Robust Solutions to Job-Shop Scheduling Problems with Operators

Joan Escamilla*, Mario Rodriguez-Molins*, Miguel A. Salido*,
Maria R. Sierra†, Carlos Mencía† and Federico Barber*
* Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia
Valencia, Spain
† Department of Computer Science
University of Oviedo
Gijon, Spain

***Abstract*—The job-shop scheduling problem with operators is an extension of the classical job-shop scheduling problem where each operation has to be assisted by one operator from a limited set of them. We confront this problem with the objective of obtaining robust schedules and minimizing the makespan. In this way the problem becomes more difficult but more interesting from a practical point of view. We propose to solve the problem in three steps. In the first one, the JSP relaxation (without operators) is modeled and solved using a CSOP solver with the objective of minimizing the makespan. Then, the solution is modified to include operators, in this step robustness is introduced by means of a set of buffers in the operators sequences. Finally, these buffers are uniformly distributed among the operations that are not involved in a critical path. We have conducted an experimental study showing that the proposed method reaches a good trade-off between robustness and optimality.**

I. Introduction

The job-shop scheduling problem (JSP) with operators has been recently proposed by Agnetis et al. [1] this problem is denoted $JSO(n,p)$ where n is the number of jobs and p denotes the number of operators. It is motivated by manufacturing processes in which part of the work is done by human operators sharing the same set of tools. The problem is formalized as a classical job-shop problem in which the processing of a task on a given machine requires the assistance of one of p available operators.

In [1] the authors made a thorough study of this problem and established the minimal NP-hard cases. Also, a number of exact and approximate algorithms to cope with this problem were proposed and evaluated on a set of instances generated from that minimal relevant cases. The results of the experimental study reported in [1] make it clear that instances of the $JSO(n,p)$ with 3 jobs, 3 machines, 2 operators and a number of 30 tasks per job may be hard to solve to optimality.

In [13] the authors propose an exact best-first search algorithm and experiment with new instances considering more than 3 jobs and 2 operators. Also, a genetic algorithm is proposed in [8] which reaches near optimal solutions for large instances. The $JSO(n,p)$ with total flow time minimization is considered in [12] where it is solved by means of an exact best first search algorithm and in [7] by means of an depth-first search algorithm. In both cases, some problem dependent heuristics and powerful pruning rules were used.

All the developed techniques are focused on obtaining optimized solutions according to makespan and total flow time. In this paper, we extend the objective for searching robust solutions. It is well-known that real life scheduling problems are dynamic and incidences may occur so that an optimal solution remain unfeasible after the incidence. The main incidence that can occur in a $JSO(n,p)$ is that a task must be delayed due to problems with the associated machine or assigned operator. In this way, our main goal is to find robust and optimized solutions to these problems.

II. Problem Description

Formally the job-shop scheduling problem with operators can be defined as follows. We are given a set of n jobs $\{J_1,\ldots,J_n\}$, a set of m resources or machines $\{R_1,\ldots,R_m\}$ and a set of p operators $\{O_1,\ldots,O_p\}$. Each job J_i consists of a sequence of v_i tasks $(\theta_{i1},\ldots,\theta_{iv_i})$. Each task θ_{il} has a single resource requirement $R_{\theta_{il}}$, an integer duration $p_{\theta_{il}}$ and a start time $st_{\theta_{il}}$ to be determined. A feasible schedule is a complete assignment of starting times and operators to tasks that satisfies the following constraints: (i) the tasks of each job are sequentially scheduled, (ii) each machine can process at most one task at any time, (iii) no preemption is allowed and (iv) each task is assisted by one operator and one operator cannot assist more than one task at a time. The objective is finding a feasible schedule that minimizes the completion time of all the tasks, i.e. the makespan. The significant cases of this problem are those with $p < min(n,m)$, otherwise the problem is a standard job-shop problem denoted as $J||C_{max}$ according to classification scheme proposed in [4].

We use the following disjunctive model for the $JSO(n,p)$. A problem instance is represented by a directed graph $G = (V, A \cup E \cup I \cup O)$. Each node in the set V represents either an actual task, or any of the fictitious tasks introduced with the purpose of giving the graph a particular structure: starting and finishing tasks for each operator i, denoted O_i^{start} and O_i^{end} respectively, and the the dummy tasks *start* and *end*.

The arcs in A are called *conjunctive arcs* and represent precedence constraints among tasks of the same job. The arcs in E are called *disjunctive arcs* and represent capacity constraints. E is partitioned into subsets E_i with $E = \cup_{\{i=1,\ldots,M\}} E_i$. E_i includes an arc (v, w) for each pair of tasks requiring the resource R_i. The set O of *operator arcs* includes three types of arcs: one arc (u, v) for each pair of tasks of the problem, and arcs (O_i^{start}, u) and (u, O_i^{end}) for each operator node and task. The set I includes arcs connecting node *start* to each node O_i^{start} and arcs connecting each node O_i^{end} to node *end*. The arcs are weighted with the processing time of the task at the source node.

From this representation, building a solution can be viewed as a process of fixing disjunctive and operator arcs. A disjunctive arc between tasks u and v gets fixed when one of (u, v) or (v, u) is selected and consequently the other one is discarded. An operator arc between u and v is fixed when (u, v), (v, u) or none of them is selected, and fixing the arc (O_i^{start}, u) means discarding (O_i^{start}, v) for any task v other than u. Analogously for (u, O_i^{end}).

Therefore, a feasible schedule S is represented by an acyclic subgraph of G, of the form $G_S = (V, A \cup H \cup I \cup Q)$, where H expresses the processing order of tasks on the machines and Q expresses the sequences of tasks that are assisted by each operator. The makespan is the cost of a *critical path* in G_S. A critical path is a longest cost path from node *start* to node *end*.

Figure 1 shows a solution graph for an instance with 3 jobs, 3 machines and 2 operators. Discontinuous arrows represent operator arcs. So, the sequences of tasks assisted by operators O_1 and O_2 are $(\theta_{21}, \theta_{11}, \theta_{32}, \theta_{12}, \theta_{13})$ and $(\theta_{31}, \theta_{22}, \theta_{23}, \theta_{33})$ respectively. In order to simplify the picture, only the operator arc is drawn when there are two arcs between the same pair of nodes. Continuous arrows represent conjunctive arcs and doted arrows represent disjunctive arcs; in these cases only arcs not overlapping with operator arcs are drawn. In this example, the critical path is given by the sequence $(\theta_{21}, \theta_{11}, \theta_{32}, \theta_{12}, \theta_{33})$, so the makespan is 14.

III. Robustness

Robustness is a common feature in real life problems. Biological life, functional systems, physical objects, etc. [14], persist, i.e. they remain running and maintain their main features despite continuous perturbations, changes, incidences or aggressions. Thus, robustness is a concept related to the *persistence* of the system, of its structure, of its functionality, etc., against external interferences: *A system is robust, if it persists*.

A system designed to perform in an expected environment is "*robust*" if it is able to maintain its functionality under a set of incidences. In our context a solution of a $JSO(n, p)$ is robust if no rescheduling is needed after small changes in the problem.

Intuitively, the notion of robustness is easy to define, but its formalization depends on the system, on its expected functionality and on the particular set of incidences to face up [9]. No general formal definition of robustness has been proposed, except few exceptions or particular cases. Particularly, Kitano [5] mathematically defines the robustness (R) of a system (SYS) with regard to its expected functionality (F) against a set of perturbations (Z), as (in a simplified way):

$$R_{F,Z}^{SYS} = \int_Z p(z) * F(z) dz \tag{1}$$

The application of robustness definitions is highly problem-dependent. Let's apply (1) to $JSO(n, p)$:

- SYS is a solution S of the $JSO(n, p)$, which we want to assess its robustness. Robustness is a concept related to $JSO(n, p)$ solutions, not to $JSO(n, p)$ itself.
- Z is the discrete set of unexpected incidences that are directly related to the start time or the duration of tasks.
- F is the expected functionality of the system. In $JSO(n, p)$, the expected functionality of a solution is its feasibility after the disruption. Here a solution is composed by the start times and duration of all tasks, plus the buffer times allocated between tasks.
- $p(z) = \frac{1}{|z|}, \forall z \in Z$. This is the probability for incidence $z \in Z$. All tasks have the same probability due to no information is given about incidences.

Therefore, the expression (1) becomes:

$$R_{F,Z}^{S} = \sum_Z p(z) * F(z) \tag{2}$$

Where function F is defined, in the case of a $JSO(n, p)$ as:

- $F(z) = 1$ iff only the affected task is modified by z. Thus the buffer assigned to this task can absorb the incidence.
- $F(z) = 0$, iff more tasks are modified by z. This means that the buffer assigned to this task cannot absorb the incidence and it is propagated to the rest of the schedule.

A robust solution is a solution that maintains its feasibility over the whole set of expected incidences. Thus, robustness in $JSO(n, p)$ implies that:

- If the duration of a task is greater than expected, then only its final time will be affected and no other tasks will be delayed.
- If the start time of a task is delayed, then its final time is also delayed but no other tasks will be affected.

Here, we focus our attention on searching for a robust solution with a minimal makespan. To do this, we will assign buffer times to tasks. This is a usual way for introducing robustness in scheduling problems. A buffer is an extra time that is given to a task to absorb small incidences. Due to the fact that the duration of a task is directly dependent of machine and operator involved in this task, the buffer assigned to this task can be used to absorb small incidences in these two components. However, there exists a trade-off between optimality and robustness so buffer times cannot be assigned to all tasks.

Lemma 1. Let a robust schedule with a given makespan. A buffer can be assigned to a task iff this task is not involved in

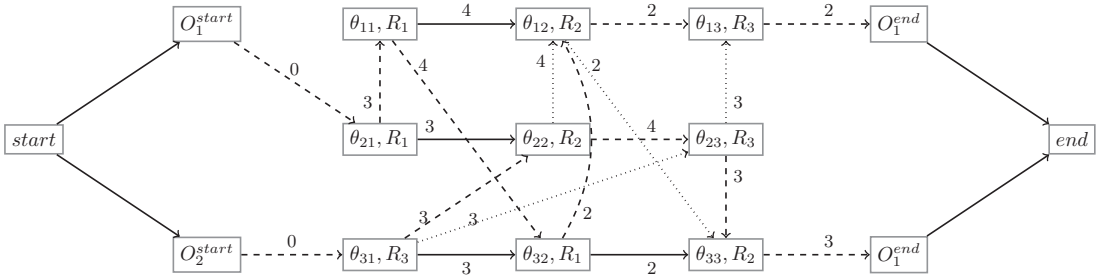

Fig. 1. A feasible schedule to a problem with 3 jobs, 3 machines and 2 operators.

any critical path.

Proof by contradiction

$\rightarrow$ If a buffer is assigned to a task and it is involved in a critical path, then this buffer could be removed to reduce makespan. Contradiction: the initial schedule has minimum makespan.

$\leftarrow$ It is straightforward. If a task is not involved in a critical path and a buffer cannot be assigned, then this task takes part of another critical path. Contradiction: this task is not involved in any critical path.

Thus, we consider that tasks involved in a critical path will not be assigned buffers to avoid increasing the makespan. Thus our main goal is to assign buffers to all tasks that are not involved in critical paths, so that we could achieve the maximum robustness with a given optimality (makespan).

IV. Modeling and Solving a $JSO(n,p)$ in Three Steps: CSOP+PP

The more natural way to solve the Jop-Shop Scheduling Problem with Operators involves all variables and constraints related to jobs, tasks and operators [1][13]. However, the solution obtained is an optimal solution that minimizes the makespan but it does not guarantee a certain level of robustness. Generally, this solution is not able to absorb any incidence and a delay in a task is propagated along the rest of the schedule.

Several reactive/proactive techniques have been developed in the literature to manage incidences in scheduling problems [3]. Thus, computing a new solution from scratch after each problem change is possible (reactive technique), but it has two important drawbacks: inefficiency and instability of the successive solutions [15]. Whilst reactive methods merely deal with the consequences of an unexpected change, taking a more proactive approach may guarantee a certain level of robustness. We are interested in this proactive approach so that our goal is searching for a equitable trade-off between robustness and optimality of a solution.

Robustness (as in section III) in job-shop scheduling can be obtained through allocating buffer times between tasks in order to absorb small disruptions (task delays, etc.) that can occur stochastically along the schedule. In an optimized solution of a $JSO(n,p)$, some natural buffers appear to satisfy the involved constraints (non-overlapping constraints). These buffers give the schedule some robustness degree. However, if more buffers must be included to make more robust the final solution, the involved tasks must be moved and the effect must be propagated to the rest of the schedule. To this end, instead of carrying out this last procedure as a post-process (PP) step, we propose to integrate this procedure with the operator allocation. Thus, we firstly solve the classical JSP and then we apply the buffer and operator allocation procedures as a post-process step.

To add robustness to $JSO(n,p)$ solutions, we divide the problem into two different phases: The first one is related to solve the classical Jop-Shop Scheduling Problem with some of the well-known techniques existing in the literature; and the second phase which is focused on the obtained solution in order to satisfy the operator constraints and to generate buffer times. Thus the final solution is an optimized solution to the $JSO(n,p)$ and it guarantees a certain level of robustness.

A. First Step: Modeling and Solving a Constraint Satisfaction and Optimization Problem

In this approach, we proposed to model the JSP in the first phase as a Constraint Satisfaction and Optimization Problem (CSOP) [2]. Due to the post-process performed, the optimal solution is not necessary since this solution will be changed. But, it is still necessary an optimized solution to try to minimize the makespan. Therefore, a near-optimal solution is looked for by the CSOP model. CSOP is an any-time solver that provides a set of solutions. Each new solution always improves the previous one, until it is the optimal or a time out is reached.

The CSOP model for the JSP is characterized by the following elements:

- A set of variables $x_1, ..., x_n$ associated with the start time of tasks. These variables take values in finite domains $D_1, ..., D_n$ that may be constrained by unary constraints over each variable. In these problems, time is usually assumed discrete, with a problem-dependent granularity.
- A set of constraints $c_1, ..., c_m$ among variables defined on the Cartesian product $D_i \times ... \times D_j$ and restrict the variable domains.
- The objective function is to minimize the makespan.

Two main constraints appear in this kind of job-shop problems:

1) Precedence constraints: the tasks θ_{ij} of each job J_i must be scheduled according to precedence constraints, i.e.,

there exists a partial ordering among the tasks of each job and may be represented by a precedence graph or tree-like structure [11].

2) Capacity constraints: resources cannot be used simultaneously by more than one task. Thus, two different tasks θ_{ij} and θ_{ik} cannot overlap unless they use different resources.

Capacity constraints involve only on type of resources: machines. In real life scheduling problems, the environment is dynamic and disruptions may occur during task execution. In these problems two types of disruptions can be generated: machine disruptions (breakdown, delays due to previously delayed task, etc.), and operator disruptions (late arrival, delays also due to previously delayed task, etc.). However, the modeled CSOP will only manage machines and the next step will also manage operators.

Algorithm 1: Calculate initial values to reduce domain

```
Data: J: set of jobs;
Result: Relative starts to each task and lineal maxtime
maxTime := 0;
cumulative_ij := 0, ∀θ_ij ∈ θ;
foreach i ∈ J do
    cumulativeJob ← {0};
    foreach θ_ij ∈ θ do
        maxTime := maxTime + p_t;
        cumulative_ij ← cumulativeJob;
        cumulativeJob ← cumulativeJob ∪ {p_θij};
return cumulative, maxTime;
```

Algorithm 2: Calculate possible values

```
Data: All tasks
Result: New domain of all θ
pValues_jt ← ∅, ∀θ_ij ∈ θ;
foreach i ∈ J do
    foreach θ_ij ∈ θ do
        tasksBefore ← tasksCanBeBefore(θ_ij)
        pValues_ij ← combineDurTasks(tasksBefore)
return pValues
```

To modeling the job-shop scheduling problem as a CSOP we have used the syntax XCSP [10]. The Extensible Markup Language presents a simple and flexible text format and it gives the facility to use some functions and structures defined in Abscon [6]. In the modeling phase, we have applied two different techniques to reduce the variable domains.

The first technique developed (Algorithm 1) calculates initials values of each tasks and reduces the domain size of the involved variables. Thus, a solution can be found more efficiently. This algorithm calculates the maximum time interval in which each task can be scheduled. On the one hand, given a θ_{ij} task, the lowest value of its $st_{\theta_{ij}}$ is the sum of the processing times of the tasks that have to be scheduled before θ_{ij} from the same job i ($cumulative_{ij}$), subject to the precedence constraints. On the other hand, the highest value of all the domains for $st_{\theta_{ij}}$ is the sum of all processing times ($maxTime$), since this value represents the end of the schedule where the tasks are scheduled linearly. Due to the fact that at least the following tasks from the same job must be scheduled before $maxTime$, the highest value for the domain of each task θ_{ij} can be reduced by subtracting the duration of all the tasks from the same job i that have to be scheduled after θ_{ij} (including θ_{ij}) to $maxTime$. The values $cumulative_{ij}$ and $maxTime$ obtained in Algorithm 1 are used to filter the domains by removing values that cannot take part of feasible solutions.

Algorithm 3: Select-value-forward-checking with deletion block

```
while D'_i ≠ ∅ do
    select first element a ∈ D'_i, and remove a from D_i
    forall k, i < k ≤ |D'| do
        if constraint(i,k) = MachineOrJobConstraint then
            if constraint(i,k)= JobConstraint then
                remove values if(val_k < a + dur_i) from D'_k
            else
                remove values
                if(val_k + dur_k ≥ a ∧ val_k < a + dur_i) from D'_k
        else
            forall b ∈ D'_k do
                if not CONSISTENT (a_{i-1},x_i := a, x_k := b) then
                    remove b from D'_k
        if D'_k = ∅ then
            reset each D'_k, i < k ≤ n to value before a was selected
        else
            return a
return null
```

In the second technique to reduce the variable domains (Algorithm 2), the values that cannot be possible are calculated. $st_{\theta_{ij}}$ only should get values that represent the sum of the tasks that can be executed before θ_{ij}. For example, if θ_{ij} is the first task of its job, the possible values are 0 and a set of the combination of the processing times of all the tasks T that can be scheduled before θ_{ij}. In this case, these tasks T are all the tasks of the other jobs. For the following task (θ_{ij+1}), its possible values $st_{\theta_{ij+1}}$ are the same as θ_{ij} plus the processing time of θ_{ij}. In the Algorithm 2, *tasksCanBeBefore* function returns the tasks that can be executed before a given task θ_{ij}; and, *combineDurTask* function calculates all the possible values for $st_{\theta_{ij}}$ following the precedence constraints.

Once the CSOP has been modeled with the corresponding reduced domains, it is solved by using a modified algorithm of Forward Checking (FC) (Algorithm 3). Instead of applying Maintaining Arc-Consistency, this algorithm performs a filtering procedure that removes the domains in blocks, i.e., it removes several values at a time. This is due to the fact that if a task θ_{ij} must be scheduled after task θ_{kl}, the $st_{\theta_{ij}}$ can take neither the value of the $st_{\theta_{kl}}$ nor all successive values until the ending time of θ_{kl}. Thus, all these values can be removed as a block of values. The values to delete depend on the type of constraint. If this is a precedence constraint, all the values before θ_{kl} plus its processing time will be deleted. Otherwise, if it is a capacity constraint the values between $st_{\theta_{kl}}$ (included)

Algorithm 4: Post-process

Data: S: Solution without operators; m: machines; p: operators;
Result: A solution considering operators

Order the machines by their number of tasks;
$machinesFewerTasks \leftarrow$ set of $m - p$ machines with fewer tasks;
$tasksToPut \leftarrow$ tasks of the machines $machinesFewerTasks$;
$remainingMachines \leftarrow m - machinesFewerTasks$;
Order $tasksToPut$ by Starting Times;

$actualState \leftarrow S$;
foreach $\theta_i \in tasksToPut$ **do**
  $savedState \leftarrow actualState$;
  $states \leftarrow \{\}$;
  for $r \in remainingMachines$ **do**
    $foundGap :=$ `IsThereAGap`(r);
    if *not* $foundGap$ **then**
      insert θ_i before the next task according to its Job;
    else
      insert θ_i in this gap;
    Delay the needed tasks according to the restrictions among them;
    $states \leftarrow states \cup \{actualState\}$;
    $actualState \leftarrow savedState$;
  $actualState \leftarrow$ `chooseBestState`$(states)$;

return $actualState$;

and its processing time will be deleted.

B. Second Step: A Post-process Procedure

Once the CSOP has obtained an optimized solution to the job-shop scheduling problem, this solution is used to allocate the required number of operators in order to solve the $JSO(n, p)$. The problem consists in finding a feasible schedule by minimizing makespan and maximizing number of buffers (N_{buf}) to guarantee a certain level of robustness. Note that a feasible schedule for $JSO(n, p)$ is also feasible for the standard job-shop problem and satisfies the restriction that at most p machines work simultaneously. Therefore, significant cases are those in which $p < min(n, m)$, otherwise our problem becomes a standard job-shop problem [1].

The aim of the Algorithm 4 is to convert a solution without operators in one where the operator constraints are considered. The idea is to set a number of machines ($remainingMachines$) equal to the number of operators p and try to reschedule the tasks of the other machines ($machinesFewerTasks$) within the $remainingMachines$. The tasks in $machinesFewerTasks$ must be sorted by their st ($tasksToPut$). Each θ_{ij} in $tasksToPut$ is allocated in the first available gap between two tasks of each machine in $remainingMachines$. For each machine, the search starts from the previous state ($savedState$). There are cases where θ_{ij} must be allocated without a gap between two tasks due to the precedence constraints. For instance, if we found a θ_{ik} of the same job as θ_{ij} and θ_{ik} must be scheduled after θ_{ij} according to the precedence constraints ($k > j$), θ_{ij} is allocated just before θ_{ik}, being delayed θ_{ik}. When a task is delayed, other tasks may be also delayed. The computational cost of the algorithm is $O(tasksToPut * |remainingMachines|)$.

The best state to allocate θ_{ij} is the state that maximizes the function $\frac{N_{buf}}{C_{max}}$. This function could be adjusted depending on the requirements of the user, e.g. either only minimizing the makespan or maximizing the number of buffers generated.

C. Third Step: Distributing Buffer Algorithm

The previous step gives us a solution that satisfies all constraints of the $JSO(n, p)$. This solution is an optimized solutions in terms of minimizing makespan and maximizing the number of buffers. However, the main goal for a robust solution, in a scheduling problem where no information about incidences is given, is to distribute the amount of available buffer among as many tasks as possible. It is well-known that all tasks involved in a critical path have not any associate buffer, because it will affect to makespan. The rest of tasks can be reassigned to generate a buffer after their ending time. The main goal of this algorithm is to distribute the amount of buffer without affecting the makespan. Thus, we can maximize the number of buffers to the resultant schedule of second step. In this way, the obtained solution is considered more robust due to more tasks have buffer times to absorb small incidences. Figure 2 shows the solution obtained in the second step and all critical paths. It can be observed that 10 buffers were generated, meanwhile the distributing buffer algorithm was able to find 14 buffers. We remark that our goal is to obtain the maximum number of buffers since no information is given about incidences so that all tasks have the same probability for delaying.

This third step is presented in Algorithm 5. This algorithm looks for the tasks θ_{ij} allocated just before each buffer generated in Algorithm 4, and tries to set them back. A task θ_{ij} is only set back if it generates a new buffer. In case a new buffer nb is generated, this process is repeated with the task just before nb. The computational cost of the algorithm is O(tasks in non-critical path), because tasks in the non-critical path are the only ones that can be moved to distribute the buffers.

Algorithm 5: Distributing buffers

Data: Sch: Schedule; $buffers$;
Result: New Schedule

foreach $b \in buffers$ **do**
  $sizeB :=$size of the buffer b;
  repeat
    $continue := false$;
    $\theta_{ij} :=$ task allocated before b;
    Set back θ_{ij};
    if *this movement generates another buffer* nb **then**
      $continue := true$;
      Update schedule Sch;
  until $continue$;
return Sch;

D. An example

Figure 2 shows the different schedules obtained by the CP Optimizer solver and our technique for a given instances of the $JSO(n, p)$. This instance represents a scheduling problem with 3 jobs, each with 10 tasks, and 2 operators. Each

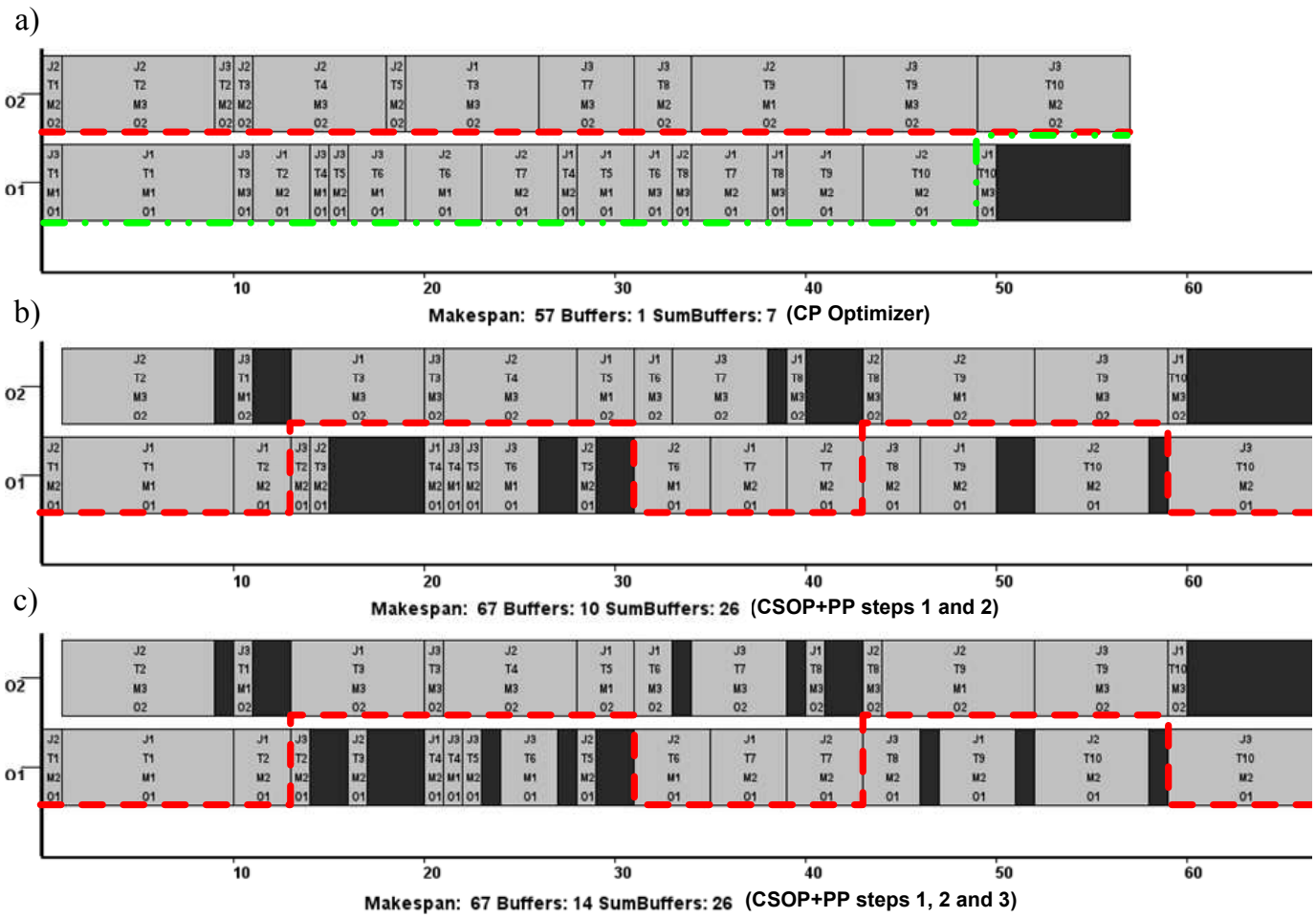

Fig. 2. A scheduling problem: an optimal and a robust solution.

rectangle represents a task whose length corresponds to its processing time. Inside the rectangle, the job, task, machine and operator are showed. The dotted lines showed in these schedules represent the critical path. The first schedule (Figure 2(a)) represents the distribution of tasks of the optimal solution obtained by CP Optimizer (only minimizing the makespan) according to the operators. It can be observed that the obtained makespan was 57 but only 1 buffer was generated due to the fact that only task $\theta_{1,10}$ was not involved in any critical path (green and red lines in Figure 2(a)). Taking into consideration the robustness within the objective function, the Figure 2(b) represents the solution obtained by applying step1 + step2 of our algorithm, where all tasks are distributed by operators. Finally, Figure 2(c) represents the schedule obtained by the third step of our algorithm. Although the makespan was increased up to 67, it can be seen that the buffers (black boxes) are distributed between all tasks that do not take part of any critical path being increased the robustness of this schedule. These buffers can be used to absorb incidences or delays from the previous task. For instance, if the resource assigned to the tasks θ_{23} suffers a small failure, the solution could not be affected.

V. Evaluation

The purpose of the experimental study is to assess our proposal CSOP+PP and to compare it with the IBM ILOG CPLEX CP Optimizer tool (CP). In CP, the p operators were modeled as a nonrenewable cumulative resource of capacity p. Also, the CP was set to exploit constraint propagation on no overlap (*NoOverlap*) and cumulative function (*CumulFunction*) constraints to extended level. The search strategy used was Depth First Search with restarts (default configuration).

We have experimented across the benchmarks proposed in [1], where all instances have $n = 3$ and $p = 2$ and are characterized by the number of machines (m), the maximum number of tasks per job (v_{max}) and the range of processing times (p_i). A set of instances was generated combining three values of each parameter: $m = 3, 5, 7$; $v_{max} = 5, 7, 10$ and $p_i = [1, 10], [1, 50], [1, 100]$.

In Figure 3, the solutions showed are the schedules for an instance $<$ 3_5_50 $>$ obtained by the CSOP+PP after both the step 2 (CSOP Step2) and step 3 (CSOP Step3). The smoothed curve of CSOP Step2 and CSOP Step3 are represented by dotted lines; by the CP Optimizer without taking into account the operators and applying steps 2 (CP

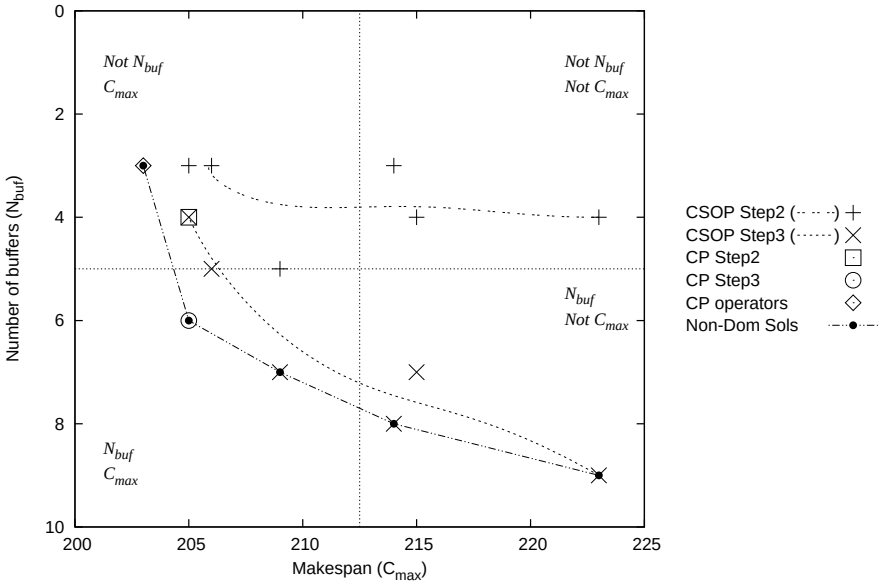

Fig. 3. Set of solutions for an instance given

Step2) and 3 (CP Step3); and, by the CP Optimizer taking into consideration operators (CP operators). Since two objective functions are considered in this problem, the solutions that are not dominated by any other solution are marked with a black point (Non-Dom Sols). It can be observed that keeping the same makespan (C_{max}), solutions given by the step 3 always outperform the ones obtained by the step 2 according to N_{buf}. It is important to note that in order to achieve the minimal C_{max}, there have to be few N_{buf}; and vice versa, to obtain more N_{buf}, it is needed to increase the C_{max}. Among the non-dominate solutions obtained, there is no optimal schedule with the lowest C_{max} and the maximum N_{buf}, therefore, the users should choose among them according to their necessities. For instance, let the solutions space be subdivided in four squares according to whether they minimize or maximize each objective. If the user just needs maximizing the N_{buf} (achieving better robustness), the solutions needed are from the right-down square.

In the next experiment the first solution given by the CSOP has been chosen to apply the post-procedure mentioned above because it is the solution that gives the opportunity to get more N_{buf}. This first solution is compared against the optimal solution. In all cases, 10 instances were considered from each combination and the average results are shown in the next tables. The sets of instances are identified by the tuple: $< m\_v_{max}\_p_i >$. The incidences (Z) to assess the robustness were modeled as a delay in a random task θ_{ij} from the schedule. For each instance, a set of 100 incidences were generated with a delay (d) that follows a uniform distribution between 1 and a 10% of p_i.

Tables I(a), I(b) and I(c) show the performance of both techniques to absorb incidences. For each technique, we report the C_{max}, the number of buffers generated (N_{buf}) in step 3, and the robustness (R) for instances for each m, v_{max} and p_i.

Following Lemma 1, the number of buffers showed in these tables are a lower bound of the number of tasks that are not involved in any critical path. For example, in instances $< 3\_10\_10 >$ the optimal solution had an average of 1.2 buffers in the 10 instances evaluated, meanwhile our technique obtained an average of 10.90 buffers in the same instances evaluated. This indicates that in average 10.90 out of 30 tasks were not involved in any critical path and a disruption in one or some of them could be absorbed and the rest of the tasks would not be involved in the disruption.

In all instances the average number of buffers obtained by CSOP+PP was bigger than the ones obtained by CP Optimizer. According to the robustness measure and the number of buffers generated, CSOP+PP procedure always outperformed the solutions given by the CP Optimizer, although the makespan turned out to be increased. For instance, in Table I(a) the instances $< 3\_10\_10 >$ increased up to 32.6% the number of incidences absorbed.

It can be seen that for CSOP+PP the greater p_i, the greater robustness values since the buffers generated could have bigger sizes, e.g., the instances with $m = 5$ and $v_{max} = 10$ increased their robustness degree obtaining an average of 25.5% for $p_i = 10$; 31.7% for $p_i = 50$; and 35% for $p_i = 100$.

Table II presents how large delays in the incidences affect the schedules. As d increases, the average of incidences absorbed are reduced, reaching the case that the CP Optimizer obtained an average robustness about 0% for most instances with $p_i = 100$. Even, CP Optimizer was unable to absorb any incidence in instances of $< 7\_10\_100 >$, whereas the CSOP+PP obtained an average of 6.9% the number of incidences absorbed for large delays.

TABLE I
AVG. MAKESPAN AND ROBUSTNESS

(a) Maximum delay 1 time units

Instance	CP Optimizer			CSOP+PP		
	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**
3_5_10	42.70	2.00	12.50	55.50	4.40	29.20
3_7_10	57.90	1.10	6.80	71.20	7.80	38.60
3_10_10	65.20	1.20	4.60	87.00	10.90	32.60
5_5_10	39.40	0.20	1.20	51.20	4.90	32.90
5_7_10	53.90	0.70	3.10	71.50	7.50	35.90
5_10_10	60.60	0.60	1.70	78.60	8.10	25.50
7_5_10	31.00	0.90	5.40	42.60	5.20	31.10
7_7_10	44.70	0.50	2.30	61.44	5.60	29.40
7_10_10	70.40	0.50	1.50	99.00	8.20	26.90

(b) Maximum delay 5 time units

Instance	CP Optimizer			CSOP+PP		
	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**
3_5_50	201.10	2.20	12.90	232.50	6.60	39.80
3_7_50	267.20	2.20	7.20	317.30	8.80	34.00
3_10_50	353.30	3.50	8.60	450.00	12.60	35.30
5_5_50	184.30	1.20	2.70	252.20	5.70	27.70
5_7_50	253.50	1.10	1.60	340.20	8.30	31.80
5_10_50	363.00	0.90	0.30	467.10	11.60	31.70
7_5_50	177.10	1.10	2.50	237.80	5.80	27.90
7_7_50	252.00	0.40	0.40	380.20	6.20	29.10
7_10_50	363.90	1.00	1.00	512.60	10.40	28.50

(c) Maximum delay 10 time units

Instance	CP Optimizer			CSOP+PP		
	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**
3_5_100	389.00	2.20	12.70	459.00	7.40	41.00
3_7_100	561.40	4.40	16.70	680.90	9.70	38.50
3_10_100	768.70	2.90	4.40	948.90	12.70	34.60
5_5_100	349.00	1.10	2.70	438.90	6.40	34.20
5_7_100	495.00	0.80	0.40	643.90	8.30	33.90
5_10_100	698.20	0.70	1.00	932.10	12.40	35.00
7_5_100	391.40	1.00	2.30	500.80	6.30	37.10
7_7_100	516.00	0.40	2.00	695.80	6.70	27.80
7_10_100	744.80	0.70	0.50	1043.20	10.20	30.00

TABLE II
AVG. ROBUSTNESS WITH LARGE INCIDENCES ($p_i = [1, 100]$)

Instance	CP Optimizer			CSOP+PP		
	d [1,20]	**d [1,50]**	**d [1,100]**	**d [1,20]**	**d [1,50]**	**d [1,100]**
3_5_100	9.90	6.10	3.10	35.20	20.80	11.90
3_7_100	8.90	6.40	3.30	34.90	22.40	12.00
3_10_100	3.80	1.90	1.30	29.10	17.10	9.50
5_5_100	1.90	1.40	0.50	26.90	12.00	8.30
5_7_100	0.20	0.00	0.00	28.60	14.80	8.50
5_10_100	1.00	0.50	0.30	26.70	16.80	10.80
7_5_100	1.90	0.50	0.40	25.00	11.30	6.70
7_7_100	0.80	0.10	0.10	20.10	10.60	5.60
7_10_100	0.00	0.00	0.00	23.50	13.80	6.90

VI. CONCLUSIONS

In this paper, unlike other optimization methods that solve the $JSO(n, p)$ by pursuing the objective function or both the objective function and the robustness measure, we have presented a three step technique to solve this problem with the aim of obtaining optimized and robust solutions. Robust solutions imply that some tasks can be delayed due to an incident in a machine or operator and the solutions remain feasible. In the first step of our approach, an optimized solution has been obtained by optimizing makespan for the JSP. In the second step, this solution is modified to take into account operators constraints by minimizing makespan and maximizing the number of buffers. And finally, in the third step, setting the previous makespan, the robustness is maximized by redistributing buffers. In this way, given solutions for the JSP, this procedure provides schedules taking into account operators in a short time. These solutions are obtained according to client needs maintaining their desired trade-off between optimality and robustness.

ACKNOWLEDGMENTS

This research has been supported by the Spanish Government under research project MEC-FEDER TIN2010-20976-C02-01 and TIN2010-20976-C02-02, by the fellowship program FPU (AP2010-4405) and by the Principality of Asturias under grant FICYT-BP09105.

REFERENCES

[1] A. Agnetis, M. Flamini, G. Nicosia, and A. Pacifici. A job-shop problem with one additional resource type. *Journal of Scheduling*, 14(3):225–237, 2011.

[2] J.C. Beck. *A schema for constraint relaxation with instantiations for partial constraint satisfaction and schedule optimization*. PhD thesis, University of Toronto, 1994.

[3] JC. Billaut, A. Moukrim, and E. Sanlaville. Flexibility and robustness in scheduling. Wiley, 2008.

[4] J. Blazewicz, W. Cellary, R. Slowinski, and J. Weglarz. Scheduling under resource constraints-deterministic models. *Annals of Operations Research*, 7:1–356, 1986.

[5] H. Kitano. Towards a theory of biological robustness. *Molecular Systems Biology*, 3(137), 2007.

[6] C. Lecoutre and S. Tabary. Abscon 112 toward more robustness. 2008.

[7] C. Mencía, M. R. Sierra, M. A. Salido, J. Escamilla, and R. Varela. Combining global pruning rules with depth-first search for the job shop scheduling problem with operators. In *Proceedings of RCRA 2012*, 2012.

[8] R. Mencía, M. R. Sierra, C. Mencía, and R. Varela. Genetic algorithm for job-shop scheduling with operators. In *Proceedings of IWINAC 2011(2). LNCS 6687*, pages 305–314. Springer, 2011.

[9] A. Rizk, G. Batt, F. Fages, and S. Solima. A general computational method for robustness analysis with applications to synthetic gene networks. *Bioinformatics*, 25(12):168–179, 2009.

[10] O. Roussel and C. Lecoutre. Xml representation of constraint networks: Format xcsp 2.1. 2009.

[11] N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86(1):1–41, 1996.

[12] M. Sierra, C. Mencía, and R. Varela. Optimally scheduling a job-shop with operators and total flow time minimization. In *Advances in Artificial Intelligence: 14th Conf. of the Spanish Association for Artificial Intelligence, Caepia 2011*, LNAI 7023, pages 193–202. Springer, 2011.

[13] M. R. Sierra, C. Mencía, and R. Varela. Searching for optimal schedules to the job-shop problem with operators. *Technical report. Computing Technologies Group. University of Oviedo*, 2011.

[14] E. Szathmary. A robust approach. *Nature*, 439:19–20, 2006.

[15] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proc. of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 307–312, 1994.