# Event and object oriented simulation to fast evaluate operational objectives of mixed model assembly lines problems

Lorenzo Tiacci*

*Università degli Studi di Perugia - Dipartimento di Ingegneria Industriale, Via Duranti, 67 – 06125 – Perugia, Italy*

**Abstract**

In this paper an event and object oriented simulator for assembly lines is presented. The tool, developed in Java, is capable to simulate mixed model assembly lines, with stochastic task times, parallel stations, fixed scheduling sequences, and buffers within workstations. The simulator is a flexible supporting tool in finding solution of the mixed model assembly line balancing problem (and the optimal sequencing and buffer allocation problems associated to it). It is capable to immediately calculate the throughput of a complex line, by simply receiving as inputs three arrays representing: the task times durations, the line configuration (number of workcentres, of buffers within them, of parallel workstations in each workcentre, of tasks assigned to each workcentre), and the sequence of models entering the line. Its fastness and flexibility allow its utilization in those algorithms and procedures where the evaluation of a fitness function (which includes the throughput as performance indicator) has to be performed several times. It allows overcoming the limit of using others measures of throughput, presented in literature, that are poorly correlated to its real value when the complexity of the line increases. The simulator is an expandable tool; in its current version provides the possibility to simulate both straight and U-shaped lines, and to generate both random and fixed sequences of models entering the line.

*Keywords:* Event-oriented simulation; object-oriented simulation; line balancing; mixed model; sequencing; buffer allocation; U-shaped lines.

## 1. Introduction

The Assembly Line Balancing Problem (ALBP) has been one of the most studied problems in the literature related to industrial engineering. It consists in assigning tasks to workstations, while optimising one or more objectives without violating any restriction imposed on the line (e.g. precedence constraints among tasks). The basic version of the general problem is the so-called Simple Assembly Line Balancing Problem (SALBP). Its main characteristics are: serial line lay out with a certain number of stations, paced line with fixed cycle time, only one product is assembled, task times are deterministic and all stations are

* Corresponding Author: Tel.: +39-075-5853741; fax +39-075-5853736.
*E-mail address:* lorenzo.tiacci@unipg.it (L.Tiacci).

equally equipped with respect to machines and workers. For an overview of exact methods and heuristics developed to solve the SALBP see Scholl and Backer (2006).

The assumptions of SALBP are very restricting with respect to real-world assembly line systems. Therefore, researchers have recently intensified their efforts to identify, formulate and solve more realistic problems, that are embraced by the term *generalized assembly line balancing problem* (GALBP). For example, in today's time of ever more personalized products, assembly lines assembling just one product are extremely rare. On the contrary, several products (models) are often manufactured on the same line. The mixed-model assembly line balancing problem (MALBP) is a GALBP that address this issue, in which units of different models can be produced in an arbitrarily intermixed sequence. For a comprehensive classification of other different possible features of the GALBP see Becker and Scholl (2006).

As the number of features that try to capture the complexity of real cases increases (e.g. mixed models, sequencing policies, parallel workstations, buffers, stochastic task times) evaluating the performance of an assembly line become more complicated. Exact analytical methods for evaluating system throughput are not available. Thus, algorithms that have been developed to solve the problem, and that need to evaluate line performances in order to compare design alternatives, use some 'measures' of the throughput, represented by parameters that are easily calculable (once the line configuration and the tasks assignments are given) and that should be correlated to the real line throughput.

The delicacy of this issue has been properly outlined by Bukchin (1998). He argued that the only practical method to accurately evaluate throughput is a simulation study, which is very time consuming and hard to perform, and that for this reason, instead, various performance measures are usually used in order to evaluate and compare design alternatives. To find measures that can be correlated with the throughput, he observes for example that the allocation of assembly times to stations is characterized by two types of variability: a 'Model Variability' (variability of assembly times of a certain model assigned to different stations) and a 'Station Variability' (variability of assembly times of different models assigned to a specific workstations). These two types of variability are often considered correlated to blockage and starvation, and, as a consequence, to high idle times within stations and, eventually, to low throughput. However, the correlation of these (and any other) performance measures with the objective in question (i.e. the throughput) has to be verified through a simulation study.

Unfortunately, results presented in literature show that the correlation between measures proposed by researchers and the real throughput is often poor, and rapidly decreases when the number of stations in the line increases. Furthermore, the discrepancy between measure and throughput is in general expected to increase with the complexity of the line (e.g. parallel stations, buffers within workstations) and with the number of aspects of the real world that are considered into the model (e.g. stochastic tasks times). Thus, the risk in this research field is to assist to the continuous development of sophisticated algorithms that try to take into consideration several aspects of real problems, but that point toward a wrong objective, because they are 'driven' by measures not correlated to the real objective.

The only way to properly evaluate the throughput would be, as said, to build a simulation model of the line, and to perform some simulation runs. This approach is very time consuming because includes the time needed for building the model for each (different) line configuration, and to execute simulation run(s). Thus, it is usually adopted only to analyze the final solution obtained by an algorithm, but cannot be utilized in the very process of finding solutions, when many evaluations of different lines configurations have to be done (consider for example how many times a fitness function has to be evaluated by a genetic algorithm).

The aim of this work is to show how to overcome the limit of using performance measures instead of simulated throughput, by the use of a parametric simulator for mixed model lines, able to model and simulate different solutions in such a quickly way that it can be used iteratively in algorithms and procedures. At this scope, a Java based event oriented simulator is presented. The tool is capable to immediately model a complex line and to calculate its throughput (and others performances indicators) by

simply receiving as inputs three arrays representing: the task times durations, the line configuration (number of workstations, of buffers within them, of parallel stations in each workstations, of tasks assigned to each workstations), and the sequence of models entering the line.

In the following, a literary review on features and operational objectives considered for the MALBP is reported. Then, modeling features, inputs required and outputs provided by the simulator are described. Section 3 summarily describes the event-oriented approach utilized and the simulator structure. Implementation issues are discussed in Section 4. In Section 5 the simulator performances are tested, and execution times are compared both with a process-oriented implementation of the same tool, and with Arena.

## 2. Operational objectives and their measures

In recent years many heuristic and meta-heuristic methods have been proposed in literature to find solution to the MALBP. Studies published in the last ten years utilize different approaches, such as: Simulated Annealing (McMullen and Frazier, 1998; Vilarinho and Simaria, 2002), Ant techniques (McMullen and Tarasewich, 2003), Genetic Algorithms (Simaria and Vilarinho, 2004; Tiacci et al., 2006; Noorul Haq et al., 2006) and other heuristics (McMullen and Frazier, 1997; Askin and Zhou, 1998; Merengo et al., 1999, Jin and Wu, 2002; Bukchin et al., 2002; Karabati and Sayin, 2003).

In order to compare design alternatives, some 'metric' have to be established. As in many industrial problems, in the MALBP a trade-off exists between costs (associated to labour and equipments) and performances. For this reason, objective functions of algorithms that find solution to MALBP usually includes both these objectives. But while to calculate costs associated to a determined line configuration is relatively easy (assuming that labour and equipments costs are known), to evaluate the performances of a mixed model assembly line may be, as already mentioned, complicated. Thus, most of the above mentioned studies uses some 'measure' to estimate the throughput (taken as the main performance objective), but do not address the problem to verify the validity of such measures. Only two studies on MALBP directly address this problem, namely Buckhin (1998) and Venkatesh and Dabade (2008).

The work of Buckhin (1998) is based on a comparison between 5 performance measures for throughput with simulation results. Three of them had already been proposed by other authors, namely the 'Smoothed Station' measure (Thomopoulos, 1979), the 'Minimum Idle Time' measure (Macaskill, 1972) and the 'Station Coefficient of Variation' (Fremerey, 1991), while the other two have been introduced by the author: the 'Model Variability' and the 'Bottleneck'. The latest one is obtained by estimating the expected value of the assembly time at the bottleneck station. Model Variability and Bottleneck have been found to outperform the other measures in showing a better correlation with the simulated throughput. On the basis of these results, Buckhin et al. (2002) incorporated the Bottleneck into a balancing heuristic algorithm in order to compare the quality of solution for a MALBP, while Tiacci et al. (2006) used a modified version of Model Variability in the fitness function of a Genetic algorithm for balancing mixed model lines with parallel workstations and stochastic task times.

Venkatesh and Dabade (2008) proposed other two measures, namely the Squared Based Model Deviation (SBDM), and the BMI, equal to the sum of the SBMD and the Smoothness Index (SI). The two performance measures along with eight others (reported in literature earlier) have been used as fitness function for a Genetic Algorithm to obtain solutions of a total of 3000 MALBP instances. Then, a statistical analysis was conducted on these solutions to compare the performance measures with regard to their ability to represent the operational objectives; the statistical analysis showed a low correlation of almost all measures with the realized throughput, with best results obtained by the SI. Contrary to what found by Buckhin, the Bottleneck showed a very low correlation with throughput. Authors selected other two operational objectives, besides the throughput, to which compare their measures: the Model Variability, and the Station Variability. At this purpose, a consideration has to be done. Model Variability

and Station Variability should not be considered 'operational objectives', but measures correlated to an operational objective (the throughput). For example, the fact that assembly times of different models are similar in each station (i.e. 'Station Variability' is low), it is not an objective in itself, but a (supposed) mean to achieve a high and stable throughput. Furthermore, while the throughput is difficult to estimate, the Model Variability and the Station Variability, for givens tasks assignments, can be calculated straightforwardly and do not need to be 'evaluated' through a measure.

More in general, methods that are developed to seek station assignments that lead to more balanced workloads across stations and across products are motivated to limit the effect, on the realized cycle time, of the sequencing of different models on the assembly line. However, as outlined by Karabati and Sayin (2003), these methods remain to be approximate approaches, for the very reason that the effects of the sequencing decision on the line throughput are not incorporated explicitly. Due to the computational complexities involved, the assembly line balancing problem and sequencing problems are usually addressed in literature independently of each other, although they are closely interrelated.

Another problem that is strictly connected to the balancing one is the buffer allocation problem (BAP). Studies in literature on BAP mostly refer to flow-lines production systems, consisting in a linear sequence of several machines. The decision to be taken is the amount of buffer space (used to avoid blockage and starvation to obtain high throughput) to install between successive stages; production rates in each stage are assumed to be known. The same problem can be transposed to assembly lines, with the difference that production rates in each station depend on which tasks are assigned to it, that is, on the result of the balancing procedure. Again, to obtain optimal results, buffer allocation and line balancing should be performed together. Literature about MALBP does not address the possibility to install buffers within stations. This is perhaps due to the additional difficulty in estimating line throughput.

In summary, the motivations for the development of the simulator that will be presented in the next sections are:

- results from the above mentioned studies (Buckhin, 1990; Venkatesh and Dabade, 2008) on the correlation between performance measures and realized throughput are not consistent each other, sign that it is difficult to find a throughput measure of general validity for all the possible issues of the problem;
- both studies consider deterministic task times and a relatively simple mixed model line configuration; a loss of correlation of the same performances measures with the simulated throughput is expected if task times are stochastic, and, in general, if the complexity of the line increases (number of workstations, buffers within workstations, parallel stations, etc.);
- the assembly sequence of the models is also important with respect to the realized cycle time, and this implies that the MALBP is also connected to a sequencing problem, that should not be treated separately from the balancing problem. Thus, the impact of models sequencing on cycle time should not be neglected.
- buffer allocation and line balancing decisions should be performed simultaneously. To do this, the impact of buffers on the line throughput has to be calculable.

## 3. The Assembling Line Simulator (ALS)

The aim of this work is to put at the scientific community and practitioners disposal an efficient, flexible and expandable tool, named Assembly Line Simulator (ASL), able to quickly model and simulate complex assembling lines. Its main characteristics are the ability to quickly build the model of a line configuration and to fast execute the simulation run. These two characteristics allow it to be easily 'embedded' in algorithms and procedures that find solutions to MALBP, overcoming the above mentioned limits imposed by the use of measures instead of the simulated throughput. In the following (Section 3.1), the types of assembling lines that can be modeled are illustrated, together with the representation of inputs

required by the simulator. In Section 3.2 the object-oriented structure and the event-oriented logic through which the simulator has been implemented are described.

*3.1. Description of the model*

In the line, each operator has a workstation (WS) where he performs one or more tasks. Each work centre (WC) consists of either one workstation, for the case of non-paralleling, or multiple parallel workstations (see Figure 1). 'Paralleling' means that when a WC consists of two or more workstations, all the tasks assigned to the WC are not shared among the WS, but each WS performs all of them. Thus an increment of production capacity of the WC is obtained through the addition of one (or more) WS which performs the same set of tasks. The aim of using parallel stations is often to perform tasks with processing time larger than the desired cycle time. However, also if any given task time does not exceed cycle time, the possibility to replicate workstations may be desirable, because it enlarges the space of feasible solutions of the balancing problem, including many feasible and potentially better balanced configurations (Vilarinho, 2002; Tiacci et al. 2006).

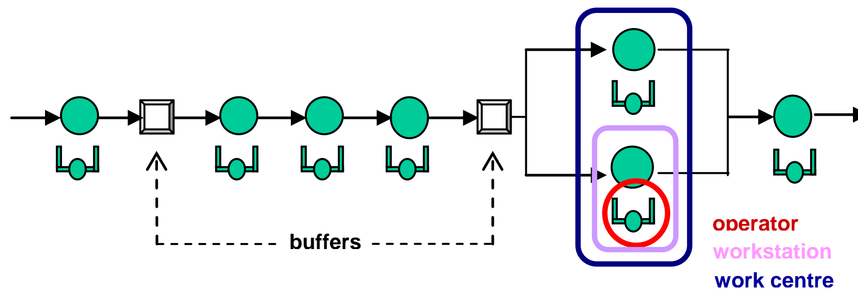



Figure 1. An assembly line with parallel workstations and buffers.

The line is asynchronous, that is as well as blockage and starvation are possible. One WC with multiple WSs is considered busy if every WS inside is busy. If a WS finishes its work on a workpiece while the subsequent WC is still busy (or the subsequent buffer is full), the workpiece can not move on, and remains in the current WS keeping it busy ('blocking after processing' policy); the WS will be released (i.e. will be able to process another workpiece) only when the workpiece leaves the WS.

Pieces are retrieved from buffers following a First In First Out rule. If a buffer is placed before a WC with parallel WSs, pieces are retrieved from the same buffer by all the WSs.

The first WC is never starved (there is always raw material for the first WC) and the last station is never blocked (there is always storage space for the finished product)

*3.1.1. Notation*

$i$ task index ($i = 0, ..., n$-1)
$j$ model index ($j = 0, ..., m$-1)
$k$ work centre index ($k = 0, ..., p$-1)
$t_{ij}$ time required by model $j$ in work centre $i$.

*3.1.2. Task times of completion*

A set of $n$ tasks (numbered with $i = 0,\dots, n-1$) has to be performed in the line in order to complete each product. Because we are dealing with mixed model lines, the number of models (types of product) to be assembled can be higher than one, and it is indicated by $m$ (numbered with $j = 0, \dots, m-1$). Input data are thus represented by an $n$ x $m$ matrix $t_{ij}$ whose elements represent the average completion time of task $i$ on model type $j$.

The 2 x 7 matrix depicted in Figure 2 represents the case in which 2 types of products (models) have to be assembled; each model requires 7 tasks to be completed. For example the average task time of task #4 of model #0 is equal to 5 minutes (or, in general, time units). It is noteworthy that if the completion of a model does not require the execution of a certain task, this would result in a 0 in the corresponding matrix element.

$t_{ij}$={{ 10, 8 },
{ 3, 9 },
{ 8, 8 },
{ 7, 8 },
{ 5, 9 },
{ 6, 10},
{ 13, 2 }}

a.

| | Model# | |
|---|---|---|
| Task# | 0 | 1 |
| 0 | 10 | 8 |
| 1 | 3 | 9 |
| 2 | 8 | 8 |
| 3 | 7 | 8 |
| 4 | 5 | 9 |
| 5 | 6 | 10 |
| 6 | 13 | 2 |

b.

Fig. 2. a. The $t_{ij}$ array representation.
b. Tabular representation of input task times.

*3.1.3. Representation of a line configuration*

A 'line configuration' represents a specific solution of the MALBP, and is characterized by the following information:

- the total number of WC in the line;
- the number of WS in each WC;
- the presence and the size of buffers before each WC;
- how many and which tasks are assigned to each WC (and are all performed by each WS assigned to the WC).

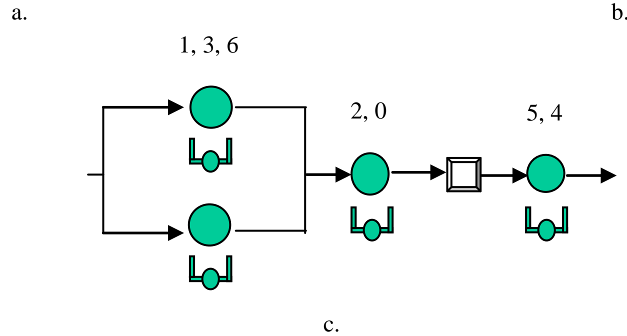
The line configuration can be represented by a two-dimensional array $lc_{kz}$ ($k = 0, \dots, p-1$), where $p$ (the number of rows) represents the total number of WC in the line. Each row represents a WC. The first element is the number of WSs assigned to the WC: a number higher than 1 means parallel WSs. The second element represents the size of the buffer placed before the WC: 0 means no buffer. The subsequent elements represent the tasks assigned to the WC. Note that rows do not necessarily contain the same number of elements. For example, Figure 3 shows a solution that represents a line composed by 3 WC. Tasks #1, #3 and #6 are assigned to WC#0, in which 2 WSs operate. Tasks #2 and #0 are assigned to WC#1 (with 1 WS), and task #4 and #5 are assigned to WC #2 (with 1 WS). A buffer of unit size is placed between WC#1 and WC#2. Note that because there are always raw materials for the first WC, the buffer before WC#0 is useless, having no impact on the line throughput, and its size should be set to 0.

$lc_{kz}$ = { { 2, 0, 1, 3, 6 },
{ 1, 0, 2, 0 },
{ 1, 1, 5, 4 } }

a.

| WC# | WSs assigned | Buffer size | Tasks assigned |
|---|---|---|---|
| 0 | 2 | 0 | 1, 3, 6 |
| 1 | 1 | 0 | 2, 0 |
| 2 | 1 | 1 | 5, 4 |

b.




c.

Fig. 3. a. The $lc_{kz}$ two dimensional array.
b. Tabular representation.
c. Graphical representation.

*3.1.4. Model sequencing*

The simulator allows also to specify the sequence of models assembled through the line: this sequence is represented by an array of integers *s*. For example, in case of two models (#0 and #1) the array *s* = {0,0,1,1} means that after two model#0, two model#1 will be assembled, and so on. *s* = {0,1,0,1} means that the two models are alternated. The sequence of numbers in the array (whose length can be arbitrarily chosen) is taken as reference for the generation of different model types entering the line.

*3.1.5 Stochastic task times*

In order to take into account another important feature of real assembly lines, stochastic task times have to be considered. The literature on stochastic ALBP is ample, and most authors assume the task times to be independent normal variates, which is considered to be realistic in most cases of human work (Whilhem, 1987). In this first version of *ALS*, task times duration can be modelled, through the definition of the string *distType*, in three ways: deterministic (*distType*=”DET”), normally distributed (*distType*=”NORM”), and exponentially distributed (*distType*=”EXP”). Deterministic task times are exactly defined by the matrix $t_{ij}$. If task times are normally distributed, the standard deviation $\sigma_{ij}$ of the completion time of task *i* for model *j* is taken equal to its mean value ($t_{ij}$) multiplied by a coefficient of variation *cv* ($\sigma_{ij} = cv \cdot t_{ij}$). If task times are exponentially distributed, the density function of completion time is characterized by the single parameter $t_{ij}$ (equal to the mean value and the standard deviation).

*3.2. Implementation*

ALS has been developed in Java, basically because the Java language is intrinsically clear and concise; the application can be easily distributed (deployed) through a single file (.jar), it is multiplatform, and there is no need of any legacy software to deal with. The simulator has been implemented using SSJ (L’Ecuyer et al., 2002; L’Ecuyer and Buist, 2005), a package for discrete event simulation in Java. SSJ

(which stands for Stochastic Simulation in Java) is an organized set of software tools offering general-purpose facilities for stochastic simulation programming in Java. It supports the event view, process view, continuous simulation, and arbitrary mixtures of these.

ALS is built using the object oriented-approach of Java and the event-oriented approach provided by SSJ. The event-oriented simulation approach is also referred in literature as “event-driven” simulation or “event-scheduling” approach. In this approach a system is modelled by identifying its characteristic events and then writing a set of event routines that give a detailed description of the state changes taking place at the time of each event. The simulation evolves over time by executing the events in increasing order of their time of occurrence (Law and Kelton, 2000, Zeigler et al., 2000, Banks et al., 2004).

To implement the simulator 4 classes, representing the physical configuration, have been defined (see Fig. 4 for the simplified UML class diagram): `Line`, `WorkCentre`, `Workstation` and `Load`.

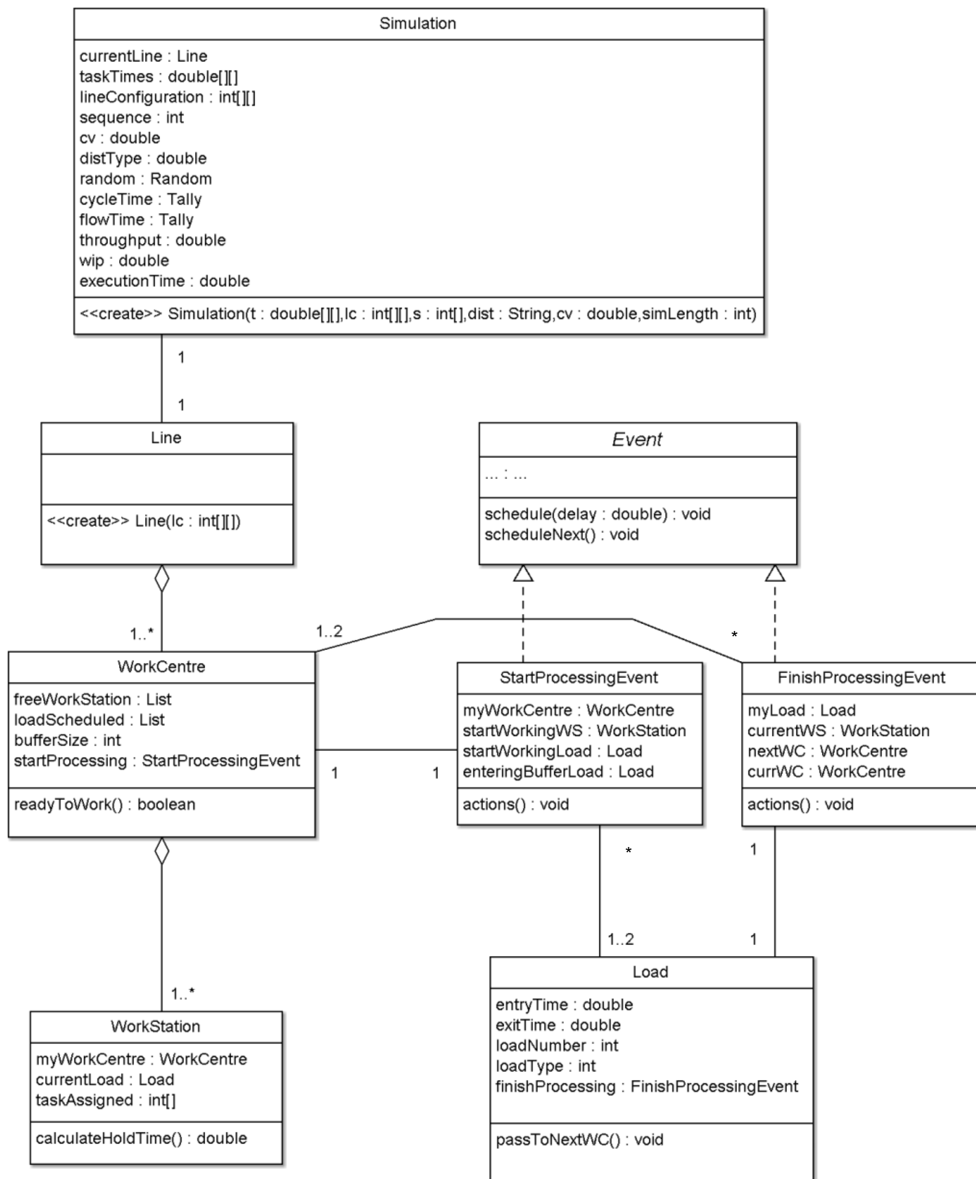



Fig. 4. A simplified UML class diagram of ALS.

The relations between these classes reproduce the relations between the corresponding entities in real world: an assembly line is formed by a set of WCs (`List workCentre`); each WC has a number of WS, some of which are available for loads processing (`List freeWorkStation`). Each WC has a virtual queue represented by the list of the loads (i.e. workpieces) that are waiting to be processed on the WC (`List loadScheduled`); a number of elements of this queue equal to the WC's buffer size (`int bufferSize`) is physically in the buffer of the WC, while the remaining elements are still waiting in the preceding WC. Each time that a load starts being processed in the WC, one WS is removed from the freeWorkStation List. If this list is empty the WC is considered busy, and cannot process any further load. The package 'SSJ' provides the abstract class `Event` that offers facilities for creating and scheduling events in the simulation. Each type of event must be defined by implementing an extension of this class. ALS is built using two main types of events, namely `FinishProcessingEvent` and `StartProcessingEvent`, which are implemented through two classes that extend the `Event` class. A FinishProcessingEvent is associated to each load (relation 1:1 in Figure 4), and it happens whenever a load has finished to be processed in a WC. A StartProcessingEvent is associated to each WC (relation 1:1), and represents the event of the WC that starts processing a load. Their method `actions()` describes the actions that are performed each time the event, associated to the corresponding entity (load or WC), happens. In this way it is not necessary to create a new event each time that a WC starts processing a load, or each time a load finishes to be processed in a WC. Instead, the same event associated to each Load or WC will be scheduled and re-scheduled.

In fact, events happen because they are scheduled. The method `scheduleNext()` schedules an event as the first event in the event list, that is, to be executed at the current time (as the next event). The method `schedule(double delay)` schedules an event to happen in `delay` time units, i.e., at current time + `delay`, by inserting it in the event list.

Figure 4 shows only the main relationships among classes. Some arrows and lines have been omitted in the UML diagram to allow an acceptable readability. The FinishProcessingEvent class has a relation with WorkCentre whose cardinality is 1:2. In fact, when a load finishes to be processed, it is linked to two WCs: the WC in which the load has just been processed (`WorkCentre currWC`) and the one subsequent to the current one (`WorkCentre nextWC`). In that moment each load can schedule the StartProcessing event associated to this two WCs (see Figure 5). Even the StartProcessingEvent class is related to Load through a relationship whose cardinality is 1:2, because, as described in the next paragraph, when a StartProcessing event happens, actions on the load that is going to be processed (`startWorkingLoad`) and on the load that is entering the buffer from the preceeding WC (`enteringBufferLoad`) have to be performed.

Through this architecture, each Load and Workstation can schedule, through their StartProcessing and FinishProcessing instances, events associated to other instances. In the following, the `actions()` methods of the two classes StartProcessingEvent and FinishProcessingEvent are described in detail. Figure 5 summarizes the events that are scheduled when StartProcessing and FinishProcessing events happens.

*3.2.1. Actions associated to StartProcessingEvent*

Actions associated to the StartProcessing event are performed when a load starts being processed in the WC, coming from the buffer or directly from the previous WC (if the buffer size is zero). The WC StartProcessing event can happen only if two conditions are both verified:

a) its loadScheduled List is not empty (there is at least one load waiting to be processed);
b) its freeWorkStation List is not empty (there is at least one WS available for process a load);

If both conditions *a* and *b* are verified, the WC is *ready to work*. The method `readyToWork()` check if this two conditions are true.

Actions performed in the StartProcessingEvent actions() method can be summarized as follows:

- remove the first load in the loadScheduled List (`startWorkingLoad`: the load starting being processed in one of the WS of the WC)
- remove one WS (`startWorkingWS`) from the set of available ones (`freeWorkStation`);
- startWorkingWS calculates the processing time (`calculateHoldTime()`) on startWorkingLoad;
- the FinishProcessing event associated to startWorkingLoad is scheduled to happen in a time equal to the processing time;
- if the number of loads in the loadScheduled List is ≥ than the buffer size (i.e. a load has already been processed in the previous WC and is waiting to enter the buffer) then :
    - a load (`enteringBufferLoad`) enters the buffer of the WC;
    - one WS (`currWS` of enteringBufferLoad) is released in the previousWC;
    - if the previous WC (`WorkCentre previousWC`) is ready to work (conditions *a* and *b* are true), the StartProcessing event associated to previousWC is scheduled to happen at the current time.

*3.2.2. Actions associated to FinishProcessingEvent*

Actions associated to the FinishProcessing event are performed when a load has just finished to be processed in a WC. The point of view of a load passing through the assembly line (from one WC to the subsequent one) is assumed, and the following notation is utilised: `WorkCentre currWC` indicates the WC the load is leaving from, that is the WC in which the load has just been processed; `WorkCentre nextWC` indicates the WC the load is arriving in, which is subsequent to currWC.

These actions are described in the FinishProcessingEvent actions() method and can be summarized as follows:

- the load claims to be processed in nextWC (the load enters the loadScheduled List of nextWC, but does not release the WS in currWC)
- if there is space in the buffer of nextWC (number of loads in the loadScheduled List of nextWC < buffer size), then:
    - the load enters the buffer of nextWC, and one WS is released in currWC (a WS is added to the set of available ones);
    - if the load is leaving the first WC in the line, then a new load is created.
    - if currWC is ready to work (conditions *a* and *b* are true), then the event StartProcessing event associated to currWC is scheduled to happen at the current time;
- if nextWC is ready to work (conditions *a* and *b* are true), then the StartProcessing event associated to nextWC is scheduled to happen at the current time;

*3.2.3. Loads creation*

In order to start the simulation, a first event, corresponding to the first load entering the line, has to be scheduled to happen at time zero. This first event corresponds to a FinishProcessing event of the load in a virtual WC numbered -1 (i.e. a virtual WC before the first WC in the line). When this event happens, the first load passes from WC#-1 to WC#0, and, triggering the StartProcessing event related to WC#0, starts up the recursive event scheduling and re-scheduling that allow the simulation to go on. When a load finishes to be processed in the first WC (WC#0), it triggers the creation of a new load entering the line, by scheduling a FinishProcessing event of a new load in WC#-1.

*3.2.4. Inputs required*

ALS is contained in a package named lineSimulator, which provides the public class `Simulation`, whose constructor requires all the inputs described in Section 3.1: the array $t_{ij}$ of tasks times completion (`double[][]t`), the array $lc_{kz}$ (`int[][]lc`) that describes the line configuration, the array $s$ (`int[]s`)

that defines the sequence of models entering the line, the string *distType* representing the task times distribution (`String distType`), the coefficient of variation *cv* (`double cv`) which is influent only if *distType* = "NORM", the simulation length (`double simLength`).

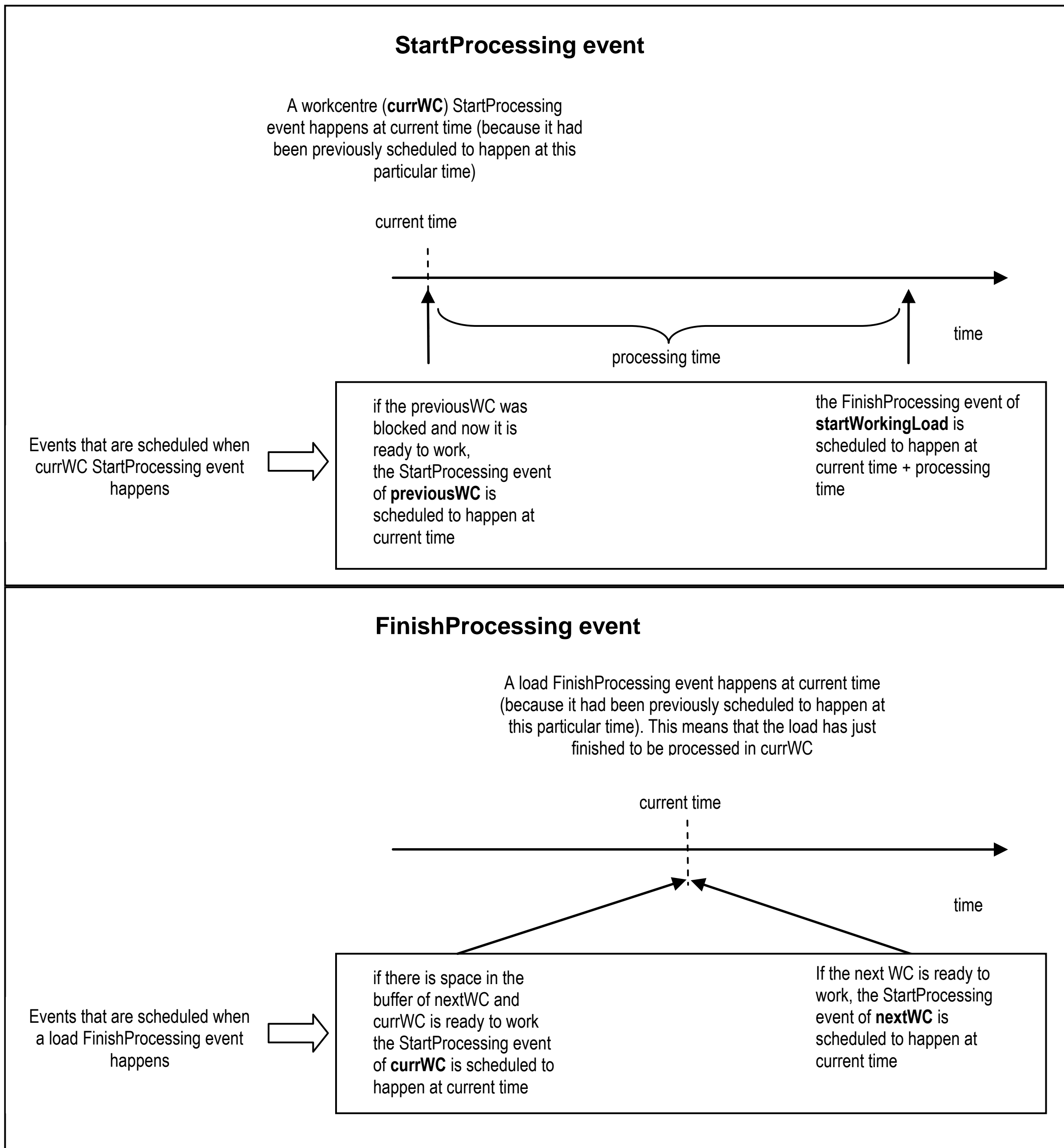



Fig. 5. Events scheduled when StartProcessing and FinishProcessing events happens.

*3.2.5. Outputs provided*

When an object Simulation is created, the model of the assembly line is created, the simulation is performed and outputs are obtainable by accessing the following public attributes of the object: `Tally cycleTime`; `Tally flowTime`. Objects of class Tally collect data that comes as a sequence of real-valued observations. It can compute sample averages, sample standard deviations and confidence intervals

on the mean based on the normality assumption. Class Tally is included in the package ‘stat’ of SSJ. Cycle time is measured as the time that passes between two consecutive loads coming out from the line. Flow Time is measured as the difference between the time of a load coming out from the line and its time of entering the line. Line throughput and WIP are derived: average throughput is calculated as the average cycle time inverse. Average WIP is calculated, using the Little law, as the product between average values of throughput and flow time.

*3.2.6. Graphical user interface (GUI)*

Although the primary utilisation of ASL is expected to be as embedded in the code of algorithms and procedures, a very simple GUI has been implemented (Figure 6). It allows to manually insert all the inputs required, included task times and line configuration information (that can also be loaded from and saved to text files), and to display outputs and execution time of the simulation run.

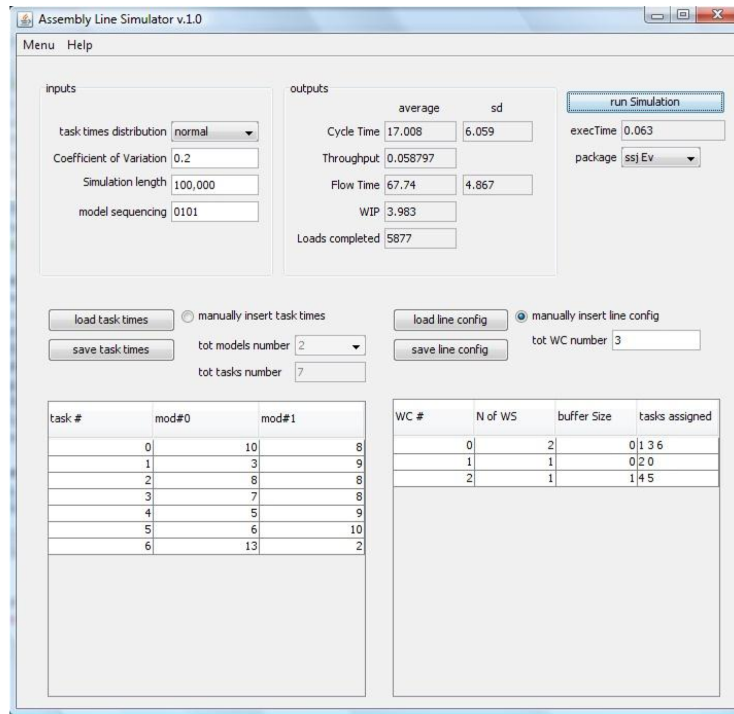



Fig. 6. The graphical user interface of ALS.

*3.2.7. Event and process oriented versions*

The final version of ALS is built using the event oriented approach, as described above. However, ALS has also been realized in other two versions, using two different ‘process-oriented’ approaches, named ‘process oriented’ and ‘2p implementation’ (for details, see Tiacci and Saetta, 2007). These two versions have been implemented using both the already mentioned package SSJ, and another package for process-based discrete event simulation, named JavaSimulation (Helsgaun, 2004). A ‘process’ is a time-ordered sequence of interrelated events separated by intervals of time, which describes the entire experience of an “entity” as it flows through a “system”. The process-oriented paradigm is a natural way of describing complex systems (Law and Kelton, 2000) and often leads to more compact code than the event-oriented view. Unfortunately many simulation environments, including the Java packages JavaSimulation and SSJ, implement processes through threads. This adds significant overhead, prevents the use of a very large

number of processes in the simulation and slows the simulation execution (L'Ecuyer and Buist, 2005), as the experiments described in the next section confirmed.

*3.3 Validation*

We validated our model through two widely adopted techniques (see Law and Kelton, 2000). The first one consists in computing exactly, when it is possible and for some combination of the input parameters, some measures of outputs, and using it for comparison. The second one, that is an extension of the first one, is to run the model under simplifying assumptions for which its true characteristics are known and, again, can easily be computed.

Using the first approach it is possible, on the basis of queuing theory, to exactly calculate the throughput of a single product assembly line with exponentially distributed task times, both for the case of two stations with a buffer of size $N$ between them, and in the case of three stations without buffers (see Altiok, 1997). The capability to appropriately model lines in which multiple model are assembled and stations can be duplicated (paralleling) has been validated through the second approach. In this case using deterministic task times it is possible to easily calculate flow time and throughput or different simple line configurations.

## 4. ALS Performances

ALS have to be evaluated on the basis of its fastness in providing the simulated throughput of a line configuration, given that all required inputs are known (see Section 3.2.4). This time corresponds to the time of creation of an object Simulation; its value (expressed in seconds) is stored in the attribute `executionTime`.

Two assembly problems have been used as testbeds. Problem 1 inputs are: the task times array reported in Figure 2, the line configuration reported in Figure 3, the sequence array $s = \{0,0,1,1\}$, normal task times distribution with coefficient of variation equal to 0.2. Problem 2 is taken from a real case, presented in Mendes et al. (2005), concerning a company that is a major manufacturer of consumer electronic goods; the system analyzed is the PC camera assembly line, in which three different versions of a PC camera with some dissimilar technical specifications are assembled. Task times and line configuration are reported in Tables 1 and 2. The sequence array is $s = \{0,1,2\}$, task times are normally distributed with coefficient of variation equal to 0.1.

Tables 3 and 4 shows ALS execution times and outputs (for Problem 1 and 2 respectively), varying the package used (JavaSim and SSJ), the implementation (process oriented, 2p, and event) and the simulation length. Problem 1 and 2 have also been modelled in Arena 11.0 (Kelton et al., 2004), and execution times in batch mode (no animation) have been recorded. The aim is to compare ALS performances with one of the most widely used software in the industrial field for the simulation of manufacturing and material handling systems (Anglani et al., 2002), which also provides one of the most efficient simulation engine among many commercial simulation software (Cimino et al., 2010). All simulations have been performed on a computer with an AMD Athlon X2 Dual Core Processor 4200+ (2.2 GHz), with JavaRuntime Environment (JRE) 1.6 running under Windows Vista Home Premium Edition.

As far as 'process oriented' and '2p implementation' versions are concerned, results show a slight superiority of package SSJ over JavaSimulation, and an appreciable superiority of the '2p implementation' over the 'process orented' version. Some data, corresponding to simulation length equal to 10,000,000 time units, is not available (*na*), due to 'thread synchronization errors' that arise when processes, like in this two versions, are implemented as threads. For other considerations about relations between execution times, throughput and WIP for these two versions, see Tiacci and Saetta (2007).

Execution times are drastically reduced using the event oriented version of ALS, which abundantly outperforms also Arena (proving to be about from 10 to 5 times faster). Furthermore, Arena's execution

times only include the simulation run time, excluding the time needed to develop the model for the specified line configurations, that are, on the contrary, intrinsically considered in execution times provided by ALS.

Table 1
Problem 2: task times (in time units)

| | Model# | | | | Model# | | | | Model# | | | | Model# | | | | Model# | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Task# | 0 | 1 | 2 | Task# | 0 | 1 | 2 | Task# | 0 | 1 | 2 | Task# | 0 | 1 | 2 | Task# | 0 | 1 | 2 |
| 0 | 2 | 2 | 2 | 8 | 2 | 2 | 2 | 16 | 6 | 6 | 6 | 24 | 4 | 4 | 4 | 32 | 4 | 4 | 4 |
| 1 | 2 | 2 | 2 | 9 | 10 | 10 | 10 | 17 | 7 | 7 | 7 | 25 | 6 | 6 | 6 | 33 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 10 | 3 | 0 | 0 | 18 | 3 | 3 | 3 | 26 | 5 | 5 | 5 | 34 | 2 | 2 | 2 |
| 3 | 2 | 2 | 2 | 11 | 11 | 11 | 11 | 19 | 28 | 37 | 33 | 27 | 0 | 0 | 2 | 35 | 1 | 1 | 1 |
| 4 | 2 | 2 | 2 | 12 | 4 | 4 | 4 | 20 | 3 | 3 | 3 | 28 | 1 | 1 | 1 | 36 | 1 | 1 | 1 |
| 5 | 0 | 11 | 11 | 13 | 0 | 4 | 4 | 21 | 8 | 8 | 8 | 29 | 3 | 3 | 3 | 37 | 1 | 1 | 1 |
| 6 | 0 | 0 | 16 | 14 | 9 | 9 | 0 | 22 | 5 | 5 | 5 | 30 | 3 | 3 | 3 | 38 | 1 | 1 | 1 |
| 7 | 21 | 39 | 37 | 15 | 13 | 13 | 12 | 23 | 7 | 7 | 9 | 31 | 0 | 0 | 3 | | | | |

Table 2
Problem 2: line configuration

| WC# | WS assigned | Buffer size | tasks assigned |
|---|---|---|---|
| 0 | 1 | 0 | 0, 1, 2, 3, 5, 6, 10, 14 |
| 1 | 2 | 0 | 4, 7, 8, 9, 12, 13, 18, 22 |
| 2 | 1 | 0 | 11, 15, 16 |
| 3 | 1 | 0 | 17, 20 |
| 4 | 2 | 0 | 19, 21 |
| 5 | 1 | 0 | 23, 24, 25, 26, 27, 28, 29, 30, 33, 36 |
| 6 | 1 | 0 | 31, 32, 34, 35, 37, 38 |

The time needed by ALS to obtain the line throughput is so much short that its usage in algorithms and procedures, that have to evaluate different line configurations several times, becomes possible. For example, solving Problem 2 through the genetic algorithm approach proposed in Tiacci et al. (2006) would require to perform a total of 50,000 fitness evaluations (i.e. throughput estimations). Considering a simulation lengths of 100,000 time units, using ALS to obtain the simulated throughput would require an extra CPU-time equal to (50,000 · 0.047 ) = 2,350 sec, that is little more than 39 minutes. This amount of time is quite acceptable for this type of problems, considering also that the experiment has been conducted on a standard PC.

It is noteworthy that the throughput estimation accuracy provided by a simulation run that lasts 100,000 time units is, for Problem 2, very high. To approximately quantify this accuracy, 40 replications of 100,000 time units length have been performed, and average ($\mu$ = 34.453) and standard deviation ($\sigma$ = 0.0432) of the simulated mean cycle times have been calculated. The interval corresponding to an amplitude of $\pm 3\sigma$ is equal to ±0.1296, that is just ±0.37% of the average value. Confounding the unbiased estimators $\mu$ and $\sigma$ with the true values and assuming a normal distribution, the approximate probability that the cycle time, provided by a single simulation run, falls between ±0.37% of the real value would be equal to the 99.7%. This accuracy is presumably much higher than any other measure, not simulation-

based, might provide. If a lower accuracy is acceptable, the simulation length can be shortened, allowing to further limit the extra CPU-time required by ALS.

Table 3
Execution times (sec) and outputs of Problem 1.

| Sim. length (time units) | ALS | | | | | Arena |
|---|---|---|---|---|---|---|
| | Process oriented | | 2p implementation | | Event (SSJ) | |
| | JavaSim | SSJ | JavaSim | SSJ | | |
| 100,000 | 1.26 | 1.12 | 0.73 | 0.501 | 0.025 | 0.265 |
| 400,000 | 4.65 | 4.07 | 2.57 | 1.98 | 0.109 | 0.777 |
| 700,000 | 8.12 | 7.04 | 4.36 | 3.46 | 0.187 | 1.281 |
| 1,000,000 | 11.30 | 10.04 | 6.26 | 4.84 | 0.266 | 1.777 |
| 10,000,000 | na | na | 62.65 | 49.29 | 2.65 | 16.51 |

Outputs (mean values expressed in time units): Cycle time: 17.00; Flow time: 68.7; WIP: 4.04.

Table 4
Execution times (sec) and outputs of Problem 2.

| Sim. length (time units) | ALS | | | | | Arena |
|---|---|---|---|---|---|---|
| | Process oriented | | 2p implementation | | Event (SSJ) | |
| | JavaSim | SSJ | JavaSim | SSJ | | |
| 100,000 | 1.93 | 1.89 | 0.72 | 0.65 | 0.047 | 0.359 |
| 400,000 | 7.11 | 6.86 | 2.66 | 2.25 | 0.181 | 0.984 |
| 700,000 | 12.49 | 11.87 | 4.4 | 3.76 | 0.32 | 1.574 |
| 1,000,000 | 18.00 | 16.90 | 6.42 | 5.44 | 0.456 | 2.183 |
| 10,000,000 | na | na | na | 54.8 | 4.47 | 20.71 |

Outputs (mean values expressed in time units): Cycle time: 34.45; Flow time: 237.25; WIP: 6.80.

## 5. Improvements

ALS is an expandable tool, and can benefit from several libraries available for Java. Once the core of the simulator, consisting in the object structure (Fig. 4) and the logic of actions methods (Sections 3.2.1 and 3.2.2), has been defined many improvements can be easily implemented. Among theses, the latest version of ALS already includes:

- more statistical distributions for representing stochastic task times; the possibility that each individual task has a completion time described by a specific statistical distribution.

The implementation of this feature has been done using the package `randvar` of SSJ, that provides a collection of classes for non-uniform random variate generation. Inputs related to task times described in section 3.2.4 (i.e. $t_{ij}$, *distType*, *cv*), has been replaced with three arrays, namely: $distType_{ij}$ = distribution of completion time of task $i$ on model type $j$ (`String[][]distType`), $p_{ij}$ = first parameter of the distribution $\mathrm{distType}_{ij}$ (`double[][]p`), $p2_{ij}$ = second parameter of the distribution $distType_{ij}$ (`double[][]p2`). In this way the following single and double parameter distributions have been modelled: “DET” = deterministic, “GEO” = geometric, “LOG” = logarithmic, “POIS” = Poisson, “EXP” = exponential, “CHI” = chi; “NORM” = normal, “UNIF” = uniform, “GAM” = gamma, “ERL” = Erlang,

"BIN" = binomial. If the distribution $distType_{ij}$ requires only one parameter, the corresponding $p2_{ij}$ values are ignored. If the distribution requires an integer parameter instead of a double one (e.g. chi and Erlang distributions), double values are casted to integer.

- the possibility to define a warm-up period;

A warm-up period $w$ (`double w`) has been introduced, during which statistics are not collected. The total duration of the simulation run will be equal to the sum of the $w$ and the *simLength* variables.

- the possibility that the sequence of models entering the line is random, while respecting a determined demand proportion among different model types (to simulate a JIT environment);

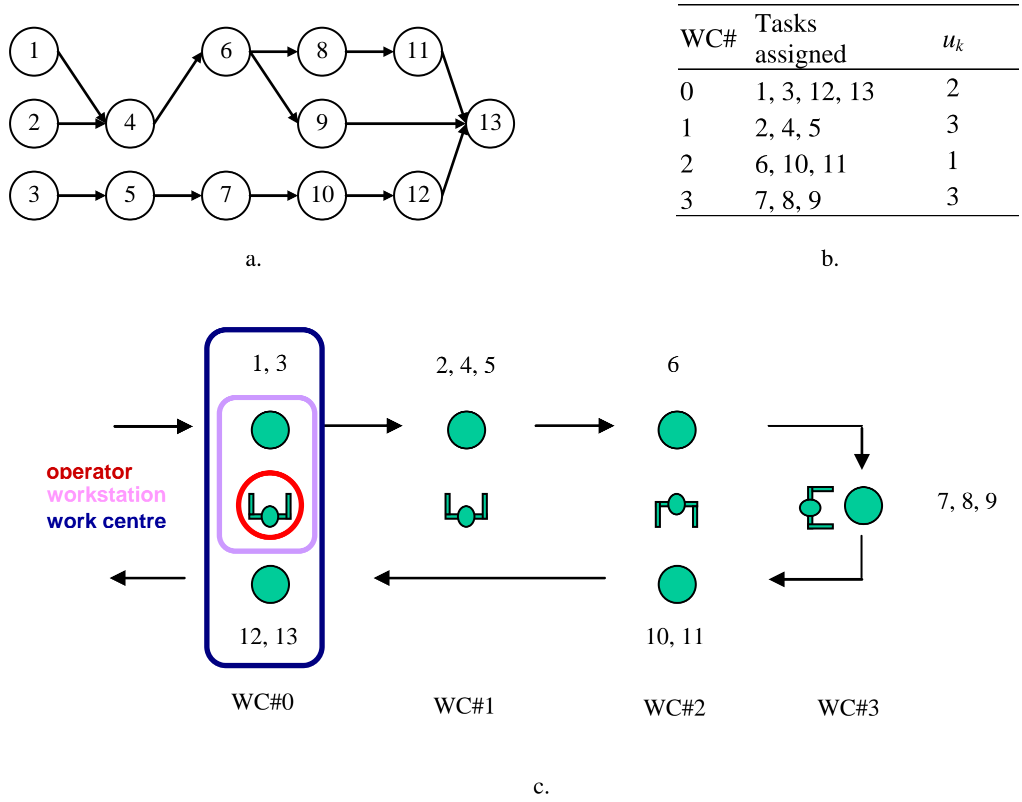
The implementation of this feature has required adding a new input, consisting in an array named $demProp_j$ (`double [] demProp`) that specifies the demand proportion of model $j$ with respect to total demand, so that each time a new load is created, the probability that its model type is $j$ is equal to:

$$P_j = \frac{demProp_j}{\sum_j demProp_j}$$

This kind of random generation is performed when the value of the boolean input *randomSeq* (`boolean randomSeq`) is true. Otherwise, the sequence of models assembled through the line is defined by the array $s$, as described in section 3.1.4.

- the possibility to model U-shaped lines (Miltenburg and Wijngaard, 1994), in which stations may work at two segments of the line facing each other simultaneously;

In U-shaped assembly lines, stations can be arranged so that two loads at different positions on the line can be handled. As depicted in Figure 7, a WC may consist of one operator and two WSs, each one at two segments of the line. In this type of WCs, named 'crossover' WCs (WC#0 and WC#2 if Fig.7), loads can be processed alternatively into one of the WSs. So the difference to the straight line is that a WC $k$ can contain not only tasks whose predecessors are assigned to one of the WC 0, . . . ,$k$, but also tasks whose predecessors will be finished until the product returns to station $k$ for the second time (cf. Monden, 1998). It is noteworthy that every solution feasible for straight lines is feasible for an U-line, because an U-line does not need to include crossover WC. However, U-shaped lines provides much more possibilities of combining tasks to WCs, and this implies increasing possibilities to find better solutions with respect to straight lines.
U-shaped lines can be simulated through ALS setting the boolean input *uShaped* (`boolean uShaped`) to true. The definition of an array $u_k$ (`int [] u`) is required in order to define the number of tasks of a WC $k$ that are performed in the first side of the U (the remaining tasks will be performed in the return side). For example (see Fig. 7) tasks assigned to WC#0 are {1, 3, 12, 13}, and $u_0 = 2$ means that the first two tasks (1, 3) are done in the first side, while the remaining tasks (12, 13) are done in the return side. Note that if $u_k = 0$, all the assigned tasks will be performed in the return side. If $u_k$ is equal to (or higher than) the number of tasks assigned to the WC, all tasks will be performed in the first side.
In the current implementation loads can enter a WC one at a time, no buffers are assumed between workcentres, and no form of paralleling is considered. So if *uShaped* is 'true' inputs data related to buffers size and number of workstations in the $lc_{kz}$ array (i.e. the first two columns) are ignored.




| WC# | Tasks assigned | $u_k$ |
|---|---|---|
| 0 | 1, 3, 12, 13 | 2 |
| 1 | 2, 4, 5 | 3 |
| 2 | 6, 10, 11 | 1 |
| 3 | 7, 8, 9 | 3 |

Fig. 7. a. Precedence diagram.
b. Tabular representation.
c. Graphical representation.

With the aim to cover the widest range of issues proposed in literature and that may be found in real cases, the next improvements of ALS include the development of the following features:

- the possibility that same task can be assigned to different WCs for different models: this relaxation is acceptable considering highly skilled workers, capable of performing a wide range of activities (for details see Bukchin et al., 2002)
- the possibility to consider buffers between WCs in U-shaped lines;
- dynamic variation of task times due to learning effects of operators;
- a version more oriented toward flow lines, with unreliable machines and stochastic failure and repair times.

## 6. Summary

The simulator presented herein, named Assembly Line Simulator (ALS), can be used as supporting tool in finding solution of the assembly line balancing problem. Its main features are the following:

- it is realistic: it is able to calculate the throughput (and other performance indicators) of an assembling line, with parallel stations, buffers within workstations, in which a variety of models have to be assembled in a mixed way, following a determined input sequence, with stochastic task times of completion;
- it is modular, because it is object-oriented: different lines configurations (with different number of workstations, tasks assignments, number of buffers, etc.) can be created by the mean of very simple array representations, so that time required for building the model is zeroed;
- it is very fast, because it is event-oriented: the simulation run is performed only by events scheduling, rescheduling, cancelling etc.; by avoiding the use of 'processes', execution times are kept very low (in some cases less than a tenth of second), arriving to outperform Arena.

These characteristics allows it to be effectively coupled to those algorithms and procedures where numerous variants of line configurations have to be simulated, and the evaluation of a fitness function (which includes some line performances indicator, such as the throughput) has to be performed several times.

ALS allows to overcome the limit of using traditional measures (not simulation-based) of the line throughput that are poorly correlated to its real value. This lack of correlation increases with the complexity of the line, and this is one of the reasons why problems, that should be approached simultaneously (line balancing, buffer allocation, model sequencing), have often been treated sequentially or separately in literature. ALS represents an effective mean to go over these traditional approaches.

ALS has been developed with the scope to be usable by researchers and practitioners. For this reason it has been developed in Java, a language that guarantees the compatibility with multiple platforms. ALS package is available as a freeware software from the author's web page (available after paper acceptance decision).

## References


Altiok Tayfur, (1997). Performances Analysis of Manufacturing Systems, Springer.

Anglani, A., Grieco, A., Pacella, M., &Tolio, T. (2002). Object-oriented modeling and simulation of flexible manufacturing systems: a rule-based procedure. *Simulation Modelling Practice and Theory*, 10, 209-234.

Askin, R.G., & Zhou, M. (1997). A parallel station heuristic for the mixed-model production line balancing problem. *International Journal of Production Research*, 35 (11), 3095-3105.

Banks, J., Carson, J., Nelson, B. &Nicol, D., (2004). *Discrete-event system simulation. Fourth edition*. Prentice Hall.

Becker., C., & Scholl, A. (2006). A survey on problems and methods in generalized assembly line balancing. *European Journal of Operational Research*, 168, 694-715.

Bukchin, J. (1998). A comparative study of performance measures for throughput of a mixed model assembly line in a JIT environment. *International Journal of Production Research*, 36, 2669-2685.

Bukchin, J., Dar-El, E.M., & Rubinovitz, J. (2002). Mixed model assembly line design in a make-to-order environment. *Computers & Industrial Engineering*, 41, 405-421.

Cimino, A., Longo, F., & Mirabelli, G. (2010). A General Simulation Framework for Supply Chain Modeling: State of the Art and Case Study, *International Journal of Computer Science Issues*, 7(2), No 3.

Fremerey, F. (1991). Model-mix balancing: more flexibility to improve the general results. In: M. Pridham, & C. O'Brien, *Production Research: Approaching the 21st Century*, pp. 314-312, London: Taylor & Francis.

Helsgaun, K. (2004). Discrete Event Simulation in Java, Department of Computer Science, Roskilde University, Denmark. Available at http://www.akira.ruc.dk/~keld/research/JAVASIMULATION/.

Jin, M., & Wu, S.D. (2002). A new heuristic method for mixed model assembly line balancing problem. *Computers & Industrial Engineering*, 44, 159-169.

Karabati, S., & Sayin, S. (2003). Assembly line balancing in a mixed-model sequencing environment with synchronous transfers. *European Journal of Operational Research*, 149, 417-429.

Kelton,W.D., Sadowski, R.P., & Sturrock, D.T. (2004). *Simulation with Arena*. New York: McGraw-Hill.

Law, A. M., & Kelton, W. D. (2000) *Simulation Modeling and Analysis*. *Third edition.* New York: McGraw-Hill.

L'Ecuyer, P., Meliani, L., & Vaucher, J. (2002). SSJ: a framework for stochastic simulation in Java. In: *Proceedings of the 2002 Winter Simulation Conference* (pp. 234-242). IEEE Press.

L'Ecuyer, P., & Buist, E. (2005), Simulation in Java with SSJ. In: *Proceedings of the 2005 Winter Simulation Conference* (pp. 611-620). IEEE Press.

Macaskill, J.L.C. (1972). Production-line balances for mixed-model lines. *Management Science*, 19, 423-434.

Mendes, A.R., Ramos, A.L., Simaria, A.S., & Vilarinho, P.M. (2005). Combining heuristic and simulation models for balancing a PC camera assembly line. *Computers & Industrial Engineering*, 49, 413-431.

Merengo, C., Nava, F., & Pozetti, A. (1999). Balancing and sequencing manual mixed-model assembly lines. *International Journal of Production Research*, 37, 2835-2860.

McMullen, P.R., & Frazier, G.V. (1997). A heuristic for solving mixed-model line balancing problems with stochastic task durations and parallel stations. *International Journal of Production Economics*, 51, 177-190.

McMullen, P.R., & Frazier, G.V. (1998). Using simulated annealing to solve a multiobjective assembly line balancing problem with parallel workstations. *International Journal of Production Research*, 36 (10), 2717-2741.

McMullen, P.R., & Tarasewich, P. (2003). Using ant techniques to solve the assembly line balancing problem. *IIE Transactions*, 35, 605-617.

Miltenburg, J., & Wijngaard, J. (1994). The U-line line balancing problem. *Management Science*, 40, 1378–1388.

Monden, Y., 1998. *Toyota production system—An integrated approach to just-in-time, 3rd*. Dordrecht: Kluwer.

Noorul Haq, A., Jayaprakash, J., & Rengarajan, K. (2006). A hybrid genetic algorithm approach to mixed-model assembly line balancing. *International Journal of Advanced Manufacturing Technologies*, 28, 337-341.

Scholl, A., & Becker, C. (2006). State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. *European Journal of Operational Research*, 168, 666-693.

Simaria, A.S., & Vilarinho, P.M. (2004). A genetic algorithm based approach to the mixed-model assembly line balancing problem of type II. *Computers & Industrial Engineering*, 47, 391-407.

Thomopoulos, N. T. (1967). Line balancing-sequencing for mixed-model assembly. *Management Science*, 14, 59-75.

Tiacci, L., Saetta, S., & Martini, A. (2006). Balancing Mixed-Model Assembly Lines with Parallel Workstations through a Genetic Algorithm Approach. *International Journal of Industrial Engineering, Theory, Applications and Practice*, 13, 402-411.

Tiacci, L., & Saetta, S. (2007). Process-oriented simulation for mixed-model assembly lines. In: *Proceedings of the 2007 Summer Computer Simulation Conference* (pp. 1250-1257). San Diego: SCSI.

Vilarinho, P. M., & Simaria, A. S. (2002). A two-stage heuristic method for balancing mixed-model assembly lines with parallel workstations. *International Journal of Production Research*, 40 (6), 1405-1420.

Wilhelm, WE. (1987). On the normality of operation times in small-lot assembly systems: a technical note. *International Journal of Production Research*, 25(1), 145-154.

Zeigler B. P., Praehofer, H., & Kim T.G (2000). *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems, second edition*, Academic Press.