Automated Manufacturing System Discovery
Report for the Course: Python for Scientific Computing

Giovanni Lugaresi

April 27, 2021

Abstract

Modern production environments are frequently subject to disruptions and consequent modifications. As a result, the development of digital twins of manufacturing systems cannot rely solely on manual operations. Further, generating models with an appropriate level of detail can avoid useless efforts and long computation times, while allowing for easier understanding and re-usability. This report describes part of a research which aims to automatically discover manufacturing systems and generate adequate digital twins. The relevant characteristics of a production system are automatically retrieved from data logs. Two main steps of the research are covered by this report: (1) a model generation procedure to automatically discover a manufacturing system from the production data; (2) a method to tune the model toward a desired level of detail, removing complexities that may hinder both the understandability and re-usability of the model for taking production planning and control decisions. Sample code is attached to this work.

Contents

- section 1 introduces the work within the research objectives;
- section 2 describes the model generation methodology;
- section 3 describes the model tuning methodology;
- section 4 applies the generation and tuning methods to a test case;
- section 5 summarizes the achievements and the future developments.

List of attachments: *config.json*; *eventlog.csv*; *test_flowline.py*; *msmlib.py*; *loopfinder.py*; *modelred.py*; *modelscores.py*; *msm_plots.py*; *other.py*.

1 Introduction

Manufacturing systems evolve regularly due to external drivers such as the irregularity of supplies or the availability of new disruptive technologies. Also, more frequent changes may occur. For instance, robots can be moved from one system to another, manufacturing cells can be reconfigured for new products, production lines may be re-shaped following new part-mixes. Besides, the ability to take appropriate decisions exploiting digital models is strongly based on the assumption that models properly aligned with the real system are either already available or obtainable within the decision time epoch [1]. Therefore, the time to develop a new model may hinder the exploitation of a digital twin along a production systems life cycle [2]. Meanwhile, the availability of real-time data suggests that if a model could be generated from datasets in a manufacturing system, the development phase may be significantly shortened, enabling digital twins to be automatically aligned with their real counterparts [3]. Recent approaches introduced the exploitation of process mining (PM) techniques for simulation model generation [4]. PM is a discipline aiming to discover and exploit valuable information from event logs available in information systems [5].

Figure 1 shows the main idea of a model generation procedure. The manufacturing system is composed by three main activities, each recording the events in an event log. The graph model is the result of mining the direct-follow relationships among activities from the dataset. Finally, the graph can be converted into a simulation model that is able to estimate the system performance indicators such as production throughput or system time. Indeed, from process mining literature we know that a graph model has a Petri Net equivalent [6].

Despite the aforementioned advancements, practical implementations of automated model generation remain scarce. One of the reasons is the difficulty in adapting the level of detail of the model [7]. The availability of an up-to-date process model can result to be insufficient since an automated model generation procedure could model some parts of the system with an excessive level of detail. A common term referring to over-complex models is the *spaghetti model* effect [8]. These models are not only excessive in terms of size, but also inaccurate in estimating performance measures. Hence, the ability to tune – or adjust – the model level of detail is also desirable in an automated modeling procedure. In its most basic form, model tuning is a method to modify a model toward a desired size. In manufacturing applications, model adaptation may refer not only to the model parameters, but also to the system layout and logical structure. A tuned model is easily understandable, and it has a higher probability of being reused [9].

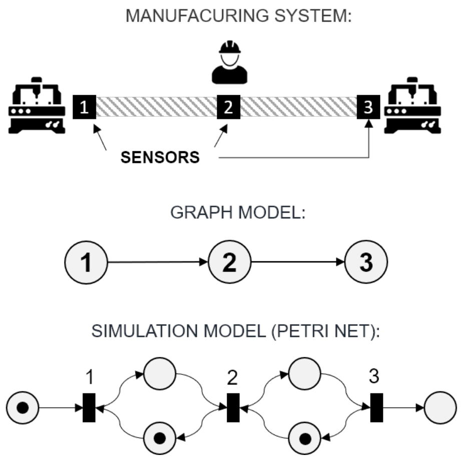

Figure 1: Representation of graph based model generation.

2 Graph-based System Discovery

In this section, we outline the data-based system discovery and model generation, while in section 3 we present the proposed model tuning method.

2.1 The Event Log

Event logs are files that aggregate all the data produced by the manufacturing system. In general, the event log may contain several types of information, such as part flows, resources identifiers, and quality check outcomes. In this work, we assume the availability of event logs containing three main information types: (1) the activity identifier $n \in \mathbb{N}$, (2) the work-piece identifier $h \in \mathbb{H}$, and (3) the timestamps $t_S(n,h)$ and $t_F(n,h)$ indicating the moment at which the n-th activity has started and finished on the h-th piece, respectively. Further, we assume that the time span covered by the event log corresponds to the time horizon of interest. Let us define each row of the log as *event* and we define *trace* $\theta_h \in \Theta$ the set of events experienced by the h-th work-piece, where Θ is the set of all the traces identified from the log. Hence, each h-th part has a corresponding trace $\theta_h = \{n^{(1)}, n^{(2)}, \dots, n^{(e_h)}, \dots, n^{(\#_h)}\}$, where $\#_h$ is the number of the activities performed by the h-th part and e_h indicates the sequential position of the activity as observed in the log for part h. Table 1 shows an example of event-log generated by the manufacturing system of Figure 1. In this example, the trace of part $h = 1$ is $\theta_1 = \{S_1, C_1, S_2, C_2, S_3\}$.

Table 1: Extract from the log of manufacturing system in Figure 1. Note: the time-stamp can be in several formats, it is assumed that pre-processing steps can convert it in any desired format.

Time-stamp	Part-ID	Activity-ID	Tag
2020-11-23 16:37:40	1	1	start
2020-11-23 16:37:44	1	1	finish
2020-11-23 16:37:47	2	1	start
2020-11-23 16:37:51	2	1	finish
2020-11-23 16:37:52	1	2	start
2020-11-23 16:37:54	3	1	start
2020-11-23 16:37:57	1	2	finish
2020-11-23 16:37:58	3	1	finish

Code. The attached file *eventlog.csv* refers to the data generated by a 6-station flow line producing 1000 parts. This event log will be used in the example case in section 4.

2.2 Digital Models Generation

We may represent a simulation model as a directed graph, in which nodes represent the manufacturing activities, and arcs represent the material flow relationships between the activities. Let us define a model Ω as a tuple $\Omega = (\mathbb{N}, \mathbb{A})$ where $\mathbb{N}$ is the set of nodes and $\mathbb{A} \subseteq \mathbb{N} \times \mathbb{N}$ is the set of arcs in the model. For instance, the graph model obtained in Figure 1 is defined by the set of nodes $\mathbb{N} = \{1,2,3\}$ and the set of arcs $\mathbb{A} = \{(1,2),(2,3)\}$. Nodes and arcs may also contain information about the system characteristics: the logical layout (i.e. precedences among activities), the capability of holding work-in-progress parts, the production volume over a certain time span, the routing policies, and the flow times.

Model generation is a procedure which links the data in the event log with a digital model Ω. It identifies the nodes and arcs and collects them in an directed graph Ω_0. The first step is the identification of the *traces*. This procedure characterizes the possible sequences of events in the system. Starting from the event log, a unique set of activities $\mathbb{N}$ is created and the traces of all the $|\mathbb{H}|$ parts are identified. Then, the traces are used to retrieve precedence relationships among activities. Hence, a node exists in the model if a certain activity has been performed by at least one part, and an arc indicates that a production step has followed another in at least one trace. Specifically, arc (n,m) exists if $\exists h \in \mathbb{H} | n,m \in \theta_h \wedge t_F(n,h) < t_S(m,h)$. Next, nodes and arcs are enriched with properties. Table 2 summarizes the notation for the nodes and arcs used in this work.

Nodes. A node identifies an activity. It can be represented as a tuple $n = (\mathbb{P}_n, \mathbb{S}_n, \kappa_n, \phi_n, \xi_n, \tau_n)$, where $\mathbb{P}_n, \mathbb{S}_n$ are sets of predecessor and successor nodes, respectively, κ_n is the buffer capacity of the node, ϕ_n the frequency of occurrence of the respective activity, ξ_n the number of events close in time, and T_n the flow times matrix.

- Each node is linked to its predecessors and successors nodes ($\mathbb{P}_n$, $\mathbb{S}_n$). Namely, two sets of nodes that represent the activities done before ($\mathbb{P}_n$) and after ($\mathbb{S}_n$) the n-th node, respectively.
- The capacity of a node κ_n is defined as the maximum amount of work-pieces that can be processed together by the corresponding production activity. It can be estimated as follows:

$$\hat{\kappa}_n = \max k \,|\, t_S(n, h-k) \leq t_F(n,h) \quad \forall n \in \mathbb{N}. \tag{1}$$

- The frequency ϕ_n indicates the number of times the corresponding activity has been observed in the log.
- ξ_n indicates the number of events that have been identified as close in time on the n-th node. In general, we define activities on the n-th node to be close in time if their time stamps satisfy $|t_F(n,h) - t_S(n,h)| \leq \zeta_N$, where ζ_N is a user-defined threshold.
- The flow times are described by a matrix, where each element indicates the time work-piece h took to flow in node n. Hence:

$$\tau_{h,n} = t_F(n,h) - t_S(n,h) \quad \forall h \in \mathbb{H}, n \in \mathbb{N}. \tag{2}$$

Arcs. An arc is a connector between two nodes. It can be represented by a tuple $a = (\eta_a, c_a, f_a, e_a, T_a)$, where η_a is a tuple of connected nodes, c_a the buffer capacity of the arc, f_a the occurrence frequency, e_a the number of close events, and T_a the arc flow times matrix.

- The nodes connected by the a-th arc are collected in a tuple $\eta_a = (n,m) \subseteq \mathbb{N} \times \mathbb{N}$.
- The capacity c_a of an arc is defined as the maximum amount of work-pieces that has resided on the arc at the same time, as retrieved from the event log. Namely:

$$c_a = \max k \,|\, t_F(n, h-k) \leq t_S(m,h) \quad \forall a \in \mathbb{A} \,|\, \eta_a = (n,m). \tag{3}$$

Table 2: Notation for digital models.

Nodes $n \in \mathbb{N}$	$\mathbb{P}_n$	Predecessor nodes set.
	$\mathbb{S}_n$	Successor nodes set.
	κ_n	Buffer capacity of a node.
	ϕ_n	Frequency of a node.
	ξ_n	Number of close events on a node.
	$T_n = \{\tau_{k,n}\}$	Nodes flow times matrix.
Arcs $a \in \mathbb{A}$	$\eta_a = (n,m)$	Nodes connected by an arc.
	c_a	Buffer capacity of an arc.
	f_a	Number of events on an arc.
	e_a	Number of close events on an arc.
	$T_a = \{t_{k,a}\}$	Arc flow times matrix.

- The number of work-pieces that have been observed flowing through the arc is indicated by f_a.
- e_a is the number of events that have been identified as close in time on the a-th arc, hence in which their time stamps satisfy $|t_S(m,h) - t_F(n,h)| \leq \zeta_A$, where ζ_A is a user-defined threshold and $\eta_a = (n,m)$.
- Flow times are defined by the matrix $T_a = \{t_{h,a}\}$ where each element indicates the time that the h-th work-piece took to flow in arc a:

$$t_{h,a} = t_S(m,h) - t_F(n,h) \quad \forall h \in \mathbb{H}, \forall a \in \mathbb{A} \,|\, \eta_a = (n,m). \tag{4}$$

Code. The attached file *test_flowline.py* refers the generation of a graph model for the a 6-station flow line depicted in figure 4 producing 1000 parts. The generated model is saved in a json file *original_model.json*. Listing 1 reports an example of model structure.

Listing 1: json example

```
"nodes":
[{"activity": 1, "predecessors": [], "successors": [2],
   ↪ "frequency": 1000, "capacity": 1, "contemp": 414,
   ↪ "cluster": 1},
{"activity": 2, "predecessors": [1], "successors": [3],
   ↪ "frequency": 1000, "capacity": 1, "contemp": 379,
   ↪ "cluster": 2},
{"activity": 3, "predecessors": [2], "successors": [4],
    ↪ "frequency": 1000, "capacity": 1, "contemp": 409,
    ↪ "cluster": 3},
{"activity": 4, "predecessors": [3], "successors": [5],
    ↪ "frequency": 1000, "capacity": 1, "contemp": 392,
    ↪ "cluster": 4},
{"activity": 5, "predecessors": [4], "successors": [6],
    ↪ "frequency": 1000, "capacity": 1, "contemp": 391,
    ↪ "cluster": 5},
{"activity": 6, "predecessors": [5], "successors": [],
    ↪ "frequency": 1000, "capacity": 1, "contemp": 416,
    ↪ "cluster": 6}],
"arcs":
[{"arc": (1, 2), "capacity": 10.0, "frequency": 1000,
    ↪ "contemp": 92},
{"arc": (2, 3), "capacity": 10.0, "frequency": 1000,
    ↪ "contemp": 149},
{"arc": (3, 4), "capacity": 10.0, "frequency": 1000,
    ↪ "contemp": 128},
{"arc": (4, 5), "capacity": 10.0, "frequency": 1000,
    ↪ "contemp": 224},
{"arc": (5, 6), "capacity": 10.0, "frequency": 1000,
    ↪ "contemp": 219}]}
```

3 Model Tuning

The focus is on how to properly build and tune digital models which are excessively exhaustive for the user purposes and have to be modified toward a reasonable level of detail. We may define model tuning as a procedure that aims to adapt an existing model in order to satisfy complexity requirements, which can be expressed in terms of maximum number of nodes or arcs. Simulation models for manufacturing systems typically represent components such as parts, resources (e.g., machines, conveyors, operators), and the paths along which the parts are flowing in the system. Hence, in a model tuning procedure, it is desirable to keep track of how much the main components of a simulation model are being represented and, possibly, reduced.

3.1 General Idea

Figure 2 graphically explains one of the possible outcomes of an automated model generation procedure, using as example the three-station production line shown in

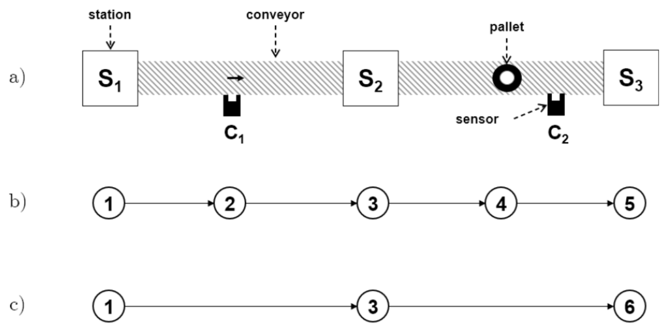

Figure 2: (a) Sequential 3-station production line; (b) graph model with 5 nodes; (c) graph model with 3 nodes.

figure 2a. Conveyors bring pallets from one station to another, and are equipped with sensors that record the correct advancement of pallets. If all stations and sensors generate data, figure 2b shows a possible outcome of an automated data-based model generation. The result is that sensors are treated as activities, thus adding unnecessary operations to the model. Let us assume to be interested in time-related performance indicators such as the system time. In this case, the information from the sensors is redundant since the elements holding the work-pieces are the three stations. Hence, for the intended goal, the model depicted in figure 2c is a more reasonable abstraction of the process, and much closer to what an experienced modeler would choose.

3.2 Methodology

In general, let us define $\Omega_0 = (\mathbb{N}_0, \mathbb{A}_0)$ the model generated by the procedure described in section 2, and $\Omega_j = (\mathbb{N}_j, \mathbb{A}_j)$ is a j-th alternative graph model. Among the alternative models satisfying the user-requirements in terms of level of detail, model tuning finds the one that maximizes an adequacy-related score. The score can be computed by the function $\Phi(\Omega)$ in equation (5), which determines how acceptably a model Ω represents the real production system.

$$\Phi(\Omega) = \sum_i w_i R_i(\Omega) \tag{5}$$

where each score $R_i(\Omega)$ is a function that describes *how well does the model* Ω *represent the i-th characteristic of the system*, and w_i is the weight of the i-th score. This way, the score of a model is directly linked to the event log and the following properties of the manufacturing system: (1) buffer sizes, (2) events close in time, (3) re-entrant flows, (4) split and merge points, and (5) activity occurrence frequency. We have defined five $R_i(\Omega)$ functions which are listed in Table 4.

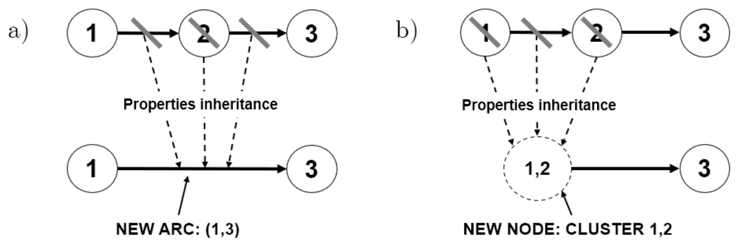

Figure 3: Properties inheritance example: a) reduction, b) aggregation.

The goal of model tuning is *to find the model* Ω *that maximizes the score* Φ *defined in equation* (5) *while satisfying a set of constraints (e.g., number of nodes).* Such problem is solved with the local search algorithm described in Algorithm 1. Algorithm 1 is based on the possibility to generate neighbors of a model. A neighbor is derived by means of either two moves: (1) aggregation, in which two nodes are merged in a cluster, (2) reduction, in which one node is removed from the model. In both aggregation and reduction moves, new arcs may be added in order to maintain the flow of parts, and clusters may be added to represent node groups. In this case, the newly introduced elements inherit the properties of the removed parts. Figure 3 represents an example of this idea. In the reduction case, node 2 is removed from the model, hence also the connected arcs $(1,2)$ and $(2,3)$ have been removed. To guarantee the flow of parts, arc $(1,3)$ is added. The properties of the added arc are derived from the ones of all the removed elements. In the aggregation case, a new node cluster is created by merging nodes 1 and 2.

Algorithm 1: Local Search – Find the most adequate model Ω_f of size U_N^{max}.

```
1 Input: Graph model $\Omega_0$, size limit $U_N^{max}$;
2 Output: Graph model $\Omega_f$;
3 STEP 0: assign $\Omega_{current} \leftarrow \Omega_0$; $i \leftarrow 0$;
4 while $\neg(\#_N(\Omega_{current}) \leq U_N^{max})$ do
5     STEP 1: Generate $\sigma_A + \sigma_R$ neighbors of $\Omega_{current}$ [10];
6     STEP 2: Sort neighbors based on score $\Phi$;
7     STEP 3: Set $\Omega_{current} \leftarrow argmax_{\{\Omega \in \mathbb{M}_A \cup \mathbb{M}_R\}} \Phi(\Omega)$, $i \leftarrow i+1$;
8 end
9 return $\Omega_f \leftarrow \Omega_{current}$;
```

Algorithm 1 is a depth-first local search algorithm. At each step, a certain number of neighbors can be generated. The neighbors generation function is described

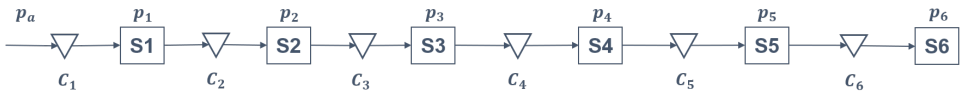

Figure 4: Six-station flow line used for the experiments in this work. Squares depict stations, while inter-operational buffers are represented as triangles.

in [10]. Let us call σ_A and σ_R the number of neighbor models generated by aggregation and reduction, respectively. Hence, at each step, two sets of neighbors $\mathbb{M}_A$ and $\mathbb{M}_R$ are generated, and $|\mathbb{M}_A| = \sigma_A$, $|\mathbb{M}_R| = \sigma_R$. Let us accept the short notation $\mathbb{N}_i$, $\mathbb{A}_i$ indicating the node and arc sets of the model selected at the i-th iteration of Algorithm 1. At any i-th iteration, the complete exploration of the neighbors set is achieved when $\sigma_A = \sigma_R = |\mathbb{N}_i|$. Since a neighbor is defined by the reduction of the model size by one node, $|\mathbb{N}_i|$ neighbors can be generated by means of reduction moves, while $|\mathbb{A}_i|$ neighbors by means of aggregation moves. For saving computation time, we may also set $\sigma_A, \sigma_R < |\mathbb{N}_i|$. As a consequence, it is necessary to define a neighbor generation rule, which defines which nodes are to be considered for either aggregation or reduction moves. In this work, neighbors are generated by aggregating or reducing first the nodes with the lowest frequency.

Code. The attached code *test_flowline.py* is used in section 4 for the model tuning. Specifically, a model with 6 activities is reduced to a model with 4. The generated model is saved in a json file named *tuned_model.json*.

4 Test case with code

In this section a test case is presented. The test case is provided together with a sample code that can be used (file: *test_flowline.py*). In this test, a flow line consisting of six stations depicted in Figure 4 is taken as reference. The flow line processes one part type. Stations process parts in sequence and each station can process one part at a time. Inter arrival and processing times are distributed according to an exponential distribution with mean $1\,min$. Each station s has an input buffer C_s. The first buffer is infinite, hence all arriving parts are accepted, while buffers C_2 to C_6 can store maximum 10 parts each. A simulation model of the flow line has been developed in Rockwell Arena and used to generate an independent event logs through a simulation experiment representing the production of 1000 parts.

This test consists in the following steps:

1. Dataset loading.

2. Graph-model generation.
3. Model tuning.

The following python packages are required: *pandas, numpy, json, tqdm, warnings, operator, copy, time, graphviz, matplotlib*.

4.1 Dataset loading

The first step is to load the database. This is done exploiting the python package *pandas* (pd), with the following command:

```
data = pd.read_csv( 'eventlog.csv', sep=",", header='infer')
```

4.2 Graph-model generation

In this step, the graph-model of the system of figure 4 is generated.

The model generation steps are wrapped in the python function *gen_model_init*, in the following command:

```
model, unique_list, *args = gen_model_init(data, config, tag = True)
```

In this function, the **input data** is the pandas dataframe defined in section 4.1, together with a configuration dictionary called *config* that collects all the parameters of interest (e.g., the weights of function (5)). An additional boolean *tag* specifies if the column of the eventlog called *tag* is to be used or not in the model generation procedure to determine the arc capacities. The **output** of the model generation function are the model in the format as in listing 1, and a list called *unique_list*, which collects all the unique trace variants in the model (e.g., if the trace $\theta = \{1,2,3\}$ is observed for more than one part ID, it is counted once as variant). Additional outputs are optional: the list of the traces removed by outliers check, and the list of the traces identified for each single part ID.

The function *gen_model_init* belongs to the developed library *msmlib.py* and exploits other functions. Briefly:

- The function *traces* identifies the traces and removes outliers based on frequency thresholds.
- The function *activity_rel_data* identifies the activities (i.e., nodes) in the model and their precedence relationships.
- The function *capacity_calc* identifies the arcs of the model and computes the capacity property of each arc.

- The functions *batching_on_nodes* and *batching_on_arcs* compute the close events on nodes and arcs, respectively.

Figure 5a shows the generated model using *gen_model_init*.

4.3 Model Tuning

The following step in the test is model tuning. In this step, we assume to be interested in simplifying the model of the line and having as output a model with 4 nodes. We also assume to be interested in a clustered model, hence in which activities are clustered, not removed.

The model tuning steps are wrapped in the python function *local_search*, in the following command:

```
record = local_search(model, unique_list, config, pen = 0)
```

In this function, the **input data** is the model generated in section 4.2, together with the unique list of trace variants, which is used to identify loops in the model, the configuration dictionary called *config* that collects all the parameters of interest (e.g., the weights of function (5)), and a penalty parameter that optionally penalizes nodes aggregation. The function implements Algorithm 1, and it is supported by the following fuctions, collected in the developed library *modelred.py*:

- The function *score_calc* computes the model score as defined by equation (5).
- The function *neighbours_aggregate* identifies a required number of neighbours of a model by aggregating nodes in clusters.
- The function *neighbours_reducing* identifies a required number of neighbours of a model by removing nodes from the model.

The **output** of function *local_search* is a list of dictionaries containing the following information: (1) the iteration step, (2) the solution identified at that step, (3) the score of the solution, (4) the type of solution (i.e., if obtained by aggregation or reduction), and (5) the computation time of the iteration. Figure 5b shows the tuned model using the function *local_search*.

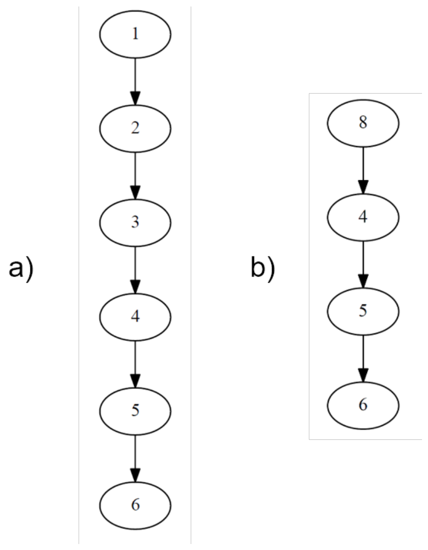

Figure 5: Test case example: a) graph model generation, b) model tuning.

5 Conclusions

The capability to generate an accurate discrete event simulation model in a short time is essential to achieve digital twin potentials. In this report, a method for discovering production systems structures and automatically develop digital models starting from the event logs of manufacturing systems has been described. The proposed automated generation and tuning method can positively contribute to real-time simulation applications [11], since it guarantees that an updated and reasonably detailed model of the physical system can be obtained at any time with no manual intervention. Several limitations still have to be solved and may be addressed by future works. For instance:

- *Investigate the application to other manufacturing system types.* Indeed, there may be significant differences among other system types. For instance, job shops are characterized by several independent part flows, which may result in a more complex identification of the system structure.
- *Remove the single part ID hypothesis.* The model generation procedure and the tuning method are not suited for assembly/disassembly lines in which multiple components may be tracked, and new part assembly identifiers may be added along the production.
- *Develop smarter model tuning rules.* For instance, to aggregate parallel ac-

tivities or remove more than two nodes at a time if the model structure suggests it.

- *Investigate the resilience against big datasets*. Outcomes of realistic applications where production systems could be composed by a consistent number of activities and such cases shall be of interest.

APPENDIX A: How this approach has been used

This section elaborates on one of the experiments that has been done to validate the approach proposed in this work. Specifically, the test is a six-station flow line described in figure 4. The scope is to verify the behavior of the proposed approach by comparing the performance estimated by the obtained simulation models with the real system one (i.e. validation). The model has been used to generate 10 independent event logs, each representing the production of 1000 pieces. The objective of the experiment is to verify if a reduced model can still correctly estimate the performances of the real system. This has been done by varying the required size in the interval $[4,6]$. Hence, for each log, three alternative models have been generated. In order to validate the generated models, each of them has been used to simulate the production of 1000 pieces in experiments replicated five times each. For this purpose, we have converted each obtained graph model Ω_f into a simulation model in Rockwell Arena, which we used to estimate the performances in terms of throughput $TH\,[parts/min]$ and system time $ST\,[min]$. Table 3 summarizes the obtained performances (mean values among the five replications). The maximum computation time to obtain a tuned model is $43.29s$. Figure 6 shows the results obtained for the fifth log of the original system, where the throughput and system time of the original system are marked with a red dashed line and are $0.846\,parts/min$ and $29.0\,min$, respectively. We can notice how both system time and throughput are changing depending on which model version we are using to simulate. As expected, the estimated performances are worsening with smaller dimensions of the models. However, it is worth to notice that for the most reduced model ($U_N^{max} = 4$), the mean absolute error is 5% for the throughput and 7% for the system time. These errors can be acceptable depending on the intended use of the digital models. For instance, the model could be used for initial estimations and definition of lower or upper bounds on decision variables. Further, in the best cases observed, the mean absolute error is 0.23% for the throughput and 0.62% for the system time. More details on other test cases and experiments are available in related works [10].

Table 3: Test Case 1 – Results obtained in terms of throughput (TH) and system time (ST) for both the complete and reduced models. For all the experiments, the Absolute Error (AE) is provided. The maximum half width confidence interval is 0.2% for the throughput and 2.8% for the system time.

		Event logs										
	U_N^{max}	1	2	3	4	5	6	7	8	9	10	**Mean**
TH	*Original*	0.849	0.828	0.860	0.868	0.846	0.862	0.812	0.818	0.864	0.813	0.842
	4	0.780	0.807	0.831	0.820	0.808	0.835	0.729	0.763	0.810	0.809	0.799
TH	5	0.801	0.788	0.822	0.822	0.788	0.795	0.772	0.769	0.812	0.774	0.794
	6	0.824	0.823	0.848	0.845	0.844	0.849	0.807	0.810	0.843	0.842	0.834
	4	0.069	0.021	0.028	0.048	0.038	0.028	0.083	0.055	0.054	0.004	0.043
AE	5	0.048	0.040	0.037	0.046	0.058	0.067	0.040	0.050	0.053	0.039	0.048
	6	0.025	0.005	0.011	0.022	0.002	0.014	0.005	0.008	0.021	0.029	0.014
ST	*Original*	30.9	33.2	37.0	31.4	29.0	38.0	34.5	34.0	30.9	32.1	33.1
	4	29.0	27.2	24.4	25.4	24.2	26.1	30.9	27.0	26.1	26.6	26.7
ST	5	36.4	32.5	33.5	30.5	33.4	34.4	38.6	36.6	30.7	34.6	34.1
	6	34.7	31.7	33.7	33.0	32.6	34.6	31.9	35.1	33.6	32.3	33.3
	4	1.9	6.1	12.5	5.9	4.7	11.8	3.5	7.0	4.8	5.5	6.4
AE	5	5.5	0.7	3.5	0.8	4.5	3.6	4.2	2.7	0.2	2.6	2.8
	6	3.8	1.5	3.2	1.6	3.7	3.3	2.6	1.1	2.7	0.2	2.4

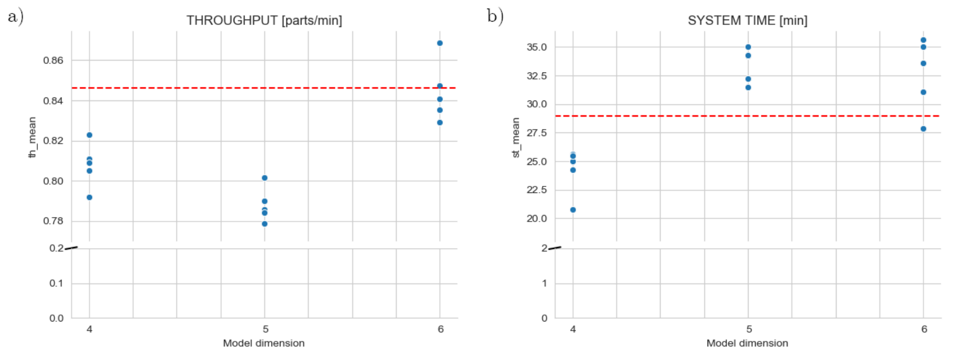

Figure 6: Test Case 1 – Results of the experiments in the flow line case (fifth event log): a) throughput; b) system time.

References

[1] Giovanni Lugaresi, Vincenzo Valerio Alba, and Andrea Matta. Lab-scale models of manufacturing systems for testing real-time simulation and production control technologies. *Journal of Manufacturing Systems*, 58:93 – 108, 2021.

[2] Anders Skoogh, Terrence Perera, and Björn Johansson. Input data management in simulation–industrial practices and future trends. *Simulation Modelling Practice and Theory*, 29:181–192, 2012.

[3] Heiner Reinhardt, Marek Weber, and Matthias Putz. A Survey on Automatic Model Generation for Material Flow Simulation in Discrete Manufacturing. *Procedia CIRP*, 81:121–126, jan 2019.

[4] Anne Rozinat, Ronny S Mans, Minseok Song, and Wil MP van der Aalst. Discovering simulation models. *Information systems*, 34(3):305–327, 2009.

[5] Wil Van Der Aalst, Arya Adriansyah, Ana Karla Alves De Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter Van Den Brand, Ronald Brandtjen, Joos Buijs, et al. Process mining manifesto. In *International Conference on Business Process Management*, pages 169–194. Springer, 2011.

[6] Willibrordus Martinus Pancratius van der Aalst, KM Van Hee, and GJ Houben. Modelling and analysing workflow using a petri-net based approach. In *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50. of, 1994.

[7] Riku-Pekka Nikula, Marko Paavola, Mika Ruusunen, and Joni Keski-Rahkonen. Towards online adaptation of digital twins. *Open Engineering*, 10(1):776–783, 2020.

[8] W.M.P. Van der Aalst. *Process Mining - Data Science in Action*. Springer, second edition edition, 2016.

[9] Stewart Robinson, Richard E Nance, Ray J Paul, Michael Pidd, and Simon JE Taylor. Simulation model reuse: definitions, benefits and obstacles. *Simulation modelling practice and theory*, 12(7-8):479–494, 2004.

[10] Giovanni Lugaresi and Andrea Matta. Automated manufacturing system discovery and digital twin generation. *Journal of Manufacturing Systems*, 59:51–66, 2021.

[11] Giovanni Lugaresi, Vincenzo Valerio Alba, and Andrea Matta. Lab-scale models of manufacturing systems for testing real-time simulation and production control technologies. *Journal of Manufacturing Systems*, 58:93–108, 2021.

[12] Keith Paton. An algorithm for finding a fundamental set of cycles of a graph. *Communications of the ACM*, 12(9):514–518, 1969.

[13] Martin Prodel. *Modélisation automatique et simulation de parcours de soins à partir de bases de données de santé*. PhD thesis, Lyon, 2017.

Table 4: Loops are identified as cycles in a directed graph $\Omega = (\mathbb{N}, \mathbb{A})$ with the algorithm defined in [12].

Name	Expression	Meaning
Buffers	$R_1(\Omega) = r_1^{(A)} \frac{\sum_{a\in\mathbb{A}} c_a}{\sum_{a\in\mathbb{A}_0} c_a} + r_1^{(N)} \frac{\sum_{n\in\mathbb{N}} \kappa_n}{\sum_{n\in\mathbb{N}_0} \kappa_n}$	R_1 is a function that represents the buffer capacity of the system. The aim is to favor the inclusion of nodes and arcs with a higher buffer capacity. ($r_1^{(N)}$ and $r_1^{(A)}$ balance the relative weight of nodes and arcs, respectively)
Close events	$R_2(\Omega) = \frac{r_2^{(A)}}{\lvert\mathbb{A}\rvert} \sum_{a\in\mathbb{A}} (1 - \frac{e_a}{f_a}) + \frac{r_2^{(N)}}{\lvert\mathbb{N}\rvert} \sum_{n\in\mathbb{N}} (1 - \frac{\xi_n}{\phi_n})$	R_2 is a function related to the number of events in the system which occur within a short time window. For example, in a production line the moment a part leaves a buffer may correspond to the recorded time it enters the downstream station. R_2 favors the exclusion of nodes and arcs related to activities that are done within short time frames ($r_2^{(N)}$ and $r_2^{(A)}$ balance the relative importance of node and arc properties).
Re-entrant flows	$R_3(\Omega) = \frac{1}{\lvert\mathbb{A}\rvert} \sum_{n\in\mathbb{N}} \sum_{m\in\mathbb{N}} \gamma_{nm} \iota_{nm}$	R_3 is related to the loops in the material flow. A loop may simply represent a station that withdraws parts from a conveyor, performs a certain activity, and then releases the parts back to the conveyor. Since the performance of the loop influences the production rate of the system, it is desirable to preserve it in the representation. This function encourages the inclusion of parts in the model which are involved in loops (ι_{nm} is 1 if the arc (n,m) is a portion of a loop in the model, 0 otherwise).
Split and merge	$R_4(\Omega) = r_4^{(in)} \sum_{n\in\mathbb{N}} \sum_{x\in\mathbb{S}_n} \gamma_{nx} + r_4^{(out)} \sum_{n\in\mathbb{N}} \sum_{l\in\mathbb{P}_n} \gamma_{ln}$	R_4 is a routing score defined in [13]. For instance, in a manufacturing system the logic of the material flow may be defined by splitting or merging points along the physical conveyors, i.e. positions in which alternative activities following or preceding the n-th node are possible. In such locations, relevant decisions may have to be taken (e.g., prioritizing). Therefore, it may be desirable to keep the nodes with multiple connected arcs ($r_4^{(in)}$ and $r_4^{(out)}$ balance the relative significance of inbound and outbound arcs).
Frequency	$R_5(\Omega) = r_5^{(A)} \frac{\sum_{a\in\mathbb{A}} f_a}{\sum_{a\in\mathbb{A}_0} f_a} + r_5^{(N)} \frac{\sum_{n\in\mathbb{N}} \phi_n}{\sum_{n\in\mathbb{N}_0} \phi_n}$	R_5 is a function representing the number of parts flowing in the nodes and arcs of the model. The aim is to favor the inclusion of nodes and arcs which are visited with a higher frequency (i.e. the preferred path). $r_5^{(N)}$ and $r_5^{(A)}$ balance the relative importance of nodes and arcs.