algorithms

MDPI

Article

A Biased-Randomized Discrete-Event Algorithm for the Hybrid Flow Shop Problem with Time Dependencies and Priority Constraints

Christoph Laroque [1], Madlene Leißau [1], Pedro Copado [2], Christin Schumacher [3], Javier Panadero [2] and Angel A. Juan [4,*]

[1] Institute for Management and Information, University of Applied Sciences Zwickau, 08056 Zwickau, Germany; christoph.laroque@fh-zwickau.de (C.L.); madlene.leissau.gel@fh-zwickau.de (M.L.)
[2] IN3—Computer Science Department, Universitat Oberta de Catalunya, 08018 Barcelona, Spain; pcopadom@uoc.edu (P.C.); jpanaderom@uoc.edu (J.P.)
[3] Informatik 4—Modeling and Simulation, TU Dortmund University, 44227 Dortmund, Germany; christin.schumacher@cs.tu-dortmund.de
[4] Department of Applied Statistics and Operations Research, Universitat Politècnica de València, 03801 Alcoy, Spain
* Correspondence: ajuanp@upv.es

Abstract: Based on a real-world application in the semiconductor industry, this article models and discusses a hybrid flow shop problem with time dependencies and priority constraints. The analyzed problem considers a production where a large number of heterogeneous jobs are processed by a number of machines. The route that each job has to follow depends upon its type, and, in addition, some machines require that a number of jobs are combined in batches before starting their processing. The hybrid flow model is also subject to a global priority rule and a "same setup" rule. The primary goal of this study was to find a solution set (permutation of jobs) that minimizes the production makespan. While simulation models are frequently employed to model these time-dependent flow shop systems, an optimization component is needed in order to generate high-quality solution sets. In this study, a novel algorithm is proposed to deal with the complexity of the underlying system. Our algorithm combines biased-randomization techniques with a discrete-event heuristic, which allows us to model dependencies caused by batching and different paths of jobs efficiently in a near-natural way. As shown in a series of numerical experiments, the proposed simulation-optimization algorithm can find solutions that significantly outperform those provided by employing state-of-the-art simulation software.

Keywords: machine scheduling; hybrid flow shop; priority; batching; based-randomization; discrete-event heuristics

Citation: Laroque, C.; Leißau, M.; Copado, P.; Schumacher, C.; Panadero, J.; Juan, A.A. A Biased-Randomized Discrete-Event Algorithm for the Hybrid Flow Shop Problem with Time Dependencies and Priority Constraints. *Algorithms* **2022**, *15*, 54. https://doi.org/10.3390/a15020054

Academic Editor: Maciej Drozdowski

Received: 30 December 2021
Accepted: 27 January 2022
Published: 2 February 2022

1. Introduction

Increasing competition and the associated intensification of global business are characteristic factors for modern manufacturing companies. Other factors, such as the ongoing digital transformation and a growing demand for customized products have to be considered as well. An economically optimal utilization of the production chains was often considered as the essential goal in the past. During the last years, however, this goal has increasingly shifted towards a more customer-oriented production. Accordingly, the focus of all companies is now on the time feasibility and adherence to promised delivery dates, which confronts operational production planning with questions about the latest possible release of a certain batch, so that it can still be manufactured and delivered on time. A constantly increasing complexity of production systems, in conjunction with a high degree of automation, repeatedly poses challenges for many production companies. As

a result, there is an increasing need to use optimization algorithms in practice in order to handle the complexity of planning problems. This is especially the case in the use case of this study, which comes from the field of semiconductor manufacturing. The use of simulation-optimization approaches [1] could provide a differentiated answer. In planning, design, and ramp-up, these methods are well established but are not frequently used yet for operational support in decision-making.

In our real-life scheduling application, and other production use cases, it is possible for jobs to take different paths through the production system. The specific path might depend upon a particular specification or parameter, which defines the machines to be visited by the job. The existence of different paths through the production system might cause time dependencies, so the order in which the jobs leave the system might be different from the order in which the jobs entered it. Consequently, this can lead to problems in the context of the planning of batch-related insertion dates, especially if batch processes are also found within the production system. Therefore, we need a simulation to model our system. In this study, a multi-path version of the hybrid flow shop scheduling problem [2] was analyzed. This version also considers two batch processes at the same time, and it is based on the real case from a German manufacturing industrial partner. Furthermore, a global priority rule and a "same setup" rule were considered. The before-mentioned elements make the computation of the makespan a non-trivial task: due to the existence of time-dependencies and batches, the makespan cannot be computed using a closed analytical expression anymore.

Accordingly, the main contributions of this study are described next: (i) a simulation model of a hybrid flow shop problem with pre-determined paths, which are predefined by individual parameters and specifications and batching; (ii) a fast discrete-event heuristic that is able to deal with the complexity of the modeled system and compute the makespan associated with a proposed solution; (iii) the extension of the previous heuristic to a biased-randomized algorithm [3], which introduces some degree of randomness into the heuristic constructive process; and (iv) the evaluation of computational experiments, which show the performance of the proposed methodology for solving this scheduling problem. Concepts of discrete-event simulation and heuristic algorithms can be combined into discrete-event driven heuristics. Some typical applications include scenarios in which synchronization issues are relevant [4] or in which rare events have to be modeled [5,6]. Furthermore, biased-randomized algorithms may generate alternative solutions based on heuristic dispatching rules. By making use of Monte Carlo simulation and skewed probability distributions, biased-randomized techniques introduce a non-uniform random behavior into a dispatching rule. Employing parallelization techniques, these heuristics can be applied in the same computational time as the original dispatching rule, thus making these algorithms much more powerful than the original heuristics they build upon. In the literature, applications of these algorithms can be found in flow shop problems [7]. Despite its importance in the semiconductor industry, to the best of our knowledge, a flow shop problem such as the one described here has never been solved in the related literature.

The article is organized as follows. Section 2 provides a more-detailed description of the scheduling problem studied in this article. Section 3 reviews related work on similar flow shop problems. Section 4 describes the algorithms proposed in this work in as much detail as possible, so they can be reproduced by other researchers or practitioners. Section 5 carries out a series of numerical experiments to test our methodology and analyzes the obtained results. Finally, Section 6 summarizes the main findings of this study and points out some future research lines.

2. A Detailed Description of the Problem

This section describes a hybrid flow shop problem with realistic constraints. The example described here is based on the specifics and constraints of a pre-assembly process in semiconductor manufacturing, and notice that these constraints commonly appear in semiconductor manufacturing in general.

In parallel with the realistic use case, in this section, we formally categorize our problem. Therefore, we use the notation first introduced by [8]. Using the scheme, any scheduling problem can be described by means of a parameter tuple $(\alpha|\beta|\gamma)$ [9]. The parameter α defines the number, the type, and the arrangement of the machines and stages. The second parameter, β, can contain any number of entries. The entries are separated by commas. β represents characteristic features and restrictions of the production process. The last expression, γ, within the tuple stands for the objective function. Single objective functions, combinations of objective functions in a mathematical expression, or different tested objective functions can be specified here [9].

Due to production systems getting more complex and specified during the time, Graham's notation [8] was further specified by [9,10] with additional production environments (α), constraints (β), and objectives (γ). In this section, elements of their notations are also utilized to characterize our problem. Further, we explain the specifics of the use case and characterize the components of the problem by Graham's notation in brackets.

Figure 1 shows the problem we are considering. The production contains 10 machines and 2 batch processes. Flow shops with multiple machines on one of the processing stages are referred to as hybrid or flexible flow shops (FHM) [9,10]. Hybrid flow shop environments with machines that need different types (or a specified amount) of jobs to start their production can be categorized as assembly flow shops [11,12]. In contrast, hybrid flow shops with machines that may process jobs of different types simultaneously (see machine $Ba1$ and $Ba2$ in Figure 1) are referred to as FHM with batching machines ($batch$) [9,13].

The described machines $i \in M$ process jobs $j \in N$ of different product types $t_j \in T$ or families (in Graham's notation known as $fmls$) [9], where $M = M1, ..., M10, Ba1, Ba2$. The four possible product types t_j are $T = A1, A2, B1, B2$. Depending on the product type, the jobs are processed along a specific route, which consists in a defined sequence of machines.

While the jobs j are all processed on machine M1 after being loaded into the modeled system, the jobs j are subsequently divided into different paths depending on their product types t_j. On the one hand, jobs of product types $j \in A1, A2$ are processed by machines $M2$, $M4$, and $M5$, while jobs of product type $j \in A2$ are processed on machine $M9$. Jobs of product types $B1$ and $B2$ ($j \in A1, A2$) are not processed on machine $M2$ but on machine $M3$ and, subsequently, by machine $M6$ in the case of product type $B1$. In the case of product type $B2$, they are processed by machines $M7$ and $M8$. Our model includes machine qualifications (M_j). In contrast, for example, jobs of type $j \in A1$ skip machine $M9$. Accordingly, the model contains the constraint of skipping stages ($skip$). According to a load of jobs with different product types—and thus different processing sequences—it is possible that jobs are processed at different speeds through the flow production scenario.

In general, a global-priority-rule procedure applies to the modeled system. Accordingly, the orders of product type $A2$ are prioritized, i.e., orders $A2$ (if available) are always given priority on a machine ($prec$). In addition, a "same setup" priority rule applies to the entire flow production scenario (apart from the batch processes). This rule sorts the jobs in the queue depending on the setup status of the machines in order to keep setup times and their influence on the makespan to a minimum. These constraints are based on a "real-world use-case" from the field of semiconductor manufacturing and are depicted in a simplified form in this model. Accordingly, in semiconductor manufacturing, countless product types are often fed into the system, which requires a special setup on the machines. In order to shorten the setup times and subsequently the overall makespan of the jobs, jobs of the same product type are given priority if possible (same setup). The prioritization of a specific product type (in this case, product type $A2$) is also used to prioritize product types with a high order volume, high makespans, and/or complex setups.

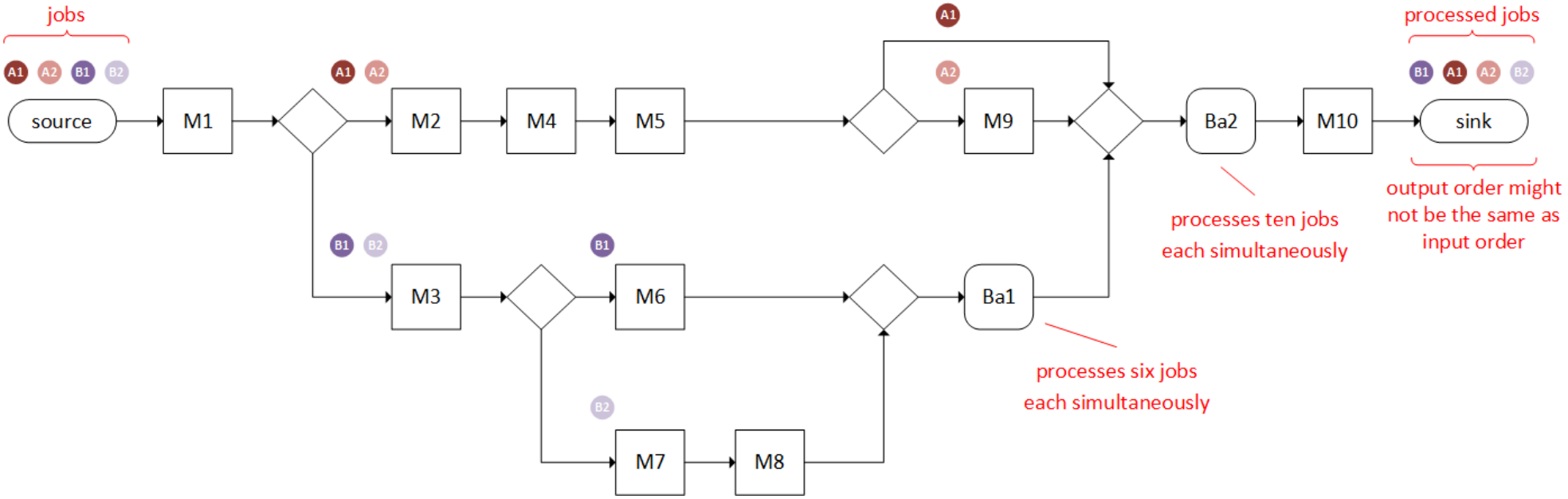

Figure 1. A simple example of the considered flow shop problem.

Unlike other machines, the machines $i \in Ba1, Ba2$ process several jobs j simultaneously. Hence, they are referred to as batch machines, whereby the product types are negligible. Accordingly, six jobs j are processed simultaneously on machine $Ba1$. The processing of the job j begins as soon as a total of six jobs of product types $B1$ and $B2$ are waiting in the machine. The ratio of jobs j of product type $j \in B1$ or product type $j \in B2$ is irrelevant. On machine $Ba2$, a total of ten jobs is processed simultaneously, whereby these can be jobs of product types $A1$, $A2$, $B1$, and $B2$ [13].

The objective of the study was to find a permutation of jobs (solution) that reduces the makespan C_{max} (which is the last component of Graham's notation). In Graham's notation [8], which was extend by [9,10], this problem can be completely formalized as:

$$FHM \mid M_j,\ batch,\ prec,\ skip,\ fmls \mid C_{max}.$$

3. Related Work

Flow shop problems have been studied since the early 1950s [14]. Several authors have employed mixed-integer programming, heuristics, and discrete-event simulation to deal with hybrid flow shops in different application areas [13,15,16]. The flow shop problem is NP-hard even for a problem with two-states, two identical parallel machines on one of both stages, and one machine on the other stage [17]. Hence, the problem we analyzed in this study is classified as NP-hard. In contrast to exact optimization methods, heuristics cannot verify the optimality of a solution. However, they can provide high-quality (or even near-optimal) solutions in short computational times, something that cannot be generally achieved with exact methods [18,19].

Several reviews on assembly flow shops can be found in the literature [11,12]. A review of hybrid flow shops with the integration of batching components is provided in [13]. As stated by some authors, the makespan and time-based objectives dominate the literature in flow shop environments [11,15]. Hence, the makespan is one of the most relevant objectives both in theoretical as well as in applied works.

In the hybrid flow shop literature, two possible expressions of priority sets can be observed. First, there is a fixed defined ranking of priority groups such as $P_1 = \{j_1\}$, $P_2 = \{j_2, j_3\}$, $P_3 = \{j_4\}$. Secondly, for each job j, a set of preceding jobs P_j is defined [20].

Some studies consider hybrid flow shop problems with batching machines, priority rules, and makespan objectives [20]. In the aforementioned work, a complex and realistic flow shop problem with changeover times, machine qualifications, and priority rules was analyzed. To deal with this problem, the authors have extended the NEH [21] considering machine qualification, skipping stages, and rule priorities. The authors concluded that the NEH heuristic provides good results for this type of flow shop problem.

Additionally, for the hybrid flow shop problem with setup times, machine qualifications, delayed machine availability, lag times between the stages, and the makespan objective, other authors compared the priority rules of the "shortest processing time," the NEH, and the "longest processing time", concluding that the NEH offers the best results [22]. A genetic algorithm for a hybrid flow shop with batching machines was introduced in [23]. In addition, the latter work allows jobs to skip stages. The initial solution for the genetic algorithm was generated by a "longest processing time" dispatching rule, followed by the "earliest completion time" heuristic for the following stages. A tabu search has also been proposed for a hybrid flow shop in [24]. This algorithm swaps two jobs in each iteration. The tabu list tracks pairs of jobs that were swapped in previous iterations to avoid loops. Three dispatching rules were compared for generating the initial solution: two variations of the "longest processing time," with the sum of the processing times over all stages per order, and an alphanumeric ordering. As a result, the authors showed that none of these strategies performs significantly better than the other.

A "shortest processing time" heuristic for a two-stage hybrid flow shop problem with polynomial runtime was introduced in [25]. Here, jobs can only be processed by one dedicated machine on stage one. In the following stages, all jobs are processed on

one machine in the batch processing mode. The "shortest processing time" heuristic was employed for allocating the jobs on the first stage, and a combination of the "earliest completion time" and a minimization of the processing times for all batches was employed on the second stage. Due to the high complexity of the problem considered in our study, a large number of possible permutations need to be explored. As mentioned before, simulation can be useful for modeling, but it needs to be combined with optimization components in order to generate high-quality or even near-optimal solutions.

4. A Biased-Randomized Discrete-Event Algorithm

In order to solve the hybrid flow-shop problem with batch processes, setup times, and priorities, a multi-start approach is proposed in Algorithm 1. This multi-start method calls a biased-randomized heuristic (Algorithm 2) that makes use of the NEH heuristic. As discussed above, variations of this heuristic can provide good results for many hybrid flow shops [6]. The original NEH was designed to minimize the makespan in a simple flow shop environment.

Due to the complexity of the flow shop system considered in this work, we made use of an original discrete-event based heuristic to compute the makespan. This procedure is outlined in Algorithm 3. Our approach, which is depicted in Algorithm 1, receives as input parameters a job list *jobsList* (where $jobsList[i]$ is the job situated in the $i-th$ position in the sequence), some parameters *params*, and the β parameter ($\beta \in (0,1]$) of the geometric distribution employed to induce the biased-randomization effect [26]. This algorithm works as follows: firstly, the initial solution *best* is generated by means of the heuristic STNEH ($\beta = 1$) at line 2. In the main loop, the algorithm iterates until the termination criterion is not reached. For each iteration, a new solution *sol* is generated by using the biased-randomized version of the STNEH heuristic, BR-STNEH. This is achieved by setting β in the interval $(0,1)$, as explained in [27]. If the makespan of *sol* is lower than solution *best*, then *best* is replaced by *sol*; otherwise *sol* is discarded.

Concerning the biased-randomized algorithm BR-STNEH (Algorithm 2), it chooses in each iteration the next job according to a two-level criterion: the setup times in the first level and the processing times in the second-level. Consider the following parameters: the jobs list *jobList*, the β parameter, the job processing times pT, job setup times sT, a list of job types *typesPriority*, and a non-negative integer mL. The algorithm starts by listing the job types $typeJob \in \{A1, A2, B1, B2\}$ to split *jobsList* into a list of jobs of type *typejob* called $jobs_{typejob}$. The NEH heuristic is applied to each list $jobs_{typejobs}$ and collected in a list of lists *typesJobs* (lines 3–7). In line 8, *typesJobs* is sorted in descending order according to setup times. This is done because the setup time occurs mainly when two jobs, of different types, are processed by the same machine consecutively. Hence, if we classify the jobs by type, and then sort by the setup time sT, we can expect a reduction in the number of times that two jobs of different types are processed by the same machine. In lines 9–12, those jobs that are a priority ($typejob \in typesPriority$) must be scheduled at the beginning of the *jobsList*. The main loop iterates until all jobs are scheduled. For each iteration, the algorithm picks a type *typeJob* from *typeJobs* list and gets randomly an integer number *lgth* between 1 and mL, i.e., the latter fixes the number of iterations of the inner loop. In the inner loop, a random job is selected from the list $jobsList_{typeJob}$ and appended to the partial solution in each turn (lines 16–21). Finally, the complete solution *jobsList* is returned. This procedure is then extended into a biased-randomized algorithm by introducing the geometric distribution $G(\beta)$ with parameter $\beta \in (0,1)$.

Algorithm 1 Multi-Start Framework

Multi-Start($jobsList$, β, $params$)
$best \leftarrow$ **BR-STNEH**(jL,$\beta = 1$, $params$)
while end criteria not reached **do**
  $sol \leftarrow$ **BR-STNEH**(jL, β, $params$)
  if makespan(sol,$params$) $<$ **makespan**($best$,$params$) **then**
    $best \leftarrow sol$
  end if
end while
return $best$

Algorithm 2 Biased Randomized STNEH

BR-STNEH($jobsList$, β, pT, sT, $typesPriority$, mL)
$typesJobs \leftarrow emptyList$
for all $typeJob \in types(jobsList)$ **do**
  $jobs_{typeJob} \leftarrow \{job \mid \forall job \in jobsList : type(job) = typeJob\}$
  $jobs_{typeJob} \leftarrow$ SortingByNEH($jobs_{typeJob}$, pT)
  $typesJobs \leftarrow$ append($typesJobs$,$jobs_{typeJob}$)
end for
$typesJobs \leftarrow$ SortingBySetupTimes($typesJobs$,sT)
for all $typeJob \in typesPriority$ **do**
  $jobsList \leftarrow$ extend($jobsList$,$jobs_{typeJob}$)
  $typesJobs \leftarrow$ remove($typesJobs$,$jobs_{typeJob}$)
end for
while $jobsList$ is not complete **do**
  $typeJob \leftarrow$ pickList($typesJobs$,β)
  $lgth \leftarrow$ randomInt(1,mL)
  while $lgth > 0$ **do**
    $job \leftarrow$ pickJob($jobs_{typeJob}$,β)
    $jobsList \leftarrow$ append($jobsList$,job)
    $jobs_{typeJob} \leftarrow$ remove($jobs_{typeJob}$,job)
    $lgth \leftarrow lgth - 1$
  end while
  if $jobs_{typeJob} = emptylist$ **then**
    $typesJobs \leftarrow$ remove($typesJobs$,$jobs_{typeJob}$)
  end if
end while
return $jobsList$

In order to compute the makespan, we developed a discrete-event based procedure. The idea of this methodology is to use a discrete-event list to manage complex time dependencies that arise as events occur over time [4]. This event list is constantly sorted, taking into account the chronological order of each event (e.g., the assignment of a job to an available machine or the arrival of a job to a batch queue), and it is iteratively processed until no events are left. Making all decisions in chronological order avoids complex and time-consuming back-rolling issues. Hence, when a new event is scheduled , it is chronologically inserted into a list *eventList*. The makespan is iteratively computed as this *eventList* is processed, and others are scheduled, following a chronological order. Algorithm 3 shows an overview of the proposed discrete-event procedure. The algorithm receives as input parameters the *jobsList*, which is the complete list of jobs in the specific order that will enter in the system to be processed the machine-dependent processing times of each job (pT) and the setup times of each job-machine (sT). The algorithm requires auxiliary data structures, such as: $availableTime[m]$ (i.e., at which time a machine m will be available), $previousTypeMachine[m]$ (which keeps track of the machine that previously processed the job), $machinesTypes[m]$ (which stands for the type $typeJob$ that machine m can process),

and $jobsWaiting[m]$ (which stores a set of jobs in the batch m). In the beginning, all jobs are being processed in machine 1, in concordance with the scheduling $jobsList$. Therefore, the algorithm is initialized, creating ending events at machine 1 and inserting them into the event list $eventList$ (lines 9 to 16). In line 17, $eventList$ is sorted by time t of occurrence. The main loop iterates until $eventList$ is empty.

Algorithm 3 Discrete-Event Makespan Method

```
1:  makespam(jobsList,pT,sT)
2:  availableTime ← initAvailableTimes()
3:  previousTypeMachine ← initPreviousTypeMachine()
4:  machinesType ← initMachinesType()
5:  jobsWaiting ← initJobsWaiting()
6:  eventList ← emptyList
7:  mkSpn ← 0
8:  for all job ∈ jobsList do
9:     setupTime ← 0
10:    if type(job) ≠ previousTypeMachine['machine1'] then
11:       setupTime ← sT['machine1'][job]
12:    end if
13:    t ← pT['machine1'][job] + setupTime
14:    event ← createEndingEvent(t,job,'machine1')
15:    eventList ← append(eventList,event)
16: end for
17: eventList ← sortingByTime(eventList)
18: while eventList ≠ emptylist do
19:    event ← first(eventList)
20:    if isStarting(event) then
21:       processingStartingEvent(event, pT, sT, availableTime, previousTypeMachine,
          machineTypes,eventList)
22:    else
23:       processingEndingEvent(event, pT, sT, availableTime, previousTypeMachine,
          machineTypes, WaitingJobs,eventList)
24:    end if
25:    mkSpn ← time(event)
26: end while
27: return mkSpn
```

At each iteration, the next event, $event$, is selected, which can belong to one of the following types: job starts in a machine (starting-event) or job ends in a machine (ending-event):

- In case of a starting-event, job job is just placed at the entrance of machine j at time t. Algorithm 4) proceed as follows: job job, machine $machine$, time t, and type $typeJob$ are recovered in lines 2–4; in line 6, we check whether job belongs to the $machine$ path; if it does not, then a new ending-event is created with the same job, machine, and time (lines 18–19); otherwise, the algorithm advances the clock time t; next, the algorithm determines the $setupTime$, which is set to 0 if the previous job type and the current one match; otherwise, $sT[machine][job]$; hence, the new time t' is set to previous time t plus the job-processing time $pT[machine][job]$ and the $setupTime$ (lines 7–14); finally, we create a new ending-event $nEvent$ with the same job and machine but at time t'; this event is insert into the event list.

Algorithm 4 Processing Starting Events

```
1: processingStartingEvent(event, pT, sT, availableT, previousTypeMachine,
   machineTypes,eventList)
2: job ← job(event)
3: typeJob ← type(job)
4: machine ← machine(event)
5: t ← time(event)
6: if job ∈ machineType[typeJob] then
7:    setupTime ← 0
8:    previousType ← previousTypeMachine[machine]
9:    if previousType ≠ typeJob then
10:      setupTime ← sT[machine][job]
11:   end if
12:   t′ ← t + pT[machine][job] + setupTime
13:   previousTypeMachine[machine] ← typeJob
14:   availableTime[machine] ← t′
15:   nEvent ← createEndingEvent(t′,job,machine)
16:   eventList ← insertEventSorted(eventList,nEvent)
17: else
18:   nEvent ← createStartingEvent(t,job,machine)
19:   eventList ← insertEventSorted(eventList,nEvent)
20: end if
```

- In case of an ending-event (Algorithm 5), like a job *job* of type *typeJob* leaving machine *machine* at time *t*, the algorithm works as follows: initially, job, type, and time are obtained from event in lines 2–3; in line 4, this machine release time is computed with the maximum between the current time *t* and the current machine release time *availableTime*[*machine*]; then, the algorithm verifies whether the job belongs to this machine path; if it does not, then a new starting-event is created with the same job and time but in the next machine (lines 18–19); otherwise, it continues to the next step; in the next step, our algorithm assumes that the size of all batches is set to 1 except for *Ba*1 and *Ba*2; hence, the current job is added to the list of jobs in the current machine *waitingJobs* (lines 8–9); in lines 10–16, the algorithm checks if the number of jobs in this machine reaches the batch size, in which case a new starting event *nvEvent* is scheduled at time *t* for each job *job*′ in the *waitingJobs* (these new events are inserted into the *eventList*).

Finally, when *eventList* is empty (line 18 in Algorithm 3, the algorithm finishes returning the time *mkSpn* of the last event processed, which is an ending-event of the last job in the last machine. This time is the makespan of the processed *jobsList*.

Algorithm 5 Processing Ending Events

processingEndingEvent(*event*, *pT*, *sT*, *availableT*, *previousTypeMachine*, *machineTypes*, *waitingJobs*,*eventList*)
job ← job(*event*)
typeJob ← type(*job*)
machine ← machine(*event*)
t ← time(*event*)
availableTime[*machine*] ← *max*(*t*, *availableTime*[*machine*])
if *machine* ∈ *machineTypes*[*typeJob*] **then**
  nbJobsMachine[*machine*] ← *nbJobsMachine*[*machine*] + 1
  waitingJobs[*machine*] ← add(*waitingJobs*[*machine*],*job*)
  if *nbJobsMachine*[*machine*] = *sizeBatch*(*machine*) **then**
    nbJobsMachine[*machine*] ← 0
    for *job*′ ∈ *waitingJobs*[*machine*] **do**
      nEvent ← createStartingEvent(*t*,*job*′,nextMachine(*machine*))
      eventList ← insertEventSorted(*eventList*,*nEvent*)
    end for
  end if
else
  nEvent ← createStartingEvent(*t*,*job*,nextMachine(*machine*))
  eventList ← insertEventSorted(*eventList*,*nEvent*)
end if

5. Computational Experiments

The proposed algorithm was implemented using Python 3.7. All experiments were run in a computer with an Intel Xeon E5-2650 v4 with 32GB of RAM. As far as we know, there are no public instances for the studied problem. Hence, we generated a set of instances, which are available at https://www.researchgate.net/publication/356874077_instances_flowShop (access on 1 February 2022). This set consists of 20 instances, and it is based on the production scenario defined in Section 2, which includes ten processing machines and two additional batch machines, distributed in four different paths. A total of four different types of jobs were considered, each type being processed on a specific path. The instances were identified following the nomenclature *j_m_y*, where *j* is the sum of jobs to be processed; *m* defines the sum of machines (including both regular and batching machines); and *y* is a sequential number, used to identify the instances with the same number of jobs and machines in an easy and comprehensive way. Notice that, depending on the instance, the number of jobs varies between 30 and 1600. We defined unrelated and machine-dependent setup times for every job. Thus, every job has different set-up times on every machine. We also defined unrelated and machine-dependent processing times. Hence, for each job *j*, a processing time at each machine *i* was defined. Our algorithms were run considering 60 s of computing time for instances with 30 jobs and 300 s for instances with 200 and 400 jobs, while it employed a maximum of 900 s for instances with 800 jobs or more.

Table 1 shows the results of the algorithm described in Section 4 for the defined instances. The first column identifies the instance. Subsequently, the following two columns present the solutions obtained using the AnyLogic simulation tool, which simulates the production scenario. In that sense, we provided the total makespan of each solution when the jobs are processed in the system using a first-in-first-out (FIFO) strategy and the computation time—in seconds—required to reach them. Using this strategy, the jobs enter in the system without any logical ordering, i.e., following the alphanumerical order as provided in the instances. We provide this information with the aim of validating the quality of our algorithm. Similarly, the next two columns report the makespan, with its corresponding computational time, provided by our approach using the above-mentioned FIFO strategy. The following two columns present the results of our approach using the same strategy combined with biased-randomization techniques. This allows us to enter jobs in the system in a different order at each algorithm iteration. In our case, a geometric

probability distribution, driven by a single parameter β was employed to induce this behavior. The value for this parameter was set after a quick tuning process over a random sample of instances, establishing a good performance whenever β falls between 0.3 and 0.4 (i.e., any random value inside this interval will generate similar results). Specifically, these columns provides the best-found solution (makespan) and the computational time—in seconds—to reach it. Similarly, the next four columns display the makespans and the computational times when the BR-NEH and BR-STNEH sorting strategies were applied. Finally, the last three columns of the table report the gaps for the different strategies with respect to the initial FIFO one.

Our results show that our approach is highly competitive regarding computational times with respect to the Anylogic simulator. Notice that for large instances (e.g., 1600 jobs), our approach provides solutions in times less than 0.2 s using the FIFO deterministic strategy, while Anylogic requires times greater than five seconds of computation to provide the same solutions. Regarding the different sorting methods, Figure 2 summarizes the results provided in Table 1, where the vertical axis of the boxplot represents the gap obtained with respect to the FIFO strategy. Notice also that, after applying biased-randomized techniques on the original FIFO strategy, the average gap outperformed the former by about 5.52% on the average. The original FIFO strategy does not contain any optimization strategy, it just dispatches jobs in the original order. Hence, improvements are achieved as more-intelligent sorting methods are used to dispatch the jobs. In that sense, the BR-NEH sorting method, which is based on sorting the jobs by total processing time in decreasing order, is able to enhance the original heuristic by 13.81% on average. Notice that the setup times of each job–machine pair may induct to lose the logic behavior of the NEH heuristic, since jobs with short processing times and large setup times are not prioritized in the sorted list of jobs. Thus, to address this limitation, the BR-STNEH sorting method also considers the setup times of each job-machine pair during the sorting process, outperforming the BR-NEH by about 2.05% (obtaining an average gap of 15.86% with respect to the initial FIFO strategy). Regarding the computational times, on average, there were no significant differences between the BR-NEH and BR-STNEH sorting methods to reach the best-found solution.

Table 1. Computational results.

	AnyLogic		Det-FIFO		BR-FIFO		BR-NEH		BR-STNEH		GAP (%)		
Instance	**Makespan (1)**	**Time (s)**	**Makespan (2)**	**Time (s)**	**Makespan (3)**	**Time (s)**	**Makespan (4)**	**Time (s)**	**Makespan (5)**	**Time (s)**	**(1)–(3)**	**(1)–(4)**	**(1)–(5)**
30_12_1	619.50	0.01	619.50	0.01	514.50	7.81	511.50	0.50	477.00	21.79	−16.95%	−17.44%	−23.00%
30_12_2	645.00	0.01	645.00	0.01	539.00	7.03	528.00	1.04	493.00	9.82	−16.44%	−18.14%	−23.57%
30_12_3	589.50	0.01	589.50	0.01	489.50	6.74	483.50	7.90	454.50	4.70	−16.97%	−17.98%	−22.90%
30_12_4	911.50	0.01	911.50	0.01	803.00	3.40	806.00	21.09	735.50	4.55	−11.90%	−11.57%	−19.31%
200_12_1	3229.02	1.05	3229.02	0.02	3064.00	79.28	2735.50	0.12	2682.00	129.89	−5.11%	−15.28%	−16.94%
200_12_2	3174.92	1.09	3174.92	0.02	2735.50	145.87	2233.65	74.80	2313.35	11.26	−13.84%	−29.65%	−27.14%
200_12_3	3298.02	1.07	3298.02	0.02	2956.30	64.34	2604.70	1.95	2570.20	0.41	−10.36%	−21.02%	−22.07%
200_12_4	3146.20	1.02	3146.20	0.02	3041.20	86.90	2649.60	0.02	2561.30	102.97	−3.34%	−15.78%	−18.59%
400_12_1	6515.52	1.86	6515.52	0.06	6313.00	77.98	5473.00	6.59	5391.00	0.04	−3.11%	−16.00%	−17.26%
400_12_2	7785.17	1.66	7785.17	0.04	7572.40	284.42	6602.10	6.16	6460.50	0.04	−2.73%	−15.20%	−17.02%
400_12_3	8321.12	1.60	8321.12	0.05	8124.40	43.18	7371.05	32.72	7237.40	2.52	−2.36%	−11.42%	−13.02%
400_12_4	6503.08	1.93	6503.08	0.05	6455.43	163.31	6424.40	0.04	6343.40	231.98	−0.73%	−1.21%	−2.46%
800_12_1	12,263.02	4.28	12,263.02	0.08	12,111.50	218.10	10,597.00	0.95	10,515.00	0.14	−1.24%	−13.59%	−14.25%
800_12_2	15,198.22	3.55	15,198.22	0.09	14,974.25	369.42	12,784.40	3.36	12,614.15	0.08	−1.47%	−15.88%	−17.00%
800_12_3	13,633.32	3.70	13,633.32	0.08	13,444.20	274.19	11,759.50	167.08	11,562.10	0.08	−1.39%	−13.74%	−15.19%
800_12_4	12,253.45	3.44	12,253.45	0.08	12,216.85	272.01	12,233.20	0.08	12,078.70	25.35	−0.30%	−0.17%	−1.43%
1600_12_1	28,735.40	7.00	28,735.40	0.17	28,512.65	312.46	24,465.25	485.94	24,109.60	517.42	−0.78%	−14.86%	−16.10%
1600_12_2	25,785.30	6.96	25,785.30	0.17	25,589.60	313.86	22,248.70	465.15	22,014.70	55.66	−0.76%	−13.72%	−14.62%
1600_12_3	25,800.20	7.76	25,800.20	0.17	25,660.10	314.38	22,248.00	780.75	22,000.90	57.45	−0.54%	−13.77%	−14.73%
1600_12_4	23,188.40	5.72	23,188.40	0.19	23,154.70	278.74	23,233.80	0.25	23,034.30	858.51	−0.15%	0.20%	−0.66%
AVERAGE	10,079.80	2.69	10,079.80	0.07	9913.60	166.17	8899.64	102.82	8782.43	101.73	−5.52%	−13.81%	−15.86%

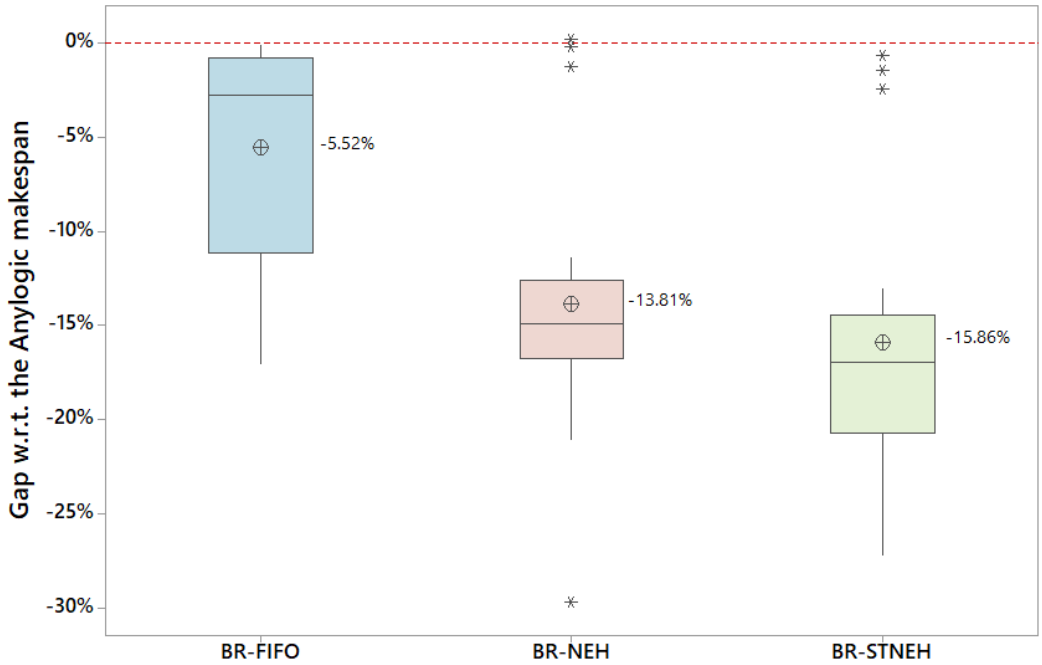

Figure 2. Gaps of the different approaches with respect to the Anylogic results.

6. Conclusions

This study considered a hybrid flow shop problem with time dependencies, batching requirements, and priority rules. The model analyzed is based on a real-life system in the semiconductor industry. Due to the existence of time dependencies, it is not possible to compute the makespan associated with a proposed solution by simply using a analytical expression, and a simulation needs to be carried out each time a new solution is provided. Apart from being time-consuming, using a pure simulation does not allow us to generate high-quality solutions, in general. In order to solve this problem, a fast discrete-event-based heuristic was proposed. This heuristic is capable of proposing a promising solution—one based on a certain constructive logic—while, at the same time, it computes its associated makespan. Moreover, the heuristic is extended into a full probabilistic algorithm by incorporating biased-randomization techniques. Hence, making use of a skewed probability distribution (a geometric one in our case), our biased-randomized discrete-event algorithm is capable of generating high-quality solutions in fast computational times, thus easily outperforming the ones provided by a state-of-the-art simulation software.

Several research lines can be considered for future work, among them: (i) the "longest processing time first" or other priority rules can be used as a basic procedure for the biased-randomized discrete-event algorithm and (ii) our approach could be extended into a full simheuristic [28] to account for scenarios where processing times have a stochastic nature.

Author Contributions: Conceptualization, A.A.J. and C.L.; methodology, A.A.J. and J.P.; software, M.L. and P.C.; validation, M.L., P.C. and C.S.; formal analysis, M.L., P.C. and C.S.; investigation, M.L., P.C., C.S. and J.P.; resources, M.L., P.C., C.S. and J.P.; data curation, M.L., P.C., C.S. and J.P.; writing—original draft preparation, M.L., P.C., C.S. and J.P.; writing—review, C.L. and A.A.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: https://www.researchgate.net/publication/356874077_instances_flowShop (accessed on 29 December 2021).

Acknowledgments: We want to express our gratitude to Dominik Mäckel, for his engagement and support along the project.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Amaran, S.; Sahinidis, N.V.; Sharda, B.; Bury, S.J. Simulation optimization: A review of algorithms and applications. *Ann. Oper. Res.* **2016**, *240*, 351–380. [CrossRef]
2. Tosun, Ö.; Marichelvam, M.; Tosun, N. A literature review on hybrid flow shop scheduling. *Int. J. Adv. Oper. Manag.* **2020**, *12*, 156–194.
3. Juan, A.A.; Corlu, C.G.; Tordecilla, R.D.; de la Torre, R.; Ferrer, A. On the use of biased-randomized algorithms for solving non-smooth optimization problems. *Algorithms* **2020**, *13*, 8. [CrossRef]
4. Fikar, C.; Juan, A.A.; Martinez, E.; Hirsch, P. A discrete-event driven metaheuristic for dynamic home service routing with synchronised trip sharing. *Eur. J. Ind. Eng.* **2016**, *10*, 323–340. [CrossRef]
5. Gholami, M.; Zandieh, M.; Alem-Tabriz, A. Scheduling hybrid flow shop with sequence-dependent setup times and machines with random breakdowns. *Int. J. Adv. Manuf. Technol.* **2009**, *42*, 189–201. [CrossRef]
6. Allaoui, H.; Artiba, A. Integrating simulation and optimization to schedule a hybrid flow shop with maintenance constraints. *Comput. Ind. Eng.* **2004**, *47*, 431–450. [CrossRef]
7. Ferrer, A.; Guimarans, D.; Ramalhinho, H.; Juan, A.A. A BRILS metaheuristic for non-smooth flow-shop problems with failure-risk costs. *Expert Syst. Appl.* **2016**, *44*, 177–186. [CrossRef]
8. Graham, R.L.; Lawler, E.L.; Lenstra, J.K.; Rinnooy Kan, A.H.G. Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Annals of Discrete Mathematics*; Elsevier: Amsterdam, The Netherlands, 1979; Volume 5, pp. 287–326.
9. Pinedo, M. *Scheduling*; Springer: Berlin/Heidelberg, Germany, 2012; Volume 29.
10. Ruiz, R.; Vázquez-Rodríguez, J.A. The hybrid flow shop scheduling problem. *Eur. J. Oper. Res.* **2010**, *205*, 1–18. [CrossRef]
11. Komaki, G.M.; Sheikh, S.; Malakooti, B. Flow shop scheduling problems with assembly operations: A review and new trends. *Int. J. Prod. Res.* **2019**, *57*, 2926–2955. [CrossRef]
12. Nikzad, F.; Rezaeian, J.; Mahdavi, I.; Rastgar, I. Scheduling of multi-component products in a two-stage flexible flow shop. *Appl. Soft Comput.* **2015**, *32*, 132–143. [CrossRef]
13. Morais, M.d.F.; Filho, M.G.; Perassoli Boiko, T.J. Hybrid flow shop scheduling problems involving setup considerations: A literature review and analysis. *Int. J. Ind. Eng.* **2013**, *20*, 614–630.
14. Johnson, S.M. Optimal two-and three-stage production schedules with setup times included. *Nav. Res. Logist. Q.* **1954**, *1*, 61–68. [CrossRef]
15. Lee, T.S.; Loong, Y.T. A review of scheduling problem and resolution methods in flexible flow shop. *Int. J. Ind. Eng. Comput.* **2019**, *10*, 67–88. [CrossRef]
16. Fan, K.; Zhai, Y.; Li, X.; Wang, M. Review and classification of hybrid shop scheduling. *Prod. Eng.* **2018**, *12*, 597–609. [CrossRef]
17. Gupta, J.N. Two-stage, hybrid flowshop scheduling problem. *J. Oper. Res. Soc.* **1988**, *39*, 359–364. [CrossRef]
18. Naderi, B.; Ruiz, R.; Zandieh, M. Algorithms for a realistic variant of flowshop scheduling. *Comput. Oper. Res.* **2010**, *37*, 236–246. [CrossRef]
19. Brucker, P. Scheduling algorithms. *J. Oper. Res. Soc.* **1999**, *50*, 774.
20. Ruiz, R.; Şerifoğlub, F.S.; Urlings, T. Modeling realistic hybrid flexible flowshop scheduling problems. *Comput. Oper. Res.* **2008**, *35*, 1151–1175. [CrossRef]
21. Nawaz, M.; Enscore, E.E., Jr.; Ham, I. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega* **1983**, *11*, 91–95. [CrossRef]
22. Zandieh, M.; Mozaffari, E.; Gholami, M. A robust genetic algorithm for scheduling realistic hybrid flexible flow line problems. *J. Intell. Manuf.* **2010**, *21*, 731–743. [CrossRef]
23. Wilson, A.D.; King, R.E.; Hodgson, T.J. Scheduling non-similar groups on a flow line: Multiple group setups. *Robot.-Comput.-Integr. Manuf.* **2004**, *20*, 505–515. [CrossRef]
24. Logendran, R.; de Szoeke, P.; Barnard, F. Sequence-dependent group scheduling problems in flexible flow shops. *Int. J. Prod. Econ.* **2006**, *102*, 66–86. [CrossRef]
25. He, L.; Sun, S.; Luo, R. A hybrid two-stage flowshop scheduling problem. *Asia-Pac. J. Oper. Res.* **2007**, *24*, 45–56. [CrossRef]
26. Ferone, D.; Gruler, A.; Festa, P.; Juan, A.A. Enhancing and extending the classical GRASP framework with biased randomisation and simulation. *J. Oper. Res. Soc.* **2019**, *70*, 1362–1375. [CrossRef]
27. Ferone, D.; Hatami, S.; González-Neira, E.M.; Juan, A.A.; Festa, P. A biased-randomized iterated local search for the distributed assembly permutation flow-shop problem. *Int. Trans. Oper. Res.* **2020**, *27*, 1368–1391. [CrossRef]
28. Hatami, S.; Calvet, L.; Fernández-Viagas, V.; Framinan, J.M.; Juan, A.A. A simheuristic algorithm to set up starting times in the stochastic parallel flowshop problem. *Simul. Model. Pract. Theory* **2018**, *86*, 55–71. [CrossRef]