PACS project

Robust Configuration tool for Reconfigurable Automotive Assembly lines

Massimo Manzini

Politecnico di Milano,
Mechanical Engineering Department

December 10, 2017

Summary

List of Figures

List of Tables

Abstract

The project presented in this document grounds on a real industrial case the student worked on. This industrial case regards the configuration and reconfiguration of an assembly line used by a company in the automotive sector. In this sector the demand is composed by a large set of products with a low demand, that does not allow the company to have dedicated assembly line but only universal ones. These lines can be adapted and quickly reconfigured when the production changes, but, for doing this, a management approach is needed. This approach has to consider the assembly line looking at the future in order to predict the market request and design the line, its initial configuration and its future reconfiguration, accordingly. In addition to this, the market is considered uncertain, thus the optimization has to take into account a stochastic dimension during the identification of the sequence of line layouts that minimizes the total investment and management cost.

Outline of the document After the introduction and industrial motivation exposed in Section 1, we propose the formalization and the solution framework in Section 2. The presented framework has been implemented in *C++* which development is addressed in Section 3. After that, an application use case is presented and solved in Section 4, and some conclusions highlighted in Section 5. At least, a user manual is included in Section 6.

1 Introduction and motivation

In the last decade, manufacturing industry has been confronted with an increasing level of customization (and thus of variety of products) and the consequent reduction of production volumes (Wiendahl et al., 2007). In this context, concepts like *flexibility* and *changebility* (Tolio, 2009) have to be considered for managing the *co-evolution* between production system and market (Tolio et al., 2010). In order to be applied in a profitable way, these concepts have to be taken into account during the system's design phase. On the contrary, add flexibility to an existing system can cost a significant investment. Hence, the design of an assembly system is a key phase needing to be managed with a long time vision, considering a multi-period approach, an uncertain and evolving environment. The automotive assembly sector is heavily affected by these trends, especially the spare parts market due to high variety of products, small lot sizes and different assembly technologies and materials.

Indeed, it is possible to represent automotive market divided in three different stages, as represented with the average production volumes on years of the lifecycle of a car in Figure 1.1. The main stages, from the *OEM* (*Original Equipment Manufacturer*) point

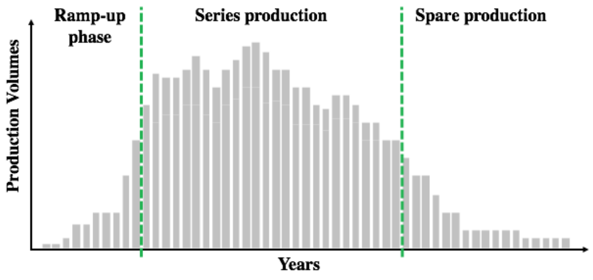


Figure 1.1: Automotive demand trend through the years.

of view, are the following:

1. *Ramp-up phase*: it is the initial part of life of a car, in which volumes start from 0 and increase, during which the *OEM* buy part from suppliers for testing and certifications;

2. *Series production phase*: it is the maturity phase of the lifecycle, during which the *OEM* produces itself all the parts;

3. *Spare production phase*: it is the last phase the car are no more produced in series but only on request, with a *make-to-order* process; during this stage the *OEM* outsources the production to a supplier.

The approach described in this document is specifically focused on the suppliers of the last stage, that have to cope with:

- high number of different products, that are bodyworks like hoods, doors and fenders;

- multiple technologies: like hemming joining, adhesive joining (manual or automated), spot welding, stud welding, nut welding, clinching, riveting, nut press and manual operations;

- different materials, aluminum and steel mainly;

- small lot sizes, in the order of a hundred of products.

In this context, consider a company that works for two of the main European *OEM*s and has to produce components for all their cars no more in series production. In addition, every component has a different shape and different material, and it has its own assembly process with different technologies involved[1]. From this point of view, the design of an assembly system able to cope with this complexity level can be very challenging.

In order to address this problem, the company developed a new assembly line architecture that is easily and quickly adaptable through a set of modular devices, represented in Figure 1.2. It is composed by

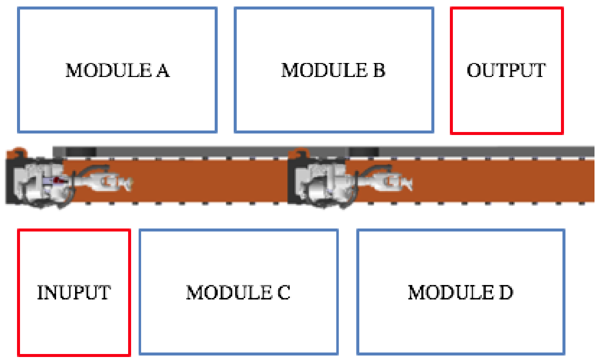


Figure 1.2: Example of modular assembly line.

- an handling robot able to move on its track and transport parts through the line, put in the center of the line;

- an input and an output station;

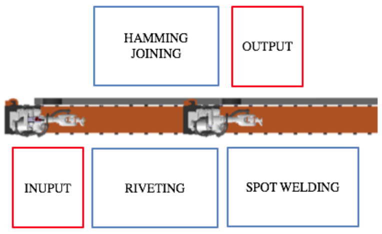
- a set of modular devices able to host technological assembly equipment requirement like machines, tools and fixtures.

Due to this modularity, it is possible to quickly change a module with another in order to introduce a new technology or a new equipment in the line. The result is the change of the set of technologies and thus the set of assembly processes the line can handle.

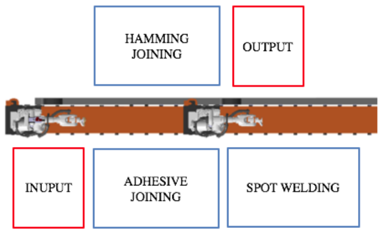
Using such an architecture, the company is able to face the change of part to be assembled with quickly substitutions of modules in the line. As an example, the two

[1]The set of assembly technologies used are addressed as *Functional Assembly Groups* (*FAG*); they are limited to the ones enlisted above, that are: mechanical joining, resistance joining, MIG joining, adhesive joining, hemming, manual operations and other.

assembly lines in Figures 1.3a and 1.3b share the same architecture, with three modular devices (in blue) able to host different machines. The first one hosts a hamming joining station, a spot welding and an adhesive joining, instead, the second one has a riveting station instead of the hamming one. It means that, with the same line architecture, it is possible to allocate in the line different sets of machine in order to change the assembly processes the line can handle.



(a) First exemplar assembly line.



(b) Second exemplar assembly line.

Figure 1.3: An exemplar configuration with two possible module sets allocated.

Thus, by simply change one module in the line, it is possible to change the set assembly process the line can handle. Nevertheless, since the number of modules in the line able to accommodate machines is fixed, the number of technologies (and thus processes) is limited. Considering again the example in Figures 1.3a and 1.3b, the line can host 3 maximum different technologies and thus a limited range of processes. We define the layout of the line as a *configuration*, it describes the line in terms of:

- the number of modules in the line, that represents the maximum number of machines to be included in it and thus the set of possible processes to be handled;
- each module position;
- additional equipment (like input station, control unit, output station, etc.) position.

Grounding on this, it is possible to change a modular device included in the line with a new one but, if we want to add a new module or change its position in order to achieve a larger (or different) set of processes, the line has to undergo a reconfiguration. It means to block the production for a couple of weeks and invest additional economic resources.

As a consequence of these technological considerations, the design phase in which the number of modules and their position in the line are decided has a huge impact on the management of the line itself in the future. Hence, this design phase has to be conducted with a long-vision, that is looking at the future demand in order to minimize the reconfiguration processes and, thus, minimize the management and investment cost.

Another complexity layer is added by the estimation of future demand. Indeed, the company under study receives a demand update from the *OEM*s every three months, thus it has a deterministic demand only for the upcoming three months. By the way, it needs an estimation of the demand for the period after the given three months, thus, they developed a forecasting method whose final output is a scenario evolution three like the one depicted in Figure 1.4. In this representation we have a set of 4 scenarios $\omega \in \Omega$, one for each leave of the tree, described by an occurrence probability π_ω and a series of scenario nodes, one for each time bucket considered, represented by $t \in T$. In other words, a scenario node is particular situation of the demand, while a scenario is a possible evolution of such a situation. In the example proposed, we have a root scenario node (D_0) that is the three-months deterministic demand, known by the company and 4 possible evolutions, along 3 time buckets. Thus, the part demand is represented as an uncertainty value whose evolution is not sure but only predictable. Grounding on these

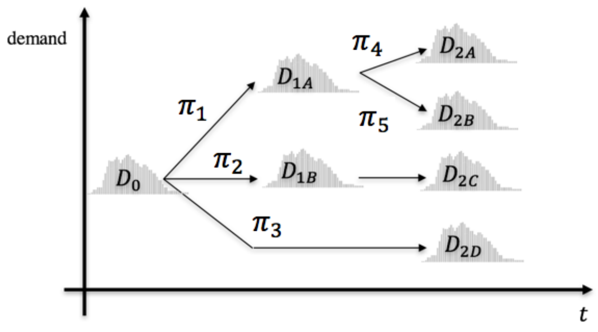


Figure 1.4: Example of scenario evolution tree.

consideration, the assembly line's design phase has to consider a set of technological rules and an uncertain environment in which the demand could evolve in different ways. As a consequence, it is needed a analytical approach that support the design and management phase by considering the strong influence the stochastic environment has on the problem. In this context the industrial and scientific problem to be faced is the following: *the development of a method for the initial configuration and future reconfiguration and management of a modular assembly line used in the automotive sector with uncertain demand.*

2 Solution framework

The problem under study is addressed using the framework in Figure 2.1. The approach takes as input the uncertain demand coming from the market (represented with a scenario evolution tree) and the technology features about products and machines. It gives the initial configuration of the line and its reconfiguration plan defined as *configuration evolution plan* together with the investment and management costs as the output. The approach is composed by three steps:

- the *Assembly line configurator*, during which the information coming from market and *OEM*s are analyzed; a set of candidate layout configurations are then generated. These configurations lie on the real information about shape and measure of the machines and on their real technological features.

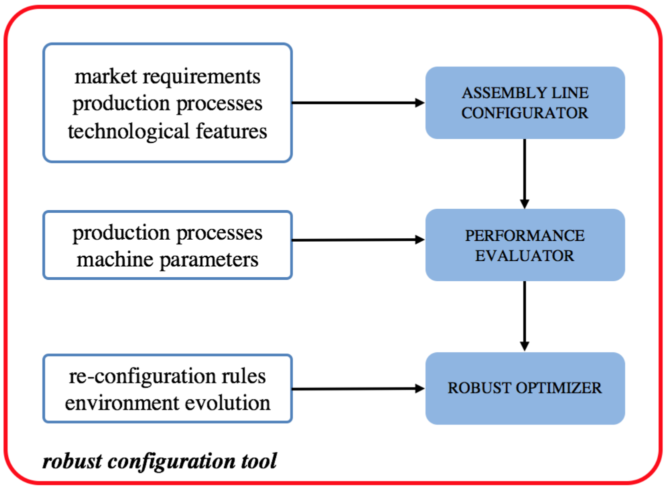


Figure 2.1: Approach framework.

- the *Performance evaluation*, during which every layout candidate performances are evaluated considering the assembly process of the products under study;
- the *Robust optimizer*, during which the *configuration evolution plan* that minimize a cost function is generated and identified.

The *Assembly line configurator* manages and analyze the input coming from the market, as deterministic demand in the first period and the estimated ones for the following, the description of machines, modules and lines in terms of size and costs, the description of products to be assembled in terms of assembly processes and technologies needed. Grounding on these information, the first step generates a set of candidate configurations for every scenario node $\omega \in \Omega$, in every time bucket $t \in T$ considered, namely set Z. Each candidate $z \in Z$ addresses the demand technological request of the related scenario node by including in the layout the set of machines and technologies (namely $j \in J$) requested by the product assembly processes. The set of products is represented by $p \in P$. The configurations are generated by considering an architecture like the one in Figure 1.2, in which the handling robot is at the center of the line and the other modules are arranged at both its sides. In addition to this, we consider fixed the position of input and output stations on the opposite sides of the line. Once fixed these three components, the other modules are randomly positioned all along the line. The output of this process is a set of configurations which layout is described by the position in the space of each component of the line.

During the second step, the performances of every configuration are estimated with the *Performance evaluation tool*. We work under the assumption that a different position of a module in the layout can ensure different performances due to the different movement the handling robot has to execute. In particular, we use a stochastic estimator developed at the Mechanical Department of the Politecnico di Milano that uses a Markov Chain formalization to asses the transient/steady state performance.[2] In particular, it generates dynamic model of system behavior by building a net of all possible system states and evaluate its performances analytically. The processing times are represented by *Phase-type Distributions* that allow to model the system with a *Continuous Time Markov Chain.* A system state is described by a vector $s = | c_1, c_2, \ldots, c_N, r |$, where each module of the configuration c_i could be *Operative* (O), *Starved* (S) or *Blocked* (B) and the handling robot r takes values $[0, \ldots, N, N+1]$, where N is the number of modules included in the line. The dynamics of the assembly cell are described by logical expressions, modeling the sequence of events considering failure and repair rate of every machine included in the line. Grounding on this, the tool estimates the absorbing time of the state representing the exit of a finished part from the line. This absorbing time is then translated into performances in terms of throughput (parts/second) for each type of product p considered by the configuration z, namely $TH(z,p)$. This value is then used by the approach in the following step.

At least we have the *Robust optimizer* that is able to identify the *configuration evolution plan* that minimizes a cost function on the entire evaluation horizon. This tool generates a set of configuration evolutions, defined as the sequence of configurations of the assembly line, by considering every possible sequence using the layouts generated in the first step. Then, it evaluates their expected cost considering investment, operative and reconfiguration costs. The idea on which the configuration evolution grounds on is that every three months, with the update of the demand from the *OEM*s, it is possible to reconfigure the line, if needed. The optimization problem, fully described in (Manzini et al., 2017), is reported in the following.

$$minimize \left(c^{inv}(z_0) + c^{op}(z_0) + \sum_{\omega \in \Omega} \pi_\omega \frac{E[c^{inv}(z_t \mid z_0)] + E[c^{op}(z_t \mid z_0)]}{(1+q)^{t_\omega}} \right) \tag{2.1}$$

subject to

$$c^{inv}(z_{t_\omega}) = \sum_{j \in J} c^{inv}(j \in z_{t_\omega}),\ \forall \omega \in \Omega \tag{2.2}$$

$$c^{op}(z_{t_\omega}) = T_{total}(z_{t_\omega}) \cdot c^{hour},\ \forall \omega \in \Omega \tag{2.3}$$

$$T_{total}(z_{t_\omega}) = \sum_{p}^{P} TH(z_{t_\omega}, p) \cdot d_p^\omega < T_{available},\ \forall \omega \in \Omega \tag{2.4}$$

The *Robust optimizer* aims at minimizing the Objective Function (2.1), representing the expected value of the overall cost over all the scenarios, by selecting a sequence of configurations z_t, with $t \in T$. The configurations considered into this sequence will be chosen from the set Z. In particular, c^{inv} and c^{op} are investment and operational cost respectively, calculated by using the layout z_t in scenario node $\omega \in \Omega$, then represented

[2]The *Performance evaluator tool* has been developed by the student together with colleagues Angius and Ratti within the EU founded Research Project *Robust PlaNet.* The tool has been completely presented in Project Deliverable number $D\,2.3$ and $D\,2.5$, in (Angius et al., 2016) and in (Manzini et al., 2017).

with z_{t_ω}. Equation (2.1) is made up of two parts, the first one takes into account the initial configuration decision z_0 while the second one considers future decisions z_t, computing the expected costs in time period t based on the initial configuration decision z_0. In this case, the expected cost is computed by considering the occurrence probability π_ω of every scenario node in every time bucket t_ω. A discount rate q is applied, using t_ω as the time stage of the considered scenario node. The investment cost is calculated in Equation 2.2 as the sum of investment costs for every machine and equipment included in the layout, $c^{inv}(j \in z_{t_\omega})$. The operative cost is calculated in Equation 2.3 as the hourly cost c^{hour} multiplied for the total production time, estimated in Equation 2.4 using the throughput and the product demand d_p^ω for every product and every scenario node. This total time has to be shorter than the available time in that time bucket that is usually the amount of time in which the plant is open during the three months. The *Robust optimizer* estimates the expected total cost for every *configuration evolution* generated considering every demand scenario and their occurrence probability. After that, the optimizer picks up the *configuration evolution* that minimizes the expected cost.

3 Implementation

The software application has been implemented in a *Windows 8* environment using *Microsoft Visual Studio*. In addition to this, an application developed in *Java* is called and used during the computation.

The implementation is organized through a set of functions in which the same set of data is passed via *reference*. In the following subsections the implementation is described in detail. In particular, the data structure designed for the problem is addressed together with the input phase in Sections 3.1 and 3.2. After that, the data are analyzed and elaborated in order to generate a set of configuration candidates in Section 3.3, which performances are estimated in Section 3.4. Using the resulting performances, the optimal *configuration evolution* is identified and printed on screen and file in Sections 3.5 and 3.6.

3.1 Data structure

All the data used by the application (already described in previous sections) are aggregated in *struct* variables in the *data_input.h* header. In this section, all the *struct*s created and used will be described.

Each configuration (or line layout) is represented by the *struct* called *configuration*, whose structure is reported in Table 3.1. Each configuration description is composed by the number of *FAG*s (production machines) included in the layout (*num_FAG*), limited to 7[3]; for each *FAG* included, its name and position in plant are described. The position of each piece of equipment is represented in terms of the position of its central point on the floor, on the X and the Y axis. Also the position of the robot (with its average transport time), input station, central unit (CU) and output station

[3]We consider only 7 *FAG*s because in the industrial problem only 7 technologies are taken into account, as described in Section 1.

are included. The characteristic of each equipment (machines, input and output station, etc.) are included in the *struct* named *equipment*, reported in Table 3.3. For each equipment the following characteristics are included: name (FAG), $2D$ dimensions (X and Y), failure and repair rate (useful for the performance evaluation) and the investment cost. For each configuration generated during the execution of the application a set of performances will be estimated, thus, a *struct* has been created, it is reported in Table 3.4. Since the performances are estimated in terms of throughput (part produced per second), the *struct* includes a value for each product considered (4 maximum). The representation of the environment in which the problem is solved is represented through the *struct* named *scenario* and *scenario_ node*, in Table 3.6 and Table 3.5 respectively. For each scenario, its sequence of scenario nodes is represented by including in the *struct* their names, one for each time bucket (3 time buckets are considered); also its occurrence probability is included. Instead, for each scenario node, we included its name, the time bucket it belongs to (*tnode*), the number of products considered, and the name and demand for each one. The characteristic about each product are stored using the *struct* reported in Table 3.7. In particular, there are included the name of the product, the name of the machine used in each production step (3 steps are always considered) and the machining time needed (*step1_ prodrate*, *step2_ prodrate* and *step3_ prodrate*).

At least, some miscellaneous information are included in the *struct* called *miscellaneous*, reported in Table 3.2. In this *struct* some general information as the number of products, scenarios and scenario nodes used for characterizing the problem, the operative, reconfiguration cost and time, the robot speed (mm/sec), input and output time needed by the input and output stations are included. Also the time available in terms of hours for each time bucket, the cost of one millimeter of track of the robot and the discounted factor (*k_ attualizzazione*) are included.

Grounding on these representations, the set of data used during the computation is formalized used the vector command of the standard library, *std::vector<>*.

3.2 Input process

The input process is executed using a series of functions that read the information from text files and organize them into structures we already described. These functions are included in the file *input_ process.cpp* and uses the command *myfile.open(filename)* where *myfile* is the *ifstream* variable and *filename* is the name of the file to be opened. Once the file has been opened, the information are saved in the corresponding variable and then the file closed using the command *myfile.close()*.

For every input process the procedure is the same as the one described, with some differences on the management of information and files due to the *structure* to which the information belongs to.

3.3 Configuration generation

The configuration generation process is the one that elaborates the data coming from the user and generates a set of feasible configurations (or layouts) for each scenario node

```
struct configuration
        int num_FAG;
        int FAG1_name;
        int FAG1_posX;
        int FAG1_posY;
        int FAG2_name;
        int FAG2_posX;
        int FAG2_posY;
        int FAG3_name;
        int FAG3_posX;
        int FAG3_posY;
        int FAG4_name;
        int FAG4_posX;
        int FAG4_posY;
        int FAG5_name;
        int FAG5_posX;
        int FAG5_posY;
        int FAG6_name;
        int FAG6_posX;
        int FAG6_posY;
        int FAG7_name;
        int FAG7_posX;
        int FAG7_posY;
        int robot_posX;
        int robot_posY;
        double time_transport
        int CU_posX;
        int CU_posY;
        int input_posX;
        int input_posY;
        int output_posX
        int output_posY;
```

Table 3.1: *Configuration struct* declaration.

```
struct miscellaneous
        int num_scenario;
        int num_scenarionodes;
        int num_products;
        int op_cost;
        int rec_cost;
        int rec_time;
        double robot_speed;
        double input_time;
        double output_time;
        int time_available;
        double cost_track;
        double k_attualizzazione;
```

Table 3.2: *Miscellaneous struct* declaration.

considered. This process is executed with two functions.

The first one, *equipment_elaboration_2* elaborates the information and create the binary matrix called *equipment_matrix* in which, there is 1 if a particular equipment is requested in a particular scenario node, 0 otherwise. This matrix is elaborated with the exploration of every scenario node needs in terms of products and then in terms of equipment for each product. The command *switch* is used.

struct equipment
  int FAG;
  int x;
  int y;
  double fail_rate;
  double repair_rate;
  int cost;

Table 3.3: *Equipment struct* declaration.

struct performances
  double prod_1;
  double prod_2;
  double prod_3;
  double prod_3;

Table 3.4: *Performances struct* declaration.

struct scenario_node
  int name;
  int tnode;
  int num_prod;
  int name_1;
  int demand_1;
  int name_2;
  int demand_2;
  int name_3;
  int demand_3;
  int name_4;
  int demand_4;

Table 3.5: *Scenario_node struct* declaration.

struct scenario
  int step1_name;
  int step2_name;
  int step3_name;
  double occur_prob;

Table 3.6: *Scenario struct* declaration.

struct product
  int name;
  int step1_FAG;
  double step1_prodrate;
  int step2_FAG;
  double step2_prodrate;
  int step3_FAG;
  double step3_prodrate;

Table 3.7: Product *struct* declaration.

The second one, *configuration_ gen_ 0* uses the binary matrix *equipment_ matrix* and randomly arranges all the equipment needed around the handling robot track, as the example line in Figure 1.2. It disposes the equipment above and below the track randomly and then generates a *configuration* variable. This file is added to a vector of *configuration struct* using the command *push_ back()*. The output is a vector of *configuration*s, one for each scenario node considered.

3.4 Performance evaluation

This phase estimates the performances (in terms of throughput) for every configuration generated in each scenario node and considering the production of every product demanded in that scenario node. Function *performance_ evaluator_ 2* is used. This phase exploits an application developed in *Java*[4] included in the project folder (together with the folder *lib*). This *Java* application can be started from the *command prompt* by indicating the name of input and output *.txt* files. The *performance_ evaluator_ 3* function main duties are:

- to write the input files;
- to start the application;
- to read and store the output information.

The input file is a *.txt* file containing all the information needed in a specific format. This file has to contain as much line as the number of operations in the assembly process, in which each line contains the word *Exp* (exponential distribution), the failure rate and the repair rate of the equipment involved in the operation, and the production rate of that operation on the specific product under study. This file is written using the *file_ write* command.

The application is executed using the *system("java -jar VoestAlpine.jar input.txt output.txt")* command that also indicates the input and output files, *input.txt* and *output.txt* respectively. The output of the application is a *.txt* file containing the throughput (expected value) of the entire line in the first line and other information about equipment utilization in following lines. Using the *file_ read* command, the *performance_ evaluator_ 2* function is able to read the output of the evaluation procedure and store this information into the performance vectors.

3.5 Robust optimization

The robust optimization is executed within the *evolution_ gen_ v1* function that generates all the possible configuration evolution (using all the evaluated layouts), estimates their expected total cost and then identifies the best solution among all the evolutions.

Considering all the layouts generated and evaluated in the previous steps, this function generates every possible evolution combination on the three time buckets. After that,

[4]Its reference file is the one called *VoestAlpine.jar*.

the expected cost presented in Equation 2.1 is calculated for every evolution by taking into account:

- an investment cost as the cost for purchasing the set of equipment and reconfigure the line;
- an operative cost as the cost for each time unit used for producing part and reconfigure the line.

The total expected cost is then calculated as the total purchasing and management (or operative) costs of the line in each scenario weighted for its occurrence probability. Once every possible configuration evolution has been identified and evaluated, the one that ensures the minimum expected total cost is then stored together with its total cost.

3.6 Output process

The output process, executed by the *output_0* function works on two levels, a file output and a video output. In both cases, the output of the computation is composed by the representation of the configuration in the first, second and third time bucket (*configuration_evolution.txt*), and the relative total expected cost (*opt_cost.txt*).

4 Application

The application case considers a configuration problem in which assembly lines like the one described in Section 1 are used. In particular, this application case considers 3 time buckets over which 3 scenarios are modeled; the resulting *scenario evolution tree* is reported in Figure 4.1 in which also occurrence probability for each scenario node is included. Each scenario is characterized by an occurrence probability and a scenario node for every time bucket (reported in Table 4.1). Each scenario node is a possible

	Occurrence prob.	**Time** 1	**Time** 2	**Time** 3
Scenario 1	0.5	1	2	4
Scenario 2	0.4	1	3	5
Scenario 3	0.1	1	3	6

Table 4.1: For each scenario considered, its occurrence probability and the scenario node for each time bucket are included.

market situation that could occur in a given time bucket and characterized by product demands, reported in Table 4.2. In this application case, only 3 products are considered, each one characterized by a three step assembly process and reported in Table 4.3.

The application case considers a set of 11 possible *FAG*s including the mandatory ones (handling robot, input and output stations, and the control unit). In Table 4.4, we report

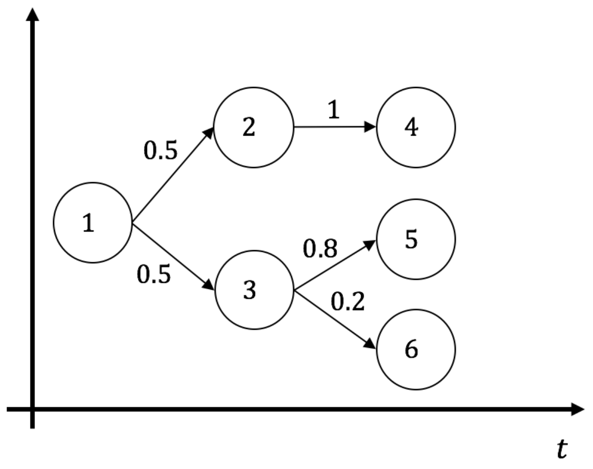


Figure 4.1: Application case scenario evolution tree.

	Time bucket	**Prod.** 1	**Prod.** 2	**Prod.** 3
Scenario node 1	1	$75\,u$	–	$100\,u$
Scenario node 2	2	$55\,u$	$75\,u$	$80\,u$
Scenario node 3	2	$50\,u$	$70\,u$	$90\,u$
Scenario node 4	3	$100\,u$	–	$65\,u$
Scenario node 5	3	$55\,u$	–	$150\,u$
Scenario node 6	3	$15\,u$	–	$55\,u$

Table 4.2: For each scenario node considered, its time bucket and the volume of demanded products (in units $[u]$) are included.

	Prod. 1	**Prod.** 2	**Prod.** 3
First operation FAG	5	6	5
First operation time	$10\,sec$	$18\,sec$	$19\,sec$
Second operation FAG	6	7	6
Second operation time	$15\,sec$	$20\,sec$	$22\,sec$
Third operation FAG	7	8	7
Third operation time	$20\,sec$	$30\,sec$	$29\,sec$

Table 4.3: For each product considered, its assembly process in terms of FAG and operation time of each step is included.

the complete description of each piece of equipment in terms of X and Y size (in mm)[5],

[5]Since we use modular equipment for arranging equipment in the line, we report the same size for the equipment number $5 - 11$. For the handling robot we did not report any Y dimension because it is automatically addressed during the computation as the length of the line; by the way, a random

failure and repair rate[6], and investment cost. At least, miscellaneous set of information

	X	Y	**Fail. rate**	**Rep. rate**	**Cost**
Handling robot (1)	$500\,mm$	–	0.03	0.8	150,000 €
Control Unit (2)	$600\,mm$	$600\,mm$	–	–	200,000 €
Output station (3)	$500\,mm$	$700\,mm$	0.031	0.82	20,000 €
Input station (4)	$550\,mm$	$720\,mm$	0.025	0.7	25,000 €
Mechanical joining (5)	$700\,mm$	$700\,mm$	0.022	0.9	80,000 €
Resistance joining (6)	$700\,mm$	$700\,mm$	0.03	0.84	70,000 €
MIG joining (7)	$700\,mm$	$700\,mm$	0.018	0.9	90,000 €
Adhesive joining (8)	$700\,mm$	$700\,mm$	0.035	0.88	88,000 €
Hemming (9)	$700\,mm$	$700\,mm$	0.03	0.8	95,000 €
Manual operations (10)	$700\,mm$	$700\,mm$	0.028	0.99	50,000 €
Other (11)	$700\,mm$	$700\,mm$	0.03	0.8	10,000 €

Table 4.4: For each piece of equipment considered, its assembly dimensions, repair and failure rate, and investment cost are included.

are included in Table 4.5. The problem under study considers a demand for the product

Number of scenarios	3
Number of scenario nodes	6
Number of products	3
Operative cost	10 €$/h$
Reconfiguration cost	20,000 €
Reconfiguration time	$80\,h$
Robot speed	$5\,mm/sec$
Input time	$6\,sec$
Output time	$5\,sec$
Time available	$160\,h$
Unitary track cost	2 €$/mm$
Discount rate	0.07

Table 4.5: Miscellaneous data.

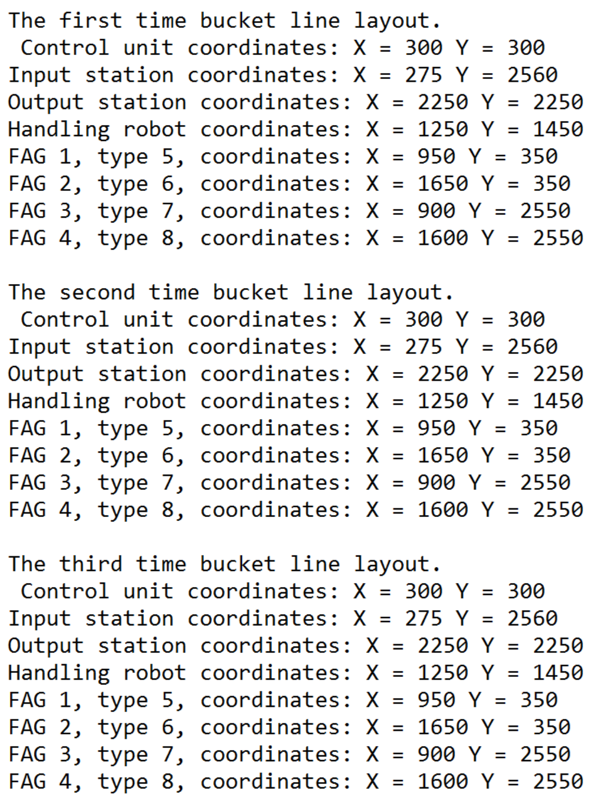
number 2 only in the second time bucket. Looking at the equipment requirements of each assembly process, we have that product number 2 asks for *FAG* number 8 instead of the number 5 of product 1 and 3. It means that only in the second time bucket the *FAG* number 8 is requested. Grounding on this, a possible solution could be to adopt a line with 3 *FAG*s in the first and third time bucket, and a line with 4 *FAG*s only in the

dimension has to be included in the input *.txt* file, this dimension will not affect the computation.

[6]The control unit reliability does not affect the line performances, thus we included two *null* values. By the way, random values have to be included in the input files, these values do not affect the computation.

second time bucket, undergoing a reconfiguration twice.

Instead, the solution obtained using the presented application considers to have the same layout for the whole time horizon, in order to not consider any reconfiguration cost that could heavily affect the total cost. The optimal *configuration evolution plan* is the one depicted in Figure 4.2 with a total expected cost of $730,229$ €. In Section 6, it is shown how to read this output and how it depicts a schematic $2D$ representation of the line layout.



```
The first time bucket line layout.
 Control unit coordinates: X = 300 Y = 300
Input station coordinates: X = 275 Y = 2560
Output station coordinates: X = 2250 Y = 2250
Handling robot coordinates: X = 1250 Y = 1450
FAG 1, type 5, coordinates: X = 950 Y = 350
FAG 2, type 6, coordinates: X = 1650 Y = 350
FAG 3, type 7, coordinates: X = 900 Y = 2550
FAG 4, type 8, coordinates: X = 1600 Y = 2550

The second time bucket line layout.
 Control unit coordinates: X = 300 Y = 300
Input station coordinates: X = 275 Y = 2560
Output station coordinates: X = 2250 Y = 2250
Handling robot coordinates: X = 1250 Y = 1450
FAG 1, type 5, coordinates: X = 950 Y = 350
FAG 2, type 6, coordinates: X = 1650 Y = 350
FAG 3, type 7, coordinates: X = 900 Y = 2550
FAG 4, type 8, coordinates: X = 1600 Y = 2550

The third time bucket line layout.
 Control unit coordinates: X = 300 Y = 300
Input station coordinates: X = 275 Y = 2560
Output station coordinates: X = 2250 Y = 2250
Handling robot coordinates: X = 1250 Y = 1450
FAG 1, type 5, coordinates: X = 950 Y = 350
FAG 2, type 6, coordinates: X = 1650 Y = 350
FAG 3, type 7, coordinates: X = 900 Y = 2550
FAG 4, type 8, coordinates: X = 1600 Y = 2550
```

Figure 4.2: Optimal *configuration evolution plan* for the given application case.

5 Conclusion

In this document, we presented an approach for the configuration and reconfiguration of an assembly line used in the automotive sector. In particular, the approach considers an uncertain evolution of the demand and thus manages a stochastic problem. In addition to this, we include in this approach (i) a *layout generator* that considers the particular technology used, (ii) a *performance evaluator tool* that models the assembly process as a *Markov Chain* and estimates the average throughput of the line, and (iii) a *robust optimizer* that takes into account the demand uncertainty and minimize the expected investment and management cost of the line through a fixed time horizon.

Possible future extension of this tool could be focused on (i) the generalization of the number of products and time buckets, or (ii) the adoption of another system type to be designed.

References

Angius, A., Colledani, M., Manzini, M., Ratti, A., Urgo, M., 2016. Equipment selection and evaluation approach for an adaptable assembly line. IFAC-PapersOnLine 49 (12), 47–52.

Manzini, M., Unglert, J., Gyulai, D., Colledani, M., Jauregui-Becker, J. M., Monostori, L., Urgo, M., 2017. An integrated framework for design, management and operation of reconfigurable assembly systems. Omega In press.

Tolio, T., 2009. Designing of Flexible Production Systems, Methodologies and Tools. Springer.

Tolio, T., Ceglarek, D., ElMaraghy, H. A., Fischer, A., Hu, S. J., Laperriere, L., Newman, S. T., Váncza, J., 2010. Species - co–evolution of products, processes and production systems. CIRP Annals–Manufacturing Technology 59 (2), 672–694.

Wiendahl, H.-P., ElMaraghy, H. A., Nyhuis, P., Zäh, M. F., Wiendahl, H.-H., Duffie, N., Brieke, M., 2007. Changeable manufacturing-classification, design and operation. CIRP Annals-Manufacturing Technology 56 (2), 783–809.

6 User manual

In this section, the user manual is included, in order to help the final user of the application to use it in the correct way.

6.1 Prerequisites

The application has been developed in a *Visual Studio* environment for *Windows 8* but, it could be possible to compile the application in any operative system. In addition to this, the performance evaluator tool has been developed in *Java*, thus the *Java Runtime Environment* has to be installed on the machine.

The way in which the application has been developed limits the size of the problem to the following:

- maximum 4 product can be considered;
- each product has 3 production steps;
- maximum 7 *FAG*s can be considered (plus the 4 mandatory *FAG*s);
- 3 time bucket are considered, thus every scenario is composed by 3 scenario nodes

6.2 Input format

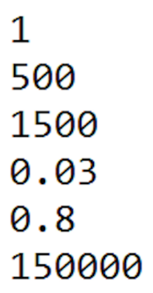
All the input files have to follow a given template in order to allow the application read the information. In the following, for every input type, we propose an example to show the reader the right template. Every file named in this Section has to be included in the *input* folder, subfolder of the project folder.

The input files that describes each equipment have to be named *equipment_#.txt* where the # is the reference number of each equipment, following the list:

1. handling robot;
2. control unit;
3. output station;
4. input station;
5. mechanical joining;
6. resistance joining;
7. MIG joining;
8. adhesive joining;
9. hemming;
10. manual operations;

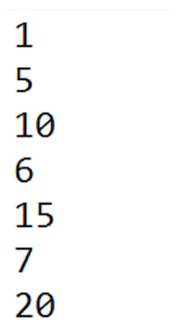
11. other.

In such a text file, data reported in Table 3.3 are included, one per rows, as depicted in Figure 6.1a. The product description is depicted in the file named *product_#*, where # is the reference number of each product. In this file, the data described in Table 3.7 are included, one item for each row, as in exemplar Figure 6.1b. Miscellaneous information have to be included in file *miscellaneous.txt* in the same order as depicted in Table 3.2 and in the exemplar Figure 6.2a. Each scenario is represented with a text file named



```
1
500
1500
0.03
0.8
150000
```

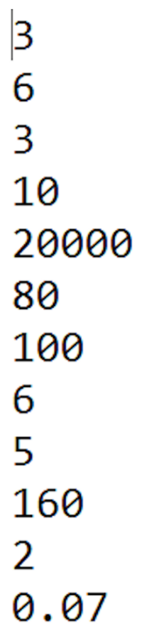
(a) Exemplar *equipment_#.txt* file.



```
1
5
10
6
15
7
20
```

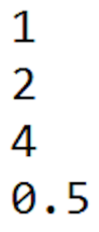
(b) Exemplar *product_#.txt* file.

Figure 6.1: Exemplar input files.



```
3
6
3
10
20000
80
100
6
5
160
2
0.07
```

(a) Exemplar *miscellaneous.txt* file.

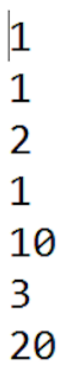


```
1
2
4
0.5
```

(b) Exemplar *scenario.txt* file.

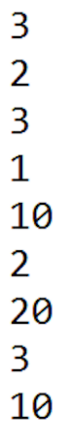
Figure 6.2: Exemplar input files.

scenarionode_#.txt, where # is the reference name of each scenario node. In this file, all the information have to be reported, one per line, as depicted in Table 3.5. Examples with 2 and 3 products are depicted in Figure 6.3a and Figure 6.3b respectively. At least, each scenario is included in a text file named *scenario_#.txt* , where # is reference number of each scenario set. The information listed in 3.6 are included in the file, one per each row, as in Figure 6.2b.



```
1
1
2
1
10
3
20
```

(a) Exemplar *scenarionode_#.txt* file, with 2 products.



```
3
2
3
1
10
2
20
3
10
```

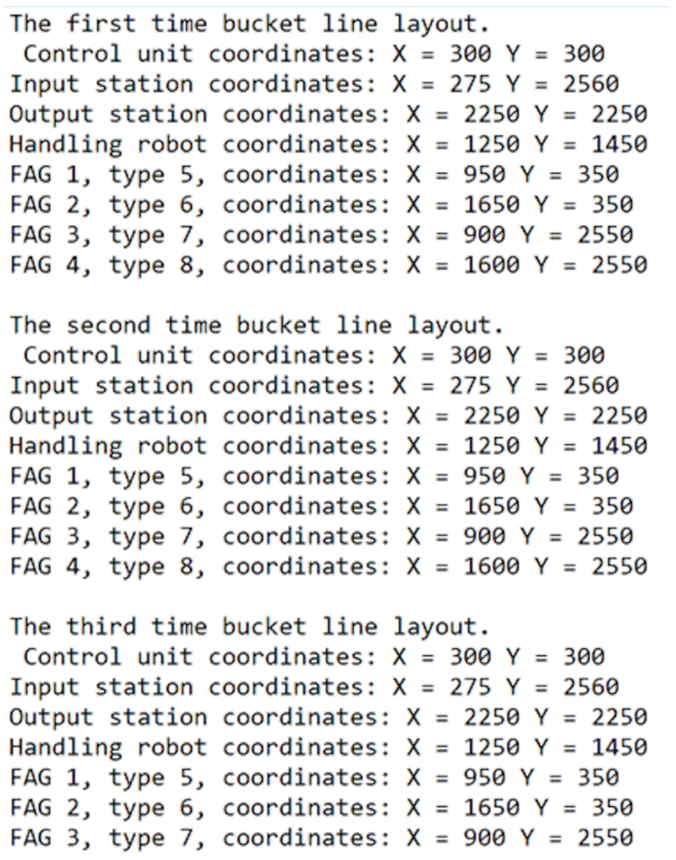
(b) Exemplar *scenarionode_#.txt* file, with 3 products.

Figure 6.3: Exemplar input files.

In addition to these, two additional *.txt* files named *input.txt* and *output.txt* have to be included directly in the application folder. These files will be used by the *performance evaluator* tool.

6.3 Output format

As done for the input files, we propose also a guide for the outputs. We have only two output files (which information are also proposed on screen), the first one (*opt_ cost.txt*) that simply includes the (optimal) total expected cost, and the second one (*configuration_ evolution.txt*) including the layout evolution. Both *.txt* files have to be included in the same folder as the application before starting the computation. An exemplar *configuration_ evolution.txt* is depicted in Figure 6.4a. In this file, the chosen layout for each time bucket is described through the position of its equipment on the floor. As done previously for the configuration generation, for each equipment, the coordinates (X and Y) of its central point in the space are reported. In the exemplar output, we have a layout with 4 *FAG*s in the first and in the second time bucket and a layout with only 3 *FAG*s in the last one. For each equipment, we have its name and its position. Once the user has these coordinates, it is possible to plot them in *Cartesian graph* and build the perimeter of each equipment around them, as done in Figure 6.4b for the first time bucket layout. Using this representation, the user can understand the disposition of each equipment included in the layout and thus use the obtained results.

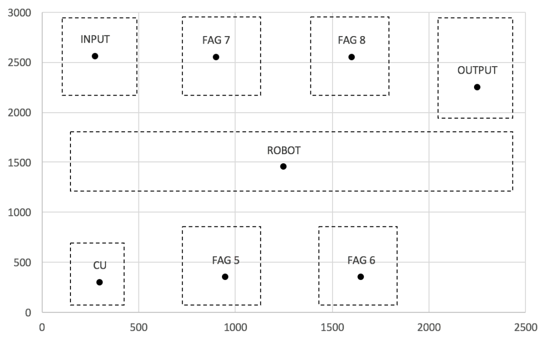


```
The first time bucket line layout.
 Control unit coordinates: X = 300 Y = 300
Input station coordinates: X = 275 Y = 2560
Output station coordinates: X = 2250 Y = 2250
Handling robot coordinates: X = 1250 Y = 1450
FAG 1, type 5, coordinates: X = 950 Y = 350
FAG 2, type 6, coordinates: X = 1650 Y = 350
FAG 3, type 7, coordinates: X = 900 Y = 2550
FAG 4, type 8, coordinates: X = 1600 Y = 2550

The second time bucket line layout.
 Control unit coordinates: X = 300 Y = 300
Input station coordinates: X = 275 Y = 2560
Output station coordinates: X = 2250 Y = 2250
Handling robot coordinates: X = 1250 Y = 1450
FAG 1, type 5, coordinates: X = 950 Y = 350
FAG 2, type 6, coordinates: X = 1650 Y = 350
FAG 3, type 7, coordinates: X = 900 Y = 2550
FAG 4, type 8, coordinates: X = 1600 Y = 2550

The third time bucket line layout.
 Control unit coordinates: X = 300 Y = 300
Input station coordinates: X = 275 Y = 2560
Output station coordinates: X = 2250 Y = 2250
Handling robot coordinates: X = 1250 Y = 1450
FAG 1, type 5, coordinates: X = 950 Y = 350
FAG 2, type 6, coordinates: X = 1650 Y = 350
FAG 3, type 7, coordinates: X = 900 Y = 2550
```

(a) Exemplar *configuration_ evolution.txt.*



(b) Simplified representation of the first time bucket line layout.

Figure 6.4: Exemplar output files.