary basis and sending a response with the start and end time of the schedule position. The first-come-first-serve principle is honored by prohibiting the postponement of already scheduled operations, thus new operations are inserted into gaps within the schedule or appended at the end.

Machine agents can handle multiple requests at the same time. When an offer is declined, a previously blocked time-slot is freed up, and the schedule is checked for potential improvements. Updated offers are then sent out for operations that can be advanced. In case the previous offer has not been processed yet by the job agent, the new offer overwrites the previous one.

**Algorithm 4** Schedule Insertion by Machine Agents (FCFS)

**Precondition:** machine agent with schedule positions $S =< s_1..s_n >$, with $s_x$ being operations, breakdown periods, or gaps

**Precondition:** machine agent has received a scheduling request for operation $o_{jx}$

**Precondition:** $startConstraint$ has been communicated by the requesting job agent

**Precondition:** $processingTime$ is known by the machine agent
**Postcondition:** $o_{jx}$ inserted into schedule, $offer$ generated

**function** OFFERFCFSBASED($o_{jx}$)
  $gaps \leftarrow \{s \in S \mid s$ is gap **and** $s$ is big enough to insert $o_{jx}\}$ ▷ *filter gaps*
  $earliestGap \leftarrow gap \in gaps$ with $\min(gap_{start})$ ▷ *find earliest gap*
  **if** $startConstraint + processingTime < s_{1_{start}}$ **then** ▷ *insert at start if possible*
    $offer_{start} \leftarrow startConstraint$
  **else if** $earliestGap$ **is null then** ▷ *append at end if necessary*
    $offer_{start} \leftarrow \max(startConstraint, endOfSchedule)$
  **else** ▷ *else, insert at earliest gap where $o_{jx}$ can fit*
    $offer_{start} \leftarrow \max(startConstraint, earliestGap_{start})$
  return $offer$

**SVO Machine Agents** The scheduling process of `SVOMachine` agents is depicted in Algorithm 5. They base their scheduling decisions on utility values for the affected parties, choosing the option that they deem most beneficial for the common good.

The procedure builds on the FCFS method described above and additionally evaluates insertions at schedule positions currently occupied by other operations. Such insertions lead to postponements of other operations, so their utility loss has to be weighed against the utility gain for the inserted operation. Utility changes are calculated by the machine agents based on the information collected with the scheduling requests, without the need for further communication. Since all job agents share the same SVO value of $\pi/4$ (see Subsection 5.5), which is the intersection point of sine and cosine curves, machine agents can sum up individual job utilities to evaluate the overall benefit of schedule insertions from the perspective of the job collective without further modifications.

Machine agents compute their own utility from the expected idle time change in their schedule. Current machine usage is linearly mapped to SVO angles (see Equation 13 in Section 5.5), and aggregated utility values are calculated according to Equation 1. An offer is then derived by insertion at the schedule position with the highest aggregated utility value.

In summary, `SVOMachine` agents apply equal weight to job interests, while factoring in machine interest to a varying degree, depending on the current machine usage. To achieve good scheduling results, well-modeled utility functions are critical, and will be discussed in the next subsection.

**Algorithm 5** Schedule Insertion by Machine Agents (SVO)

**Precondition:** machine agent with schedule positions $S =< s_1..s_n >$, with $S = O \cup G \cup B$, and $O$: set of operations, $G$: set of gaps, $B$: set of breakdown periods

**Precondition:** machine agent has received a scheduling request for operation $o_{jx}$

**Precondition:** $startConstraint$ has been communicated by the requesting job agent

**Precondition:** $processingTime$ is known by the machine agent

**Postcondition:** $o_{jx}$ inserted into schedule, $offer$ generated

**function** OFFERSVOBASED($o_{jx}$)
- $SVO_m \leftarrow workloadFraction \cdot \frac{\pi}{2}$
- $offer \leftarrow$ offerFcfsBased($o_{jx}$) ▷ *offer without postponements*
- $P \leftarrow \{p \in O \cup G \mid startConstraint < p_{end} < offer_{end}\}$
- $M \leftarrow$ initialize map of $p$ to utility
- **for** $p \in P$ **do** ▷ *map insertion positions to utilities*
  - $start \leftarrow \max(startConstraint, p_{start})$
  - $gain \leftarrow$ utilityFunction($o_{jx}$, $p$)
  - $O_{affected} \leftarrow o \in O | o_{end} > start$
  - $loss \leftarrow \sum_{o_a \in O_{affected}}$ utilityFunction($o_a$, new position)
  - **if** $loss + gain > 0$ **then** ▷ *only consider positions beneficial for jobs*
    - $M[p] \leftarrow$ utilityFunction($SVO_m$, usageChange)
- $offer \leftarrow$ new offer from max utility in M
- return $offer$

## 5.4 Utility Functions

Utility values should reflect the goal of the agent in an appropriate way, and be easily computable. While exploring all possibilities in depth would exceed the scope of this thesis, a limited set of utility functions was evaluated in the static environment, and the most promising one chosen for dynamic environments.

We model utilities based on deviations from the ideal value, which leads to a value range from $-\infty$ to zero. The simplest deviation function is a linear comparison, and should require the least computational effort. Inspired by the least squares method, a common technique applied in optimization problems where function parameters are fitted to observed values [69], we also explored squaring deviation values. This punishes larger deviations, and could potentially improve results.

For job agent utilities, an additional modification was tested, where utilities are weighted according to the expected duration of all following operations in the job's operation sequence. This is supposed to broaden the perspective in low level scheduling decisions by favoring operations of jobs with longer remaining processing time.

In total, we arrived at four utility function variations: Linear, weighted linear, squared, and weighted squared deviation, which are formally defined in the following paragraphs.

**Job Utility** Job agents intend to minimize the end time of their last operation, which should minimize makespan of the job shop as an emergent behavior on the system level. To this end, job agents simply pick the offer with the earliest end time for each scheduling decision, but `SVOMachine` agents base those offers on utility values for job agents (see Algorithm 5) .

To calculate the ideal end time of an operation, the processing time on the machine is added to the start constraint, which corresponds to the maximum of the operation predecessor's end time and the current simulation time. The deviation between real and ideal end is calculated according to Equation 5:

$$\begin{aligned} dev_{lin} &= end_{real} - end_{ideal} \\ &= end_{real} - (startConstraint + processingTime) \end{aligned} \tag{5}$$

Deviations from the ideal end represent the absolute value of utilities associated with schedule slots. They are used to calculate utility values for schedule slot changes by subtracting new deviation values from prior values according to Equation 6:

$$u_{lin}(o_x) = dev_{lin}(prior) - dev_{lin}(new) \tag{6}$$

As a result, positive $u_{lin}$ values indicate an improvement for the corresponding job agent. As outlined in Algorithm 5, schedule changes are only considered if the sum of utility values of all affected job agents is positive, i.e. the benefit outweighs potential drawbacks.

For the squared utility function, we simply square the deviation from the ideal value to evaluate utilities:

$$dev_{squ} = (end_{real} - end_{ideal})^2 \tag{7}$$

$$u_{squ}(o_x) = dev_{squ}(prior) - dev_{squ}(new) \tag{8}$$

The weighting modification utilizes the minimal processing time $min(pt)$ of the job's remaining operations $o_{x+1}$ to $o_n$, since the actual processing time is at first unspecified, and subject to change later on:

$$u_{weighted}(o_x) = u(o_x) \cdot (1 + \sum_{o_{x+1}}^{o_n} min(pt)) \tag{9}$$

By applying the weighting term, the slope of the utility function gets steeper with longer subsequent processing time. As a result, operations that are more likely to extend the makespan when they are postponed are favored, which might positively influence makespan optimization.

**Machine Utility** Machine agents intend to maximize their utilization rate, which corresponds to minimizing their idle times. By applying SVO modifiers to the utility values, workloads are expected to balance out between machines as an emergent behavior of the system.

The ideal idle time value for a machine is zero, corresponding to a schedule where operations are seamlessly concatenated. Analogous to the job agent, we base the utility calculation on the deviation from the ideal value in Equation 10, and derive the squared function accordingly (Equation 11):

$$u_{lin}(machine) = idleTime_{prior} - idleTime_{new} \tag{10}$$

$$u_{squ}(machine) = (idleTime_{prior} - idleTime_{new})^2 \tag{11}$$

Machine utility is aggregated with utilities for all affected jobs $j_1$ to $j_n$ on the basis of Equation 1, with the reward corresponding to the utility value:

$$u_{aggr} = u_{machine} \cdot cos(\theta_{machine}) + \sum_{i=1}^{n} u_{j_i} \cdot sin(\theta_{machine}) \tag{12}$$

Unweighted job utilities are used for this aggregation to avoid distortion by different scaling of job and machine utility.

## 5.5 Social Value Orientation Settings

`SVOMachine` agents aggregate their individual utility with the utilities of all involved parties in a scheduling decision based on their social value orientation. We consider all jobs to be equally important, thus we assign them a uniform SVO angle. To achieve maximum cooperativeness, a pro-social value of $\pi/4$ was chosen. As a consequence, the job's SVO angle does not have to be communicated to machine agents, which are responsible for operation sequencing and thus for aggregating and comparing job utilities.

In contrast, when a machine agent runs near full capacity, maximizing its workload is of lower concern than if its buffer is nearly empty. To reflect that, a machine agent's SVO angle gets tied to its current workload. We chose a simple linear mapping provided with Equation 13, ranging from self-interested to altruistic attitude.

$$SVO = workload \cdot \frac{\pi}{2} \tag{13}$$

On the one hand, this mapping aims to promote workload balancing between machines, and on the other hand, it introduces a flexible weighting between job agents' and machine agents' optimization goals depending on the machine's utilization rate. For future applications, more sophisticated mapping functions could be applied when the specific conditions of a shop are known, e.g. with a centering around the average workload of a machine.

Workloads are calculated as given by Equation 14, by dividing busy time in the schedule, i.e. operations and breakdowns, by the remaining time until all operations the machine is aware of have been processed. The remaining time is derived from the $maxEnd$ of operations within the machine's own schedule as well as in its knowledge base of eligible operations that are scheduled elsewhere.

$$workload = \frac{\sum_{t=now}^{t=maxEnd} busyDuration}{maxEnd - now} \tag{14}$$

Although breakdown periods are not normally included in workload calculations, we opted for including them when calculating SVO angles, since machines are effectively blocked during that time and an exclusion would lead to a stronger preference for assignments to recently broken machines. This would not make much sense in our opinion, since the end time of the breakdown is only known due to idealized assumptions, and subject to a high degree of uncertainty in reality.

## 5.6 Reference Algorithms

Results of the `FCFS` and `CPSat` algorithms serve as references to evaluate the `SVO` algorithm's schedule quality. `FCFS` was implemented as described within Subsection 5.3, and minimizes makespan by letting job agents minimize their completion time. `CPSat` was adapted from previous work performed by Fraunhofer IPK, which utilizes OR-TOOLS' CP-SAT solver and was described in Subsection 4.2.

To address Research Goal RG4, the Python implementation of the CP-SAT wrapper was ported to Kotlin and adapted to the multi-agent environment by adding an abstract superclass `CentralizedAlgorithm` with a `CentralAgent` singleton that is activated by the environment dynamically and calls the `solve()` method of the `CPSat` class. The solver was configured to minimize the equally weighted makespan and workload objectives.

Additionally, the following changes were made: The workload objective was originally defined as minimizing the maximum machine usage deviation from the mean (Equation 15), with usage being defined as the sum of processing times on a machine.

$$\begin{aligned} workloadObjective &= \min(workloadTarget) \\ &= \min(\max(usageDeviation)) \\ &= \min(\max(|\frac{usage_{total}}{no_{machines}} - usage_{machine}|)) \end{aligned} \tag{15}$$

Since the CP-SAT solver operates with integer variables, the use of the mean value restricted the solution space to solutions where the mean usage is an integer, which was obfuscated by the dynamic typing in Python. To overcome this issue, the usage deviation function (Equation 16) was expanded by the denominator $no_{machines}$ to Equation 17:

$$usageDeviation_{original} = |\frac{usage_{total}}{no_{machines}} - usage_{machine}| \tag{16}$$

$$usageDeviation_{expanded} = |usage_{total} - usage_{machine} \cdot no_{machines}| \tag{17}$$

The function itself is not changed by expanding it, and the integer constraint is no longer a concern. To compensate for the dimension change of the function value when aggregating objectives, the makespan target value gets multiplied by the same factor (Equation 18).

$$\begin{aligned} target_{aggr} &= no_{machines} \cdot makespanTarget + workloadTarget \\ &= no_{machines} \cdot \max(op_{end}) + \max(|usage_{total} - usage_{machine} \cdot no_{machines}|) \end{aligned} \tag{18}$$

The aggregated target function is then minimized by the CP-SAT solver. To speed up the search and make sure that solutions are at least as good as the `FCFS` solution, a solution hint from static `FCFS` runs is provided to the solver. During dynamic simulations, assignments differ between `FCFS` and CP-SAT, thus only an initial hint can be provided. In all subsequent `solve()` calls, the solution of the previous iteration is modified and used as a dynamic hint: active operations are right-shifted in case of a breakdown and injected operations are appended to the eligible machine that can offer the earliest end time, as shown in Algorithm 6.

**Algorithm 6** Dynamic solution hints

**Precondition:** $S$: Solution from a previous iteration
**Postcondition:** $S'$: Solution respecting breakdown period and extended by injected operations

```
function ADAPTHINT(currentOps, breakdownLength)
    if breakdownLength > 0 then
        for op ∈ S ∩ currentOps do                ▷ right-shift current operations
            right shift op by breakdownLength
        for op ∈ currentOps \ S do                ▷ append new operations
            append op to machine schedule which can offer min end
```

The CP-SAT solver strictly adheres to the given optimization objectives, i.e. minimizing makespan and optionally minimizing differences in utilization rates of machines. As a consequence, it does not guarantee that operations are scheduled for the earliest time possible, as the makespan objective only enforces that for the very last operations. This proved to be detrimental in dynamic runs, as gaps in the schedule are not always efficiently used when new jobs are injected or breakdowns occur. Adding a minimization objective for the mean completion time of jobs would amend this, but would likely increase runtime. Therefore, we opted for re-using the `advance()` function implemented for `FCFS` to move operations into previous gaps at the end of a `solve()` call, which can solve the issue rapidly.

For the larger benchmarking instances, it was necessary to restrict the time the CP-SAT solver spends searching for solutions in order to get results in a reasonable time frame, so optimal solutions are not guaranteed in those cases. The provided solution hints ensure that at least a feasible solution is found even when the search is stopped by the timeout. The allotted time can be adjusted by a parameter when calling the program, e.g. for quick test runs, and defaults to 10 min for static runs and 60 s per iteration for dynamic runs.

### 5.7 Summary

To tackle Research Goal RG1, the discrete event simulation engine `kalasim` was used to implement static and dynamic environments. Job and machine agents were designed as components within the framework, and a negotiation algorithm was developed to enable production scheduling, as needed for Research Goals RG2 and RG3. To complement the SVO scheduling implementation, utility functions and social value orientation settings were defined. Research goal RG4 was addressed by porting an existing Python implementation of a CP-SAT wrapper for FJSP scheduling to Kotlin, and adapting it to the needs of dynamic multi-agent environments.

## 6 Evaluation

To evaluate the implemented `SVO` algorithm (Research Goal RG5–RG8), we performed benchmarking runs and compared the resulting solutions with the reference algorithms `FCFS` and `CPSat`. The program saves results of single benchmarking instances as Gantt charts in .png format and summarized in .csv format. An example for the Gantt chart output is given in Figure 14.

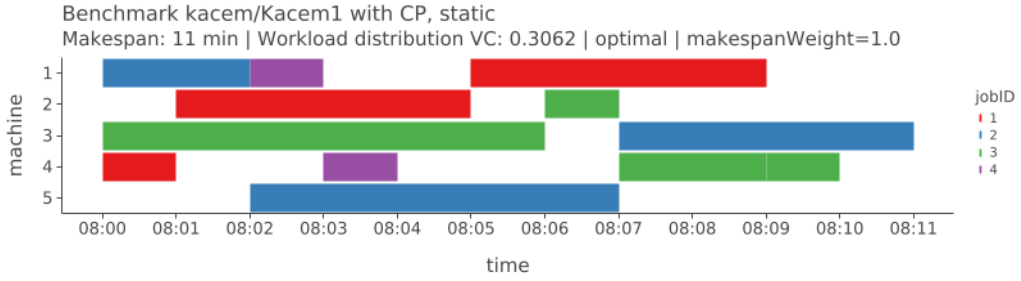



Figure 14: Gantt chart example

The .csv files contain parameters of the run, makespan, runtime, and the machines' workload variation coefficient, which serves as a measure for workload balancing. The variation coefficient $vc$ is defined as the standard deviation $\sigma$ divided by the mean value $\mu$ (Equation 19), so smaller values indicate a more equal distribution of workloads across machines.

$$vc = \frac{\sigma}{\mu} \tag{19}$$

The experiments were performed on a desktop computer with an AMD Ryzen 7 5700G CPU (8 x 3.8 GHz) and 32 GB RAM, using the benchmarking sets introduced in Subsection 4.3. Runtime results were averaged over five separate runs. The raw data has been included in the GitHub repository `https://github.com/ComfyChair/KalasimFJSP` under `results`. In the following subsections, we address Research Goals RG5 to RG8 one by one, starting with the evaluation of the static runs (RG5).

## 6.1 Static Environment

The performance under static conditions was evaluated using all 397 benchmark instances. As a preliminary step, we investigated the influence of different job utility functions on makespan minimization in order to decide on a utility function for further analysis.

### Makespan Objective

The utility functions defined in Subsection 5.4 are abbreviated here as "lin" for linear, "squ" for squared, and postfixed with an x for their extended (weighted) versions ("linx" / "squx").

Table 2: Optimally solved problems per benchmarking set. BKS refers to best known solutions from the literature.

| set | | total | optimal | % | BKS | opt ∪ BKS | % |
|---|---|---|---|---|---|---|---|
| Barnes | | 21 | 18 | 85.7 | 3 | 21 | 100.0 |
| Behnke | | 60 | 3 | 5.0 | 49 | 52 | 86.7 |
| Brandimarte | | 10 | 5 | 50.0 | 3 | 8 | 80.0 |
| Dauzere | | 18 | 2 | 11.1 | 0 | 2 | 11.1 |
| Fattahi | | 20 | 18 | 90.0 | 0 | 18 | 90.0 |
| Hurink | edata | 66 | 56 | 84.8 | 7 | 63 | 95.5 |
| | rdata | 66 | 23 | 34.8 | 22 | 45 | 68.2 |
| | sdata | 66 | 62 | 93.9 | 1 | 63 | 95.5 |
| | vdata | 66 | 22 | 33.3 | 22 | 44 | 66.7 |
| Kacem | | 4 | 4 | 100.0 | 0 | 4 | 100.0 |
| total | | 397 | 213 | 53.7 | 107 | 320 | 80.6 |

In order to eliminate distortions by simultaneous usage optimization, a set of `SVO` runs was performed without machine utility aggregation and compared to results of `CPSat` that were also limited to makespan optimization. As results of the distributed algorithms partially outperformed `CPSat`, the constraint programming algorithm was at first provided solution hints of `FCFS` runs in order to guarantee improved solutions. Since `CPSat` still undershot `SVO` results in some cases, hints

generated from SVO-linx runs were finally employed. 81 % of all instances could be solved to optimality or to the best known solution from the literature [62], [70]–[73] within a time limit of 10 min. Table 2 breaks this number down by benchmarking set, and detailed results are listed in Table A.1–A.7. Best known solution (BKS) means that a solution has not been proven to be optimal, since the respective instances are very hard to solve by exact methods, but so far, no better solutions has been reported.

Figure 15 gives an overview over the makespan achieved by FCFS and SVO scheduling in relation to the constraint programming reference. Divergence of makespan from CPSat solutions was averaged over benchmarking sets and plotted as relative errors. Smaller bars indicate better approximation of the best known solutions, and error bars represent the standard error of the mean $\sigma_s$, which is defined as the standard deviation $\sigma$ divided by the square root of the sample size $n$:

$$\sigma_s = \frac{\sigma}{\sqrt{n}} \tag{20}$$

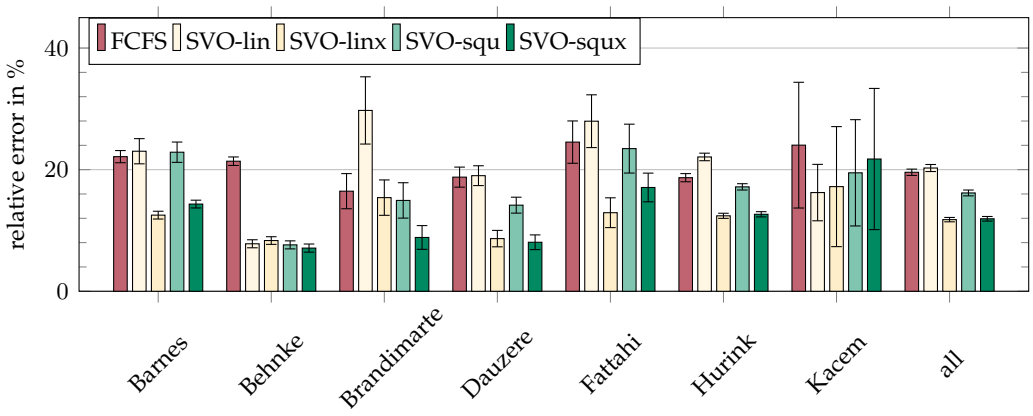



Figure 15: Mean makespan of distributed algorithms compared to CPSat results. Error bars indicate the standard error of the mean $\sigma_s$.

The results for different utility functions are not entirely consistent across benchmarking sets, reflecting the differences in difficulty level and challenge type they pose. For example, Kacem's set contains only four problems with very short optimal makespan and large differences in processing times between machines, so small deviations from the ideal packing lead to large relative errors, and small differences in solution quality of single instances result in large standard errors of the mean for the set. Behnke's set on the other hand contains large problems constructed around so called work centers of identical machines, which are supposed to be more practice-oriented than other benchmarking sets [62]. The design results in dense packing of operations in two work centers, that always process the first and last operation of a job, respectively, and a more loose packing in the others machine pools. The best re-

sults for SVO scheduling were obtained with this problem type, with relative errors ranging from 6.2 % to 8.3 % between utility functions.

Despite the variability between benchmarking sets, an overall advantage of the weighted utility functions linx and squx can be derived from Figure 15. The linear utility function lin did not produce better results than FCFS scheduling on average. It seems likely that this simple function is just not powerful enough to have a large impact on makespan, since it neither punishes large deviations harshly like squ, nor does it prefer operations with longer subsequent operations like the weighted functions. For the Brandimarte set, were the job encoding order already yields relatively good FCFS solutions, SVO-lin results were even substantially worse. The best solutions were obtained with linx and squx, with overall makespan errors of about 12 %. These weighted utility functions also exhibited the least inter-instance variability, as shown in the statistical overview in Table 3, indicating the best stability for different problem types.

Table 3: Makespan statistics over all instances.

| | FCFS | SVO | | | |
|---|---|---|---|---|---|
| | | lin | linx | squ | squx |
| mean relative error | 19.57 | 20.27 | 11.78 | 16.17 | 11.92 |
| variance | 106.71 | 134.81 | 50.95 | 92.97 | 56.04 |
| std. dev. $\sigma$ | 10.33 | 11.61 | 7.14 | 9.64 | 7.49 |
| std. error $\sigma_s$ | 0.52 | 0.58 | 0.36 | 0.48 | 0.38 |

Differences between utility functions were further evaluated by performing pairwise two-sided t-tests using Libre Office (Table 4). This statistical test is commonly used to test for significance of differences between two sets of measurements. Here, relative errors were compared on a per instance basis, with pairwise comparisons of different utility function to each other and to FCFS scheduling. The statistical analysis confirms that SVO-lin results are on par with FCFS with a p-value of 0.33, while significant differences are seen between FCFS and SVO scheduling with the other utility functions. linx and squx, the two functions with the best approximation of CPSat results, are very similar to each other with a p-value of 0.6.

Table 4: P-values of two-sided pairwise t-tests comparing utility functions. The confidence level was set to 95 %. Values indicating a significant difference are written in bold face.

| tested against | lin | linx | squ | squx |
|---|---|---|---|---|
| FCFS | 0.33 | $\mathbf{4.92{\cdot}10^{-48}}$ | $\mathbf{6.03{\cdot}10^{-9}}$ | $\mathbf{2.75{\cdot}10^{-48}}$ |
| lin | | $\mathbf{1.06{\cdot}10^{-43}}$ | $\mathbf{1.78{\cdot}10^{-18}}$ | $\mathbf{5.63{\cdot}10^{-43}}$ |
| linx | | | $\mathbf{3.33{\cdot}10^{-22}}$ | 0.60 |
| squ | | | | $\mathbf{6.45{\cdot}10^{-25}}$ |

Upon inspection of the instances with the largest relative errors, we noticed that differences between processing times of operations were not properly considered in our utility functions so far, which became problematic when differences were large, like in Kacem's instances. Therefore, two additional utility functions were included in the experiments, linx+ and squx+, where the minimal processing time of the operation itself is included in the weighting term:

$$u_{weighted}(o_x) = u(o_x) \cdot \sum_{o_x}^{o_n} min(pt) \tag{21}$$

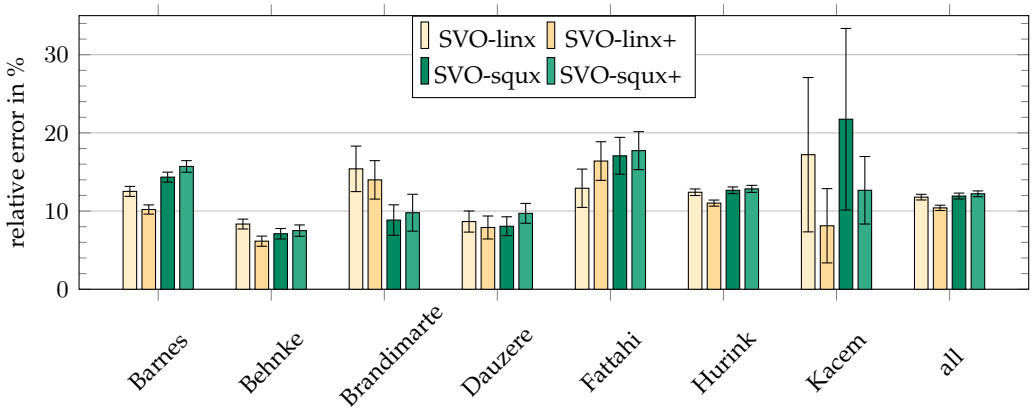



Figure 16: Makespan analysis of the modified utility functions linx+ and squx+.

Table 5: Makespan errors of `FCFS` and `SVO` with different utility functions in percent. The best result per benchmarking set is marked in bold face.

| rel. errors | FCFS | SVO | | | | | |
|---|---|---|---|---|---|---|---|
| in % | | lin | linx | linx+ | squ | squx | squx+ |
| Barnes | 22.14 | 23.03 | 12.52 | **10.20** | 22.87 | 14.34 | 15.71 |
| Behnke | 21.39 | 7.81 | 8.34 | **6.15** | 7.63 | 7.10 | 7.50 |
| Brandimarte | 16.46 | 29.74 | 15.40 | 13.99 | 14.94 | **8.85** | 9.79 |
| Dauzere | 18.77 | 19.01 | 8.66 | **7.90** | 14.16 | 8.06 | 9.71 |
| Fattahi | 24.53 | 27.97 | **12.92** | 16.40 | 23.46 | 17.07 | 17.73 |
| Hurink | 18.68 | 22.09 | 12.41 | **11.02** | 17.17 | 12.66 | 12.84 |
| Kacem | 24.03 | 16.23 | 17.21 | **8.12** | 19.48 | 21.75 | 12.66 |
| all | 19.57 | 20.27 | 11.78 | **10.41** | 16.17 | 11.92 | 12.21 |
| rel. to FCFS [%] | | +3.6 | −39.8 | −46.8 | −17.4 | −39.1 | −37.6 |

Figure 16 compares the modified utility functions with their unmodified counterparts. While both result in an improvement for the Kacem set, only the linear

function performs better on average, with a mean relative error of 10.4 %, an improvement of about 47 % compared to FCFS scheduling. However, two out of seven benchmarking sets obtained better results with other utility functions, as can be seen in Table 5, which summarizes relative errors for all tested utility functions.

As we are striving for an algorithm suitable for dynamic scheduling, we additionally analyzed utility functions for computational efficiency. Table 6 gives a first impression on the runtime differences between the algorithms CPSat, FCFS and SVO, and a comparison between the most competitive utility functions linx, squx, and linx+. Only a single run was performed for CPSat due to time restrictions, but for FCFS and SVO, the average of five simulation runs is listed.

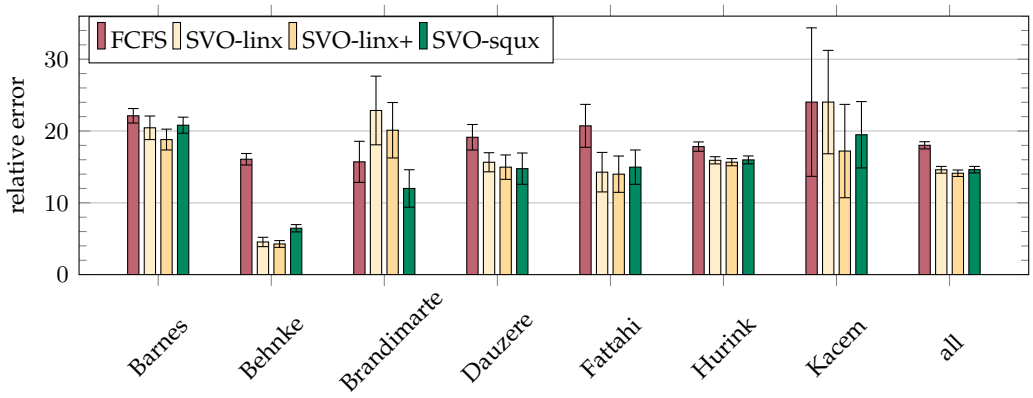
Table 6: Runtime comparison for utility functions. FCFS and SVO values refer to the average of five simulations.

| | CPSat | FCFS | SVO | | |
|---|---|---|---|---|---|
| | | | linx | squx | linx+ |
| mean [min] | 1944 | 0.405 | 19.76 | 19.25 | 17.89 |
| std. dev. $\sigma$ | | 0.005 | 0.68 | 0.15 | 0.18 |
| std. error $\sigma_s$ | | 0.002 | 0.30 | 0.07 | 0.08 |

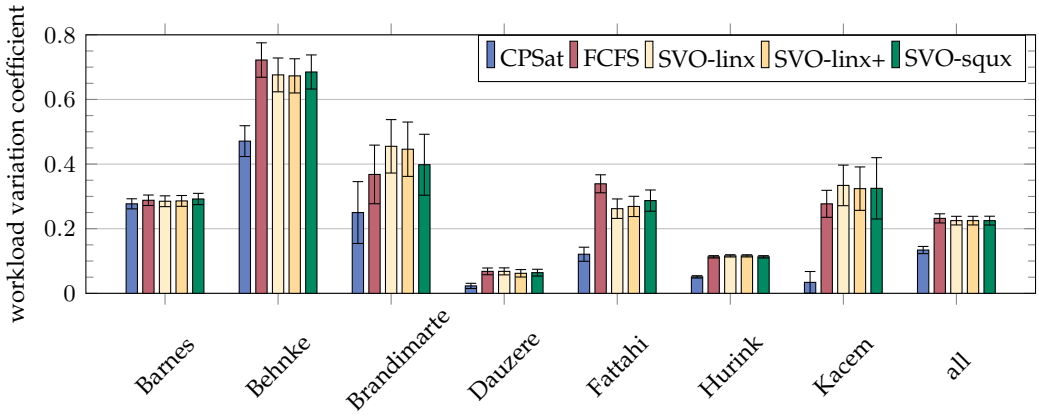
As expected, CPSat is by far the most time-consuming method, due to the combinatorial nature of the FJSP and the size of some of the benchmark instances. The given sum over all the benchmark runs of over 32 hours is already restricted by the employed timeout of 10 minutes per problem. Although some of the feasible solutions matched or even surpassed the best known solution from the literature (column BKS in Table 2), proving optimality for all the non-optimal instances would take significantly longer, despite the speed-up by giving solution hints. The FCFS algorithm on the other hand took only 24 seconds for all benchmarks on our setup, but its results were about 20 % worse than CPSat on average, as was shown in Table 5. SVO simulations finished after 18–20 minutes for all combined sets, with slightly better performance with linx+. We continued our static evaluations with these three utility functions in order to assess their effectiveness for the combined optimization goal.

### Combined Objectives

For a full evaluation of the SVO based algorithm under static conditions (Research Goal RG5), new benchmarking runs of CPSat and SVO were performed that included the workload balancing goal. As FCFS scheduling does not explicitly optimize workload balancing, its previous simulation runs were re-used as a reference. The weight between makespan and workload balancing objective was set to 0.5 in case of CPSat, while for the SVO algorithm, a weight parameter below 1 was given to enable machine goals, but the actually applied weight during simulations varies depending on the current machine usage.




(a) Makespan errors for combined optimization objectives.




(b) Workload variation coefficient as a measure of workload balancing.

Figure 17: Static results with combined objectives. (a): relative makespan error, (b): workload balancing comparison. Error bars represent standard errors of grouped results.

Figure 17 illustrates solution quality in comparison to the reference algorithms. With the inclusion of the workload objective, the relative error to `CPSat` results has increased significantly, and is only about 19–22 % better than `FCFS` (Figure 17a), while it was 39–47 % when optimizing the single makespan objective (Table 5). The second optimization goal of balancing workload across machines is visualized in Figure 17b. Workload variation coefficients did not improve significantly compared to `FCFS` scheduling (p-value of 0.69), indicating that the consideration of idle time changes in operation sequencing is insufficient to achieve the desired workload balancing effect. With job agents deciding on assignments of operations to machines solely based on their own completion time, machine objectives are not adequately considered, while the makespan objective still takes a massive hit due to subpar sequencing choices.

### Workload Optimization Version 2

Given the negligible positive effect and large drawback of this first attempt at workload optimization, a modified version was implemented, where assignment decisions made by job agents include machine utility.

Idle times are no longer taken into account when machines optimize their internal schedule, but instead, machines report their utility gain for scheduling an operation when sending offers to job agents. Machine utility is calculated according to Equation 22, by multiplying the operation's processing time $pt$ with the SVO weighting term $cos(\theta)$. The minimal processing time of an operation $min(pt_{o_x})$ is used instead of the actual processing time on the machine to avoid giving preference to machines with longer processing times.

$$u_{machine}(o_x) = min(pt_{o_x}) \cdot cos(\theta_{machine}) \tag{22}$$

Job agents aggregate their own utility with the reported machine utility according to Equation 23, with $dev$ referring to the linear or squared deviation function without weighting as given in Subsection 5.4.

$$u_{aggr}(offer) = u_{machine} - cos(\theta_{job}) \cdot dev(offer) \tag{23}$$

`FcfsJob` was subclassed to `SVOJob` to implement this algorithmic change: Whenever job agents make a scheduling decision, they pick the offer with the highest utility value instead of the offer with the earliest end time (line 8 in Algorithm 7).

**Algorithm 7** Operation Scheduling by `SVOJob` Agents, v2

**Precondition:** job agent $j$ with operation sequence $O_j =< o_1..o_n >$ has scheduled all operations preceding $o_x$

**Precondition:** eligible machines are known to the job agent
**Postcondition:** operation $o_x$ is scheduled on an eligible machine

```
1:  function SCHEDULE(o_x)
2:      startConstraint ← if x = 1 now else end_{o_{x-1}}
3:      M_x ← eligible machines for o_x
4:      for m ∈ M_x do
5:          send scheduling request to machine agent m with startConstraint
6:      collect responses in offers (non-blocking)
7:      if activated and offers contains valid offers then
8:          acceptedOffer ← o ∈ offers with max(u_aggr)
9:          send Decision in reply to all o ∈ offers
10:         if size(M_x) > size(offers) then ▷ also inform other agents about new best end
11:             send RequestUpdate to all agents in M_x
```

Since the change in assignment logic increased schedule instability, with circular reordering of machine schedules, and operation assignments switching endlessly between machines in some scenarios, additional changes became necessary:

- Machines choose the best schedule reordering move among all possibilities, instead of the best move for the operation with the highest deviation value like in version 1.
- Schedule optimization is performed earlier in the machine's process, before new requests are processed or switch offers are considered.
- All time slot insertions and removals are immediately followed by an additional schedule optimization step without pushbacks.
- To eliminate the influence of fluctuating start constraints, an operation's ideal end time was re-defined as the hypothetical end time if the whole operation sequence was processed on the fastest machines without any delays, instead of just the operation itself. This is calculated recursively by Equation 24:

$$op_{optimalEnd} = predecessor_{optimalEnd} + \min(processingTime) \tag{24}$$

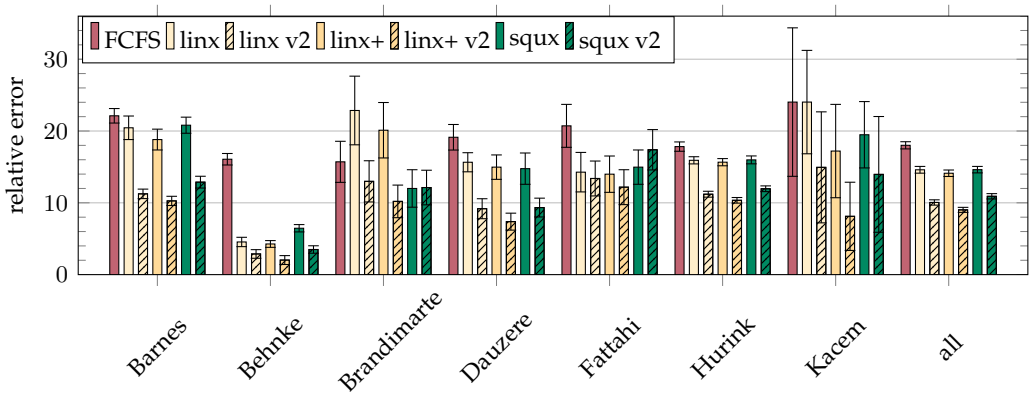
- To prevent cyclic assignment switching when machines reach full capacity, machine SVO angles were scaled down to a range of 0° - 77.4° by modifying the mapping function to Equation 25:

$$\theta = usage \cdot 0.86 \cdot \frac{\pi}{2} \tag{25}$$

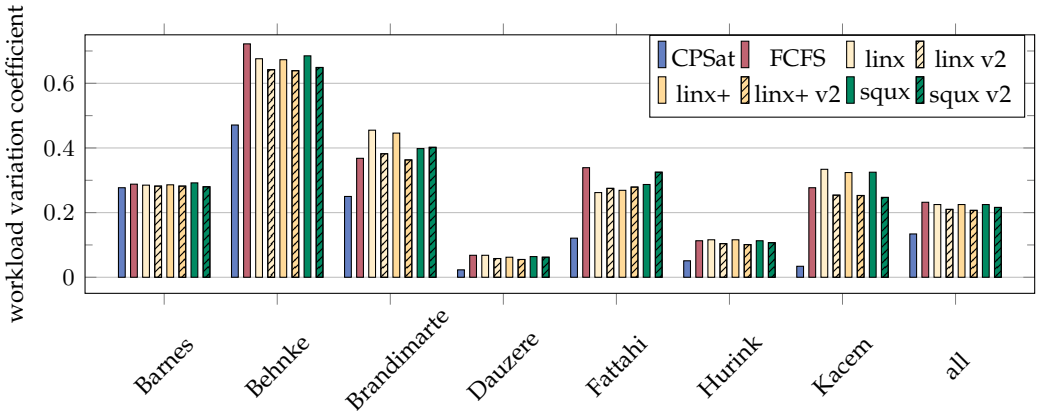
Table 7: Single makespan objective results, comparing algorithm v1 to v2.

| rel. errors | FCFS | SVO v1 | | | SVO v2 | | |
|---|---|---|---|---|---|---|---|
| in % | | linx | squx | linx+ | linx | squx | linx+ |
| Barnes | 22.1 | 12.5 | 14.3 | 10.2 | 11.5 | 12.8 | 10.8 |
| Behnke | 21.4 | 8.3 | 7.1 | 6.2 | 7.8 | 8.5 | 7.0 |
| Brandimarte | 16.5 | 15.4 | 8.9 | 14.0 | 16.8 | 15.1 | 11.7 |
| Dauzere | 18.8 | 8.7 | 8.1 | 7.9 | 8.6 | 9.0 | 7.0 |
| Fattahi | 24.5 | 12.9 | 17.1 | 16.4 | 16.2 | 22.3 | 14.4 |
| Hurink | 18.7 | 12.4 | 12.7 | 11.0 | 12.3 | 12.6 | 11.3 |
| Kacem | 24.0 | 17.2 | 21.8 | 8.1 | 25.3 | 14.0 | 8.1 |
| all | 19.6 | 11.8 | 11.9 | 10.4 | 11.8 | 12.4 | 10.6 |
| rel. to v1 [%] | | | | | +0.4 | +4.0 | +1.5 |

To assess whether these changes influence the effectiveness of the algorithm, we compare relative errors of single makespan objective solutions between the two algorithm versions in Table 7. While there are some bigger changes in relative errors on the level of single benchmarking sets, the average stayed roughly the same, with algorithm v2 obtaining 0.4 – 4.0 % worse solutions, depending on the utility function.




(a) Makespan errors with modified workload optimization.




(b) Workload variation coefficient with modified workload optimization.

Figure 18: Static results with algorithm v2. (a): relative makespan error, (b): workload balancing comparison.

Results of benchmarking runs with combined objectives are visualized in Figure 18 and numerically compared to the prior version in Table 8 (makespan) and Table 9 (workload). The modified algorithm version performed consistently better in regards to both optimization goals. The best results were again obtained using deviation function linx+, with an average makespan error of 9.0 % compared to `CPSat`. Interestingly, the modified algorithm version reached even lower makespan errors in the combined runs than in the single objective runs. It seems that this way of encouraging a more balanced spread of operations across machines also promotes approaching the makespan goal instead of hindering it.

Workload variation coefficients improved by a smaller margin, with the best results also being obtained by linx+ v2. Still, variation coefficients deviated much more from the `CPSat` reference than makespan results, with a 54.6 % deviation as opposed to the 9.0 % makespan error. This can be attributed to the way we balance

Table 8: Makespan with combined objectives, comparing algorithm v1 to v2.

| rel. errors | FCFS | linx | | squx | | linx+ | |
|---|---|---|---|---|---|---|---|
| in % | | v1 | v2 | v1 | v2 | v1 | v2 |
| Barnes | 22.1 | 20.5 | 11.3 | 20.8 | 12.9 | 18.8 | 10.3 |
| Behnke | 16.1 | 4.6 | 2.9 | 6.5 | 3.5 | 4.3 | 2.0 |
| Brandimarte | 15.7 | 22.9 | 13.0 | 12.0 | 12.1 | 20.1 | 10.2 |
| Dauzere | 19.1 | 15.7 | 9.2 | 14.8 | 9.3 | 15.0 | 7.4 |
| Fattahi | 20.7 | 14.3 | 13.4 | 15.0 | 17.4 | 14.0 | 12.2 |
| Hurink | 17.8 | 15.9 | 11.2 | 16.0 | 12.0 | 15.7 | 10.4 |
| Kacem | 24.0 | 24.0 | 14.9 | 19.5 | 14.0 | 17.2 | 8.1 |
| all | 18.0 | 14.6 | 10.1 | 14.6 | 10.9 | 14.1 | 9.0 |
| rel. to FCFS [%] | | −18.9 | −44.2 | −18.8 | −39.4 | −21.6 | −49.9 |

Table 9: Workload variation coefficient, algorithm v1 compared to v2.

| | FCFS | linx | | squx | | linx+ | |
|---|---|---|---|---|---|---|---|
| benchmark | | v1 | v2 | v1 | v2 | v1 | v2 |
| Barnes | 0.288 | 0.285 | 0.282 | 0.292 | 0.280 | 0.286 | 0.282 |
| Behnke | 0.722 | 0.676 | 0.642 | 0.685 | 0.649 | 0.673 | 0.639 |
| Brandimarte | 0.368 | 0.455 | 0.382 | 0.398 | 0.402 | 0.446 | 0.363 |
| Dauzere | 0.068 | 0.068 | 0.058 | 0.064 | 0.062 | 0.062 | 0.055 |
| Fattahi | 0.339 | 0.262 | 0.275 | 0.287 | 0.325 | 0.269 | 0.279 |
| Hurink | 0.113 | 0.116 | 0.104 | 0.113 | 0.107 | 0.116 | 0.101 |
| Kacem | 0.277 | 0.334 | 0.254 | 0.325 | 0.247 | 0.324 | 0.253 |
| all | 0.232 | 0.225 | 0.210 | 0.225 | 0.216 | 0.225 | 0.207 |
| rel. to CP-SAT [%] | +73.2 | +68.4 | +56.8 | +67.8 | +61.3 | +68.0 | +54.6 |

job and machine goals in the aggregation function that job agents employ when deciding between offers from different machine agents. While job agents always use a factor of $cos(45°) \approx 0.7$ for their own utility, machine utility gets weighted by factors ranging from $cos(0.86 \cdot usage \cdot 90°) \approx 0.2 - 1$, depending on their usage. The variability of the machine utility weighting factor as a mechanism to promote assignments on machines with low usage results as a side effect in an increased weighting of job over machine utilities for usage levels above 0.58. This effect could be seen as a dynamic adjustment of the weighting between makespan and workload balancing, with increasing weight being put on the makespan objective when machine workloads increase. Since most benchmarking scenarios do results in higher usage levels than 0.58 for most of their machines, an increased weight on the makespan objective is to be expected.

To evaluate efficiency changes introduced by the modified workload optimization strategy, the time needed to complete benchmarking runs using algorithm v2 is averaged over five separate runs in Table 10. With the modified workload optimization, runtime has decreased drastically from approximately 25 min to 2 min for all benchmarking sets, most likely due to inefficiencies in how idle time was calculated in v1. The speed advantage over `CPSat` increased to three orders of magnitude, and it is now only five times slower than `FCFS` scheduling. The longest problem instance (Behnke59) took 6.8 s to solve, which seems still adequate for dynamic applications.

Table 10: Runtime comparison for algorithm v2 under static conditions.

| | | CPSat | FCFS | linx | squx | linx+ |
|---|---|---|---|---|---|---|
| single goal | [min] | 1944.41 | 0.41 | 1.90 | 1.82 | 1.91 |
| dual goal | [min] | 2169.13 | | 2.01 | 1.85 | 2.00 |
| longest instance | [s] | > 600 | 2.20 | 6.10 | 6.10 | 6.80 |

#### Summary

Social value orientation was shown to be beneficial in static scheduling compared to the base algorithm, although the size of the effect varied across the evaluated benchmarking sets. The best results were obtained using the modified algorithm v2 and the weighted utility function linx+. This combination led to averaged makespan errors between 10.6 % (single makespan goal) and 9.0 % (dual goal) relative to the exact method. Workload balancing results deviated more severely from `CPSat` with 54.6 % higher variation coefficients, indicating that our SVO implementation favors the makespan objective. Runtime analysis showed that algorithm v2 is highly efficient and therefore applicable for dynamic environments.

### 6.2 Dynamic Job Arrivals

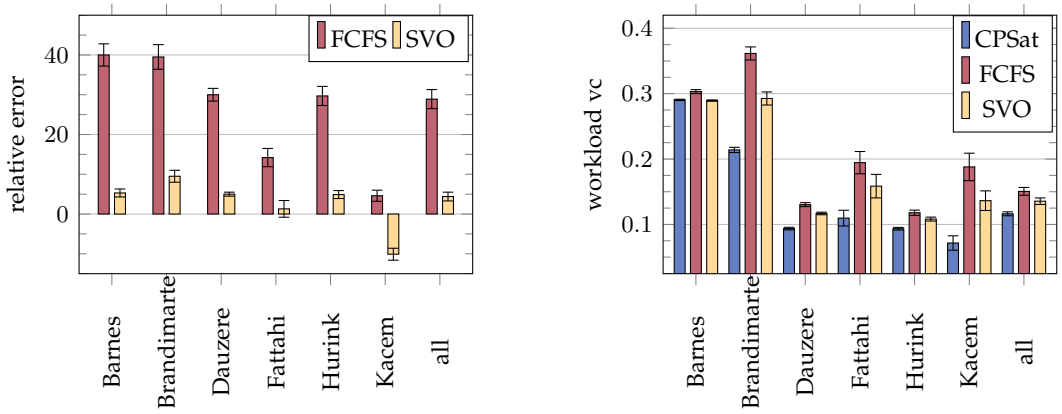
For all further analysis, the most effective SVO setup was employed, namely workload balancing version 2 with utility function linx+. Research Goal RG6 was addressed by including dynamic job arrivals into the simulation environment, with problem instances derived from static benchmarking data as described in Subsection 5.1.

In the `CPSat` wrapper, each job injection triggers a new `solve()` call. Along with the newly added job, all prior operations that are not in PROCESSING or DONE state are included in the optimization. To keep execution times reasonable, `CPSat` timeouts were set to 60 seconds per solving increment.

A subset of 191 benchmark instances for which optimal reference results were obtained during the static runs was chosen for dynamic evaluation (marked by a plus sign in the appendix, Tables A.1 – A.7). Those easier problems were expected to still

yield acceptable reference data, while the results of harder instances would deteriorate more severely with the stricter time limit. This excluded Behnke's and most of Dauzère's benchmarking set, as both contain problems which were too difficult to solve within reasonable time.

Results were averaged over five separate runs with different randomization seeds, which influence the injection order of jobs. Relative makespan errors and workload variation coefficients for the medium workload setting are displayed in Figure 19, with error bars indicating standard errors of runs with different seeds. Deviations from the `CPSat` reference are summarized and compared with static results in Table 11.




(a) Makespan errors relative to `CPSat`.

(b) Workload variation coefficients.

Figure 19: Dynamic job arrivals with medium workload. Error bars represent standard errors between runs with different seeds.

Table 11: Dynamic job solutions compared to static runs of the instance subset. Deviations from the `CPSat` reference are given in %.

| environment | makespan error | | workload error | |
|---|---|---|---|---|
| | FCFS | SVO | FCFS | SVO |
| static | 20.4 | 15.2 | 56.5 | 48.4 |
| dynamic jobs | 28.9 | 4.4 | 29.5 | 16.6 |

Solutions of the `FCFS` reference algorithm deteriorated more severely than the `CPSat` reference under dynamic conditions: The average makespan error of 20.4 % for the evaluated subset of instances under static conditions increased to 28.9 %. This might be due to amplification effects of suboptimal scheduling choices, as the FCFS rule does not allow much reordering in reaction to dynamic changes.

`SVO` scheduling on the other hand achieved significantly better solutions relative to `CPSat` under dynamic conditions, with makespan errors decreasing from 15.2 %

to 4.4 %, and workload variation coefficients approximately halfway between both reference solutions, at a 16.6 % difference from `CPSat` results instead of 48.4 % as under static conditions. This decrease can to some extend be ascribed to worse `CPSat` solutions due to tighter time restrictions, but was also observed with small instances that were not terminated early, e.g. Kacem1–3. In total, five instances were solved to equal and 19 to better solutions than calculated by `CPSat`, and in only five of them, the solution search was aborted due to solver timeouts (Table A.8 in Appendix 7). Another explanation could be that the different weighting between makespan and workload balancing objectives leads to superior makespan results of SVO scheduling in some cases. Like we observed with `FCFS` solutions, slightly worse scheduling decisions might get amplified in the dynamic setting, thus explaining the difference to static results.

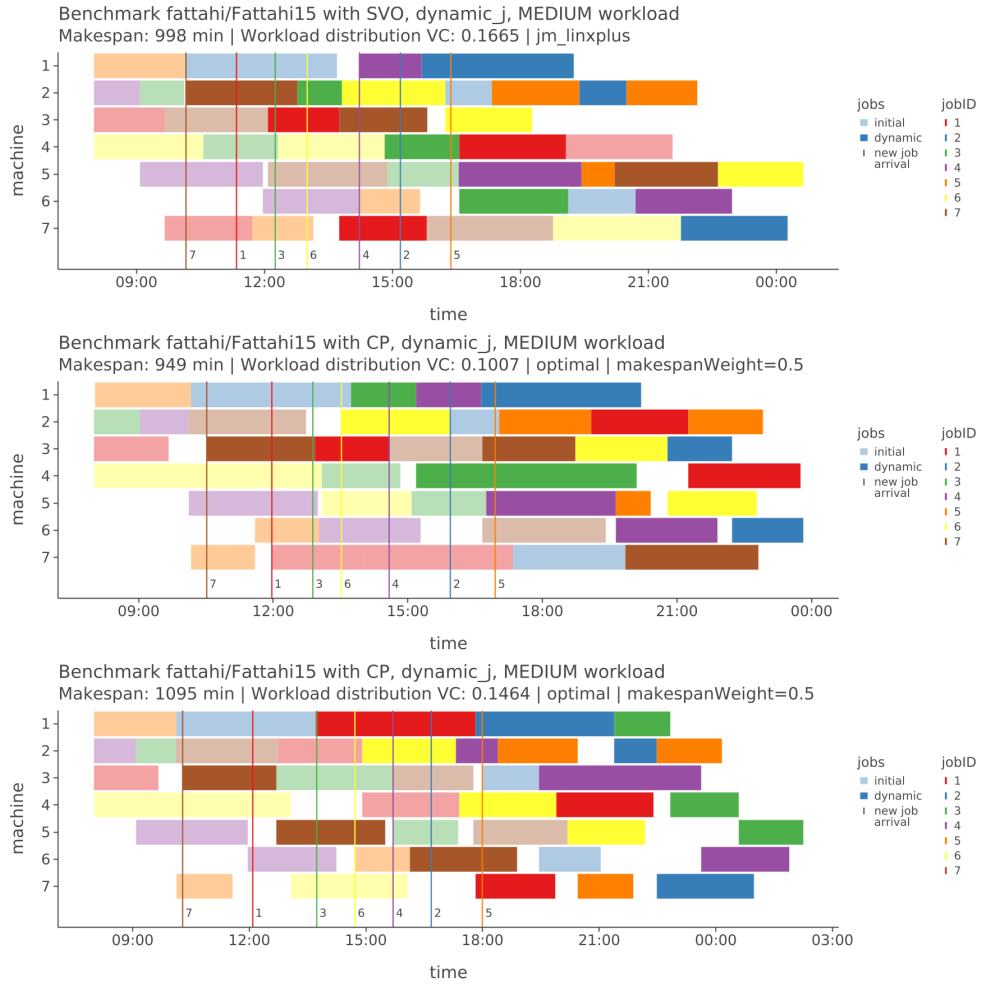



Figure 20: Gantt charts of Fattahi15 solutions with dynamic job arrivals and the same seed. Upper panel: `SVO` algorithm. Middle panel: `CPSat` - low makespan, lower panel: `CPSat` - high makespan.

OR-TOOLS' CP-SAT solver does not differentiate between solutions with the same objective values, and it has a random element integrated into its search process. This can lead to different ordering of operations in solutions that are equal in regard to the objective when optimizing multiple goals at once or if a problem instance contains sufficient flexibility. These subtle differences can cause diverging objective results in later solver increments of dynamic simulations, as shown in Figure 20. The middle and lower panel are solutions of two separate `CPSat` runs with the same seed, resulting in a roughly 15 % higher makespan in the second case, although the schedule is almost identical in the beginning. This discrepancy is most likely aggravated by the job injections being tied to completed workloads and would probably be lower if we removed that coupling. However, the fact that `SVO` scheduling undercut `CPSat` solutions in roughly 10 % of the problem instances indicates that the heuristics applied in the `SVO` algorithm can be extremely effective. It would be interesting to investigate this finding further by collecting more data of repeated `CPSat` runs, and examining the factors for deteriorating makespan results in the dynamic setting, but this was considered beyond the scope of this thesis.

Computational times of the different algorithms are compared in Table 12. The `SVO` algorithm was in the nanosecond to second range per instance, with an average of 0.6 s and the hardest problem taking 7.0 s, while `CPSat` took 47.7 s on average, with runtimes ranging from 6.8 milliseconds to 13.1 minutes for the hardest instance. Although the median value of `CPSat` runtimes is much lower than its mean value, the exact method is still more than 10 times slower than `SVO` scheduling, although we restricted the evaluated benchmarks to the easier instances, and the allowed time per solving increment. In summary, the spread between easy and hard instances is much lower with the `SVO` algorithm, in line with our goal to reduce the time complexity for handling the combinatorial FJSP.

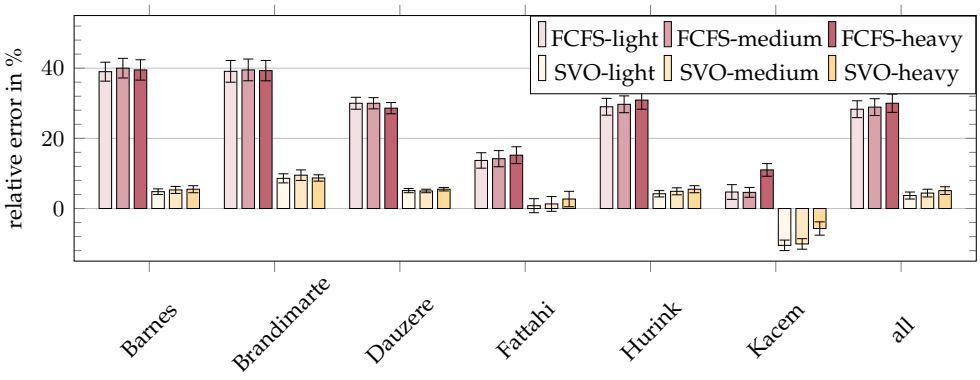
Table 12: Runtime comparison under dynamic job conditions.

| | | CPSat | FCFS | SVO |
|---|---|---|---|---|
| mean | [ms] | 47,654.5 | 34.9 | 610.4 |
| median | [ms] | 1,290.5 | 20.5 | 108.6 |
| max | [s] | 785.5 | 0.3 | 7.0 |

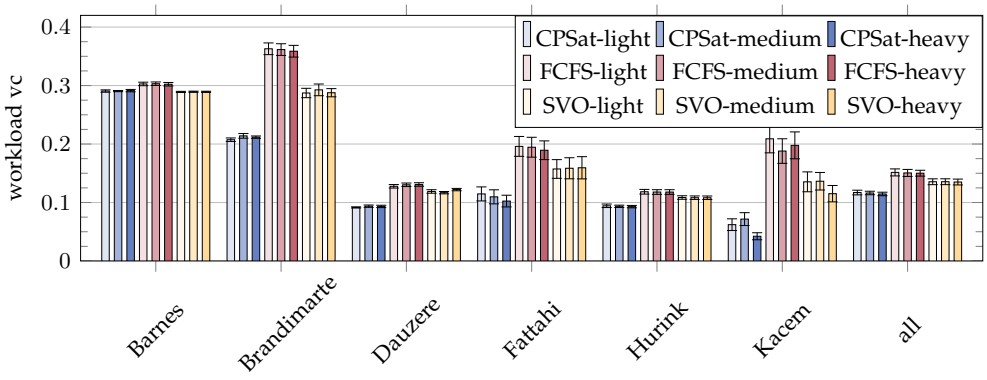
With the reduced computational effort and reasonably close approximation of reference results, the algorithm seems to be suitable for realistically sized instances in dynamic applications.

### Workload Influence

To address Research Goal RG7, dynamic job arrivals were simulated under light (70%), medium (77%) and heavy (85%) shop workloads. The effects on relative makespan errors and workload variation coefficients are visualized in Figure 21.




(a) Makespan errors relative to `CPSat`.




(b) Workload variation coefficient.

Figure 21: Dynamic runs under light, medium and heavy total load.

All algorithms were relatively stable in the evaluated workload range, but a tendency towards higher makespan errors under increasing workloads can be observed. This can be explained to some extend by the capping of `CPSat`'s optimization potential when fewer operations are available in the system to shuffle around, so differences get emphasized when the algorithm nears its full potential. On the other hand, a comparison of workload influence on raw makespan results in Table 13 indicates that `SVO` scheduling is less affected by workload than the reference algorithms. Pairwise t-tests on the instance level (Table 14) confirm that the `SVO` algorithm's makespan results are much more stable. `FCFS`'s p-values are already higher than `CPSat`'s, indicating a lower susceptibility to shop workloads. The lowest influence is observed in the `SVO` solutions, where medium and heavy workload results are no longer significantly different with a p-value of 0.063.

It seemed likely that the variable weight between makespan and workload balancing goals introduced by utilization-dependent machine SVO angles stabilizes makespan results under different workloads. To support this notion, the simulation was re-run with a makespan weight of 1.0. Surprisingly, excluding the workload

balancing goal led to even lower differences of SVO's makespan results under different workloads with p-values of 0.7 and 0.2, respectively (Table A.9 and A.10 in Appendix 7).

Table 13: Workload influence on mean makespan.

| | mean makespan | | | drop [%] | | | rel. error [%] | |
|---|---|---|---|---|---|---|---|---|
| workload | CPSat | FCFS | SVO | CPSat | FCFS | SVO | FCFS | SVO |
| light | 2,723 | 3,394 | 2,844 | | | | 28.3 | 3.7 |
| medium | 2,697 | 3,377 | 2,835 | −1.0 | −0.5 | −0.3 | 28.9 | 4.4 |
| heavy | 2,677 | 3,366 | 2,827 | −0.7 | −0.3 | −0.3 | 30.0 | 5.1 |

Table 14: Pairwise t-tests of makespans under varying shop loads. P-values indicating a significant difference are marked in bold.

| | CPSat | | FCFS | | SVO | |
|---|---|---|---|---|---|---|
| workload | light | heavy | light | heavy | light | heavy |
| medium | $\mathbf{7.6 \cdot 10^{-14}}$ | $\mathbf{2.7 \cdot 10^{-11}}$ | $\mathbf{4.0 \cdot 10^{-6}}$ | $\mathbf{7.5 \cdot 10^{-4}}$ | **0.005** | 0.063 |

Workload balancing results on the other hand were barely influenced by shop workload levels. According to pairwise t-tests (Table 15), only one significant deviation was detected on the instance level, between heavy and medium workloads in CPSat. This stems mostly from differences in the smaller instances in Kacem's and Fattahi's set, which also experience a higher variability with different seeds, and is no longer significant when looking at mean values of benchmarking sets (p-value of 0.23).

Table 15: Pairwise t-tests of workload variation coefficients under varying shop loads. P-values indicating a significant difference are marked in bold.

| | CPSat | | FCFS | | SVO | |
|---|---|---|---|---|---|---|
| workload | light | heavy | light | heavy | light | heavy |
| medium | 0.348 | **0.010** | 0.104 | 0.307 | 0.932 | 0.502 |

Taken together, shop workloads barely affected scheduling results within the evaluated range. Workload balancing was not significantly altered for any algorithm, but makespan results of the CPSat reference improved marginally under heavier workloads, resulting in a slight increase in relative errors of the less affected FCFS and SVO solutions. Makespan results of the SVO algorithm were the least affected by different workloads. Although the reason to this remains elusive, it underlines its suitability for dynamic environments.

## 6.3 Dynamic Machine Breakdowns

In order to approach Research Goal RG8, the simulation environment was extended to machine breakdown events as described under Subsection 5.1. The mean distance between breakdowns was set to the minimal makespan, aiming for one breakdown per machine per run. Separate simulations were performed with and without dynamic job arrivals, for which a medium shop workload was chosen. As before, five runs with different randomization seeds were averaged per problem instance. Here, seeds influence breakdown distances and lengths in addition to job injection orders. We obtained a mean number of breakdowns per machine of 1.3 with and 1.4 without dynamic job arrivals, with values ranging from 0.4 to 2.9 and 0.3 to 3.6, respectively. Besides tight schedule packing, fewer breakdowns per machine can occur due to the non-preemptiveness condition, i.e. operations in processing are not interrupted, which can lead to imminent breakdowns not being recorded at the end of a schedule.

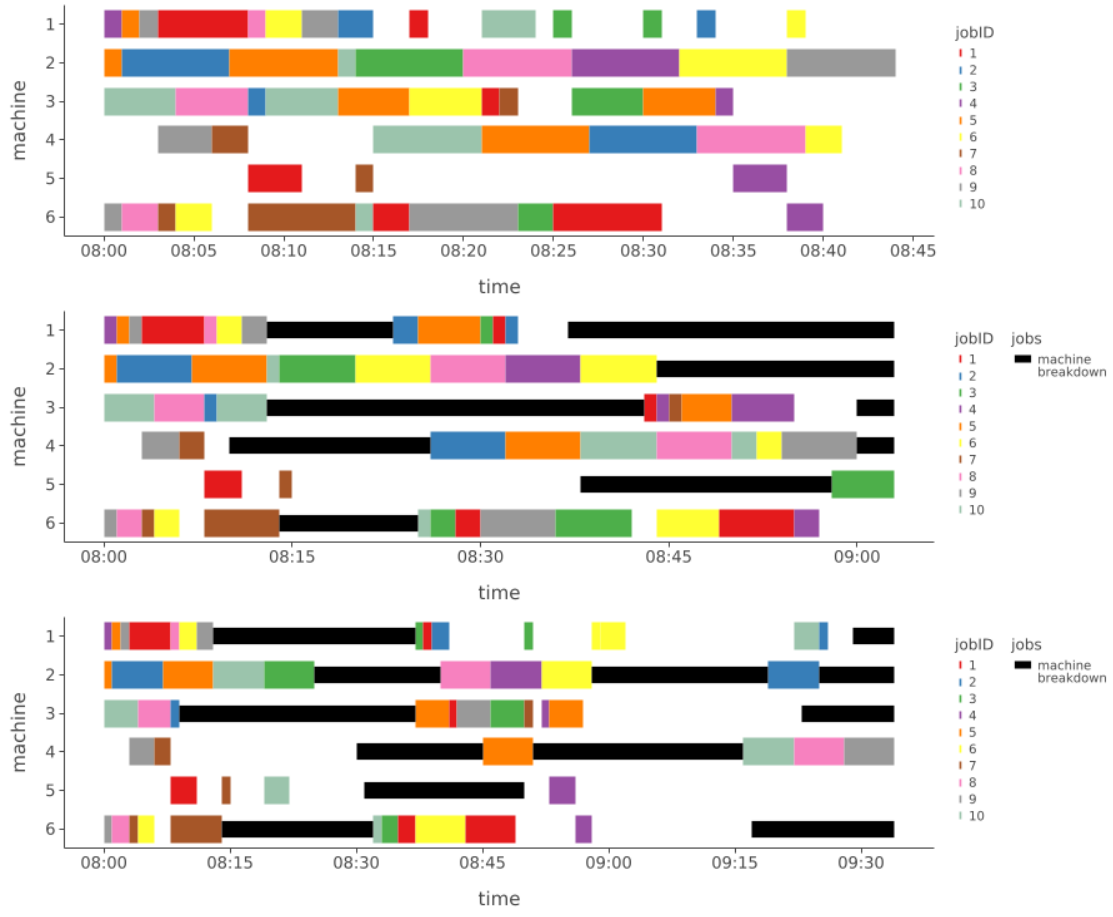


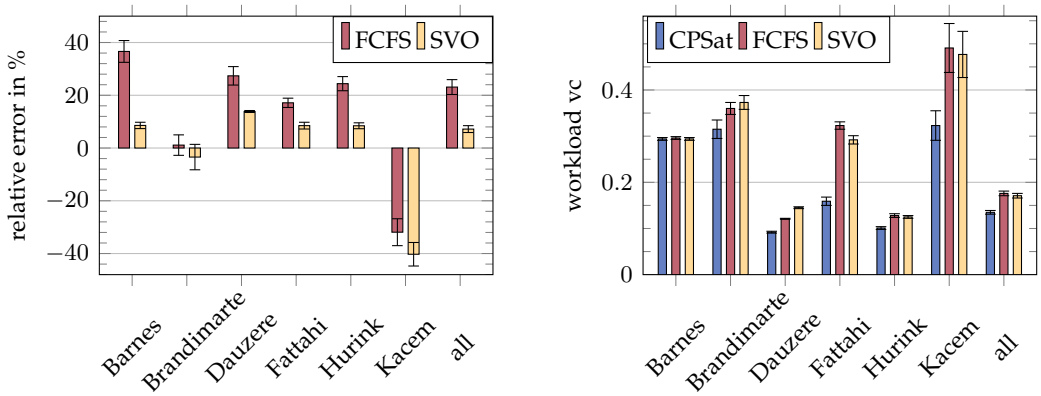

Figure 22: Gantt chart comparison of SVO solutions for the Brandimarte Mk01 problem. Upper panel: static run, makespan 44; middle panel: seed 23, makespan 63, lower panel: seed 42, makespan 94.

With machine breakdowns, seeds have a bigger influence on the outcome, if critical paths are disrupted. To illustrate this, Figure 22 compares Gantt charts of an instance with extreme deviations in makespan depending on the randomization seed. The breakdown pattern produced by seed 23 disrupts the original schedule to a much smaller extend than the one produced by seed 42, where machine 2 - the busiest machine in the original schedule - breaks down repeatedly. This leads to a large discrepancy in makespan, with an increase of 114 % compared to the static schedule for seed 42 as opposed to 43 % for seed 23. When disregarding shifts due to the non-preemptiveness condition, breakdown patterns are the same for all algorithms when the same seed is used, therefore relative errors are not affected to the same extend as raw makespan values. Nevertheless, critical paths in the schedules produced by `CPSat` and the `SVO` algorithm can differ. Slightly larger variations between runs with different seeds were observed when including machine breakdowns, but F-tests on the standard deviations did not indicate significance. The F-test is mostly analogous to the t-test, but is applicable to examine the distribution of variations between groups.

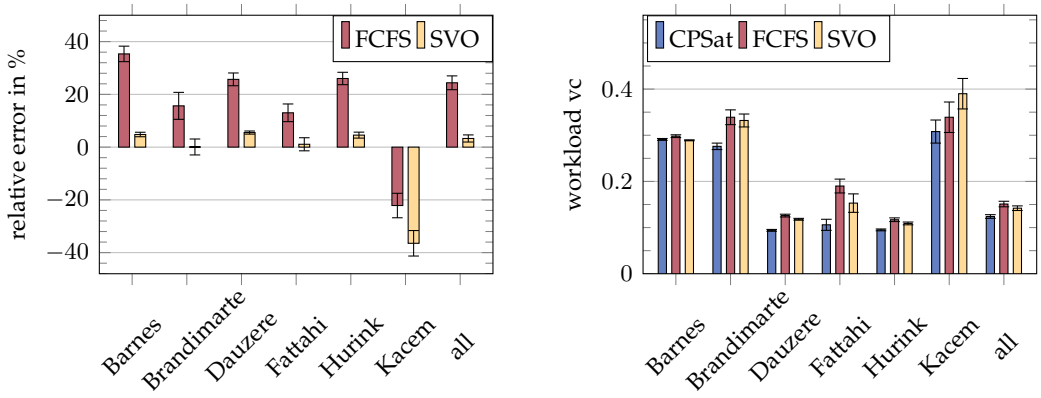
Scheduling results averaged over benchmarking set are visualized in Figure 23, with error bars representing standard errors between simulations with different seeds. Both with and without dynamic job arrivals, lower makespan values were obtained by the `SVO` algorithm compared to `FCFS`, but the improvement was more pronounced when dynamic job arrivals were included. Since `FCFS` scheduling results showed no significant difference (p-value of 0.09), this indicates that `SVO` scheduling is especially well-suited to react to dynamic job arrivals. This is underlined by a comparison of makespan results obtained by `SVO` scheduling under different simulation conditions as presented in Table 16. The lowest relative makespan errors were obtained when both jobs and machines were modeled dynamically (3.3%), followed by dynamic jobs combined with static machines (4.4%).

Table 16: Comparison of relative makespan errors in different simulation environments (all values in %).

| set | static | | dyn. jobs | | dyn. machines | | both dyn. | |
|---|---|---|---|---|---|---|---|---|
| | FCFS | SVO | FCFS | SVO | FCFS | SVO | FCFS | SVO |
| Barnes | 22.7 | 19.3 | 40.0 | 5.3 | 36.6 | 8.6 | 35.3 | 4.8 |
| Brandimarte | 20.1 | 23.1 | 39.5 | 9.5 | 1.1 | −3.4 | 15.7 | 0.1 |
| Dauzere | 23.2 | 19.0 | 30.0 | 5.0 | 27.4 | 13.9 | 25.7 | 5.5 |
| Fattahi | 18.8 | 12.6 | 14.2 | 1.3 | 17.1 | 8.5 | 13.0 | 1.1 |
| Hurink | 20.2 | 14.7 | 29.7 | 4.9 | 24.4 | 8.4 | 26.0 | 4.6 |
| Kacem | 24.0 | 17.2 | 4.6 | −10.1 | −31.9 | −40.3 | −22.1 | −36.5 |
| all | 20.4 | 15.2 | 28.9 | 4.4 | 23.1 | 7.2 | 24.4 | 3.3 |
| rel. to FCFS | | −25.5 | | −84.9 | | −68.9 | | −86.6 |




(a) Static jobs and dynamic machines.




(b) Dynamic jobs and dynamic machines.

Figure 23: Results with dynamic machine breakdowns.

Workload balancing effectiveness was reduced for all algorithms under machine breakdown conditions (Table 17). This was to be expected, since the amount of balancing by operation assignments is limited by blocked schedule stretches, and those breakdown periods are not included in the workload variation coefficient calculations. The differences between the reference algorithms `CPSat` and `FCFS` are therefore smaller, and improvements by `SVO` scheduling compared to `FCFS` scheduling are not as pronounced.

As shown in Subsection 6.1, workload variation coefficients of `SVO` scheduling were not significantly different from `FCFS` scheduling in static environments, but under dynamic job arrival conditions. With the extension to machine breakdowns, the same observation is made: We observed only slight improvements of workload balancing with static jobs (25.9% compared to 30.4% higher than `CPSat`), and more noticeable improvements when dynamic job arrivals were included (14.5% compared to 21.8% higher than `CPSat`).

Table 17: Comparison of workload variation coefficients in different simulation environments.

| set | static | | dyn. jobs | | dyn. machines | | both dyn. | |
|---|---|---|---|---|---|---|---|---|
| | FCFS | SVO | FCFS | SVO | FCFS | SVO | FCFS | SVO |
| Barnes | 0.298 | 0.292 | 0.303 | 0.290 | 0.296 | 0.294 | 0.298 | 0.289 |
| Brandimarte | 0.321 | 0.315 | 0.361 | 0.286 | 0.360 | 0.365 | 0.339 | 0.327 |
| Dauzere | 0.122 | 0.149 | 0.130 | 0.116 | 0.121 | 0.144 | 0.126 | 0.119 |
| Fattahi | 0.332 | 0.285 | 0.195 | 0.157 | 0.323 | 0.287 | 0.190 | 0.151 |
| Hurink | 0.130 | 0.125 | 0.118 | 0.108 | 0.128 | 0.124 | 0.117 | 0.109 |
| Kacem | 0.277 | 0.253 | 0.188 | 0.137 | 0.491 | 0.480 | 0.339 | 0.390 |
| all | 0.173 | 0.164 | 0.151 | 0.135 | 0.176 | 0.170 | 0.151 | 0.142 |
| rel. to CPSat [%] | 56.5 | 48.0 | 30.2 | 16.4 | 30.4 | 25.9 | 21.8 | 14.5 |
| rel. to FCFS [%] | | −5.3 | | −10.6 | | −3.4 | | −6.0 |

### 6.4 Summary

Social value orientation was shown to be a useful concept in production scheduling, especially under dynamic job arrival conditions. With a much shorter runtime than `CPSat`, the `SVO` algorithm reached makespan errors of 9.0 %, surpassing `FCFS` scheduling by 50 % under static conditions. In dynamic job arrival environments, makespan errors dropped to 4.4 %, an improvement of 85 % relative to `FCFS` results, and even surpassed the `CPSat` reference in 10 % of the evaluated instances. Workload distributions deviated more severely from the `CPSat` reference due to the inherently flexible weighting between job and machine goals, with deviations ranging from 55 % (static, all sets) to 17 % (dynamic jobs, smaller subset), an improvement relative to `FCFS`' deviation values of 75 % or 56 %, respectively.

Different shop workload levels influenced solutions only by small amounts, with `SVO` scheduling producing the most stable makespan results, and workload balancing remaining mostly unaffected for all algorithms. Inclusion of dynamic machine breakdowns led to a further decrease in makespan errors of the `SVO` solutions to 3.3 % relative to `CPSat`, and a reduction of workload balancing deviations to 15 %.

Overall, the SVO scheduling implementation was shown to approach `CPSat` reference results closely under dynamic conditions, with a greatly reduced time complexity.

## 7 Conclusions

We successfully developed a multi-agent scheduling system for FJSP with makespan minimization and workload balancing objectives. A discrete event simulation framework was utilized to model the environment and implement an SVO-based schedul-

ing algorithm along with two reference algorithms.

Six utility functions were tested, of which linear deviations from ideal values, weighted by the remaining workload of a job, delivered the best results. An initial algorithm version was revised to improve integration of machine and job goals, which lead to a close approximation of the constraint programming reference in regards to makespan (9.0 %) and 54.6 % higher workload variation coefficients. Time complexity of `SVO` scheduling was significantly lower than with the exact method. With the largest instance taking only a few seconds to solve, the algorithm should be fast enough for online scheduling.

The discrepancy between makespan and workload balancing objective can be explained by the variable weighting between job and machine goals. SVO angles are usage-dependent for machine agents, while they are constant for job agents, so the weighting between makespan and workload balancing objectives is not fixed when aggregating utility values. Although this limits the algorithm's workload balancing capabilities, it seems to improve makespan results compared to single objective optimization. It would be possible to create a more equal weighting when aggregating job and machine utilities, but the flexibility might in fact be beneficial to a shop's efficiency: Makespan can be considered its key indicator, while machine workload is mainly of concern when deviations are large, e.g. high strain on some machines contributes to their increased wear, or large differences in workload prevent efficient deployment of human operators. Such large differences in workload should still be prevented, since machine workload gains an increased weight at low utilization rates, but workload balancing is not enforced as much as with a fixed weight. To support this notion, a more detailed evaluation of workload differences between individual machines than the workload variation coefficient would be necessary in future work.

Evaluation under dynamic conditions revealed that the SVO based algorithm excels when job arrivals are included, with makespan errors dropping to 4.4 %. Interestingly, a relatively large portion of the evaluated problem instances was even solved to better solutions compared to the reference. The heuristics applied by the `SVO` algorithm produces deterministic solutions, while slight variations in `CPSat` solution increments occur due to instance flexibility. These differences can lead to different outcomes under dynamic conditions, which were worse than `SVO` makespan results in 10 % of the cases. Further research would be needed to verify these numbers, with data collection of a sufficiently large number of repeated `CPSat` runs for the whole dataset. It would be interesting to explore the factors that lead to superior `SVO` results in some benchmarking instances in depth, e.g regarding flexibility and duration variety.

Overall shop workload levels had only minor effects on the results of all algorithms, with a tendency towards larger makespan errors under heavier workloads. This can be explained by the decreasing capability of the `CPSat` reference to optimize the schedule when jobs enter the system at a later time point, highlighting its limitations under dynamic conditions. While machine breakdown events could be

handled by all algorithms, SVO makespan errors decreased to 3.3 % when both jobs and machines were modeled dynamically, underlining its effectiveness in handling such dynamic events.

It should be noted that we had to apply several measures to break cyclic re-ordering and re-assignment of operations in the revised algorithm version, and it cannot be excluded that this issue arises again in novel problem instances with the current settings. When two very similar solutions exist, the system might end up in a state where operations are shifted back and forth, e.g. between two machines that aim to increase their usage. We alleviated cyclic re-assignments by restricting the range of SVO angles that machines adopt, but different scenarios might need even tighter restrictions. Future work should address this possible issue, e.g. by considering restriction of machine SVO angles to values between 45° (pro-social) and 0° (self-interested). As a side effect, this might shift the balance between the two optimization objectives towards a more equal, yet still flexible weighting. Experiments with randomized instance generation could be used to increase confidence in the ability of the algorithm to terminate, and either a graph-based cycle detection or a timeout could be applied as fail-safe mechanisms.

Taken together, the results show that the developed SVO based algorithm is well-suited for scheduling in dynamic environments, with a good approximation of reference results, and low computational effort. Further work is necessary to ensure convergence of the algorithm in currently untested scenarios, and to evaluate its performance in less idealized settings, when transportation and setup times, or issues of real-time communication are taken into account.

# References


[1] J. Barbosa and P. Leitão, „Modelling and simulating self-organizing agent-based manufacturing systems“, in *IECON 2010 - 36th Annual Conference on IEEE Industrial Electronics Society*, Nov. 2010, pp. 2702–2707.

[2] J. Xie, L. Gao, K. Peng, X. Li, and H. Li, „Review on flexible job shop scheduling“, *IET Collaborative Intelligent Manufacturing*, vol. 1, no. 3, pp. 67–77, 2019.

[3] Z. Qin and Y. Lu, „Self-organizing manufacturing network: A paradigm towards smart manufacturing in mass personalization“, *Journal of Manufacturing Systems*, vol. 60, pp. 35–47, Jul. 1, 2021.

[4] R. L. Sisson, „Methods of Sequencing in Job Shops - A Review“, *Operations Research*, vol. 7, no. 1, pp. 10–29, Feb. 1959.

[5] H. Xiong, S. Shi, D. Ren, and J. Hu, „A survey of job shop scheduling problem: The types and models“, *Computers & Operations Research*, vol. 142, Article 105731, Jun. 2022.

[6] P. Brandimarte, „Routing and scheduling in a flexible job shop by tabu search“, *Annals of Operations Research*, vol. 41, no. 3, pp. 157–183, Sep. 1993.

[7] S. Dauzère-Pérès, J. Ding, L. Shen, and K. Tamssaouet, „The flexible job shop scheduling problem: A review“, *European Journal of Operational Research*, vol. 314, no. 2, pp. 409–432, Apr. 2024.

[8] B. Roy and B. Sussmann, „Les problémes dordonnancement avec constraintes disjoncties“, *Note ds*, vol. 9, 1964.

[9] E. Balas, „Machine Sequencing Via Disjunctive Graphs: An Implicit Enumeration Algorithm“, *Operations Research*, vol. 17, no. 6, pp. 941–957, Dec. 1969.

[10] S. Dauzère-Pérès and J. Paulli, „An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search“, *Annals of Operations Research*, vol. 70, no. 0, pp. 281–306, Apr. 1, 1997.

[11] D. Y. Lee and F. DiCesare, „Scheduling flexible manufacturing systems using Petri nets and heuristic search“, *IEEE Transactions on Robotics and Automation*, vol. 10, no. 2, pp. 123–132, Apr. 1994.

[12] P. J. M. van Laarhoven, E. H. L. Aarts, and J. K. Lenstra, „Job Shop Scheduling by Simulated Annealing“, *Operations Research*, vol. 40, no. 1, pp. 113–125, 1992.

[13] J. Błażewicz, E. Pesch, and M. Sterna, „The disjunctive graph machine representation of the job shop scheduling problem“, *European Journal of Operational Research*, vol. 127, no. 2, pp. 317–331, Dec. 1, 2000.

[14] M. R. Garey, D. S. Johnson, and R. Sethi, „The Complexity of Flowshop and Jobshop Scheduling“, *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117–129, May 1976.

[15] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan, „Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey“, in *Annals of Discrete Mathematics*, ser. Discrete Optimization II, P. L. Hammer, E. L. Johnson, and B. H. Korte, Eds., vol. 5, Elsevier, Jan. 1, 1979, pp. 287–326.

[16] J. Carlier and E. Pinson, „An Algorithm for Solving the Job-Shop Problem“, *Management Science*, vol. 35, no. 2, pp. 164–176, 1989.

[17] L. Meng, C. Zhang, Y. Ren, B. Zhang, and C. Lv, „Mixed-integer linear programming and constraint programming formulations for solving distributed flexible job shop scheduling problem“, *Computers & Industrial Engineering*, vol. 142, Article 106347, Apr. 1, 2020.

[18] K. Darby-Dowman, J. Little, G. Mitra, and M. Zaffalon, „Constraint Logic Programming and Integer Programming approaches and their collaboration in solving an assignment scheduling problem“, *Constraints*, vol. 1, no. 3, pp. 245–264, 1997.

[19] R. Gedik, C. Rainwater, H. Nachtmann, and E. A. Pohl, „Analysis of a parallel machine scheduling problem with sequence dependent setup times and job availability intervals“, *European Journal of Operational Research*, vol. 251, no. 2, pp. 640–650, Jun. 1, 2016.

[20] P. J. Stuckey, „Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving“, in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, A. Lodi, M. Milano, and P. Toth, Eds., Berlin, Heidelberg: Springer, 2010, pp. 5–9.

[21] L. Perron and F. Didier, *CP-SAT*, version v9.11, Google, May 7, 2024. [Online]. Available: `https://developers.google.com/optimization/cp/cp_solver/` (visited on 07/04/2025).

[22] E. Nowicki and C. Smutnicki, „A fast tabu search algorithm for the permutation flow-shop problem“, *European Journal of Operational Research*, vol. 91, no. 1, pp. 160–175, May 24, 1996.

[23] P. S. Ow and T. E. Morton, „Filtered beam search in scheduling“, *International Journal of Production Research*, vol. 26, no. 1, pp. 35–62, Jan. 1, 1988.

[24] S.-J. Wang, B.-H. Zhou, and X. Li-Feng, „A filtered-beam-search-based heuristic algorithm for flexible job-shop scheduling problem“, *International Journal of Production Research*, vol. 46, no. 11, pp. 3027–3058, Jun. 2008.

[25] L. A. Hall and D. B. Shmoys, „Jackson's Rule for Single-Machine Scheduling: Making a Good Heuristic Better“, *Mathematics of Operations Research*, vol. 17, no. 1, pp. 22–35, 1992.

[26] V. Sels, N. Gheysen, and M. Vanhoucke, „A comparison of priority rules for the job shop scheduling problem under different flow time- and tardiness-related objective functions“, *International Journal of Production Research*, vol. 50, no. 15, pp. 4255–4270, Aug. 1, 2012.

[27] Y. Gui, Z. Zhang, D. Tang, H. Zhu, and Y. Zhang, „Collaborative dynamic scheduling in a self-organizing manufacturing system using multi-agent reinforcement learning“, *Advanced Engineering Informatics*, vol. 62, Article 102646, Oct. 1, 2024.

[28] F. Glover, „Tabu Search: A Tutorial“, *Interfaces*, vol. 20, no. 4, pp. 74–94, Aug. 1990.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, „Optimization by Simulated Annealing“, *Science*, vol. 220, no. 4598, pp. 671–680, May 13, 1983.

[30] B. F. Skinner, „Reinforcement today“, *American Psychologist*, vol. 13, no. 3, pp. 94–99, 1958.

[31] M. E. Aydin and E. Öztemel, „Dynamic job-shop scheduling using reinforcement learning agents“, *Robotics and Autonomous Systems*, vol. 33, no. 2, pp. 169–178, Nov. 30, 2000.

[32] S. Baer, J. Bakakeu, R. Meyes, and T. Meisen, „Multi-Agent Reinforcement Learning for Job Shop Scheduling in Flexible Manufacturing Systems“, in *2019 Second International Conference on Artificial Intelligence for Industries (AI4I)*, Sep. 2019, pp. 22–25.

[33] G. E. Vieira, J. W. Herrmann, and E. Lin, „Rescheduling Manufacturing Systems: A Framework of Strategies, Policies, and Methods“, *Journal of Scheduling*, vol. 6, no. 1, pp. 39–62, Jan. 1, 2003.

[34] D. Ouelhadj and S. Petrovic, „A survey of dynamic scheduling in manufacturing systems“, *Journal of Scheduling*, vol. 12, no. 4, pp. 417–431, Aug. 1, 2009.

[35] O. Cardin, D. Trentesaux, A. Thomas, P. Castagna, T. Berger, and H. Bril El-Haouzi, „Coupling predictive scheduling and reactive control in manufacturing hybrid control architectures: State of the art and future challenges“, *Journal of Intelligent Manufacturing*, vol. 28, no. 7, pp. 1503–1517, Oct. 1, 2017.

[36] S. V. Mehta, „Predictable scheduling of a single machine subject to breakdowns“, *International Journal of Computer Integrated Manufacturing*, vol. 12, no. 1, pp. 15–38, Jan. 1, 1999.

[37] R. O'Donovan, R. Uzsoy, and K. N. McKay, „Predictable scheduling of a single machine with breakdowns and sensitive jobs“, *International Journal of Production Research*, vol. 37, no. 18, pp. 4217–4233, Dec. 1, 1999.

[38] B. Waschneck, A. Reichstaller, L. Belzner, *et al.*, „Optimization of global production scheduling with deep reinforcement learning“, *Procedia CIRP*, 51st CIRP Conference on Manufacturing Systems, vol. 72, pp. 1264–1269, Jan. 1, 2018.

[39] C. J. C. H. Watkins and P. Dayan, „Q-learning“, *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1, 1992.

[40] Y. Gui, D. Tang, H. Zhu, Y. Zhang, and Z. Zhang, „Dynamic scheduling for flexible job shop using a deep reinforcement learning approach“, *Computers & Industrial Engineering*, vol. 180, Article 109255, Jun. 1, 2023.

[41] F. Ren and H. Liu, „Dynamic scheduling for flexible job shop based on MachineRank algorithm and reinforcement learning“, *Scientific Reports*, vol. 14, no. 1, Article 29741, Nov. 29, 2024.

[42] Y. Li, Q. Wang, X. Li, *et al.*, „Real-Time Scheduling for Flexible Job Shop With AGVs Using Multiagent Reinforcement Learning and Efficient Action Decoding“, *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–13, 2025.

[43] H.-P. Wiendahl and V. Ahrens, „Agent-Based Control of Self-Organized Production Systems“, *CIRP Annals*, vol. 46, no. 1, pp. 365–368, Jan. 1, 1997.

[44] A. Giret and V. Botti, „Identifying and specifying holons in manufacturing systems“, in *2008 7th World Congress on Intelligent Control and Automation*, Jun. 2008, pp. 404–409.

[45] B. Hayes-Roth, „An architecture for adaptive intelligent systems“, *Artificial Intelligence*, vol. 72, no. 1, pp. 329–365, Jan. 1, 1995.

[46] M. Wooldridge, *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.

[47] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge, „The belief-desire-intention model of agency“, in *International Workshop on Agent Theories, Architectures, and Languages*, Springer, 1998, pp. 1–10.

[48] Z. Li, C. H. Sim, and M. Y. Hean Low, „A Survey of Emergent Behavior and Its Impacts in Agent-based Systems“, in *2006 4th IEEE International Conference on Industrial Informatics*, Aug. 2006, pp. 1295–1300.

[49] J. Barbosa, P. Leitão, E. Adam, and D. Trentesaux, „Behavioural Validation of the ADACOR2 Self-organized Holonic Multi-agent Manufacturing System“, in *Industrial Applications of Holonic and Multi-Agent Systems*, V. Mařík, A. Schirrmann, D. Trentesaux, and P. Vrba, Eds., Cham: Springer International Publishing, 2015, pp. 59–70.

[50] Smith, „The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver“, *IEEE Transactions on Computers*, vol. C-29, no. 12, pp. 1104–1113, Dec. 1980.

[51] A. Madureira, I. Pereira, P. Pereira, and A. Abraham, „Negotiation mechanism for self-organized scheduling system with collective intelligence“, *Neurocomputing*, Innovations in Nature Inspired Optimization and Learning Methods, vol. 132, pp. 97–110, May 20, 2014.

[52] W. B. G. Liebrand and C. G. McClintock, „The ring measure of social values: A computerized procedure for assessing individual differences in information processing and social value orientation“, *European Journal of Personality*, vol. 2, no. 3, pp. 217–230, Sep. 1, 1988.

[53] N. Buckman, A. Pierson, W. Schwarting, S. Karaman, and D. Rus, „Sharing is Caring: Socially-Compliant Autonomous Intersection Negotiation“, in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Macau, China: IEEE, Nov. 2019, pp. 6136–6143.

[54] W. Schwarting, A. Pierson, J. Alonso-Mora, S. Karaman, and D. Rus, „Social behavior for autonomous vehicles“, *Proceedings of the National Academy of Sciences*, vol. 116, no. 50, pp. 24 972–24 978, Dec. 10, 2019.

[55] C. Zhao, D. Chu, Z. Deng, and L. Lu, „Human-Like Decision Making for Autonomous Driving With Social Skills“, *IEEE Transactions on Intelligent Transportation Systems*, vol. 25, no. 9, pp. 12 269–12 284, Sep. 2024.

[56] G. Dagkakis and C. Heavey, „A review of open source discrete event simulation software for operations research“, *Journal of Simulation*, vol. 10, no. 3, pp. 193–206, Aug. 1, 2016.

[57] H. Brandl, *Kalasim*, version 1.0.3, Dec. 13, 2024. [Online]. Available: `https://www.kalasim.org/` (visited on 07/04/2025).

[58] J. Hurink, B. Jurisch, and M. Thole, „Tabu search for the job-shop scheduling problem with multi-purpose machines“, *Operations-Research-Spektrum*, vol. 15, no. 4, pp. 205–215, Dec. 1, 1994.

[59] J. W. Barnes and J. B. Chambers, „Solving the job shop scheduling problem with tabu search“, *IIE Transactions*, vol. 27, no. 2, pp. 257–263, Apr. 1, 1995.

[60] I. Kacem, S. Hammadi, and P. Borne, „Pareto-optimality approach for flexible job-shop scheduling problems: Hybridization of evolutionary algorithms and fuzzy logic“, *Mathematics and Computers in Simulation*, Intelligent Forecasting, Fault Diagnosis, Scheduling, and Control, vol. 60, no. 3, pp. 245–276, Sep. 30, 2002.

[61] P. Fattahi, M. Saidi Mehrabad, and F. Jolai, „Mathematical modeling and heuristic approaches to flexible job shop scheduling problems“, *Journal of Intelligent Manufacturing*, vol. 18, no. 3, pp. 331–342, Jun. 1, 2007.

[62] D. Behnke and M. J. Geiger, „Test Instances for the Flexible Job Shop Scheduling Problem with Work Centers“, *Research Paper, Helmut-Schmidt-University*, 2012.

[63] M. Mastrolilli. „Flexible Job Shop Problem“, Flexible Job Shop Problem. (2025), [Online]. Available: `https://people.idsia.ch/~monaldo/fjsp.html` (visited on 07/04/2025).

[64] S. Dauzère-Pérès, W. Roux, and J. B. Lasserre, „Multi-resource shop scheduling with resource flexibility“, *European Journal of Operational Research*, vol. 107, no. 2, pp. 289–305, Jun. 1, 1998.

[65] D. Hutter, T. Steinberger, and M. Hellwig. „A Benchmarking Environment for Worker Flexibility in Flexible Job Shop Scheduling Problems“. arXiv: 2501.16159 [cs]. (Apr. 14, 2025), pre-published.

[66] E. M. Dar-El and R. A. Wysk, „Job shop scheduling – A systematic approach“, *Journal of Manufacturing Systems*, vol. 1, no. 1, pp. 77–88, Jan. 1, 1982.

[67] L. Lv, J. Fan, C. Zhang, and W. Shen, „A multi-agent reinforcement learning based scheduling strategy for flexible job shops under machine breakdowns“, *Robotics and Computer-Integrated Manufacturing*, vol. 93, Article 102923, Jun. 1, 2025.

[68] W. He and D. Sun, „Scheduling flexible job shop problem subject to machine breakdown with route changing and right-shift strategies“, *The International Journal of Advanced Manufacturing Technology*, vol. 66, no. 1, pp. 501–514, Apr. 1, 2013.

[69] H. Abdi, „The method of least squares“, *Encyclopedia of measurement and statistics*, vol. 1, pp. 530–532, 2007.

[70] M. A. González, C. R. Vela, and R. Varela, „Scatter search with path relinking for the flexible job shop scheduling problem“, *European Journal of Operational Research*, vol. 245, no. 1, pp. 35–45, Aug. 16, 2015.

[71] A. A. García-León, S. Dauzère-Pérès, and Y. Mati, „An efficient Pareto approach for solving the multi-objective flexible job-shop scheduling problem with regular criteria“, *Computers & Operations Research*, vol. 108, pp. 187–200, Aug. 1, 2019.

[72] H. Zupan, N. Herakovič, and J. Žerovnik, „A Robust Heuristics for the Online Job Shop Scheduling Problem“, *Algorithms*, vol. 17, no. 12, Article 568, 12 Dec. 2024.

[73] R. Reijnen, I. G. Smit, H. Zhang, Y. Wu, Z. Bukhsh, and Y. Zhang. „Job Shop Scheduling Benchmark: Environments and Instances for Learning and Non-learning Methods“. arXiv: 2308.12794 [cs]. (Mar. 17, 2025), pre-published.

# Appendix

## Makespan results of `CPSat`

BKS: best known solution, ** optimal solution, * result matching or better than BKS, + optimal solution with 0.5 makespan weight, and thus inclusion in dynamic runs.

Table A.1: `CPSat` results for Barnes [59] and Brandimarte [6] sets.

| set | instance | BKS | | makespan | | |
|---|---|---|---|---|---|---|
| barnes | mt10c1 | 927 | [70] | 927 | ** | + |
| | mt10cc | 908 | [70] | 908 | ** | + |
| | mt10x | 918 | [70] | 918 | ** | + |
| | mt10xx | 918 | [70] | 918 | ** | + |
| | mt10xxx | 918 | [70] | 918 | ** | + |
| | mt10xy | 905 | [70] | 905 | ** | + |
| | mt10xyz | 847 | [70] | 847 | ** | + |
| | setb4c9 | 914 | [70] | 914 | ** | + |
| | setb4cc | 907 | [70] | 907 | ** | + |
| | setb4x | 925 | [70] | 925 | ** | + |
| | setb4xx | 925 | [70] | 925 | ** | + |
| | setb4xxx | 925 | [70] | 925 | ** | + |
| | setb4xy | 910 | [70] | 910 | ** | + |
| | setb4xyz | 903 | [70] | 902 | ** | + |
| | seti5c12 | 1,170 | [70] | 1,169 | ** | + |
| | seti5cc | 1,135 | [70] | 1,135 | * | |
| | seti5x | 1,198 | [70] | 1,198 | ** | + |
| | seti5xx | 1,197 | [70] | 1,194 | ** | + |
| | seti5xxx | 1,194 | [70] | 1,194 | ** | + |
| | seti5xy | 1,135 | [70] | 1,135 | * | |
| | seti5xyz | 1,125 | [70] | 1,125 | * | |
| brandimarte | Mk01 | 40 | [70] | 40 | ** | + |
| | Mk02 | 26 | [70] | 26 | * | + |
| | Mk03 | 204 | [70] | 204 | ** | |
| | Mk04 | 60 | [70] | 60 | ** | + |
| | Mk05 | 172 | [70] | 172 | * | |
| | Mk06 | 57 | [70] | 60 | | |
| | Mk07 | 139 | [70] | 139 | * | + |
| | Mk08 | 523 | [70] | 523 | ** | + |
| | Mk09 | 307 | [70] | 307 | ** | |
| | Mk10 | 196 | [70] | 207 | | |

Table A.2: `CPSat` results for Dauzère [64]and Fattahi [61] benchmarking sets.

| set | instance | BKS | | makespan | | |
|---|---|---|---|---|---|---|
| dauzere | 01a | 2,505 | [70] | 2,505 | ** | + |
| | 02a | 2,229 | [70] | 2,240 | | |
| | 03a | 2,228 | [70] | 2,231 | | |
| | 04a | 2,503 | [70] | 2,503 | ** | + |
| | 05a | 2,211 | [70] | 2,221 | | |
| | 06a | 2,183 | [70] | 2,207 | | |
| | 07a | 2,266 | [70] | 2,352 | | |
| | 08a | 2,064 | [70] | 2,076 | | |
| | 09a | 2,062 | [70] | 2,072 | | |
| | 10a | 2,267 | [70] | 2,315 | | |
| | 11a | 2,051 | [70] | 2,078 | | |
| | 12a | 2,018 | [70] | 2,064 | | |
| | 13a | 2,248 | [70] | 2,362 | | |
| | 14a | 2,163 | [70] | 2,193 | | |
| | 15a | 2,162 | [70] | 2,177 | | |
| | 16a | 2,244 | [70] | 2,305 | | |
| | 17a | 2,130 | [70] | 2,179 | | |
| | 18a | 2,119 | [70] | 2,171 | | |
| fattahi | Fattahi1 | 66 | [62] | 66 | ** | + |
| | Fattahi2 | 107 | [62] | 107 | ** | + |
| | Fattahi3 | 221 | [62] | 221 | ** | + |
| | Fattahi4 | 355 | [62] | 355 | ** | + |
| | Fattahi5 | 119 | [62] | 119 | ** | + |
| | Fattahi6 | 320 | [62] | 320 | ** | + |
| | Fattahi7 | 397 | [62] | 397 | ** | + |
| | Fattahi8 | 253 | [62] | 253 | ** | + |
| | Fattahi9 | 210 | [62] | 210 | ** | + |
| | Fattahi10 | 516 | [62] | 516 | ** | + |
| | Fattahi11 | 468 | [62] | 468 | ** | + |
| | Fattahi12 | 446 | [62] | 446 | ** | + |
| | Fattahi13 | 466 | [62] | 466 | ** | + |
| | Fattahi14 | 554 | [62] | 554 | ** | + |
| | Fattahi15 | 514 | [62] | 514 | ** | + |
| | Fattahi16 | 634 | [62] | 634 | ** | + |
| | Fattahi17 | 931 | [62] | 879 | ** | + |
| | Fattahi18 | 884 | [62] | 884 | ** | + |
| | Fattahi19 | 1,070 | [62] | 1,055 | * | |
| | Fattahi20 | 1,208 | [62] | 1,196 | * | |

Table A.3: `CPSat` results for Hurink edata [58] benchmarking set, compared to best known solutions from [62].

| instance | BKS | makespan | | | instance | BKS | makespan | | |
|---|---|---|---|---|---|---|---|---|---|
| abz5 | 1,167 | 1,167 | ** | + | la21 | 1,009 | 1,009 | * | |
| abz6 | 925 | 925 | ** | + | la22 | 880 | 880 | ** | + |
| abz7 | 622 | 622 | * | | la23 | 950 | 950 | ** | + |
| abz8 | 644 | 644 | * | | la24 | 908 | 908 | ** | + |
| abz9 | 648 | 655 | | | la25 | 936 | 936 | ** | + |
| car1 | 6,176 | 6,176 | ** | + | la26 | 1,106 | 1,112 | | |
| car2 | 6,327 | 6,327 | ** | + | la27 | 1,181 | 1,181 | ** | |
| car3 | 6,856 | 6,856 | ** | + | la28 | 1,147 | 1,142 | * | + |
| car4 | 7,789 | 7,789 | ** | + | la29 | 1,107 | 1,116 | | |
| car5 | 7,229 | 7,229 | ** | + | la30 | 1,204 | 1,198 | * | |
| car6 | 7,990 | 7,990 | ** | + | la31 | 1,532 | 1,532 | ** | + |
| car7 | 6,123 | 6,123 | ** | + | la32 | 1,698 | 1,698 | ** | + |
| car8 | 7,689 | 7,689 | ** | + | la33 | 1,547 | 1,547 | ** | + |
| la01 | 609 | 609 | ** | + | la34 | 1,599 | 1,599 | ** | + |
| la02 | 655 | 655 | ** | + | la35 | 1,736 | 1,736 | ** | + |
| la03 | 550 | 550 | ** | + | la36 | 1,160 | 1,160 | ** | + |
| la04 | 568 | 568 | ** | + | la37 | 1,397 | 1,397 | ** | + |
| la05 | 503 | 503 | ** | + | la38 | 1,141 | 1,141 | * | |
| la06 | 833 | 833 | ** | + | la39 | 1,184 | 1,184 | ** | + |
| la07 | 762 | 762 | ** | + | la40 | 1,144 | 1,144 | * | |
| la08 | 845 | 845 | ** | + | mt06 | 55 | 55 | ** | + |
| la09 | 878 | 878 | ** | + | mt10 | 871 | 871 | ** | + |
| la10 | 866 | 866 | ** | + | mt20 | 1,088 | 1,088 | ** | + |
| la11 | 1,103 | 1,103 | ** | + | orb1 | 977 | 977 | ** | + |
| la12 | 960 | 960 | ** | + | orb2 | 865 | 865 | ** | + |
| la13 | 1,053 | 1,053 | ** | + | orb3 | 951 | 951 | ** | + |
| la14 | 1,123 | 1,123 | ** | + | orb4 | 984 | 984 | ** | + |
| la15 | 1,111 | 1,111 | ** | + | orb5 | 842 | 842 | ** | + |
| la16 | 892 | 892 | ** | + | orb6 | 958 | 958 | ** | + |
| la17 | 707 | 707 | ** | + | orb7 | 387 | 389 | ** | + |
| la18 | 842 | 842 | ** | + | orb8 | 894 | 894 | ** | + |
| la19 | 796 | 796 | ** | + | orb9 | 933 | 933 | ** | + |
| la20 | 857 | 857 | ** | + | orb10 | 933 | 933 | ** | + |

Table A.4: `CPSat` results for Hurink rdata [58] benchmarking set, compared to best known solutions from [62].

| instance | BKS | makespan | | | instance | BKS | makespan | | |
|---|---|---|---|---|---|---|---|---|---|
| abz5 | 954 | 954 | ** | | la21 | 835 | 862 | | |
| abz6 | 807 | 807 | ** | + | la22 | 760 | 774 | | |
| abz7 | 544 | 544 | * | | la23 | 842 | 846 | | |
| abz8 | 555 | 576 | | | la24 | 804 | 803 | * | |
| abz9 | 545 | 572 | | | la25 | 787 | 796 | | |
| car1 | 5,034 | 5,035 | | + | la26 | 1,061 | 1,065 | | |
| car2 | 5,985 | 5,985 | * | | la27 | 1,056 | 1,103 | | |
| car3 | 5,623 | 5,624 | | | la28 | 1,080 | 1,079 | * | |
| car4 | 6,514 | 6,514 | * | | la29 | 998 | 1,003 | | |
| car5 | 5,615 | 5,615 | * | | la30 | 1,078 | 1,098 | | |
| car6 | 6,147 | 6,147 | ** | | la31 | 1,521 | 1,524 | | |
| car7 | 4,425 | 4,425 | ** | + | la32 | 1,659 | 1,663 | | |
| car8 | 5,692 | 5,692 | ** | | la33 | 1,499 | 1,499 | * | |
| la01 | 570 | 570 | * | + | la34 | 1,536 | 1,539 | | |
| la02 | 529 | 529 | ** | + | la35 | 1,550 | 1,554 | | |
| la03 | 477 | 477 | * | + | la36 | 1,023 | 1,049 | | |
| la04 | 502 | 502 | * | + | la37 | 1,066 | 1,084 | | |
| la05 | 457 | 457 | * | + | la38 | 954 | 972 | | |
| la06 | 799 | 799 | * | | la39 | 1,011 | 1,039 | | |
| la07 | 749 | 749 | * | + | la40 | 955 | 979 | | |
| la08 | 765 | 765 | * | + | mt06 | 47 | 47 | ** | + |
| la09 | 853 | 853 | * | | mt10 | 686 | 686 | ** | |
| la10 | 804 | 804 | * | + | mt20 | 1,022 | 1,022 | * | |
| la11 | 1,071 | 1,071 | * | | orb1 | 746 | 746 | ** | |
| la12 | 936 | 936 | * | | orb2 | 696 | 696 | ** | + |
| la13 | 1,038 | 1,038 | * | | orb3 | 712 | 712 | ** | |
| la14 | 1,070 | 1,070 | * | | orb4 | 753 | 753 | ** | |
| la15 | 1,089 | 1,089 | * | | orb5 | 639 | 639 | ** | |
| la16 | 717 | 717 | ** | | orb6 | 754 | 754 | ** | |
| la17 | 646 | 646 | ** | + | orb7 | 302 | 302 | ** | |
| la18 | 666 | 666 | ** | + | orb8 | 639 | 639 | ** | |
| la19 | 700 | 700 | ** | + | orb9 | 694 | 694 | ** | + |
| la20 | 756 | 756 | ** | + | orb10 | 742 | 742 | ** | |

Table A.5: `CPSat` results for Hurink sdata [58] benchmarking set, compared to best known solutions from [72].

| instance | BKS | makespan | | | instance | BKS | makespan | | |
|---|---|---|---|---|---|---|---|---|---|
| abz5 | 1,234 | 1,234 | ** | + | la21 | 1,046 | 1,046 | ** | + |
| abz6 | 943 | 943 | ** | + | la22 | 927 | 927 | ** | + |
| abz7 | 665 | 667 | | | la23 | 1,032 | 1,032 | ** | + |
| abz8 | 670 | 678 | | | la24 | 935 | 935 | ** | + |
| abz9 | 691 | 687 | * | | la25 | 977 | 977 | ** | + |
| car1 | 7,038 | 7,038 | ** | + | la26 | 1,218 | 1,218 | ** | + |
| car2 | 7,166 | 7,166 | ** | + | la27 | 1,235 | 1,235 | ** | + |
| car3 | 7,312 | 7,312 | ** | + | la28 | 1,216 | 1,216 | ** | + |
| car4 | 8,003 | 8,003 | ** | + | la29 | 1,152 | 1,162 | | |
| car5 | 7,702 | 7,702 | ** | + | la30 | 1,355 | 1,355 | ** | + |
| car6 | 8,313 | 8,313 | ** | + | la31 | 1,784 | 1,784 | ** | + |
| car7 | 6,558 | 6,558 | ** | + | la32 | 1,850 | 1,850 | ** | + |
| car8 | 8,264 | 8,264 | ** | + | la33 | 1,719 | 1,719 | ** | + |
| la01 | 666 | 666 | ** | + | la34 | 1,721 | 1,721 | ** | + |
| la02 | 655 | 655 | ** | + | la35 | 1,888 | 1,888 | ** | + |
| la03 | 597 | 597 | ** | + | la36 | 1,268 | 1,268 | ** | + |
| la04 | 590 | 590 | ** | + | la37 | 1,397 | 1,397 | ** | + |
| la05 | 593 | 593 | ** | + | la38 | 1,196 | 1,196 | ** | + |
| la06 | 926 | 926 | ** | + | la39 | 1,233 | 1,233 | ** | + |
| la07 | 890 | 890 | ** | + | la40 | 1,222 | 1,222 | ** | + |
| la08 | 863 | 863 | ** | + | mt06 | 55 | 55 | ** | + |
| la09 | 951 | 951 | ** | + | mt10 | 930 | 930 | ** | + |
| la10 | 958 | 958 | ** | + | mt20 | 1,165 | 1,165 | ** | + |
| la11 | 1,222 | 1,222 | ** | + | orb1 | 1,059 | 1,059 | ** | + |
| la12 | 1,039 | 1,039 | ** | + | orb2 | 888 | 888 | ** | + |
| la13 | 1,150 | 1,150 | ** | + | orb3 | 1,005 | 1,005 | ** | + |
| la14 | 1,292 | 1,292 | ** | + | orb4 | 1,005 | 1,005 | ** | + |
| la15 | 1,207 | 1,207 | ** | + | orb5 | 887 | 887 | ** | + |
| la16 | 945 | 945 | ** | + | orb6 | 1,010 | 1,010 | ** | + |
| la17 | 784 | 784 | ** | + | orb7 | 397 | 397 | ** | + |
| la18 | 848 | 848 | ** | + | orb8 | 899 | 899 | ** | + |
| la19 | 842 | 842 | ** | + | orb9 | 934 | 934 | ** | + |
| la20 | 902 | 902 | ** | + | orb10 | 944 | 944 | ** | + |

Table A.6: `CPSat` results for Hurink vdata [58] benchmarking set, compared to best known solutions from [73].

| instance | BKS | makespan | | | instance | BKS | makespan | | |
|---|---|---|---|---|---|---|---|---|---|
| abz5 | 859 | 859 | ** | | la21 | 803 | 812 | | |
| abz6 | 742 | 742 | ** | | la22 | 736 | 747 | | |
| abz7 | 495 | 515 | | | la23 | 811 | 821 | | |
| abz8 | 509 | 537 | | | la24 | 775 | 784 | | |
| abz9 | 500 | 530 | | | la25 | 756 | 760 | | |
| car1 | 5,006 | 5,007 | | + | la26 | 1,054 | 1,058 | | |
| car2 | 5,929 | 5,929 | * | | la27 | 1,084 | 1,090 | | |
| car3 | 5,599 | 5,599 | * | | la28 | 1,070 | 1,073 | | |
| car4 | 6,514 | 6,514 | * | | la29 | 994 | 998 | | |
| car5 | 4,919 | 4,915 | * | | la30 | 1,069 | 1,076 | | |
| car6 | 5,486 | 5,486 | ** | | la31 | 1,520 | 1,524 | | |
| car7 | 4,281 | 4,281 | ** | | la32 | 1,658 | 1,661 | | |
| car8 | 4,613 | 4,613 | ** | | la33 | 1,497 | 1,499 | | |
| la01 | 570 | 570 | * | | la34 | 1,535 | 1,540 | | |
| la02 | 529 | 529 | * | | la35 | 1,549 | 1,550 | | |
| la03 | 477 | 477 | * | | la36 | 948 | 968 | | |
| la04 | 501 | 502 | | | la37 | 986 | 986 | * | |
| la05 | 457 | 457 | * | | la38 | 943 | 943 | * | |
| la06 | 799 | 799 | * | | la39 | 922 | 928 | | |
| la07 | 749 | 749 | * | | la40 | 955 | 955 | * | |
| la08 | 765 | 765 | * | | mt06 | 47 | 47 | ** | + |
| la09 | 853 | 853 | * | | mt10 | 655 | 655 | ** | |
| la10 | 804 | 804 | * | | mt20 | 1,022 | 1,022 | * | |
| la11 | 1,071 | 1,071 | * | | orb1 | 695 | 695 | ** | + |
| la12 | 936 | 936 | * | | orb2 | 620 | 620 | ** | |
| la13 | 1,038 | 1,038 | * | | orb3 | 648 | 648 | ** | + |
| la14 | 1,070 | 1,070 | * | | orb4 | 753 | 753 | ** | + |
| la15 | 1,089 | 1,089 | * | | orb5 | 584 | 584 | ** | |
| la16 | 717 | 717 | ** | | orb6 | 715 | 715 | ** | |
| la17 | 646 | 646 | ** | + | orb7 | 275 | 275 | ** | |
| la18 | 663 | 663 | ** | + | orb8 | 573 | 573 | ** | |
| la19 | 617 | 617 | ** | | orb9 | 659 | 659 | ** | |
| la20 | 756 | 756 | ** | + | orb10 | 681 | 681 | ** | |

Table A.7: `CPSat` results for Behnke [62] and Kacem [60] benchmarking sets.

| instance | BKS | makespan | | | instance | BKS | makespan | | |
|---|---|---|---|---|---|---|---|---|---|
| Behnke1 | 91 | 90 | ** | | Behnke31 | 272 | 234 | * | |
| Behnke2 | 91 | 91 | ** | | Behnke32 | 259 | 230 | * | |
| Behnke3 | 91 | 91 | ** | | Behnke33 | 245 | 232 | * | |
| Behnke4 | 97 | 97 | * | | Behnke34 | 265 | 227 | * | |
| Behnke5 | 91 | 92 | | | Behnke35 | 253 | 220 | * | |
| Behnke6 | 131 | 133 | | | Behnke36 | 531 | 402 | * | |
| Behnke7 | 130 | 133 | | | Behnke37 | 536 | 414 | * | |
| Behnke8 | 128 | 132 | | | Behnke38 | 527 | 407 | * | |
| Behnke9 | 129 | 131 | | | Behnke39 | 516 | 400 | * | |
| Behnke10 | 133 | 134 | | | Behnke40 | 521 | 440 | * | |
| Behnke11 | 259 | 247 | * | | Behnke41 | 87 | 87 | * | |
| Behnke12 | 251 | 241 | * | | Behnke42 | 87 | 87 | * | |
| Behnke13 | 252 | 250 | * | | Behnke43 | 86 | 86 | * | |
| Behnke14 | 258 | 245 | * | | Behnke44 | 85 | 86 | | |
| Behnke15 | 262 | 244 | * | | Behnke45 | 87 | 87 | * | |
| Behnke16 | 566 | 453 | * | | Behnke46 | 124 | 117 | * | |
| Behnke17 | 535 | 430 | * | | Behnke47 | 126 | 122 | * | |
| Behnke18 | 555 | 430 | * | | Behnke48 | 134 | 127 | * | |
| Behnke19 | 532 | 444 | * | | Behnke49 | 121 | 116 | * | |
| Behnke20 | 522 | 438 | * | | Behnke50 | 131 | 130 | * | |
| Behnke21 | 85 | 85 | * | | Behnke51 | 259 | 224 | * | |
| Behnke22 | 87 | 87 | * | | Behnke52 | 255 | 218 | * | |
| Behnke23 | 86 | 86 | * | | Behnke53 | 257 | 221 | * | |
| Behnke24 | 87 | 87 | * | | Behnke54 | 267 | 234 | * | |
| Behnke25 | 87 | 88 | | | Behnke55 | 256 | 229 | * | |
| Behnke26 | 122 | 120 | * | | Behnke56 | 538 | 399 | * | |
| Behnke27 | 132 | 129 | * | | Behnke57 | 535 | 407 | * | |
| Behnke28 | 123 | 119 | * | | Behnke58 | 531 | 409 | * | |
| Behnke29 | 125 | 122 | * | | Behnke59 | 532 | 420 | * | |
| Behnke30 | 127 | 124 | * | | Behnke60 | 537 | 409 | * | |
| Kacem1 | 11 | 11 | ** | + | Kacem3 | 7 | 7 | ** | + |
| Kacem2 | 11 | 11 | ** | + | Kacem4 | 11 | 11 | ** | + |

### Dynamic Job environment

Table A.8: Makespan results equal or better than `CPSat` solutions under dynamic job arrival conditions.

| set | | instance | makespan | | CPSat | diff. |
|---|---|---|---|---|---|---|
| | | | CPSat | SVO | timeout | in % |
| Brandimarte | | Mk08 | 1,047.6 | 1,047.0 | | 0.1 |
| Fattahi | | Fattahi5 | 230.8 | 229.4 | | 0.6 |
| | | Fattahi6 | 589.6 | 564.4 | | 4.3 |
| | | Fattahi7 | 661.4 | 634.0 | | 4.1 |
| | | Fattahi10 | 968.6 | 904.8 | | 6.6 |
| | | Fattahi11 | 860.0 | 841.2 | | 2.2 |
| | | Fattahi13 | 907.4 | 847.8 | | 6.6 |
| | | Fattahi14 | 1,049.6 | 1,045.4 | | 0.4 |
| | | Fattahi15 | 1,038.2 | 993.8 | | 4.3 |
| Hurink | rdata | mt06 | 85.8 | 84.8 | | 1.2 |
| | sdata | la06 | 1,852.0 | 1,852.0 | | 0.0 |
| | | la10 | 1,916.0 | 1,916.0 | | 0.0 |
| | | la11 | 2,444.0 | 2,444.0 | | 0.0 |
| | | la13 | 2,300.0 | 2,300.0 | | 0.0 |
| | | la14 | 2,584.0 | 2,584.0 | | 0.0 |
| | vdata | la17 | 1,118.0 | 1,092.0 | * | 2.3 |
| | | la18 | 1,228.8 | 1,210.8 | * | 1.5 |
| | | la20 | 1,366.6 | 1,341.0 | * | 1.9 |
| | | mt06 | 80.8 | 80.2 | | 0.8 |
| | | orb4 | 1,389.0 | 1,364.4 | * | 1.8 |
| Kacem | | Kacem1 | 21.4 | 19.8 | | 7.5 |
| | | Kacem2 | 23.2 | 21.2 | | 8.6 |
| | | Kacem3 | 16.6 | 13.8 | | 16.9 |
| | | Kacem4 | 24.8 | 23.0 | * | 7.3 |

Table A.9: Workload influence on mean makespan without workload balancing.

| | mean makespan | | | drop [%] | | | rel. error [%] | |
|---|---|---|---|---|---|---|---|---|
| workload | CPSat | FCFS | SVO | CPSat | FCFS | SVO | FCFS | SVO |
| light | 2,708 | 3,394 | 2,854 | | | | 28.3 | 5.4 |
| medium | 2,686 | 3,377 | 2,844 | −0.8 | −0.5 | −0.4 | 28.9 | 5.9 |
| heavy | 2,666 | 3,366 | 2,839 | −0.8 | −0.3 | −0.2 | 30.0 | 6.5 |

Table A.10: Pairwise t-tests of makespans under varying shop loads without workload balancing. P-values indicating a significant difference are marked in bold.

| | CPSat | | FCFS | | SVO | |
|---|---|---|---|---|---|---|
| workload | light | heavy | light | heavy | light | heavy |
| medium | $\mathbf{2.7 \cdot 10^{-7}}$ | $\mathbf{4.0 \cdot 10^{-7}}$ | $\mathbf{4.0 \cdot 10^{-6}}$ | $\mathbf{7.5 \cdot 10^{-4}}$ | 0.244 | 0.660 |