Process-oriented simulation for mixed-model assembly lines

Lorenzo Tiacci, Stefano Saetta
Dipartimento di Ingegneria Industriale
Università degli Studi di Perugia
lorenzo.tiacci@unipg.it, stefano.saetta@unipg.it

Keywords: process-oriented simulation, object-oriented simulation, assembly lines, balancing, scheduling.

Abstract
In this paper a process-oriented simulator for assembly lines is presented. The simulator considers mixed model lines, with stochastic task times, the possibility to have parallel stations, and to consider different scheduling sequences. The simulator is a flexible supporting tool in finding solution of the assembly line balancing problem (and the optimal sequencing problem associated to it) and can be used in those algorithms and procedures where the evaluation of a fitness function (which includes some line performances indicator, such as the throughput) has to be performed several times. Different implementations of the simulator have been tested with two different packages for discrete event simulation in Java (‘JavaSimulation’ and ‘Stochastic Simulation in Java’).

1. INTRODUCTION
An assembly line is a set of sequential workstations linked by a material handling system. In each workstation a set of tasks is performed using a predefined assembly process. The Assembly Line Balancing Problem (ALBP) consists in assigning tasks to workstations, while optimising one or more objectives without violating any restriction imposed on the line (e.g. precedence relations among some of the tasks impose a partial ordering, reflecting which task has to be completed before others). The speed of the assembly line and length of each uniform station dictates time available per operator, known as cycle time (ct).

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 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. At 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.

In a mixed-model line, an operation may be carried out on only a subset of the products assembled in the line. This implies that task times in each workstations may considerably vary. For this reason also the assembly sequence of the model units is important with respect to the efficiency of a line. Thus, the MALBP is also connected to a sequencing problem [Merengo et al. 1999].

Another additional decision by which the MALBP may be accompanied concerns the utilization of parallel stations that perform the same task set. The aim of using parallel stations is often to perform tasks with processing time larger than the 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].

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]. For a comprehensive classification of other different possible features of the GALBP see [Becker and Scholl 2006].

To evaluate performances of a mixed model assembly line may be complicated. This complication is a result of blockage and starvation caused by the arrival of different models to the line, having different assembly time requirements at each station. Considering the throughput as the main operational design objective, the effects of these phenomena on line throughput are very difficult to evaluate. Unfortunately its evaluation is fundamental in almost all procedures and algorithms developed to solve MALBP,
since the estimation of objective functions that includes performance indicators is often required. The delicacy of this issue has been accurately 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. These performance measures are required to be highly correlated with the objective in question (i.e. throughput). So he uses simulation as a methodology for the validity study. Results show that the absolute quality of all measures examined decrease with an increase of line length.

The aim of this work is to overcome the limit of using performance measures instead of simulated throughput, by developing a parametric simulator for mixed model lines that can quickly simulate different line configurations and can be used iteratively in algorithms and procedures. At this scope, a Java based process oriented simulator that considers all the above mentioned features (mixed models, sequencing issues, parallel stations, stochastic task times) is presented. The aim is to put at the scientific community and practitioners disposal an efficient, flexible and expandable tool able to accurately calculate real assembly line performances.

In the following, inputs of the MALBP are summarized, the simulator structure is described, implementation issues are discussed and the simulator performances, with respect to different line configurations, are commented.

**Notation**

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

2. **INPUTS OF THE PROBLEMS**

An assembly line with parallel workstation can be depicted as in Figure 1.

![Figure 1. An assembly line with parallel workstations.](image)

In the line, each operator has a workstation where he performs one or more tasks. Each work centre (WC) consists of either one or multiple parallel workstations. ‘Parallelising’ 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.

A set of $n$ tasks (numbered with $i = 0, \ldots, n-1$) has to be performed in the line in order to complete each product. Each task has a stochastic completion time, which is assumed normally distributed. 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 $j = 0, \ldots, m-1$). Input data are thus represented by an $n \times m$ matrix $t_{ij}$ whose elements represent the average completion time for task $i$ of model type $j$.

The $2 \times 7$ 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, units time). 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.

<table>
<thead>
<tr>
<th>Task#</th>
<th>Model#</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
</tbody>
</table>

![Figure 2. The $t_{ij}$ input matrix.](image)

In our model the standard deviation $\sigma_{ij}$ of the completion time of task $i$ for model $j$ is equal to its mean value multiplied by the coefficient of variation $cv$.

$$\sigma_{ij} = cv \cdot t_{ij} \tag{1}$$

A solution of a MALBP has to specify:
- the total number of WC in the line;
- the number of WS in each WC;
- how many and which tasks are assigned to each WC (and are all performed by each WS assigned to the WC).

The solution can be represented by a two-dimensional array $s_{kz}$ ($k = 0, \ldots, p-1$), 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 WS assigned to the WC; 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 WS operate. Tasks #2 and #0 are assigned to WC#1 (with 1 WS), and task #5 is assigned to WC #2 (with 1 WS).

<table>
<thead>
<tr>
<th>WC#</th>
<th>WS assigned</th>
<th>tasks assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1 3 6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2 0 /</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5 /</td>
</tr>
</tbody>
</table>

**Figure 3.** The $S_{k,t}$ two dimensional array.

The simulator allows also to specify the sequence of models assembled through the line: this sequence is represented by an array of integers. For example, in case of two models (#0 and #1) the array \{0,0,1,1\} means that after two model#0, two model#1 will be assembled, and so on. The Array \{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.

Resuming, input data consist of: the matrix $t_{ij}$ (average tasks times completion); the two-dimensional array $S_{k,t}$ (line configuration, as described above); the value of $cv$ (to calculate the standard deviation of tasks times) and the sequence array.

### 3. The SIMULATOR STRUCTURE

#### 3.1. The process-oriented approach

In the process-oriented approach a process describes the actions performed during the life cycle of an entity in the 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. Processes may represent “autonomous” objects such as machines and robots in a factory, customers in a retail store, vehicles in a transportation or delivery system, etc.

In object-oriented simulation a simulated system is considered to consists of objects (e.g. an entity or a server) that interact with each other as the simulation evolves. Therefore, the implementation of Process-oriented simulation in an object-oriented programming environment is very intuitive. Objects contain data and have methods; data describe the state of an object at a particular point in time, while methods describe the actions that the object is capable of performing. Processes are active objects whose behaviour in time is described by their method actions(). In our case, the entities of the system are loads (products to be completed), workcentres and workstations. Thus, three corresponding classes have been created in order to model the system.

Both packages ‘JavaSimulation’ and ‘SSJ’, which are shortly described in Paragraph 4 and which have been used in this work for the implementation of the simulator, provide an abstract class that represents an object-process. In the following we refer to JavaSimulation package names for methods (SSJ provides corresponding methods with different names). The abstract class is named Process, and its method actions() has to be implemented when defining a class that extends Process. The classes Load, WorkCentre and WorkStation are three classes extending the Process class; their method actions() describes the actions associated to the corresponding entity. The method can be called through the method activate() by any other entities of the system, putting between brackets the instance that has to be activated. In this way, each instance can interact with other instances activating (and also suspending) their processes.

Figure 4 shows how methods actions() of classes Load, WorkCentre and WorkStation interacts each other. Arrows represent the activations calls between entities. These interactions are described in the next paragraphs.

In our implementation, when a process is activated for the first time, actions associated to the process will be cyclically repeated (being eventually suspended and re-activated several times) until the end of the process. This is obtained by putting the instructions that describe entities actions into a ‘while’ cycle, the exit flag of which is that the current time is equal to the simulation length (as in the case of WorkStation, see below).

```java
public class WorkStation extends Process {
    public void actions() {
        while (time() < Simulation.length) {
            //here the actions to be performed
            //during the WS process
            cancel (this);
        }
    }
}
```

When the time limit of the simulation length is reached, the method cancel() definitely put an end to the entity process. A process can be suspended by the command passivate(). When it is re-activated (by an activate call from another entity), it continues its process from the point where it has been interrupted. Note that the cycle in the actions() method of the Load class contains two interruptions (while cycles of classes WorkCentre and WorkStations contain only one interruption each). The actions between these two interruptions correspond, as described in paragraph 3.1.3, to the processing time spent within a workstation and to a (possible) blocking time. The process of a Load instance may end or because the time
limit of the simulation length is reached (as in the WS case), or because the load is leaving the line, after having been processed by the last work centre of the line (see below the ‘break’ instruction).

```java
public class Load extends Process {
    public void actions() {
        while (time() < Simulation.length) {
            // part of the process
            passivate();
            // part of the process
            passivate();
            // part of the process
            if (currWCIndex == line.size()) break;
            cancel(this);
        }
    }
}
```

In the description of the `actions()` methods of the three classes, 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:

- ‘WC-‘ indicates the current WC, that is the WC where the load is physically in;
- ‘WC+‘ indicates the WC subsequent to the current one.

### 3.1.1. WorkCentre

Each WC has a virtual queue (it is virtual because there are no buffer between work centres) represented by the list of the loads that have been scheduled for the processing on the WC (List `loadScheduled`). Each WC has also a list of WS available for loads processing. Each time that a load starts being processed in the WC, one WS is removed from the list of the available ones. If the list is empty, a load, though scheduled to WC+, can not be processed and has to wait in WC- until one WS of WC+ become available.

The WC process consists in removing the first load from the `loadScheduled` List (following a FIFO logic) and in passing it to one of the available WS for processing (if there is any one). The process has to be activated when:

- a load is leaving the current work centre (WC-), and thus frees one WS, that becomes available. In this case WC- has to be activated;
- a load claims to be processed in WC+: the load enters the `loadScheduled` List of WC+, but still occupies the WS in WC-. In this case WC+ has to be activated.

In both cases the activation is performed by a load. Thus, the activation call of the `WorkCentre` process has to be placed in the `action()` method of the `Load` class. However, these activation call have to be performed only if two conditions are both verified:

a) the `loadScheduled` List of the WC that has to be activated is not empty (there is at least one load waiting to be processed);

b) the list of available WS of the WC that has to be activated 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 and its process can be (re)activated; otherwise the process is not (re)activated.

### 3.1.2. WorkStation

The WS process consists in processing a load, that is in ‘holding’ the load for a random amount of time (being this time normally distributed, with mean value equal to the average task time and the standard deviation calculated as in eq. 1). The process has to be activated when a load physically passes in the WC where the WS is. This action is performed, as illustrated in the previous paragraph, by `WorkCentre`. Thus, the activation call of the `WorkStation` process have to be placed in the `action()` method of the `WorkCentre` class.

### 3.1.3. Load

The Load process consists of two phases, which alternate each other since the load enters the line until it completes the last processing in the last WC of the line. The two phases are:

1) a processing phase in a WC, corresponding to the workstation process that is working the load (see the previous paragraph);

2) a (possible) waiting phase in the WC-, after that it has been processed. The waiting phase ends (and the load is re-activated) when the WC+ is ready to work (conditions `a` and `b` are true).

Phase 1) is activated by WC+. The activation takes place when WC+ is active (see paragraph 3.1.1), that is when conditions `a` and `b` have been verified (are true), and the load can physically pass from WC- to WC+ (in particular, at least one WS is available in WC+). When this phase takes place, if WC+ is the first WC in the line a new load is created and enters the `loadScheduled` List of WC#0. Note that new loads will be generated in the system only as needed by the first WC (see [Melani et al. 1999] for issues related to the formation of infinite queues when representing the first operation in an assembly line with some software languages).

Phase 2) starts when load processing in a WS is ended. This phase is activated by the WS that completed the processing on the load. In this phase:

- if the load has finished its processing in the last WC of the line, it activates WC- (if conditions `a` and `b` are true) and ends its process.

- if the load has finished its processing in an intermediate WC, or has just been created:
  - claims to be processed in WC+ (the load enters the `loadScheduled` List of WC+, but still occupies the WS in WC-)
  - if WC+ is ready to work (conditions `a` and `b` are true), activates the WC+
• claims to be processed in WC+ (the Load enters the 'loadScheduled' List of WC+, but still occupies the WS in WC-)
  • if WC+ is ready to work (conditions a and b are true), activates the WC+
  interruption
  re-activation (phase 1)
  the Load physically passes from one WC to the subsequent WC:
  • if WC- is ready to work (conditions a and b are true), activates the WC-
  • if the load is leaving the first WC in the line, creates a new load that enters the line.
  interruption
  re-activation (phase 2)
  the Load processing is finished. If the load has completed its processing in the last WC of the line
  • activates the last WC (if conditions a and b are true, that is if the WC is ready to work).
  • Ends the Load process

Figure 4. The ‘actions()’ methods of classes Load, WorkCentre and WorkStation

• the Load passes from WC- to WC+:
  • activates the first load in the loadScheduled List (the Load starts being processed in one of the WS of the WC)
  • activates one of the available WS (the processing starts)
  interruption
  re-activation / creation

• the WS processes the Load (it holds the Load for a time equal to the task time); when processing is finished
  • activates the current Load
  interruption
  re-activation / creation

Class Workstation does not contain any more an ‘action()’ method (it is not a class extending the abstract class Process)
It just contains a method to calculate the hold Time.

Figure 5. The ‘actions()’ methods of classes Load and WorkCentre in the two Process Classes (2p) implementation.
3.2. An implementation with only two Process classes

The above described process oriented approach allows to easily and effectively model an assembly line by the use of an intuitive correspondence of process to system entities.

Unfortunately many programming environments, including the two Java packages that are analysed in the next paragraph, implement processes as true threads from the operating system. This adds significant overhead, prevents the use of a very large number of processes in the simulation and slows the simulation execution.

A possible way to overcome this limit is to reduce the number of Process objects that simultaneously operates during the simulation. At this purpose, an alternative implementation, that makes use of only two classes extending the Process class (namely Load and WorkCentre) instead of three (i.e. WorkStation is no longer a process), has been developed. Figure 5 shows the action() methods of the two classes Load and WorkCentre, and their interactions. It is noteworthy that class WorkStation does not extend the abstract class Process, but only provides a method that calculates the hold time of the load being processing in the WC. The re-activation of the load, that was performed by the WS assigned to the load, is now performed by the WC through the schedule() method. This method schedules the reactivation of the Load process in a time equal to the hold time (calculated by the WS).

In order to reduce overheads due to synchronisation, the two phases partition of the Load process has been replaced by a single phase, corresponding to the actions performed by the load when it physically passes from one WC to the subsequent one. This allows to avoid an activation call from WorkCentre. It is noteworthy that the number of activation calls between entities needed to perform the same actions set are considerable reduced in comparison with the three process classes implementation.

4. IMPLEMENTATION

The simulator has been implemented using two different packages for discrete event simulation in Java, namely JavaSimulation [Helsgaun, 2004] and SSJ [L’Ecuyer and Buist 2005].

JavaSimulation is a Java package for process-based discrete event simulation which facilities are based on the simulation facilities provided by programming language SIMULA, one of the first object-oriented programming languages. SIMULA provides the standard class SIMULATION, a very powerful tool for discrete event simulation. The package JavaSimulation may be seen as an implementation of class SIMULATION in Java. It’s implementation is based on Java’s threads.

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. Two implementations of processes are available in SSJ. The first uses Java threads, as described in Section 4 of [L’Ecuyer et al. 2002]. The second is taken from DSOL [Jacobs and Verbraeck 2004]. For the implementation of the simulator presented herein, the first one has been used (package simprocs of SSJ).

The simulator presented herein is contained in a package named lineSimulator, which provides the class Simulation, whose constructor (reported below) requires all the inputs described in Paragraph 2:

```java
public Simulation(double coefficientOfVariation, int simulationLength, int[] sequence, double[][]taskTimes, int[][]solution)
```

When a Simulation object is created, the simulation is performed and outputs are obtainable by accessing the following public fields of the object:

```java
public Tally cycleTime;
public Tally flowTime;
public Tally wip;
```

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.

Both packages implements processes as true Java threads. As outlined by [L’Ecuyer and Buist 2005] this adds significant overhead and prevents the use of a very large number of processes in the simulation. This can be critical when an infinite queue of loads objects is present in the system. [Melani et al. 1999] outlined that when modelling an assembly line in which the first operation draws materials from warehouse with readily available stock, one problem may arise in the representation of this first operation in the model when using software languages such as GPSS, SLAM, SIMAN. The problem arises when an artificial load inter-arrival, that is very small in order to assure non-starving of the first WC, is assumed; and it is demonstrated by the loss of the desired statistics generated by the software, the inability of the software to complete the execution of the simulation, or an excessive simulation time that is sometimes terminated abruptly. As described in Paragraph 3.1.3, in our case the load entering the line is generated only as needed by the first WC, thus avoiding an increasing number of loads in the first (virtual) queue and limiting the total number of simultaneous processes. Thus, the approach presented herein overcomes the limit of the maximum number of simultaneous processes sustainable by
the operating system, since the formation of very large queues is avoided.

Two assembly problems have been used as testbeds for the two packages analysed, for both the two implementation with three processes (called 'process oriented' in the table) and two processes (called '2p implementation'). Problem 1 inputs are: the task times matrix reported in Figure 1, the solution array of Figure 2, the sequence array \{0,0,1\} and a coefficient of variation equal to 0.1. 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 \{0,1,2\}, and the coefficient of variation is equal to 0.1.

Table 1. Problem 2: task times.

<table>
<thead>
<tr>
<th>Task#</th>
<th>Model#</th>
<th></th>
<th>Model#</th>
<th></th>
<th>Model#</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2 2 2</td>
<td>13</td>
<td>0 4 4</td>
<td>26</td>
<td>5 5 5</td>
</tr>
<tr>
<td>1</td>
<td>2 2 2</td>
<td>14</td>
<td>9 9 0</td>
<td>27</td>
<td>0 0 2</td>
</tr>
<tr>
<td>2</td>
<td>2 2 2</td>
<td>15</td>
<td>13 13 12</td>
<td>28</td>
<td>1 1 1</td>
</tr>
<tr>
<td>3</td>
<td>2 2 2</td>
<td>16</td>
<td>6 6 6</td>
<td>29</td>
<td>3 3 3</td>
</tr>
<tr>
<td>4</td>
<td>2 2 2</td>
<td>17</td>
<td>7 7 7</td>
<td>30</td>
<td>3 3 3</td>
</tr>
<tr>
<td>5</td>
<td>0 11 11</td>
<td>18</td>
<td>3 3 3</td>
<td>31</td>
<td>0 0 3</td>
</tr>
<tr>
<td>6</td>
<td>0 0 16</td>
<td>19</td>
<td>28 37 33</td>
<td>32</td>
<td>4 4 4</td>
</tr>
<tr>
<td>7</td>
<td>21 39 37</td>
<td>20</td>
<td>3 3 3</td>
<td>33</td>
<td>2 2 2</td>
</tr>
<tr>
<td>8</td>
<td>2 2 2</td>
<td>21</td>
<td>8 8 8</td>
<td>34</td>
<td>2 2 2</td>
</tr>
<tr>
<td>9</td>
<td>10 10 10</td>
<td>22</td>
<td>5 5 5</td>
<td>35</td>
<td>1 1 1</td>
</tr>
<tr>
<td>10</td>
<td>3 0 0</td>
<td>23</td>
<td>7 7 9</td>
<td>36</td>
<td>1 1 1</td>
</tr>
<tr>
<td>11</td>
<td>11 11 11</td>
<td>24</td>
<td>4 4 4</td>
<td>37</td>
<td>1 1 1</td>
</tr>
<tr>
<td>12</td>
<td>4 4 4</td>
<td>25</td>
<td>6 6 6</td>
<td>38</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

Table 2. Problem 2: line configuration.

<table>
<thead>
<tr>
<th>WC#</th>
<th>WS assigned</th>
<th></th>
<th>tasks assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1 2 3 5 6 10 14</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7 8 9 12 13 18 22</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11 15 16</td>
<td>/ / / / / / / /</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>17 20</td>
<td>/ / / / / / / /</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>19 21</td>
<td>/ / / / / / / /</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>23 24 25 26 27 28 29 30 33 36</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>31 32 34 35 37 38</td>
<td>/ / / /</td>
</tr>
</tbody>
</table>

Tables 3 and 4 shows execution times and outputs (for Problem 1 and 2 respectively) of different simulations, varying the package used, the implementation (process oriented, 2p) and the simulation length. 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.

Table 3. Execution times and outputs for Problem 1

<table>
<thead>
<tr>
<th>Simulation length (time units)</th>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Process oriented</td>
</tr>
<tr>
<td></td>
<td>JavaSim</td>
</tr>
<tr>
<td>100,000</td>
<td>1.26</td>
</tr>
<tr>
<td>400,000</td>
<td>4.65</td>
</tr>
<tr>
<td>700,000</td>
<td>8.12</td>
</tr>
<tr>
<td>1,000,000</td>
<td>11.30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs (time units):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle time: 17.00; Flow time: 59.00; WIP: 3.47</td>
</tr>
</tbody>
</table>

Table 4. Execution times and outputs for Problem 2

<table>
<thead>
<tr>
<th>Simulation length (time units)</th>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Process oriented</td>
</tr>
<tr>
<td></td>
<td>JavaSim</td>
</tr>
<tr>
<td>100,000</td>
<td>1.93</td>
</tr>
<tr>
<td>400,000</td>
<td>7.11</td>
</tr>
<tr>
<td>700,000</td>
<td>12.49</td>
</tr>
<tr>
<td>1,000,000</td>
<td>18.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs (time units):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle time: 34.45; Flow time: 234.25; WIP: 6.80</td>
</tr>
</tbody>
</table>

Results also show that the simulator implemented with SSJ is slightly faster than the one implemented with JavaSimulation, both for the full process oriented and for the 2p implementations. Execution times increases linearly with the simulation length, i.e. with the number of loads completed. For the same simulation length, one would expect the execution time increases when WIP increases, and decreases when cycle time increases. In effect, WIP is equivalent to the average number of loads in the line, that is, to the number of processes that run simultaneously. Throughput (equal to the cycle time inverse) is proportional to the activation rate of loads (and other entities) processes.

Problem 2 is a problem with a higher number of WIP, but with a lower throughput than Problem 1. It is noteworthy that execution times for Problem 2 are considerably higher, when using the process oriented implementation, with respect to Problem 1. This means that the higher number of simultaneous processes is not balanced by the lower rate of activations. At the contrary, the 2p implementation does not show the same increase, meaning that the higher number of WIP in this case is balanced by the lower throughput. This is certainly due to the reduction of activation calls between entities needed to perform the same actions (see Fig. 4 and 5). Table 3 and 4 show that the implementation with only two Process classes is always faster, for both packages JavaSim and SSJ. In the 2p
implementation the execution time seems to be sensitive to the ratio WIP/cycleTime (or the product between WIP and throughput). This ratio is in effect correlated to the number of activation calls per unit time during the simulation, considering that the throughput is the rate of the activation call of each load, while the WIP is the number of loads that are simultaneously in the line.

5. CONCLUSIONS

The two packages for discrete event simulation in Java utilised in this work allow a very fast and easy construction of a simulator for a quite complex system as a mixed-model assembly line with parallel workstations. The simulator presented herein can be used as supporting tool in finding solution of the assembly line balancing problem (and the optimal sequencing problem associated to it).

In this context, a simulator has to provide performances of extreme flexibility and fastness of execution, in order to be effectively coupled to those algorithms and procedures where numerous variants of a base system may 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.

As far as flexibility concerns, different lines configurations can be easily simulated simply passing to the constructor of the Simulation class, which is provided in the package, the array representing the solution to be evaluated. Similarly, different sequences of model can be simulated by simply changing the array passed to the constructor.

With respect to speed, it is worth noticing that both packages implement processes as true Java threads, and this adds significant overhead and prevents the use of a very large number of processes in the simulation. However, the implementations proposed limit the number of processes in the system, preventing the formation of infinite queues and consequent system crashes, and provides acceptable execution times. In particular, an implementation that limits the activation calls rate between processes has been presented, and obtained satisfactory results.

The simulator presented is expandable, and authors are working to future version in order to expand its flexibility (including for example the possibility to consider parametric buffers between workcentres) and to speed up its execution (through the implementation of an event-oriented approach).

The LineSimulator system is available from the first author’s web page [Tiacci 2007].

References


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/].


