Overlap Algorithms in Flexible Job-shop Scheduling

Celia Gutiérrez

Universidad Complutense de Madrid, Spain

***Abstract* — The flexible Job-shop Scheduling Problem (fJSP) considers the execution of jobs by a set of candidate resources while satisfying time and technological constraints. This work, that follows the hierarchical architecture, is based on an algorithm where each objective (resource allocation, start-time assignment) is solved by a genetic algorithm (GA) that optimizes a particular fitness function, and enhances the results by the execution of a set of heuristics that evaluate and repair each scheduling constraint on each operation. The aim of this work is to analyze the impact of some algorithmic features of the overlap constraint heuristics, in order to achieve the objectives at a highest degree. To demonstrate the efficiency of this approach, experimentation has been performed and compared with similar cases, tuning the GA parameters correctly.**

***Keywords*— Algorithm, Flexible Job-Shop Scheduling, GA parameters, Local improvement, Overlap heuristics.**

I. Introduction

A Job-shop Scheduling Problem (JSP) is based on the concept of jobs, which are composed of operations that must be processed by the resources of different type in a sequential order. Each operation has a completion time. One machine can only process one job at a time and an operation cannot be pre-empted. The objective is to minimize the total makespan (the time to complete all jobs). The simplification of this problem is enunciated like this: there are *n* jobs to be scheduled on *m* machines in a general job-shop problem, *G*, minimizing the total completion operation time, C_{max}, $n/m/G/C_{max}$.

Flexible Job-shop Scheduling Problem is a generalization of the JSP, where the resource is selected among a set of suitable ones, giving place to two subproblems: routing and allocation of operations. The first one produces the start-time of the operations, and the second one the assignment of operations on resources.

Both JSP and fJSP have been solved by the use of metaheuristic algorithms, like GAs.The application of a GA on the simple basis as in [1] has poor performance because no domain knowledge is inserted, leading to non-feasible results. One way to insert knowledge into the algorithm is by hybridizing the GA with heuristics that provide local search. This paper follows the last approach, and goes beyond a deep analysis of GAs. It fact, it is an extension of [2], that explains how to achieve optimal results in the hybridization of GA with local search techniques to solve fJSP. This work provides a further analysis of the overlap constraint operators. In this way, the previous work provides a macroperspective view of the whole solution, and the present work is a microperspective view. It is structured in this way: section 2 covers the problem background; section 3 introduces the complete algorithm and the codification of information regarding the resources and fitness functions; section 4 shows the algorithms of a heuristic operator variants; section 5 shows the results of the experimentation phase; section 6 contains the comparison with similar approaches; and section 7 has the conclusions and future work.

II. Problem Background

Hybrid approaches that mix GA and heuristics are a well-known solution that has proven to be efficient, as heuristics provide domain knowledge that the simple GA cannot [3]. This focus can be applied in two ways: embedding the heuristics into the GA loop (integrated approach), or outside it (hierarchical approach), [3].

Literature shows examples of hybrid GA with intelligent genetic operators than produce optimal schedules. This is the case of [4], that describe an effective hybridation of both techniques, applying improved crossover and mutation operators when there are non-feasible schedules.[5] describes a hybrid GA solution by the use of two vector chromosome and bottleneck shifting procedure. The representation is made by two vectors: one for the machine assignment and the another one for the operation sequence. [6] solve the same problem by the use of an artificial immune algorithm. It uses several strategies for generating an initial population and selecting the best individuals. It also has operators that reorganize the operations (by a mutation). [7] adopt the hybrid GA by the use of the approach by localization to initialize the GA, and improving it by reordering jobs and machines, and by searching for a global minimum [4] have improved operators constraint and mutator operators that consider constraint violations.

The second way to include the heuristics has also been widely implemented, though the existing algorithms vary in the order of application, heuristic methods, goal of the application,

DOI: 10.9781/ijimai.2014.265

and even domain. [8] follows this paradigm by means of a local search by the definition of the neighborhood.

This work follows the second approach. Having proven the efficiency of the mentioned algorithms, the objective of this research is to provide the designer with relevant issues that improve the algorithm performance when using local improvements within a hybrid GA under a hierarchical architecture. This is also considered a multi-objective fJSP, because the solution achieves three goals:

- To minimize the makespan of the operations.
- To minimize the maximal machine overload, i.e., the maximum working time spent at any machine.
- To satisfy the maximum number of constraints.

There are also recent approaches to solve the problem of JSP, like [9], where they solve the problem of scheduling independent tasks in a grid computing system. They use a new evaluation (distributed) algorithm inspired by the effect of leaders in social groups, the group leaders' optimization algorithm (GLOA). In contrast, the present work analyzes some design features of the hybrid algorithm, preferably the overlap constraint repairer.

III. HIERARCHICAL DESIGN FEATURES

This work constitutes the extended version of the previous work, providing deeper details of the heuristics design and argumentation for the parameters tuning. So, whereas [2] and [10] provide a solution to a general fJSP, the current work provides design and execution details in order to achieve the goals of the algorithm.

This research has been analysed following a hierarchical approach that decomposes the resource and the start-time assignment in two different problems solved by different and independent GA, like in [5]. Previous to both GA running, there is a module that calculates the limits for the start-time for each operation, and after both GA running the module of the heuristics solve the unfulfilled constraints. The adaptation of the algorithm to JSP claims a simpler architecture, where the resource GA module does not appear. Other variations concerning the heuristics are also discussed in the section 4.

A. Codification of the Resource GA Chromosome

The chromosome and fitness function for both GA are described in the previously cited works. There are subtle differences in the morphology of both chromosomes: while the solution for time GA is directly codified into the chromosome, the chromosome for resource GA stores as many genes as operations, which must be decoded to get the resource number. For example, for the set of 4 orders, 3 products per order (maximum), 1 product instance per product (maximum), 5 operations per instance (maximum), and 4available resources in the job-shop, the gene value must cover 4 x 3 x 1 x 5 x 4 values, so the range is [0-239]. To decode a gene value, successive divisions must be applied using this algorithm that involves equation (1) to equation (8):

$$\textit{Resource number} = \textit{gene MOD number of resources} \quad (1)$$

$$\textit{cant} = \textit{gene / number of resources} \quad (2)$$

$$\textit{product instance identification} = \textit{gene MOD number of product instances} \quad (3)$$

$$\textit{cant} = \textit{cant / number of product instances} \quad (4)$$

$$\textit{operation number} = \textit{cant MOD number of operations} \quad (5)$$

$$\textit{cant} = \textit{cant / number of operations} \quad (6)$$

$$\textit{product identification} = \textit{cant MOD number of products} \quad (7)$$

$$\textit{order number} = \textit{cant / number of products} \quad (8)$$

For a gene value of 69, the decoding process gives the following values for the parameters:

- resource number = 1
- product instance identification = 0
- operation number = 2
- product identification= 0
- order number = 1

B. GA fitness functions

There is one fitness function for each GA. Both functions incorporate penalizations that depend on the domain they are evaluating. For both GAs, the objective is to minimize the values obtained by the fitness functions.The following subsections contain their codification:

1) Fitness function for Resource GA

This function evaluates the sums of deviations between the assignment of operations to certain resource and the ideal assignment. In other words, this fitness function penalizes non-balanced assignments of operations among the resources of the same type. The ideal assignment is the number of operations assigned to the resources of the same type, divided into the number of resources of that type, as equation (9) shows:

$$\textit{Fitness} = f \times \sum_{i=0} |O_{i,t} - (O_t / R_t)| \quad (9)$$

where:

f is a the penalty factor (For simplicity, f=1),
i represents each resource in the job-shop,
$O_{i,t}$ is the number of operations assigned to the i resource, that belongs to the t type of resource,
Ot is the number of operations assigned to the resources of t type,
R_t is the number of resources of t type.

2) Fitness function for Time GA

This function sums up the starting times of all operations, with a penalization when an operation violates a constraint, as in equation (10):

$$Fitness = \sum_{i=0} t_i + p_i \quad (10)$$

where:

i represents each operation in the job-shop,
t_i is the starting time of the *i* operation,
p_i is the sum of quantities derived from penalizations for order and overlap violated constraints, in the way equations (11) and (12) show:

-if an order constraint is violated, the fitness must be severely penalized, so that this chromosome does not to pass to the next generation:

$$p_i = p_i + 100000000 \quad (11)$$

-if overlap constraint is violated, the fitness is penalized proportionally to the amount of the overlap. :

$$p_i = p_i + |t_{f,j} - t_i| \quad (12)$$

where $t_{f,j}$ is the finishing time of the *j* overlapped operation.

Notice that range constraint is not contemplated in the penalization equation because the time GA assigns the start-times within the range limits. Therefore the solutions provided by the time GA are always valid according to this constraint.

C. Heuristic algorithm

A relevant design issue is the organization of constraints in the heuristic stage. In a Constraint Satisfaction Problem (CSP) like this, a dilemma appears on the order of repairment of the constraints, claiming a further analysis. As the repairment of a constraint can modify the degree of satisfaction of the remaining constraints, the evaluation of the constraint of each operation must be followed by each repairment, so its start-time is updated. The algorithm below shows the workflow of the heuristic stage. It ends when it reaches a maximum number of iterations (MAX_IT). This parameter is tuned depending on the size of the orders, as explained in subsection 5.2.

```
Step 1: Point to 1st operation
Step 2: Get operation data
Step 3: Point to 1st constraint
Step 4: Heuristic evaluator
Step 5: Heuristic repairer
Step 6: If no more constraints
              then go to step 8
        otherwise go to step 7
Step 7: Point to next constraint
Step 8: If more operations
              then go to step 9
        otherwise go to step 10
Step 9: Point next operation
Step 10: Termination condition.
        If iterations = MAX_IT
              then exit
        otherwise go to step 1
```

IV. VARIANTS FOR THE OVERLAP CONSTRAINT

As mentioned before, each constraint has one module to evaluate, and another one to repair. Whereas Range and Order heuristics are simple and described in [2], Overlap heuristics requires a deeper design: the evaluator is more complex than the other ones, and the repairer presents different variants.

Previously to running this repairer, a conflict appears about which of the overlapped operations has the priority to get repaired, which is not necessarily the operation appointed by the main algorithm. This is solved by the designation of the *critical operation*. The overlap repairer goal is to find an interval where the operation can be shifted while respecting the range constraint, so the critical operation must have the narrowest margin for start-time assignment (i.e. it is the most restrictive), as equation (13) says:
i is critical over *j* if:

$$|tmax_i - tmin_i| < |tmax_j - tmin_j| \quad (13)$$

i, *j* are the overlapped operations
$tmax_i$ is the start-time upper limit for *i* operation
$tmin_i$ is the start-time lower limit for *i* operation

Each overlap repairer solves one overlap of a pair of operations, so if an overlap has more than two operations like equation (14) says, it will be solved in *k+1* iterations of the repairer. At each iteration, there will be a different designation for the critical operation.

$$k + 2, k > 0 | \quad (24)$$

Apart from these variables, there are others that participate in subsequent algorithms:

- O is the current operation of the algorithm defined in section 3. It is the operation that is being evaluated/repaired at each iteration of the main program.
- J is the operation that is being compared to the O at each evaluator/repairer iteration.
- C is the critical operation in an overlap.
- t_i is the start time of i operation.
- I is the current interval of the R. An interval is considered when there is a period of time when R is not assigned to any operation, so it remains not active.
- R_i is the resource assigned to i operation.
- T_R is the type of R resource.
- S is the resource currently appointed to.
- L is the list of operations that overlap with O.
- L_I is the list of I.
- L_R is the list of resources of the same type as R_o

The structure for the evaluator and the repairer variants are described in the following subsections.

A. Overlap Evaluator

The following algorithm includes the steps to evaluate if the current operation overlaps other one(s) on the same resource:

```
Step 1: Store (O, L)
Step 2: Point J at the 1st operation
assigned to Ro
Step 3: Stop condition:
          if no more operations for Ro
          then stop
          oterwise go to step 4.
Step 4: If J not = O, and J overlaps O
          then store (J, L)
Step 5: Point J at the next operation in
Ro
Step 6: Go to step 3.
```

Operations are overlapped if an operation begins before the other one has finished. The information that results from this stage is a list of operations that overlaps the current one. This list is the input of the overlap repairer stage.

B. Overlap Repairer

The overlap repairer includes several stages (i.e. Interval Search, OperationExchange, Resource Mutation), which are successively executed if the previous one has not been successful, as [2] show.

Other design issues come out when handling constraints that interfere with others. In this case, there are two possibilities:

1. To consider a blind repairment, so that the constraint is repaired without considering the other ones. Such is the case of the order and range repairers.
2. To consider an intelligent repairment, so that the constraint is repaired taking the other ones into consideration. Overlap repairer follows this approach. There are several ways to incorporate these considerations, producing two variants for overlap repairer: the first one (pure variant) considers the range constraint for its amendments; the second one (hybrid variant) considers both the range and the order constraints. The mentioned stages can be designed in both ways:

1) Algorithms for Pure Variants.

a) Algorithm for Interval Search

```
Step 1: Find LI for Ro
Step 2: Find C among two overlapped in L
Step 3: Position I at the beginning of
LI
Step 4: Stop condition:
          if no more intervals in LI
          then go to step 8.
Step 5: If I suitable for C
          Then tc = max (tminC, tminI)
          Exit
Step 6: Position I at next interval of
LI
Step 7: Go to step 4.
Step 8: Exit.
```

An interval is suitable if it matches the assignment conditions for the critical operation, in terms of operation duration and start-time range limits.

b) Algorithm for OperationExchange.

```
Step 1: Find C among two overlapped in L
Step 2: Position J in previous operation
in Rc
Step 3: Stop condition:
if no more previous operations,
then exit.
Step 4: If J suitable for C
          then exchange (tj, tC)
          exit.
Step 5: Position J in the next previous
operation in Rc
Step 6: Go to Step 3.
```

A current operation is suitable if its start-time fulfills the range constraint of the critical one.

2) Algorithms for Hybrid Variants.

a) Algorithm for Interval Search.

It remains the same as the PureVariant, except the suitability condition is step 5. In *this case, a*n interval is suitable if it matches the assignment conditions for the critical operation, in terms of operation duration and start time range bounds, and not belonging to the same job (to assure it fulfills the order constraint).

b) Algorithm for OperationExchange .

It *remains the same as the PureVariant, except the suitability condition is step 4. In this case,* an operation is suitable if it does not belong to the same job (to assure it fulfills the order constraint), and its start-times fulfills the range constraint of the critical.

c) Algorithm for Mutation Operator.

This operator assigns the operation to another resource of the same type, while preserving the start-time. This amendment does not interfere with the other type of constraints, but it can produce overlaps in the new resource.

```
Step 1: Find C among two overlapped in L
Step 2: Position S in 1st resource in
the job-shop
Step 3: Stop condition:
          if no more resources
         then go to step 7.
Step 4: If S not = RC and Ts = TRC
          then store (S, LR)
Step 5: Position S in next resource in
the job-shop
Step 6: Go to step 3.
Step 7: Random assignment of RC among
the candidates in LR.
```

V. Experimentation Results

Tests have been performed for the complete algorithm, putting special emphasis on the variants of the overlap repairers. The machine has been a Sun Sparc workstation running Solaris operating system. There has been a preliminary stage, to configure the GA, and a main stage, to validate the complete algorithm.

A. Tuning the GA Parameters

Beside the algorithmic issues, the success of the algorithm lies on several factors, like the correct tuning of the GA parameters. Several works have inserted in the code the way to tune them dynamically like the fuzzy logic controller (FLC), which methods are described in [11]. The key of success of applying FLC to GA is a well-formed fuzzy sets and rules [12]. In this work there has been previous experimentation to analyse the best values for the GA, by testing the different GA isolatedly. The most successful configuration for the parameter set population size/number of generations/mutation rate/selection type is 50/60/0.01/tournament for the resource GA and 8/10/0.01/elite for the time GA.

B. Configuration of the Hybrid GA

Testbeds have been configured varying the number of orders from 1 to 4, number of jobs from 1 to 3, number of products from1 to 4, number of product instances from 1 to 2, number of operations from 1 to 4, and operation processing times from 24 to 100, 5 resources belonging to 4 types, with the total number of executions per testbed of 25. The number of iterations for the heuristics stage has varied with the number of orders: for one order only 100 have been needed, while for four orders more than 200. Results collect the average of the executions.

Heuristic optimization algorithms can be evaluated in two ways [13]: by measuring the solution quality and measuring the solution time. In this case we have measured the solution quality by two criteria:

Considering this problem as a CSP, the solution quality must measure the constraint satisfaction rate. In this work, we consider the mean error (ME) parameter, as the percentage of constraints not satisfied. Figure 1 shows the results for the pure and hybrid variants of interval and exchange operators, distributed horizontally by the number of orders and vertically by the ME. This figure reflects that for few operations the pure repairer is better, but when the number of operations increases, the hybrid one is better. In this case, the ME is higher than 0, due to the technological limitations, i.e. more operations for the same number of resources produces more operations with unfulfilled constraints, and therefore reduces the number of fulfilled constraints. The reason for this improvement using the hybrid repairer is that the design of that heuristics has been made in such a way that the improvement in the overlap does not worsen any other constraint, in contrast with the pure repairer. The disadvantage of that is that fewer amendments can be applied with this variant, because it is more restrictive.

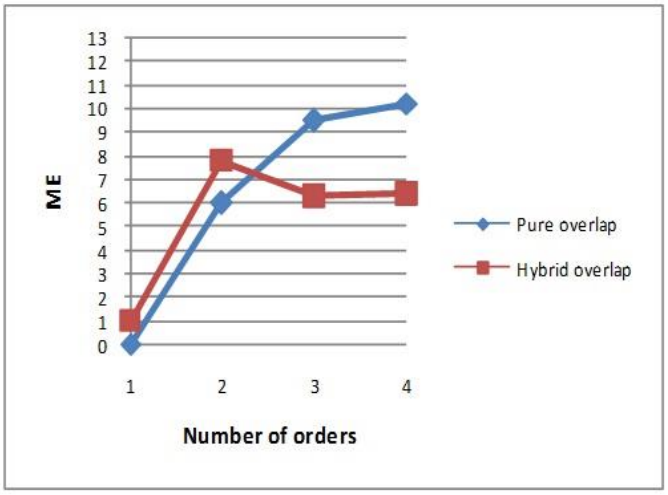

Fig. 1. ME of the two variants of overlap repairers

Considering it as a fJSP, the quality measurement is the time GA fitness. Table I shows the results for the time GA, as it is related to the constraints. PRf and HRf columns contain the Pure Repairer fitness and the Hybrid Repairer fitness respectively. Def(HRf, PRf) provides information about the percentage difference of both fitness values as equation 15 shows:

$$Def(HRf, PRf) = 100 * \frac{(HRf - PRf)}{PRf} \quad (15)$$

Table I
Fitness values for the overlap variant

Number of orders	PRf	HRf	Def(HRf, PRf)
1	300	316	5.33%
2	352	379	7.67%
3	380	397	4.47%
4	411	419	1.95%

There is a relationship between the values for ME in Figure 1 and the fitness values shown in Table I. The fitness function is penalized when the range and overlap constraints are not fulfilled. The fewer the number of orders, the lower (and better) fitness results. Results are also better for the pure variant than the hybrid one. The reason is that the former reorder the overlapped operations trying to fulfill the range constraint, and the latter must also makes sure that the reorganization also fulfills the order constraint. This complexity means that the search interval does not always find the earliest interval suitable, and even does not find and interval, delaying more operations of the jobs than in the pure variant.

Besides that, the evolution of Def(HRf, PRf) is to decrease when the number of orders increases. This also shows that the fitness values in both repairers tend to be very similar for high number of orders. Therefore, it is recommendable to use the Hybrid Repairer in these cases, because they will provide similar fitness values than the Pure Repairer but with lower ME values.

VI. Comparison with Alternative Solutions

To test the efficiency of our algorithm, Table II collects the comparison with respect the makespan using [8] benchmark. It contains the best results of a set of executions. It consists of ten problems mk1-mk10, with the number of jobs are in the range 10-20, the number of machines are in the range 6-15, number of operations are in the range 5-15. Other configuration information is: n x m, that refers to the number of jobs per number of machines; (LB, UB) with the optimum makespan if known [14]; otherwise, it reports the best lower and upper bound known; Flex. with the average number of equivalent machines per operation. This work compares the mentioned fJSP experiments of hGA from [5], AIA [6] and GA [7], and TWS for the best results achieved among the different rules in [8]. The information presented in Table 2 has been partially obtained from [2].

The proposed algorithm of GAH has achieved lower results of makespan for some fJSP instances and similar results of makespan for the remaining fJSP instances. These results combined with the ME results in section 5, demonstrate that the algorithm shows excellent quality solution as a fJSP and a CSP.

TABLE II
Comparison with best known makespan for ten fJSP instances

Problem	n x m	Flex.	(LB, UB)	hGA	AIA	GA	TWS	GAH
Mk01	10 x 6	2.09	(36, 42)	40	40	40	42	40
Mk02	10 × 6	4.10	(24, 32)	26	26	26	32	26
Mk03	15 × 8	3.01	(204, 211)	204	204	204	211	204
Mk04	15 x 8	1.91	(48, 81)	60	60	60	81	60
Mk05	15 × 4	1.71	(168, 186)	172	173	173	186	172
Mk06	10 x 15	3.27	(33, 86)	58	63	63	86	57
Mk07	20 × 5	2.83	(133, 157)	139	140	139	157	139
Mk08	20 x 10	1.43	523	523	523	523	523	523
Mk09	20 × 10	2.53	(299, 369)	307	312	311	369	308
Mk10	20 × 15	2.98	(165, 296)	197	214	212	296	196

VII. Conclusions and Future Work

This work has described the algorithms of a complex heuristic, like the overlap evaluator and repairers, in a hybrid GA applied to fJSP, a multi-objective problem. The most relevant issue concerns the use of two variants for the repairer: one that does not take into consideration the other constraints (pure), and the other one that incorporates them (hybrid). When adopting this approach, designers may consider what the experimentation has revealed: pure variant is better for fJSP with few operations, producing better ME results; in contrast, it is recommendable the use of the hybrid variant when the number of operations increases. It also shows that it maintains the level of quality of other algorithms, in terms of makespan. Finally, it is also recommendable an appropiate tuning of GA parameters.

The future work opens a high number of possibilities. Concerning the inclusion of intelligent operators, we are working in the design of hybrid variants for the range and precedence repairers. In the same way, we are making another variant of the ResourceMutation substage, which assures that the new resource assignment does not cause the overlap of other operations. Finally, new constraints adapted to concrete JSP and fJSP are to be incorporated and experimented. Re-design of the model is done using the FactoryMethod design patron, where a family of constraints can be chosen depending on the application that is used. The collection of classes in [2], will be transformed in the collection shown in Figure 2. The fJSP class is the superclass which the concrete application inherits from: in the described work, this application is GAH, which uses the order, range, and overlap concrete constraints. When using OtherApplication, it will use OtherConstraints (containing the measurer or evaluator), which has the corresponding OtherConstraint_unfulfilled subclass (containing the repairers for that constraint). The construction of the repairer will also contemplate the inclusion of pure and hybrid variants. The choice on which one to use will depend on the number of operations handled by the fJSP. The results of the mentioned modifications will be compared with the current version, to see how they affect to the ME and the *makespan*.

VIII. Acknowledgment

C. Gutiérrez thanks Jose Maria Lazaro for his continuous help on the applications of fJSP. She also wants to thank Juan Jose Merelo for his guidance in the customization of his GAGS. This research work has been funded by the Spanish Ministry for Economy and Competitiveness through the project “SOCIAL AMBIENT ASSISTING LIVING - METHODS (SociAAL)” (TIN2011-28335-C02-01).

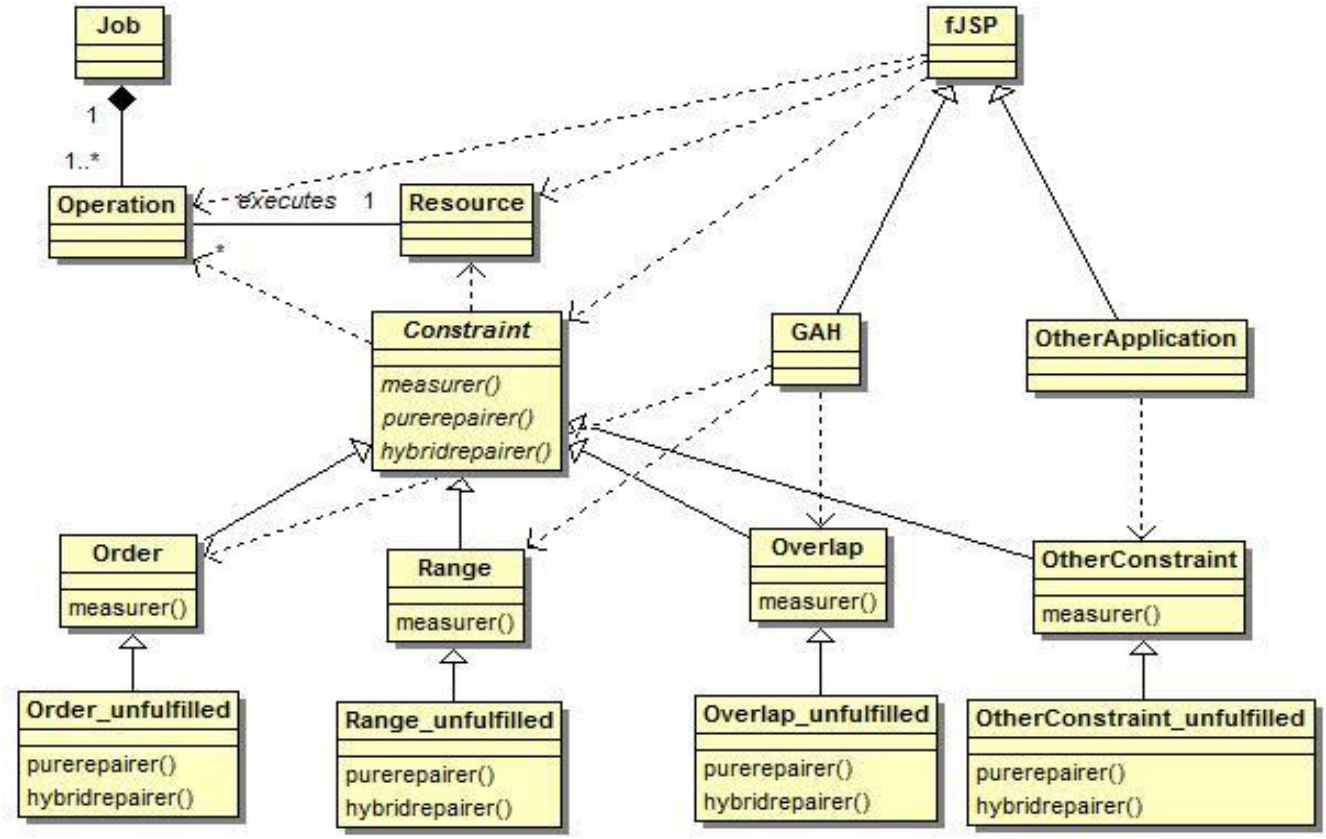

Fig. 2. Re-design of the classes for adaptation to other problems.

REFERENCES

[1] J.-J. Lin, "A GA-based Multi-Objective Decision Making for Optimal Vehicle Transportation," Journal of Information Science and Engineering, vol. 24, pp. 237-260, 2008.

[2] C. Gutierrez and I. García-Magariño, "Modular design of a hybrid genetic algorithm for a flexible job–shop," Knowledge-Based Systems, vol. 24, pp. 102–112, 2011.

[3] Y. Yun, M. Gen, M., and S. Seo, "Various hybrid methods based on genetic algorithm with fuzzy logic controller," Journal of Intelligent Manufacturing, vol. 14, pp. 401-419, 2003.

[4] J. Dorn and M. Girsch, "Genetic operators based on constraint repair," in ECAI'94 Workshop on Applied Genetic and other Evolutionary Algorithms.

[5] J. Gao, M. Gen, and L. Sun, "A hybrid of genetic algorithm and bottleneck shifting for flexible job shop scheduling problem," in 8th Annual Conference on Genetic and Evolutionary Computation, 2006, pp. 1157-1164.

[6] A. Bagheri, M. Zandieh, I. Mahdavi, and M. Yazdani, "An artificial immune algorithm for the flexible job-shop scheduling problem," Future Generation Computer Systems, vol. 26, pp. 533-541, 2010.

[7] F. Pezella, G. Morganti, and G. Ciaschetti, "A genetic algorithm for the Flexible Job-shop Scheduling Problem," Computers & Operations Research, vol. 35, pp. 3202-3212, 2008.

[8] P. Brandimarte, "Routing and scheduling in a flexible job shop by tabu search," Annals of Operations Research, vol. 41, pp. 157-183, 1993.

[9] Z. Pooranian, M. Shojafar, J. H. Abawajy, and M. Singhal, "GLOA: A New Job Scheduling Algorithm for Grid Computing," International Jorunal of Interactive Multimedia and Artificial Intelligence, vol. 2, no. 1, pp. 59-64, 2013.

[10] C. Gutierrez, "Heuristics in a General Scheduling Problem," in Proc. Conference on Evolutionary Computation, vol. 1, 2004, pp. 660-666.

[11] M. Gen and R. Cheng, Genetic Algorithms and Engineering Optimization. New Jersey: John Wiley and Sons, 2000.

[12] F. Cheong and R. Lai, "Constraining the optimization of a fuzzy logic controller using an enhanced genetic algorithm," IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics, vol. 30, pp. 31-46, 2000.

[13] R. Rardin and R. Uzsoy, "Experimental Evaluation on Heuristic Optimization Algorithms: A Tutorial," Journal of Heuristics, vol. 7, pp. 261-304, 2001.

[14] M. Mastrolilli and L.-M. Gambardella, "Effective neighbourhood functions for the flexible job shop problem," Journal of Scheduling, vol. 3, pp. 3-20, 2000.

C. Gutierrez was born in Bilbao, Spain, in 1969. She received the B.Eng. and Ph.D. degrees in Computer Science from the University of Deusto (Spain) in 1992, and from the University of the Basque Country (Spain) in 2000, respectively. She has worked for Labein technological research center, in Bilbao (Spain). She has also worked for the Basque Government and Indra, as a software Engineer. After doing teaching and research at some private universities, such as Instituto Tecnológico de Estudios Superiores de Monterrey (México D.F.), she joined Universidad Complutense de Madrid (Spain) where her current position is Assistant Professor . She is a member of the Grasia research group, on agent technologies. She has published in relevant journals, like Knowledge-based Systems. She has also directs financially supported projects. Her current interests focus on Data Mining, Software Quality, Multiagent Systems and Accessibility.