Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 53-4 (2020) 89–94

IFAC Papers Online CONFERENCE PAPER ARCHIVE

Automatic Translation of Blocking Flexible Job Shop Scheduling Problems to Automata Using the Supervisory Control Theory

Daniel Sarsur C.[*] Patrícia N. Pena[**]
Ricardo H. C. Takahashi[***]

[*] *Graduate Program in Electrical Engineering - Universidade Federal de Minas Gerais, Belo Horizonte, Brazil (e-mail: danielsc@ufmg.br).*
[**] *Department of Electronics Engineering - Universidade Federal de Minas Gerais, Belo Horizonte, Brazil (e-mail: ppena@ufmg.br).*
[***] *Department of Mathematics - Universidade Federal de Minas Gerais, Belo Horizonte, Brazil (e-mail: taka@mat.ufmg.br).*

Abstract: This work presents an algorithm to automatically translate a Blocking Flexible Job Shop Scheduling Problem modeling into automata using the Supervisory Control Theory. Different problems of the literature are analyzed in their textual form and the algorithm returns an automaton that implements the closed-loop behavior under the Supervisory Control Theory. A heuristic is applied to find, among all sequences, the sequence that minimizes the makespan. With our approach, we find makespan values near to those in the literature. This methodology faces memory usage boundaries, but it was able to find solutions for instances of some well-known problems.

Keywords: Supervisory Control Theory; Optimization; Job Shop Scheduling; Blocking; Application;

1. INTRODUCTION

Over the years the task scheduling problem has been increasingly studied due to its importance to the industry. This problem consists on a decision-making process and deals with the resource allocation in order to optimize an objective (Baker and Trietsch, 2013). In the problem formulation, a number of jobs, divided into operations, must be assigned to available machines. Different environments can be found depending on how jobs are processed, like unrelated parallel machines, flow shop, job shop, open shop, and others (Pinedo, 2016).

The job shop scheduling problem (JSSP) defines that each job has its own predetermined route to be processed and they can be different from each other. A generalization of this problem is the flexible job shop scheduling problem (FJSSP) where an operation can be processed by more than one machine. In this case, each job may have more than one route (Pinedo, 2016). Brucker et al. (1994), Adams et al. (1988) and Van Laarhoven et al. (1992) are some works dedicated to solve the FJSSP, using different techniques.

This work focuses in the Blocking FJSSP where a machine can only release the processed job if the next machine is available, otherwise it remains blocked. This scenario is pretty common in manufacturing systems where there are no intermediate buffers between the machines (Mascis and Pacciarelli, 2002).

Generally, precedence relation and processing times at each machine are presented in tables or in textual form. The first one is more suitable for visual interpretation while the second one is easier to be read by software. Each set of data of a problem can also be called an instance.

It is important to mention the common usage of a discrete event system abstraction to represent the behavior of a system. The machine's state changes with events occurrences, that initiates and ends an operation in the machine. The languages and automata framework (Hopcroft et al., 2001) is used in this work.

Along with the automata representation, we use the Supervisory Control Theory (SCT) (Ramadge and Wonham, 1989) to restrict the behavior of the plant to a minimally restrictive and nonblocking closed loop behavior. The SCT uses automata to model the open loop plant and the system restrictions to generate a supervisor, an agent that enforces the legal behavior. Besides that, the entire problem search space is contained in the supervisor.

Therefore, the main purpose of this work is to generate automatically the automaton that models a Blocking FJSSP from instances in the textual form as they are found in the literature. Other goals are: to generate a supervisor that models the closed-loop behavior of each problem and apply to them an heuristic to minimize makespan. Thus, we show that it is possible to use DES and SCT to solve a practical problem of a different area, specifically, the Operational Research Area.

The paper is organized as follows. The next section introduces preliminary concepts to help understanding the results. Section 3 presents the procedure to automatic

10.1016/j.ifacol.2021.04.009

translate the problem and some information about the supervisors found. The following section gives the makespan results when a heuristic is applied to the mentioned supervisors. Section 5 discuss the achievements and difficulties of this work.

2. PRELIMINARIES

Concepts related to the Blocking Flexible Job Shop Scheduling Problem are introduced in Section 2.1. Then, we introduce the main concepts related to languages, automata (Section 2.2) and the Supervisory Control Theory (Section 2.3). In Section 2.4 some considerations about the "blocking" term are presented.

2.1 Blocking Flexible Job Shop Scheduling Problem

The Flexible Job Shop Scheduling Problem (FJSSP) is one of the variations of the task scheduling problems, which deals with resource allocation to complete tasks. The tasks are called jobs and the resources are called machines. A set of n jobs $\{J_i\}_{1 \le i \le n}$, consisting of h operations $\{O_{ij}\}_{1 \le j \le h}$, must be processed by a set of m machines $\{M_k\}_{1 \le k \le m}$ with a processing time of p_{ijk}. Each machine can process one operation at a time.

Table 1 presents the assignment of operations' processing time in relation to the machines. A value $p \in \mathbb{R}_+$ is assigned to an operation if it can be processed by a machine and an infinite value (∞) is assigned if it cannot (Behnke and Geiger, 2012).

Table 1. Processing time assignment for the FJSSP with total flexibility

$p_{i,j,k}$		M_1	M_k	$\cdots$	M_m
J_1	$O_{1,1}$	$p_{1,1,1}$	$p_{1,1,k}$	$\cdots$	$p_{1,1,m}$
	$O_{1,2}$	$p_{1,2,1}$	$p_{1,2,k}$	$\cdots$	$p_{1,2,m}$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
	O_{1,h_1}	$p_{1,h_1,1}$	$p_{1,h_1,k}$	$\cdots$	$p_{1,h_1,m}$
J_i	$O_{i,1}$	$p_{i,1,1}$	$p_{i,1,k}$	$\cdots$	$p_{i,1,m}$
	$O_{i,2}$	$p_{i,2,1}$	$p_{i,2,k}$	$\cdots$	$p_{i,2,m}$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
	O_{i,h_i}	$p_{i,h_i,1}$	$p_{i,h_i,k}$	$\cdots$	$p_{i,h_i,m}$
$\cdots$		$\cdots$	$\cdots$	$\cdots$	$\cdots$
J_n	$O_{n,1}$	$p_{n,1,1}$	$p_{n,1,k}$	$\cdots$	$p_{n,1,m}$
	$O_{n,2}$	$p_{n,2,1}$	$p_{n,2,k}$	$\cdots$	$p_{n,2,m}$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
	O_{n,h_n}	$p_{n,h_n,1}$	$p_{n,h_n,k}$	$\cdots$	$p_{n,h_n,m}$

The FJSSP considers the existence of unlimited buffers between the machines, which means that when a machine processes an operation, it becomes immediately available to receive the next operation. That kind of statement is only possible for simulation purposes, because in the real world is not possible to conceive a storage with unlimited space. A realistic scenario considers that buffers are limited or nonexistent. In this last case, the machine plays the role of the buffer keeping the job until the next machine is available to receive it. This machine is, then, blocked to process the next job. When this restriction is considered the problem is called Blocking FJSSP.

Two different cases of blocking must be considered: *blocking with swap* (BWS) and *blocking no-swap* (BNS). In a cycle of two or more machines blocked, each one waiting for another machine of this cycle, a swap moves all jobs, simultaneously, to the next machine. When a swap is not allowed, that situation leads to a deadlock.

Some studies (Pranzo and Pacciarelli, 2016) and (Louaqad and Kamach, 2015) mention the solution infeasibility as an issue in Blocking FJSSP. Depending on how an algorithm is implemented, a solution that is found may not be feasible. An alternative is to refuse it, which is a waste of computational resources. Another alternative is to go back a few steps and try to find a new path to the solution. It's important to consider that there are no guarantees that the new path will lead to a feasible solution, so this process may happen several times, which is also a waste of computational resources. To work around this problem, some techniques can be used, such as the one chosen in this work.

2.2 Discrete Event Systems

Discrete event system are dynamic systems driven by events, which are internal and external entities that interact with the model changing its state instantly. These events can represent a sensor signal, the push of a button, the end of a task, etc. This kind of system is different from those driven by time, for example, where the state changes continuously and its solutions are based on the Classical Control Theory.

Let Σ be the finite and nonempty set of events, called alphabet. A finite sequence of events from an alphabet is called a string. The set of all possible strings built from alphabet Σ, is called Kleene Closure Σ^*, and it includes the empty string ϵ. The length of a string is defined by $|\sigma_1\sigma_2...\sigma_i| = i$, where $i > 0$. The length of the empty string is $|\epsilon| = 0$.

The concatenation of two strings is represented by su, where $s, u, su \in \Sigma^*$. A string s is prefix of $t \in \Sigma^*$, $s \le t$, if $\exists u \in \Sigma^*$ such that $su = t$. A language is defined as a subset $L \subseteq \Sigma^*$ and the prefix closure $\overline{L}$ of a language $L \subseteq \Sigma^*$ is the set of all string prefixes in L, expressed by $\overline{L} = \{s \in \Sigma^* | s \le t \text{ for some } t \in L\}$.

Automata are oriented graphs whose vertices are the states and the edges are the transitions. A deterministic finite automaton is defined as a 5-tuple $G = (Q, \Sigma, \delta, q_0, Q_m)$, where Q is the finite set of states, Σ the alphabet, $\delta : Q \times \Sigma \to Q$ the transition function, $q_0 \in Q$ the initial state and $Q_m \subseteq Q$ the set of marked states. An automaton is deterministic if $\delta(q, \sigma) = q_1$ and $\delta(q, \sigma) = q_2$ implies $q_1 = q_2$.

The transition function can be extended to strings in the form $\delta(q, \sigma s) = q'$ if $\delta(q, \sigma) = x$ and $\delta(x, s) = q'$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. The active event function $\Gamma : Q \to 2^\Sigma$ is given by the event set $\sigma \in \Sigma$, in a given state q, where $\delta(q, \sigma)$ is defined, $\delta(q, \sigma)!$.

The automaton generated language is given by $\mathcal{L}(G) = \{s \in \Sigma^* | \delta(q_0, s)!\}$ and represents the set of strings that, starting from the initial state, leads to some state of the automata G. The automaton recognized language is given by $\mathcal{L}_m(G) = \{s \in \Sigma^* | s \in \mathcal{L}(G) \text{ and } \delta(q_0, s) \in Q_m\}$ and represents the set of strings that, starting from the initial state, leads to some marked state $q \in Q_m$ of G.

An important operation to represent the combined behavior of automata is the parallel composition. This operation is applied to automata $G_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, Q_{m1})$ and $G_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, Q_{m2})$ leading to $G_{12} = G_1 \parallel G_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{12}, (q_{01}, q_{02}), Q_{m1} \times Q_{m2})$, with

$$\delta_{12} = \delta((q_1, q_2), \sigma) =$$
$$\begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)), & \text{if } \sigma \in \Gamma_1(q_1) \cap \Gamma_2(q_2) \\ (\delta_1(q_1, \sigma), q_2), & \text{if } \sigma \in \Gamma_1(q_1) \backslash \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)), & \text{if } \sigma \in \Gamma_2(q_2) \backslash \Sigma_1 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

where $\Gamma(q) \backslash \Sigma$ indicates subtraction of sets.

2.3 Supervisory Control Theory

The Supervisory Control Theory (SCT) (Ramadge and Wonham, 1989) allows to find a nonblocking and minimally restrictive supervisor that implements restrictions regarding safety and liveness. The SCT leads to a control structure that works disabling events that lead the system to unsafe behaviors.

The open loop system is called *plant* and its behavior can be found by the parallel composition of each machine $G = \|G_i$, where i is the machine index. The set of system restrictions, so called specifications, is the result of the parallel composition of each specification $E = \|E_i$.

For the SCT modeling the alphabet is splitted into two disjoint subsets $\Sigma = \Sigma_c \cup \Sigma_{uc}$. The first one, Σ_c, is composed by the controlled events and the second one Σ_{uc} is composed by the uncontrolled events, which cannot be disabled.

The parallel composition between the plant and the specifications leads to the desired closed loop behavior $K = G \parallel E$, so $\mathcal{L}_m(K) \subseteq \mathcal{L}_m(G)$. The automaton K is said to be controllable regarding G if $\overline{K}\Sigma_{uc} \cap \mathcal{L}(G) \subseteq \overline{K}$.

The controllability property establishes that uncontrollable events cannot be disabled. If the supervisor is not controllable, the supremal controllable and nonblocking sublanguage $Sup\mathcal{C}(K, G) \subseteq K$ must be found. This is done by eliminating all the states that disable uncontrolled events and then checking for the controllability property until it becomes satisfied.

In this case, the supervisor contains all sequences that respect the restrictions and it can be considered as the optimization problem search space. It means that any string chosen from the closed loop behavior is certainly feasible, so there will be no waste of computational resources dealing with infeasible solutions, which is the main advantage of using the SCT to model the Blocking FJSSP.

2.4 "Blocking" Meaning Considerations

The word "blocking" has different meanings depending on the context in which it appears.

When in SCT context the term represents a scenario in which a deadlock or a livelock occurs in a system. Deadlock characterizes the situation where the system is stuck in a state and this state is not a marked state, where a task is ended. In the Blocking FJSSP, when this happens, the system crashes because some of the processes involved are unable to complete their tasks. A livelock is when the system gets stuck in a cycle of actions that never take the system to complete a task.

When it comes to task scheduling problems, "blocking" exists when buffers are limited or nonexistent, so one machine needs to hold a job until the next machine can receive it. Thus, the job that remains on the machine blocks it from processing another job. Although a blocking in JSSP problems makes deadlocks possible, the difference in both contexts should be clear.

3. AUTOMATIC TRANSLATION

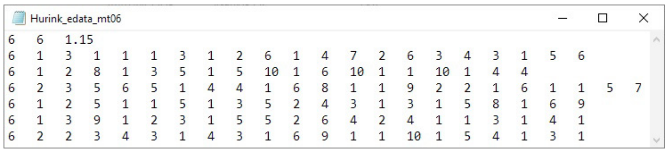
A set of benchmark problems from the literature was chosen to perform the automatic translation, presented in Table 2. The modeling was done considering that swap is not allowed.

Table 2. FJSSP Instances

Instances	Jobs	Machines	Reference
mt06, 10, 20	6..20	5..10	Muth et al. (1963)
mk01..10	10..20	4..15	Brandimarte (1993)
01..18a	10..20	5..10	Dauzère-Pérès and Paulli (1997)
la01..40	10..30	5..15	Lawrence (1984)
orb1..10	10	10	Applegate and Cook (1991)
mt10_, setb4_, seti5_	10..15	11..18	Chambers and Barnes (1996)
abz5..9	10..20	10..15	Adams et al. (1988)
car1..8	7..14	4..9	Carlier (1978)
hurink s,e,r,v data	6..30	4..15	Hurink et al. (1994)

3.1 Instance Interpretation

All the information related to machine assignment and processing times for each operation are, typically, presented in two different ways. One of them is as a table like Table 1 and the other is in a textual format as presented in Figure 1. The first one is more appropriate for visual interpretation and the second one is more appropriate for software interpretation.



```
6  6  1.15
6  1  3  1  1  1  3  1  2  6  1  4  7  2  6  3  4  3  1  5  6
6  1  2  8  1  3  5  1  5  10 1  6  10 1  1  10 1  4  4
6  2  3  5  6  5  1  4  4  1  6  8  1  1  9  2  2  1  6  1  1  5  7
6  1  2  5  1  1  5  1  3  5  2  4  3  1  3  1  5  8  1  6  9
6  1  3  9  1  2  3  1  5  5  2  6  4  2  4  1  1  3  1  4  1
6  2  2  3  4  3  1  4  3  1  6  9  1  1  10 1  5  4  1  3  1
```

Figure 1. Instance example

The first row of the textual file has at least two fields: 1) number of jobs n; 2) number of machines m; 3) (optional) mean of the number of machines able to process each operation, which indicates the problem flexibility.

In the example of Figure 1, the first row is

6 6 1.15

which represents a problem with 6 jobs, 6 machines and 1.15 machines per operation.

The first row is followed by n rows with the data of each job. The first value of each row indicates the number of operations h of the job and is followed by h sets of data for each operation: the first value is the number of machines r that can process that operation followed by r pairs (machine, processing time). An example is shown based on the instance presented in Figure 1.

The second line has all data related to the first job (J_1) as presented below:

$$\overbrace{6}^{h}\ \overbrace{\underbrace{1}_{r=1}\ \underbrace{3\quad 1}_{\text{pair}}}^{O_{1,1}}\ \overbrace{\underbrace{1}_{r=1}\ \underbrace{1\quad 3}_{\text{pair}}}^{O_{1,2}}\ \overbrace{\underbrace{1}_{r=1}\ \underbrace{2\quad 6}_{\text{pair}}}^{O_{1,3}}$$

$$\overbrace{\underbrace{1}_{r=1}\ \underbrace{4\quad 7}_{\text{pair}}}^{O_{1,4}}\ \overbrace{\underbrace{2}_{r=2}\ \underbrace{6\quad 3}_{\text{pair}}\ \underbrace{4\quad 3}_{\text{pair}}}^{O_{1,5}}\ \overbrace{\underbrace{1}_{r=1}\ \underbrace{5\quad 6}_{\text{pair}}}^{O_{1,6}}$$

In this job, the first operation ($O_{1,1}$) is performed by machine 3 and has a processing time of 1 unit. The second operation ($O_{1,2}$) is performed by machine 1 with processing time of 3 units. Only operation $O_{1,5}$ can be processed by two machines which are machine six and four, both with processing time of three units. The interpretation of this line leads to Table 3.

Table 3. Processing time assignment for the given job

$p_{i,j}$		M_1	M_2	M_3	M_4	M_5	M_6
J_1	$O_{1,1}$	∞	∞	1	∞	∞	∞
	$O_{1,2}$	3	∞	∞	∞	∞	∞
	$O_{1,3}$	∞	6	∞	∞	∞	∞
	$O_{1,4}$	∞	∞	∞	7	∞	∞
	$O_{1,5}$	∞	∞	∞	3	∞	3
	$O_{1,6}$	∞	∞	∞	∞	6	∞

3.2 *Blocking FJSSP Modeling as Automaton*

The main purpose of this work is to develop a method to automatically translate a Blocking FJSSP into automata, with the objective of minimizing makespan upon such automata. The job modeling as an automaton is done based on the following steps:

- create an automaton for each job;
- each operation is composed by a pair of events, one for its beginning and other for its ending:
  - · events initiated with the letter a are controllable and represent the beginning of an operation;
  - · events initiated with the letter b are uncontrollable and represent the end of an operation;
  - · the index of each event is $ijkl$, where:
    - - i refers to the job;
    - - j refers to the operation;
    - - k refers to the current machine;
    - - l refers to the previous machine ($l = 0$ if the event refers to the first operation);
- the automaton marked language corresponds to the job operations' sequence;
- the states are named as $sd.x$, where:
  - · d is the automaton's depth;
  - · x is a natural number to distinguish states in the same depth.

Figure 2 shows a job modeled as described above. In this example the job has six operations and only one of them can be processed by more than one machine. Notice that in state $s8.0$ there are two futures, to reach state $s9.1$ with the execution of event $a1564$ (job 1, 5th operation, machine 6 is picked, and the previous machine was 4) or state $s9.0$ by executing event $a1544$ (job 1, 5th operation, machine 4 is picked, and the previous machine was 4).

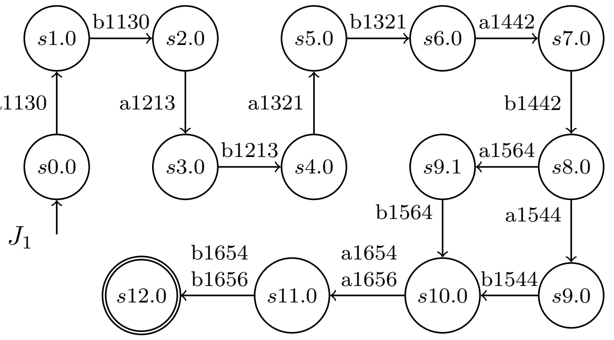


Figure 2. An example of job modeling as automaton

The machine modeling for the FJSSP without blocking can be done with only two states: one for when the machine is available ($s0$) and another for when the machine is working ($s1$). When blocking is considered an extra state ($s2$) is needed because after finishing an operation, the machine cannot go to the available state until the job is admitted in the next machine. The modeling is done as follows, Figure 3.

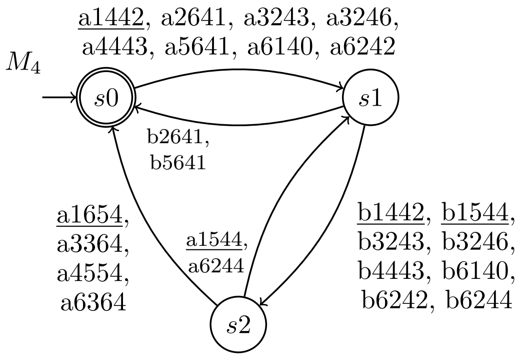

a1442, a2641, a3243, a3246, a4443, a5641, a6140, a6242
M_4
s0
s1
b2641, b5641
a1654, a3364, a4554, a6364
a1544, a6244
b1442, b1544, b3243, b3246, b4443, b6140, b6242, b6244
s2

Figure 3. Machine M_4 of instance of Figure 1 modeled as an automaton

A machine in the initial state $s0$ is available. When it starts processing a job, the machine goes to state $s1$ and at the end it goes to state $s2$. If the subsequent operation is admitted by the same machine, it returns to the state $s1$, but if the subsequent operation is admitted by another machine, the machine is then released going back to state $s0$.

A particular case should be considered regarding the scenario in which the machine processes the last operation of a job. In this case the machine is immediately released because the job is removed from the system once it has been fully processed. From an automaton standpoint, this represents a transition from state $s1$ (working) directly to state $s0$ (released). This is shown in Figure 3, where the machine four of the instance in Figure 1 is modeled.

The events' names are the same created in the job modeling, which means that the data for the machine modeling also comes from the instance. In order to illustrate the creation of the automaton for the machines, we relate the events of Machine 4 with Job 1 (Table 3). Machine 4 does operation 4 (a_{1442} and b_{1442}) and operation 5 (a_{1544} and b_{1544}) of job 1. From the initial state, machine M_4 can do event (a_{1442}) that moves it from state s_0 to state s_1, and from where it will exit with the end of such operation (b_{1442}) going to state s_2. In the case of job 1, the next operation is also performed by machine 4, so event a_{1544} will take the machine back to state s_1, from where it exits with b_{1544} to state s_2. From state s_2 (machine is off, but blocked waiting the next machine to take the job), after the 5th operation, the machine is released only after event a_{1654}. The same process of labeling is done for every job in the instance until Machine M_4 is complete. The events above mentioned are underlined to facilitate the observation.

Each machine is dealt with as an specification to the SCT problem, since they process one operation at each time, applying restriction to the execution of the jobs.

3.3 Supervisor Generation for Blocking FJSSP

To perform this step, a computer with Windows 10 64-bit operating system, RAM 64.00GB and Intel Xenon processor E5-2650/2.00 GHz was used. Procedures for modeling jobs and machines were implemented in a program, as presented in Algorithm 1 and the automata were generated using the UltraDES library (Alves et al., 2017). The monolithic supervisor was obtained from the plants (jobs models) and specifications (machines restrictions).

```
Algorithm 1: Supervisor generation from FJSSP instance
  Input: FJSSP_instance
  Output: supervisor
1 foreach $job_n$ do
2     $depth \leftarrow 0$
3     foreach $operation_h$ do
4         $x \leftarrow 0$
5         foreach $machine_m$ do
6             add $[depth.x \rightarrow a_{ijkl} \rightarrow (depth+1).x]$ to $job_n$
7             add $[(depth+1).x \rightarrow b_{ijkl} \rightarrow (depth+2).x]$ to $job_n$
8             $x \leftarrow x++$
9             add $[0 \rightarrow a_{ijkl} \rightarrow 1]$ to $maq_m$
10            add $[1 \rightarrow b_{ijkl} \rightarrow 0]$ to $maq_m$
11        depth ← depth + 2
12    G_set ← generate $job_n$ automaton
13 foreach $maq_m$ do
14    E_set ← generate $maq_m$ automaton
15 supervisor ← generate_monolithic_supervisor (G_set, E_set)
```

The number of states and transitions of each supervisor are presented in Table 4. The table presents only the instances that had a computable solution. Note that $|M_{j,k}|$ indicates the average of machines per operation (flexibility).

Although the size of the supervisors found is relatively large, the entire search space is contained within it. This means that any path chosen from the initial state to the supervisor's marked state represents a feasible, nonblocking production sequence that satisfies the constraints of the problem.

Other instances were subjected to the same procedure, but could not compute their supervisor for lack of memory on the computer where they were processed. This is due to the state space explosion problem that can be observed for example, comparing the "Hurink_sdata_la01.fjs" and "Hurink_sdata_car1.fjs" instances where a 10% increase in the number of jobs causes a $\approx 1,150\%$ increase in the number of states. A small increase in the flexibility causes a huge increase in the size of the problem, as can be seen from "Huring_edata_mt06" to "Huring_rdata_mt06".

Table 4. Size of obtained supervisors

Instance	**Dimension**	$\overline{\lvert M_{j,k}\rvert}$	**States**	**Transitions**
Hurink_sdata_mt06	6x6	1.00	346.328	1.085.803
Hurink_edata_mt06	6x6	1.15	761.494	2.782.216
Hurink_rdata_mt06	6x6	2.00	36.923.146	234.222.896
Hurink_vdata_mt06	6x6	3.00	110.097.999	1.128.024.299
Hurink_sdata_car1	11x5	1.00	121.529.344	394.521.601
Hurink_sdata_car2	13x4	1.00	111.562.752	314.163.201
Hurink_sdata_car3	12x5	1.00	383.021.056	1.253.793.793
Hurink_sdata_car4	14x4	1.00	300.498.944	850.755.585
Hurink_sdata_car5	10x6	1.00	249.672.704	915.118.081
Hurink_sdata_car7	7x7	1.00	10.787.520	40.741.569
Hurink_edata_car7	7x7	1.15	20.944.494	90.622.826
Hurink_sdata_car8	8x8	1.00	235.018.496	1.007.106.049
Hurink_sdata_la01	10x5	1.00	9.709.952	29.609.025
Hurink_edata_la01	10x5	1.15	44.678.169	154.143.493
Hurink_sdata_la02	10x5	1.00	11.018.176	33.899.137
Hurink_edata_la02	10x5	1.15	38.742.965	133.536.014
Hurink_sdata_la03	10x5	1.00	7.694.720	23.422.209
Hurink_edata_la03	10x5	1.15	30.685.423	105.835.602
Hurink_sdata_la04	10x5	1.00	8.495.232	25.624.577
Hurink_edata_la04	10x5	1.15	23.228.017	79.958.694
Hurink_sdata_la05	10x5	1.00	12.766.336	39.199.041
Hurink_edata_la05	10x5	1.15	46.370.480	162.824.604

4. AN HEURISTIC FOR THE BLOCKING FJSSP

Since it was possible to find the state space for the instances, a solution was sought for them. A heuristic was chosen based on Alves et al. (2019) with some improvements to deal with temporal feasibility and is called Heuristic Time Minimization (HTM). The HTM principle is to prioritize controllable over non-controllable events so that the branch-factor is significantly reduced. Another procedure is that whenever a path reaches a state already visited, it keeps the one with the shortest time, eliminating all other branchings.

This heuristic was applied to some instances and the results are presented in Table 5. The first and second columns identify the instance. The third and fourth columns show the size of the supervisor obtained. Next, the processing time of the heuristic applied over such supervisor is presented and the makespan found. The last two columns present the value of makespan presented in the literature. It can be noticed that there are many empty cells, meaning that there are no known results for such instances. In some cases (instances marked with and *), we used a computer with 256 GB of RAM.

The heuristic had a better performance than Mascis and Pacciarelli (2002) results and worse performance than Pranzo and Pacciarelli (2016) results. These and other works present results for larger instances, such as (10 × 10), which were not possible to compute with the proposed methodology.

The execution time of the heuristic ranges from a few seconds for instances of up to 41 millions of states to a few hours for instances around 400 millions of states. This is justified by the large number of evaluations that the algorithm has to perform given the number of paths in the supervisor.

Table 5. Makespan results for Blocking FJSSP

Instance	Dimension	States	Transitions	Execution time (s)	Makespan		
					HTM	**Pranzo & Pacciarelli (2016)**	**Mascis & Pacciarelli (2002)**
Hurink_sdata_mt06	6x6	346.328	1.085.803	0,11	**69**		74
Hurink_sdata_car1*	11x5	121.529.344	394.521.601	7.654,70	**7.409**		
Hurink_sdata_car2*	13x4	111.562.752	314.163.201	22.934,44	**7.503**		
Hurink_sdata_car3	12x5	383.021.056	1.253.793.793	Unable to find a solution			
Hurink_sdata_car5*	10x6	249.672.704	915.118.081	1.513,88	**8.218**		
Hurink_sdata_car7	7x7	10.787.520	40.741.569	6,29	**6.788**		
Hurink_sdata_car8*	8x8	235.018.496	1.007.106.049	86,85	**8.585**		
Hurink_sdata_la01	10x5	9.709.952	29.609.025	0,59	965	**881**	1066
Hurink_sdata_la02	10x5	11.018.176	33.899.137	1,16	944	**900**	1077
Hurink_sdata_la03	10x5	7.694.720	23.422.209	2,19	821	**808**	884
Hurink_sdata_la04	10x5	8.495.232	25.624.577	0,50	889	**862**	881
Hurink_sdata_la05	10x5	12.766.336	39.199.041	0,53	803	**742**	995
Hurink_sdata_la06	15x5	Unable to find a supervisor					

5. CONCLUSIONS

This work showed that it is possible to use the Supervisory Control Theory to represent Job Shop Scheduling problems. An algorithm to automatically translate a Blocking Flexible Job Shop Scheduling Problem to automata using the Supervisory Control Theory was developed and supervisors are generated for a number of instances in the literature.

The full search space is within the supervisor and other heuristics may be applied over it, with the advantage that no back-forward search is ever going to be necessary. The modeling proposed can be extended to other classes of JSSP problems.

To illustrate the use of the search space, we applied a heuristic over the supervisor and we were able to find optimized sequences for some problems and they turned out to be close to the results in the literature.

The search for other heuristics that may provide better results is left for future work. Also is left for future work, the search for a different way of storing the transition structure in order to decrease the memory usage and the processing time, expanding the computation limits.

REFERENCES

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

Alves, L.V.R., Pena, P.N., and Takahashi, R.H.C. (2019). Planning on Discrete Event Systems Using Parallelism Maximization. *arXiv e-prints*, arXiv:1912.12985. Https://arxiv.org/abs/1912.12985.

Alves, L.V., Martins, L.R., and Pena, P.N. (2017). Ultrades-a library for modeling, analysis and control of discrete event systems. *IFAC-PapersOnLine*, 50(1), 5831–5836.

Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on computing*, 3(2), 149–156.

Baker, K.R. and Trietsch, D. (2013). *Principles of sequencing and scheduling.* John Wiley & Sons.

Behnke, D. and Geiger, M.J. (2012). Test instances for the flexible job shop scheduling problem with work centers. *Logistics Management Department, Hamburg, Germany.*

Brandimarte, P. (1993). Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations research*, 41(3), 157–183.

Brucker, P., Jurisch, B., and Sievers, B. (1994). A branch and bound algorithm for the job-shop scheduling problem. *Discrete applied mathematics*, 49(1-3), 107–127.

Carlier, J. (1978). Ordonnancements a contraintes disjonctives. *RAIRO-Operations Research*, 12(4), 333–350.

Chambers, J.B. and Barnes, J.W. (1996). Tabu search for the flexible-routing job shop problem. *Graduate program in Operations Research and Industrial Engineering, The University of Texas at Austin, Technical Report Series, ORP96-10.*

Dauzère-Pérès, S. and Paulli, J. (1997). An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research*, 70, 281–306.

Hopcroft, J.E., Motwani, R., and Ullman, J.D. (2001). Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1), 60–65.

Hurink, J., Jurisch, B., and Thole, M. (1994). Tabu search for the job-shop scheduling problem with multi-purpose machines. *Operations-Research-Spektrum*, 15(4), 205–215.

Lawrence, S. (1984). An experimental investigation of heuristic scheduling techniques. *Supplement to resource constrained project scheduling.*

Louaqad, S. and Kamach, O. (2015). Scheduling of blocking and no wait job shop problems in robotic cells.

Mascis, A. and Pacciarelli, D. (2002). Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research*, 143(3), 498–517.

Muth, J.F., Thompson, G.L., and Winters, P.R. (1963). *Industrial scheduling.* Prentice-Hall.

Pinedo, M.L. (2016). *Scheduling: theory, algorithms, and systems.* Springer, New York, New York, 5^{a} edition.

Pranzo, M. and Pacciarelli, D. (2016). An iterated greedy metaheuristic for the blocking job shop scheduling problem. *Journal of Heuristics*, 22(4), 587–611.

Ramadge, P.J. and Wonham, W.M. (1989). The control of discrete event systems. *Proceedings of IEEE*, 77, 81–98.

Van Laarhoven, P.J., Aarts, E.H., and Lenstra, J.K. (1992). Job shop scheduling by simulated annealing. *Operations research*, 40(1), 113–125.