Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)

Learning Scheduling Models from Event Data

Arik Senderovich, Kyle E. C. Booth, J. Christopher Beck
Department of Mechanical & Industrial Engineering, University of Toronto, Toronto, ON, Canada
sariks@mie.utoronto.ca, kbooth@mie.utoronto.ca, jcb@mie.utoronto.ca

Abstract

A significant challenge in declarative approaches to scheduling is the creation of a model: the set of resources and their capacities and the types of activities and their temporal and resource requirements. In practice, such models are developed manually by skilled consultants and used repeatedly to solve different problem instances. For example, in a factory, the model may be used each day to schedule the current customer orders. In this work, we aim to automate the creation of such models by learning them from event data. We introduce a novel methodology that combines process mining, timed Petri nets (TPNs), and constraint programming (CP). The approach learns a sub-class of TPN from event logs of executions of past schedules and maps the TPN to a broad class of scheduling problems. We show how any problem of the scheduling class can be converted to a CP model. With new instance data (e.g., the day's orders), the CP model can then be solved by an off-the-shelf solver. Our approach provides an end-to-end solution, going from event logs to model-based optimal schedules. To demonstrate the value of the methodology we conduct experiments in which we learn and solve scheduling models from two types of data: logs generated from job-shop scheduling benchmarks and real-world event logs from an outpatient hospital.

Introduction

In declarative approaches to problem solving, the problem is modeled in a formal language and solved by a general purpose solver for the chosen formalism. Research in such areas as AI planning, mixed integer programming, and constraint programming has, therefore, addressed both how to model a problem and how to build a formalism-specific solver.

In this work, we attack the modeling challenge by developing an approach to learn models of scheduling problems from data. For many real-world scheduling applications, data comes in the form of event logs: records of the executed activities, their start and completion times, and the resources that these activities used (van der Aalst 2011). We propose to learn scheduling problems from logs using an existing process mining solution and transform them into constraint programming (CP) models that can be solved by an appropriate solver. We assume the execution as recorded in the event log does not violate any problem constraints, yet make no assumptions about the quality of the schedule. As a result, the logs may arise from the execution of schedules from any source including model-less and manual approaches (e.g., decisions made by staff).

Figure 1 presents an overview of our approach. In the first step, we apply an existing process mining method (Senderovich et al. 2015) to mine the event logs. The result is a timed Petri net (TPN), a well-established expressive formalism for modeling dynamic systems (Silva and del Foyo 2012). We then map the mined TPN to a *basic scheduling problem* (BSP) provided that the TPN obeys structural properties which can be efficiently detected. The BSP captures a class of scheduling problem, generalizing well-known classes such as job-shop scheduling. The final step converts the BSP into a CP model which can then be repeatedly solved with new data (e.g., new customer orders, new patients to be treated) to produce schedules for the target facility.

We provide a proof of concept implementation of our approach and evaluate it in a two-phased experiment. In the first phase, we learn models of job-shop scheduling benchmarks using synthetically generated data. We show that our approach can reconstruct the original benchmark problem and, given enough time, solve it to optimality. In the second phase, we learn scheduling models from real-world logs from a large outpatient cancer hospital in the United States. Our method learns an appointment scheduling model that we solve using CP.

The main contribution of our work is threefold:

1. We provide an end-to-end, data-to-model solution that learns scheduling models from event logs.
2. We introduce activity-resource Petri nets to characterize a sub-type of TPNs that correspond to basic scheduling problems.
3. We provide a mapping of the learned scheduling problem into a CP model and solve new problem instances.

Background

In this section we provide the background for our work. First, we define *timed Petri nets*, the target formalism for our process mining step. Then, we introduce event logs and present

Figure 1: An end-to-end solution for learning scheduling models from event logs.

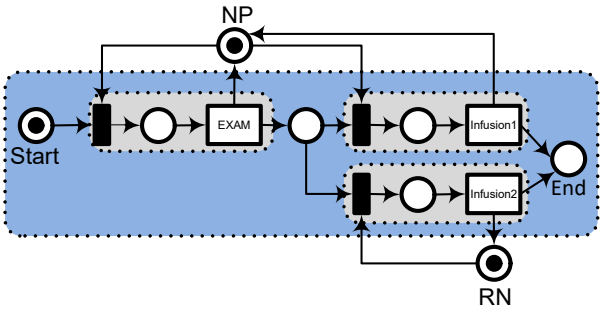

Figure 2: A TPN of a hospital process.

the state-of-the-art methodology for mining TPNs from event logs in schedule-driven systems.

Timed Petri Nets

Timed Petri nets (TPNs) are dynamic models for analyzing discrete-event dynamic systems that exhibit parallelism, synchronization, and resource consumption (David and Alla 1994; Silva and del Foyo 2012).

Figure 2 shows a TPN containing three timed transitions that model activity modes (white rectangles), labeled as Exam, Infusion1 and Infusion2, three immediate transitions that model instantaneous events (black rectangles) and eight places (circles). The Start and End places correspond to job start and end points, respectively. The places labeled NP and RN correspond to two resources: a nurse practitioner and a registered nurse, with the tokens inside the places representing the corresponding resource capacities.

Definition 1 (Timed Petri net (TPN))**.** *A timed petri net $\mathcal{N}$ is a directed graph represented by tuple $\mathcal{N} = \langle E, E', P, F, \tau \rangle$ with,*

- *E being a finite set of transitions with $E' \subseteq E$ being a (possibly empty) set of timed transitions and $E \setminus E'$ being the set of immediate (non-timed) transitions,*
- *P being a finite set of places,*
- *$F \subseteq E \times P \cup P \times E$ being the flow relation (edges) of the Petri net, and,*
- *$\tau : E' \rightarrow \mathbb{R}^+$ being a function that assigns deterministic durations to timed transitions (without loss, we assume a positive real-valued time dimension).*

The marking of a Petri net is a function $M : P \rightarrow \mathbb{N}$ that maps a place to the number of tokens that it contains. The marking in Figure 2 consists of a single job token in the Start place and a single token in the NP and RN places.

A TPN is fully characterized by the pair $(\mathcal{N}, M_0)$ with $\mathcal{N}$ being the net (Definition 1) and M_0 being its initial marking.

We denote by $\bullet p$ ($\bullet e$) the preset of place p (transition e), i.e., the set of places (transitions) that directly precede p (e). Similarly, we denote by $p\bullet$ ($e\bullet$) the postset of place p (transition e), which is the the set of places (transitions) that directly succeed p (e). We define F_P to be the set of incoming and outgoing flows that corresponds to a set of places P, i.e., $F_P = \{(x, y) \in F \mid x \in P \ \vee \ y \in P\}$.

Continuing the example, the preset of NP, $\bullet p$ with p being the place NP, includes two transitions that correspond to Exam and Infusion1, respectively, since these transitions have flows into p. The set $F_{\{p\}}$ with p being the place NP consists of four flows leading from and to NP.

The semantics of a TPN are defined by a *token game*: a transition $e \in E$ is *enabled* in a marking M, if all places in the preset of e are marked, i.e., $\forall\ p \in \bullet e : M(p) > 0$. An immediate transition that is enabled can *fire*. Firing of a timed transition $e' \in E'$ depends on its duration $\tau(e')$: once it is enabled, a deterministic clock is started and the transition can *fire* after $\tau(e')$ units have elapsed. Firing a transition e in a marking M yields a marking M', such that $M'(p) = M(p) - 1$ for all $p \in \bullet e \setminus e\bullet$; $M'(p) = M(p) + 1$ for all $p \in e \bullet \setminus \bullet e$; and $M'(p) = M(p)$ otherwise.

Returning to our example, given the initial marking in Figure 2, only the immediate transition that precedes Exam is enabled. After that transition fires, the two tokens (the job token and the NP token) merge into a single token and enable the timed transition labeled Exam. Following a delay of $\tau(Exam)$, the joined token is split: the NP token is returned to its original place and the job token continues to enable the two immediate transitions that precede Infusion1 and Infusion2.

Learning Models from Event Logs

Process mining is a rapidly evolving research field centered around methodologies for learning process models from event data (van der Aalst 2011). The assumption is that the execution of processes is recorded in event logs, which can be employed to learn models of the underlying system in the form of, for example, Petri nets (van der Aalst, Weijters, and Maruster 2004), transition systems (van der Aalst et al. 2006), process trees (Leemans, Fahland, and van der Aalst 2014), and queueing networks (Senderovich et al. 2016).

An event log comprises a set of events with each event corresponding to an executed (scheduled) activity. Every event contains information about the activity in the form of attributes including:

1. A unique job identifier.
2. The activity type.
3. The resource(s) that executed the activity.
4. The activity start and completion timestamps.

Case	Activity Type	Resources	Start	Complete
pat1	Blood-Draw	RN	9:05AM	9:10AM
pat1	Exam	MD	9:55AM	10:20AM
pat2	Exam	NP	9:30AM	9:45AM
pat2	Infusion	RN	9:35AM	10:52AM
pat3	Exam	NP	12:45PM	1:10PM
pat3	Infusion	NP	9:35AM	10:32AM

Table 1: Excerpt from an event log of a hospital process.

Table 1 presents an excerpt from a real-world event log for an outpatient cancer hospital. The events correspond to activities from a chemotherapy infusion process; the job identifier is the 'case' that identifies the patient and the resources involved are registered nurses (RN), nurse practitioners (NP), and medical doctors (MD).

To assess the quality of learned models, we let $\psi(L, \mathcal{N}) \in [0, 1]$ be a learning quality function that, given an event log L and a process model $\mathcal{N}$, evaluates the model. Higher values of ψ indicate high quality models w.r.t. the event log.

Process learning aims at finding a function γ that maps an event log L onto a timed Petri net model $\gamma(L)$ such that a quality measure $\psi(L, \gamma(L))$ is maximized (van der Aalst 2011). The measure ψ quantifies the similarity between model and log indirectly, since we do not assume that we have labeled pairs of logs and models. For example, ψ can be the *fitness* of a learned model: the proportion of events in the log that can be parsed by the TPN (Rozinat and Van der Aalst 2005). See Chapter 5 in (van der Aalst 2011) for an extensive survey on process learning and its evaluation.

Process Learning for Scheduled Processes

We learn TPNs by adopting the *SchedMiner* approach for learning scheduled processes from event logs (Senderovich et al. 2015).

SchedMiner makes the following assumptions:

- Resources and their capacities do not change over time and resources are work conserving (i.e., resources are available immediately after completing a task).
- Resources reach their capacity at some point in the event log. Otherwise, the capacity learned by the miner provides a lower bound on the true capacity.
- Logs are complete: all resources and activities that participate in the scheduled process are captured in the event log. Furthermore, all possible resources per activity type are observed in the log. For example, the activity Exam in Table 1 can be executed only by an MD or an NP.
- An activity can be executed by one of several resources.
- Non-preemptive: activities that start their execution cannot be interrupted.

With these assumptions *SchedMiner* learns a TPN as follows. First, jobs are aggregated into job variants according to the execution order of their activity types. In Table 1, the jobs $pat2$ and $pat3$ are mapped to a single job variant, ⟨Exam, Infusion⟩, while $pat1$ is mapped to job variant ⟨Blood Draw, Exam⟩.

Next, *SchedMiner* maps activity types to the sets of resources on which they can be executed. Durations for each activity type and resource pairing are learned using averages over the corresponding durations in the log.[1] For Table 1, the duration of an exam activity performed by an MD is 25 minutes and by an NP is 20 minutes (the average of 15 and 25 minutes for $pat2$ and $pat3$).

Based on the job variants, resources, and durations per activity type/resource combination, one single input/single output directed TPN is created for each job variant. The dark area in Figure 2 represents a job variant corresponding to $pat2$ and $pat3$ (see Table 1). Subsequently, resource places that are shared between job variants are created based on the activity/resource combinations and the flows between resource places and job variants are determined according to the resources that can execute each activity. These resource flows connect the separate job variant TPNs into a single TPN. Note that Figure 2 presents one out of many possible job variants.

Finally, *SchedMiner* sets the initial marking of the net. All job tokens start in the input place of their corresponding job variant TPN, while resource tokens reside in their corresponding places. The number of tokens in each resource place corresponds to the capacity of that resource: the maximum number of tasks that the resource performed in parallel in the event log.

We assume that the event log was generated by executing a solution to a scheduling problem. Our goal is to find the scheduling problem definition that generated the log. In this section, we present a formalization of a problem definition that we are able to learn: the *basic scheduling problem* (BSP). We then provide a CP formulation for a BSP, corresponding to the BSP2CP phase in Figure 1.

In the next section, we return to the mining of TPNs and their translation to BSP.

Basic Scheduling Problems

The BSP follows the definition of a *scheduling problem* presented in van der Aalst (1996).[2]

Definition 2 (Basic Scheduling Problem (BSP))**.** *Given a set of activities to be scheduled $\mathcal{A}$, a function $\theta : \mathcal{A} \to \mathcal{T}$ that maps these activities to activity types and a function $\nu : \mathcal{A} \to \mathcal{V}$ that maps activities to job variants, the BSP is a tuple $\langle \mathcal{A}, \theta, \nu, \mathcal{T}, \mathcal{R}, \mathcal{V}, \Pi, c, d \rangle$ with:*

- *$\mathcal{T}$ being the set of activity types,*
- *$\mathcal{R}$ being the set of resources,*
- *$\mathcal{V}$ being the set of job variants with $\mathcal{V} \subseteq \mathcal{T}^*$ where $\mathcal{T}^*$ is a set of finite sequences over $\mathcal{T}$,*
- *$\Pi = \{\Pi_v \subseteq \mathcal{T} \times \mathcal{T} \mid v \in \mathcal{V}\}$ being the set of precedence relations between pairs of activity types, such that for all*

[1] *SchedMiner* can learn stochastic distributions for activity durations. However, we limit our work here to learning deterministic TPNs.

[2] van der Aalst (1996) defines a *scheduling problem* and shows how it can be represented as a TPN. We go in the opposite direction in the next section, showing how a restricted (and learnable) TPN can be represented as a BSP.

$(t, t') \in \Pi_v$, *activity type* t *must complete before* t' *can start in job variant* v,

- $c : \mathcal{R} \to \mathbb{N}^+$ *being the function that maps resources to their capacities, and,*
- $d : \mathcal{T} \times \mathcal{R} \nrightarrow \mathbb{R}^+$ *being the duration partial function that maps pairs of activity types and resources (that can execute these activities) to values in the time domain.*

Note that in a scheduling problem instance that we eventually solve, $\mathcal{A}$, θ and ν are provided externally while the remaining BSP components are learned from the event log via the TPN. In other words, we are given the activities to be scheduled, along with their types and their job variants but the durations, resources, resource capacities and requirements, and precedence relations are all learned. In the hospital example, if we have new activities to be scheduled, $a_1, a_2 \in \mathcal{A}$, we assume to know that a_1 corresponds to type Exam and a_2 corresponds to type Infusion and the two activities belong to job variant $\langle$Exam, Infusion$\rangle \in \mathcal{V}$.

Since the BSP is a parameterized problem, its solution requires precedence relations and durations to be expressed on the activity level (rather than the activity type level). We denote $\Pi_{\mathcal{A}}$ the activity level precedence relations,

$$\Pi_{\mathcal{A}} = \{(a, a') \in \mathcal{A} \times \mathcal{A} \mid \nu(a) = \nu(a') = v \\ \wedge\ (\theta(a), \theta(a')) \in \Pi_v\}$$

and $d_{\mathcal{A}}(a, r)$ the activity-resource durations,

$$d_{\mathcal{A}}(a, r) = d(\theta(a), r), \forall (\theta(a), r) \in dom(d).$$

The function $d_{\mathcal{A}}$ is a partial function that is defined only for activity type and resource pairs $(\theta(a), r)$ that participate in d.

A schedule s, which satisfies a BSP, is an allocation of resource sets to activities over time, i.e., $s \in \mathcal{A} \to (\mathcal{R} \times \mathbb{R}^+)$. A feasible schedule respects resource and temporal constraints. One often considers an objective function $\phi(s)$ and the goal in optimal scheduling is to find a schedule s^* that optimizes $\phi(s)$. In this work, we do not learn the objective function and hence assume that ϕ is given.

The BSP generalizes well-known scheduling problems including the job-shop scheduling problem (JSP). For example, we extend the JSP by allowing alternative resources and durations that depend on both activity types and resources.

Mapping BSPs to CP Models

In this section we propose a generic CP model capable of modeling and solving BSP instances (i.e., the BSP2CP step in Figure 1). We employ a common formalism for modeling multi-machine scheduling problems using optional activities and alternative resources (Laborie 2009; Laborie et al. 2018). The proposed model allows us to represent instances of the BSP in CP and solve them using off-the-shelf solvers.

We use optional interval variables to efficiently represent activities. Formally, an optional interval variable has possible values within a convex interval: $\{\perp\} \cup \{[s, e) | s, e \in \mathtt{Z}, s \leq e\}$, where s and e are the start and end values of the interval and $\perp$ is a special value indicating the variable is not present in the solution. The presence, start time, and duration of an optional interval variable, var, can be expressed using $\mathtt{Pres}(var)$, $\mathtt{Start}(var)$, and $\mathtt{Length}(var)$. Absent interval variables (i.e., $\mathtt{Pres}(var) = 0$), do not participate in model constraints.

Activities. For each activity specified by the BSP instance, $a \in \mathcal{A}$, we create a mandatory (i.e., $\mathtt{Pres}(var) = 1$) interval variable, x_a, where $\mathtt{Start}(x_a) \geq 0$ and the duration is a variable. These interval variables represent resource-independent activities that are linked to resources below.

Precedence Relations. To enforce the precedence relations between the activities in the BSP, for each pair of activities $(a, a') \in \Pi_{\mathcal{A}}$, we post a constraint of the form: $\mathtt{EndBeforeStart}(x_a, x_{a'})$. This constraint ensures interval variable x_a ends before interval variable $x_{a'}$ starts.

Resource Assignment. For each activity, $a \in \mathcal{A}$, we create a set of optional interval variables, X_a, for the resources, $r \in \mathcal{R}$, with non-zero duration in $d_{\mathcal{A}}(a, r)$. Formally $X_a = \{\bar{x}_{ar} : d_{\mathcal{A}}(a, r) > 0, r \in \mathcal{R}\}$, where $\bar{x}_{ar}$ is an optional interval variable representing activity $a \in \mathcal{A}$ assigned to resource $r \in \mathcal{R}$ with duration $\mathtt{Length}(\bar{x}_{ar}) = d_{\mathcal{A}}(a, r)$. Each activity is assigned to exactly one resource and linked to the resource-independent activities with $\mathtt{Alternative}(x_a, X_a)$, enforcing that only one interval variable from the set X_a is present, and that it starts and ends together with mandatory interval variable x_a.

Resource Capacity. We model resource capacity with $\mathtt{Cumulative}(\{\bar{x}_{ar} : a \in \mathcal{A}\}, c(r)), \forall r \in \mathcal{R}$. The cumulative constraint expresses that at any time point, the total number of present and executing activity interval variables assigned to a resource is bounded by the capacity of that resource, $c(r)$.

Objective Function. Although in this work we do not learn the objective function, we conduct experiments minimizing the makespan, C_{max}, and minimizing the sum of completion times, $\sum_{a \in \mathcal{A}}(\mathtt{Start}(x_a) + \mathtt{Length}(x_a))$. The makespan is linked to the rest of the model with the constraint: $C_{max} \geq \mathtt{Start}(x_a) + \mathtt{Length}(x_a), \forall a \in \mathcal{A}$.

Activity-Resource Petri Nets

Our approach requires a process mining algorithm that learns a timed Petri net. However, deriving a schedule from a timed Petri net model is *undecidable* (Popova-Zeugmann 2013). Therefore, without making further assumptions on its structure, a TPN cannot be translated into a BSP. To bridge this expressiveness gap, we define the activity-resource Petri net (ARPN), a novel sub-type of TPN that enables the translation of learned TPNs into BSPs.

Defining ARPNs

To define ARPNs we use two building blocks:

- *Seize-delay-release constructs* (SDRCs) are TPNs that consist of three nodes (two transitions and a single place) and two flows. The nodes are (1) *seize*, an immediate transition

that seizes tokens; (2) *delay*, a place where the token is delayed; and (3) *release*, a timed transition that releases the token after a delay.
- *Activity Petri nets* (APNs) are TPNs that consist of seize-delay release constructs. APNs are feed-forward acyclic timed Petri nets that model job variants: sets of partially ordered activities.

The light gray parts of Figure 2 correspond to three SDRCs. For example, the SDRC that directly follows the start place in Figure 2 contains an immediate transition, which seizes the token from the start place. Then, the token is delayed according to the duration of an Exam. Subsequently, Exam releases the token to wait for the two Infusion SDRCs. Note that the two following SDRCs comprise a single activity type Infusion with two execution modes with potentially different durations: an infusion with a nurse practitioner (NP) and an infusion with a registered nurse (RN).

Definition 3 (Seize-Delay-Release Construct (SDRC))**.** *An SDRC is a timed Petri net, $\mathcal{S} = \langle E, E', P, F, \tau \rangle$, such that*

- *The set $E = \{e_{seize}, e_{release}\}$ contains two transitions (seize and release),*
- *The set $E' = \{e_{release}\}$ is the timed delay transition,*
- *The set $P = \{p_{delay}\}$ is a single delay place, and,*
- *The flows are $F = \{(e_{seize}, p_{delay}), (p_{delay}, e_{release})\}$.*

Given an SDRC, $\mathcal{S}$, we denote by $E_{\mathcal{S}}$ (and $E'_{\mathcal{S}}$), $P_{\mathcal{S}}$, $F_{\mathcal{S}}$ its sets of transitions (and timed transitions), places and flows, respectively. Furthermore, the set of places that precede (follow) the immediate (timed) transition of $\mathcal{S}$ is denoted by $\bullet E_{\mathcal{S}}$ ($E_{\mathcal{S}}\bullet$), i.e.,

$$\bullet E_{\mathcal{S}} = \{p \in P \mid \forall e \in E_{\mathcal{S}} \setminus E'_{\mathcal{S}} : \; p \in \bullet e\}$$
$$E_{\mathcal{S}}\bullet = \{p \in P \mid \forall e' \in E'_{\mathcal{S}} : \; p \in e'\bullet\}.$$

The complexity of detecting all SDRCs in a general TPN is linear in the number of its nodes (places and transitions) since it involves traversing the immediate transitions of the TPN and verifying that the nodes that directly follow them adhere to Definition 3.

Let $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$ be a set of SDRCs. S can be partitioned into k sets denoted by $C = \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$, with each set $\mathcal{C}_j \subseteq S, j = 1, \ldots, k$ containing SDRCs that have the same input and output places, i.e.,

$$C = \{\mathcal{C} \subseteq S \mid \forall \mathcal{S}_i, \mathcal{S}_j \in \mathcal{C} : \bullet E_{\mathcal{S}_i} = \bullet E_{\mathcal{S}_j} \; \wedge \; E_{\mathcal{S}_i}\bullet = E_{\mathcal{S}_j}\bullet\}.$$

From the existence of S, we get that C always exists and it is unique for a given TPN. From the scheduling perspective, each $\mathcal{C} \in C$ corresponds to a single activity type with multiple execution modes. In Figure 2, the two SDRCs with delay transitions Infusion1 and Infusion2 correspond to two execution modes of an activity type Infusion.

Let $E_{\mathcal{C}_j}$ ($E'_{\mathcal{C}_j}$) be the set of transitions (timed transitions) of the SDRCs in $\mathcal{C}_j$. We are now ready to define the APN.

Definition 4 (Activity Petri net (APN))**.** *An APN is a timed Petri net, $\langle E, E', P, F, \tau \rangle$, which satisfies the following conditions:*

- *The set $\bigcup_{j=1}^{m} E_{\mathcal{S}_j}$ contains only transitions from the set of SDRCs, S,*
- *The set $P = \bigcup_{j=1}^{m} P_{\mathcal{S}_j} \cup P_c$ contains the delay places in S and a finite set of $k+1$ connector places $P_c = \{p_1, \ldots, p_{k+1}\}$ with $p_1, p_{k+1} \in P_c$ being unique source and sink places, respectively, and,*
- *The flow $F = \bigcup_{j=1}^{m} F_{\mathcal{S}_j} \cup F_c$ contains both the set of SDRC flows and a set F_c such that:*

$$F_c = \{(p, e) \in P_c \times E \setminus E' \mid p = p_j \wedge$$
$$\exists \mathcal{C}_j \in C(\; e \in E_{\mathcal{C}_j} \setminus E'_{\mathcal{C}_j})\} \cup$$
$$\{(e, p) \in E' \times P_c \mid \exists \mathcal{C}_j \in C(e \in E'_{\mathcal{C}_j} \; \wedge \; p = p_{j+1})\}.$$

An APN is shown in the blue area of Figure 2: it consists of three SDRCs partitioned into two sets: an SDRC that involves Exam and two SDRCs that correspond to Infusion (transitions Infusion1 and Infusion2 are part of the same set in C). The APN is the main building block of the ARPN. Specifically, an ARPN contains a set of APNs with the addition of resource places and resource flows that connect these APNs.

Definition 5 (Activity-Resource Petri nets (ARPN))**.** *Let $\{\mathcal{N}_j\}_{i=1}^{n}$ be a set of APNs with $\mathcal{N}_j = \langle E_j, E'_j, P_j, F_j, \tau_j \rangle$ and let $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$ be a set of SDRCs that compose these APNs.*

An ARPN is a timed Petri net, $\langle E, E', P, F, \tau \rangle$, such that

- *The set $E = \bigcup_{j=1}^{n} E_j$ contains only transitions from the set of APNs, N_{apn},*
- *The set $P = \bigcup_{j=1}^{m} P_j \cup P_r$ contains places from the APNs, N_{apn}, and a finite set of resource places P_r, and,*
- *The flow set $F = \bigcup_{i=1}^{m} F_i \cup F_r$ contains the flows from N_{apn}, and a set F_r such that:*

$$F_r = \{(x, y) \in (P_r \times E \setminus E') \cup (E' \times P_r) \mid \forall (x, y) \in F_r : \; Q(x, y, S)\}$$

with,

$$Q(x, y, S) =$$
$$((x \in P_r \wedge y \in E_{\mathcal{S}_i} \setminus E'_{\mathcal{S}_i} \Rightarrow \exists (e', x) \in F_r : \; e' \in E'_{\mathcal{S}_i})$$
$$\wedge \; (x \in E'_{\mathcal{S}_i} \wedge y \in P_r \Rightarrow \exists (y, e) \in F_r : \; e \in E_{\mathcal{S}_i} \setminus E'_{\mathcal{S}_i}).$$

The set of flows F_r allows for resource tokens to be consumed only by immediate transitions and produced only by timed transitions. The property $Q(x, y, S)$ ensures that if an immediate transition consumes a resource token while being part of SDRC, $\mathcal{S}_i$, then there must be a flow between the timed transition of $\mathcal{S}_i$ back to the same resource place. Similarly, if a timed transition of an SDRC produces a resource token, there must be a flow between the resource place and the immediate transition of the same SDRC. One can easily verify that Figure 2 is an ARPN having a single APN, three SDRCs and two resource places.

Verifying that a mined TPN is an ARPN is linear in the number of Petri net nodes and flows ($\mathcal{O}(|P|+|E|+|F|)$). The procedure involves the computation of the SDRC set, S. Then, resource places are detected by validating that $Q(x, y, S)$ holds and resource places and their corresponding flows are removed from the TPN. Verifying that the remaining n components are APNs is done by computing the partition set C, and checking that each of the n components adheres to the conditions of Definition 4.

Translating ARPNs to BSPs

If the TPN is not an ARPN, then we detect it and return a mismatch. Provided that the TPN is an ARPN we can compute the following sets:

1. The set of resource places P_r.
2. The set of SDRCs $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$, which correspond to the various execution modes per activity.
3. The set of n APNs $\{\mathcal{N}_i\}_{i=1}^{n}$ that comprise the ARPN, representing different job variants.
4. The partitions of SDRCs in $\mathcal{N}_i$, namely C_i.

We are now ready to provide the translation from an ARPN into a BSP. Since $\mathcal{A}$, θ and ν are external, the procedure below computes only the parameters of the BSP.

Definition 6 (ARPN to BSP)**.** *Given an ARPN and an initial marking, $(\mathcal{N} = (E, E', P, F, \tau), M_0)$, the BSP is created as follows:*

- *The set of activity types corresponds to the sets of SDRC partitions, namely $\mathcal{T} = \bigcup_{i=1}^{n} C_i$,*
- *The resource set is given by the set of places, $\mathcal{R} = P_r$,*
- *The set of job variants is given by,*

$$\mathcal{V} = \{(\mathcal{C}_1, \ldots, \mathcal{C}_{m_i}) \mid \exists i \in [n] : \mathcal{C}_{j-1} \rightsquigarrow_i \mathcal{C}_j, j = 2, \ldots m_i\},$$

  with $\rightsquigarrow_i$ indicating that the input place into $\mathcal{C}_j$ directly follows the output place of $\mathcal{C}_{j-1}$ in $\mathcal{N}_i$,
- *The precedence relation for all job variants, $\Pi_v, v \in \mathcal{V}$, is given by,*

$$\Pi_v = \{(t, t') \in \mathcal{T} \times \mathcal{T} \mid t \rightarrow_v t'\}$$

  with $t \rightarrow_v t'$ being true if t occurs before t' in the job variant v,
- *The resource capacities, c, are equal to the initial marking of resource places, i.e., $\forall r \in \mathcal{R} : c(r) = M_0(p_r)$, and,*
- *The duration (partial) function d is computed as follows:*

$$\begin{aligned} d(\cdot) = \{((t, r), \delta) \in \mathcal{T} \times \mathcal{R} \times \mathbb{R}^+ \mid \\ \forall \mathcal{S} \in S : (\forall e \in E_{\mathcal{S}} \setminus E'_{\mathcal{S}} (t \in {\bullet e} \ \wedge \ r \in ({\bullet e} \cap \mathcal{R})) \\ \wedge \forall e' \in E'_{\mathcal{S}} (\delta = \tau(e')))\}. \end{aligned}$$

The set of job variants corresponds to the n APNs that compose the ARPN. For each APN, $\mathcal{N}_i, i \in [n]$, we construct a job variant as the sequence of the SDRC sets within C_i. The precedence relation is derived from the job variants, since the latter define a total order among activity types. Therefore, if a pair of activity types (t, t') is ordered in job variant v such that t occurs before t', the pair of activity types will appear in the corresponding precedence relation of job variant v, namely $(t, t') \in \Pi_v$. To compute durations, we iterate over all SDRCs of the ARPN and retrieve the duration values $\tau(e')$ of their corresponding timed transitions $E'_{\mathcal{S}}$. We assign a value $d(a, r)$ for every pair of activity type (given by set C) and resource that is connected to the SDRCs of that activity type in the ARPN.

Returning to the ARPN in Figure 2, we assume that the Exam by an NP has a duration of 20 minutes, while Infusion has a duration of 15 minutes when performed by an NP and 10 minutes when performed by an RN. When applying the translation in Definition 6, a BSP is created as follows: (1) the activity type set $\mathcal{T} = \{t_1, t_2\}$, that comprises the Exam and Infusion SDRC partition sets, (2) two resources, $\mathcal{R} = \{r_1, r_2\}$, derived from the NP and RN places, (3) a single job variant, $v_1 = \langle t_1, t_2 \rangle$ will be created in $\mathcal{V}$, (4) a precedence relation $\Pi_{v_1} = \{(t_1, t_2)\}$ (i.e., Exam type comes before Infusion type in variant v_1), (5) resource capacities are set to 1 ($c(R_1) = c(R_2) = 1$) and (6) durations are assigned: $d(t_1, r_1) = 20, d(t_2, r_1) = 15, d(A_2, r_2) = 10$.

Evaluation

In this section, we apply our log-to-model methodology to two types of event logs: (1) simulated event logs generated using publicly available JSP benchmarks and (2) real-world event logs from an outpatient cancer hospital in the United States (denoted *DayHospital*).

JSP Benchmarks. For the first experiment, we simulate event logs using 53 publicly available JSP benchmark instances, learn their parameterized models, and solve them.[3] Instance files contain the jobs to be scheduled, the resource requirements, durations, and required order (precedence constraints) of each of the activities. We assume that every job in the instance file is a single job variant and every activity in the job is a unique activity type. For example, consider a JSP input file that contains 20 jobs and 15 machines (300 activities). To create our logs, we assume that the job shop has 20 job variants with 300 *types* of activities.

We simulate each job variant a uniform number of times (between 100 and 200) to obtain the event log. During the simulation, we schedule the jobs using a randomized dispatch policy that adheres to precedence and capacity constraints. Returning to the example of an instance with 20 jobs over 15 machines, we produce an event log that contains approximately 3000 jobs, which corresponds to 45000 events. The event logs from the 53 instances vary with the number of events (15000 and 45000 events per log) depending on the number of jobs and machines per instance.

For each of the produced event logs, we learn the TPN, verify that it is an ARPN and create the parameter set of the corresponding BSP. Next, for each of the benchmarks, we consider the original instance file, and add its activities with their job variants and activity types to the BSP. This results in a complete BSP representation. Lastly, we map the BSP to a CP model and solve it. These experiments are conducted with a makespan minimization objective function.

Results. Mining the ARPN and mapping it into a BSP for each JSP instance is not very time consuming, taking, on average, 0.73 seconds. CP model experiments are implemented in CP Optimizer from the IBM ILOG CPLEX Optimization Studio version 12.8. We use a 10-minute time limit for the branch-and-infer search. Our CP model then solves $49/53$ of the JSP benchmark instances to proven optimality. Feasible solutions were found for all instances. The average runtime for the

[3] Instances retrieved from: https://github.com/Thiebout/JobShopScheduling/tree/master/testinstances

instances proved optimal was 15.4 seconds and the average optimality gap for the instances not solved to optimality was 6.3%.

DayHospital. In the second experiment, we examine the applicability of the model learning methodology to real-world problems. We use four months of event data from DayHospital (January-April, 2016). The acquired dataset comprises two types of information: (1) appointment book records, which provide the activities that were scheduled for a given day (including their types and resources) and (2) events that came from a real-time locating system that continuously tracks patients and health providers, which provided the timestamps of activity starts and completions. Matching these two information sources creates an event log, an excerpt of which was shown in Table 1.

With a volume of roughly 1000 patients per day, a month of log data (19 working days/month) contains approximately 30000 executed activities (events). Each activity is attributed to one out of 60 activity types and one of 240 job variants. Furthermore, the data contains 31 organizational roles (e.g., MD, RN) belonging to 80 departments (e.g., Thoracic Oncology, Gastrointestinal Oncology). These attributes can be used to identify resources, however, since the pooling of resources in DayHospital is based on departments and not roles (i.e., two MDs from different departments are not pooled for the same activities), we chose departments as our resource attribute. Department capacities range between 1 (Geriatrics) and 36 (Floor 7 infusion unit). Note that the capacity may either be dominated by roles (e.g., number of nurses) or by other resources, such as the number of rooms, number of ECG monitors, etc. Activity durations vary between 5 minutes (Blood Draw) and 10 hours (Infusion).

The first three months of data (January-March) serve as our training data, which we use for learning the BSP. Individual days from April are used as test data, providing new appointments to be scheduled. Specifically, we consider activity types that were scheduled during April to be the fresh set of activities that are fed into the learned BSP. We used the CP model that corresponds to the learned BSP to schedule these new appointments with respect to two optimality criteria, namely minimizing makespan (the objective employed by the current DayHospital scheduling system) and sum of completion times.

Results. Learning the ARPN and mapping it to a BSP takes from 200 to 450 seconds, depending on the number of events in the training set. From the BSP, we then apply our CP model, investigating both minimizing schedule makespan and sum of completion times. Our CP model uses the same experimental set-up used for the JSP experiments.

For makespan minimization, CP is able to solve all of the problem instances to proven optimality in less than one second. Upon closer inspection, the presence of a number of very long activities (e.g., chemotherapy infusions) in each instance allows the solver to find the optimal solution with little effort because the long activities determine the makespan and the solver infers further search cannot reduce it.

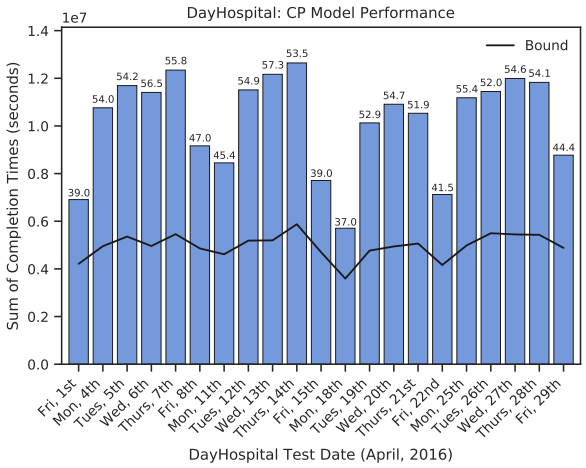

Figure 3: Real-world DayHospital experiments. Objective values (bars), lower bounds (line), and optimality gap (top of bars in %) for sum of completion times objective function (in 1×10^7 seconds).

To produce schedules that intelligently sequence the shorter activities, we also investigate minimizing the sum of activity completion times as an objective function. This problem is much more difficult for the solver, as the placement of each activity has direct impact on the objective value. Our CP model is able to find feasible solutions to all 21 instances, as illustrated in Figure 3, but with an average optimality gap of 50.3%, as measured against the lower bound determined by the CP solver.

Discussion & Limitations

As far as we are aware, our work represents the first system that can automatically learn parameterized scheduling models from log data and apply them to new instances in the same context (see the next section on related work). Nonetheless, there are a number of limitations in our work.

First, we do not necessarily learn a *good* CP model. A given problem has numerous possible CP models with differing computational performance (Smith 2006) and we rely on a single mapping from BSP to a CP scheduling model. While such models exhibit very strong (often, state-of-the-art) performance (Laborie 2018), the question of learning good models is not in the scope of this paper.

Secondly, because we assume unsupervised learning, we have no information on the quality of the schedules executed in the log or the objective function that is appropriate for our learned model. Future research may be able to adapt hypotheses testing work in process mining (Senderovich et al. 2016) to evaluate the likelihood that a particular log was produced to optimize different objective functions from a specified set of possibilities. However, in the current system, the objective function must be provided externally.

The missing objective function is a special case of the limitations of *SchedMiner*: we can only learn what is in the log. Unseen activity types or resources will not be learned,

nor will capacities for resources that have not been used fully at some point in the log. Furthermore, the events and resources named in the log determine what we can learn as activities and resources. However, it should be noted that we do not need the logs to reflect a good schedule (by whatever metric is used) or to even be the result of an optimization. Any execution of the processes in the log, provided they satisfy the constraints of the context, can be used to learn a CP model. And that model, given the desired objective function and enough time, can be solved to optimality.

In terms of generality, there is no need to restrict the optimization technology to CP. Other model-based approaches, such as mixed integer linear programming, or even non-model-based approaches, such as metaheuristics, can be used to solve the resulting scheduling problems provided the mapping can be made from the BSP.

Finally, our approach can only learn problems expressible as BSPs. While *SchedMiner* can learn more general TPNs, our mapping to BSPs will (gracefully) fail if the mined TPN is not an ARPN. We chose to use TPNs as an intermediate knowledge representation rather than directly learning a BSP in order to exploit the richer expressivity of the TPNs in the future. We plan to expand the types of scheduling problems that can be recognized and the optimization methods that can be targeted. For example, in a context with substantial duration uncertainty, we would like to learn stochastic Petri nets that can then be mapped to solvers based on queueing theory or stochastic scheduling. We can then begin to investigate automatically learning (i.e., based on the characteristics of the learned TPN) which types of scheduling problems and solution approaches are appropriate for which problem characteristics. TPNs are also expressive enough to represent planning problems and we are considering whether our framework can be used to learn planning models from execution logs (e.g., in the context of teleoperated systems that can also be run autonomously).

Related Work

Scheduling with Timed Petri nets. Research on modeling scheduling problems using TPNs dates back to 1980s (Carlier, Chretienne, and Girault 1985; van der Aalst 1996; Zuberek and Kubiak 1999). The majority of these works map basic scheduling elements, such as precedence constraints and resources with limited capacities, onto Petri net constructs. In our work, we provide a methodology that solves the inverse problem: given a (learned) TPN, we translate it into a parameterized scheduling model.

TPNs were shown to provide an inefficient platform for solving scheduling problems, since they require a global search over the entire state-space (Lee and DiCesare 1994). Due to their ineffectiveness, Petri nets were either used for inference (e.g., finding redundant constraints and providing lower and upper bounds on makespan) or solved by heuristic methods (Silva and Valette 1988; Lee and DiCesare 1994). In our work, we avoid solving the scheduling problem in its Petri net representation. Instead, we map the scheduling problem into a CP model for which effective solvers exist (Baptiste, Le Pape, and Nuijten 2001).

Learning CP Models. Freuder (2018) defines automatic model acquisition to be part of the Holy Grail of Constraint Programming. Unsurprisingly, learning CP models from data has been studied in recent literature (see (Raedt et al. 2016) and references within). Existing methods aim at finding a set of constraints that yield the most accurate classification of assignments in the data into feasible and infeasible. In Lallouet et al. (2010), the authors combine an inductive logic programming technique for learning from labeled data with background knowledge on the structure of the problem to learn a representation of the problem that correctly classifies solutions. Furthermore, the work by Bessiere et al. (2017) suggests acquiring constraints from examples classified by the user. These approaches require negative examples, i.e., assignments that do not satisfy the problem, which may be difficult to come across. Beldiceanu and Simonis (2012) propose *ModelSeeker*, a method for learning global constraints for CP using well-structured data. Our approach differs from *ModelSeeker* in that we learn parameterized models i.e., we do not assume the problem parameters (e.g., the size of the JSP problem) in advance.

Learning Action Planning Models. Our work relates to an existing body of work on learning action planning models using observed past plans (e.g., Zhuo and Kambhampati (2017), Aineto, Jiménez, and Onaindia (2018), and references within). The most closely related work on learning of planning models provides a synthesis of learned transition systems into action schemata, which are representations of the preconditions, effects and parameters of a given action (Cresswell, McCluskey, and West 2009). Instead of transition systems, we learn timed Petri nets, a formalism that better captures the various scheduling elements.

Conclusion

We presented a novel approach for learning parameterized scheduling models from event data that combines process learning, timed Petri nets, parameterized scheduling models, and constraint programming. To bridge the expressiveness gap between timed Petri nets and scheduling problems we proposed a novel sub-type of timed Petri nets, namely activity-resource Petri nets, that enabled us to translate from Petri nets into basic scheduling problems. The applicability of our end-to-end (data-to-CP-model) approach was demonstrated in an empirical evaluation. Our experiments have shown that the proposed method effectively learns parameterized scheduling models from event data. Specifically, the learning method accurately reconstructed and solved (to optimality) a set job shop scheduling problems. Additionally, using real-world event logs, the approach provided an appointment scheduling solution based solely on the learned CP model. Future work involves extending our approach in several directions. First, we plan to extend our TPN recognition algorithm beyond BSPs to include aspects of deterministic scheduling such as non-unit resource requirements and general temporal constraints. Second, we wish to generalize the Petri net framework to stochastic Petri nets, allowing us to learn and solve queueing scheduling problems.

References

Aineto, D.; Jiménez, S.; and Onaindia, E. 2018. Learning STRIPS action models with classical planning. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M. T. J., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018.*, 399–407. AAAI Press.

Baptiste, P.; Le Pape, C.; and Nuijten, W. 2001. *Constraint-Based Scheduling*. Norwell, MA, USA: Kluwer Academic Publishers.

Beldiceanu, N., and Simonis, H. 2012. A model seeker: Extracting global constraint models from positive examples. In *Principles and practice of constraint programming*, 141–157. Springer.

Bessiere, C.; Koriche, F.; Lazaar, N.; and O'Sullivan, B. 2017. Constraint acquisition. *Artificial Intelligence* 244:315 – 342. Combining Constraint Solving with Mining and Learning.

Carlier, J.; Chretienne, P.; and Girault, C. 1985. Modelling scheduling problems with timed petri nets. In *Advances in Petri Nets 1984*. Springer. 62–82.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of object-centred domain models from planning examples. In Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI.

David, R., and Alla, H. 1994. Petri nets for modeling of dynamic systems: A survey. *Automatica* 30(2):175–202.

Freuder, E. C. 2018. Progress towards the holy grail. *Constraints* 23(2):158–171.

Laborie, P.; Rogerie, J.; Shaw, P.; and Vilím, P. 2018. IBM ILOG CP Optimizer for scheduling. *Constraints* 23(2):210–250.

Laborie, P. 2009. IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 148–162. Springer.

Laborie, P. 2018. An update on the comparison of mip, cp and hybrid approaches for mixed resource allocation and scheduling. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 403–411. Springer.

Lallouet, A.; Lopez, M.; Martin, L.; and Vrain, C. 2010. On learning constraint problems. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, 45–52. IEEE.

Lee, D. Y., and DiCesare, F. 1994. Scheduling flexible manufacturing systems using petri nets and heuristic search. *IEEE Transactions on robotics and automation* 10(2):123–132.

Leemans, S. J.; Fahland, D.; and van der Aalst, W. M. 2014. Discovering block-structured process models from event logs containing infrequent behaviour. In *Business Process Management Workshops*, 66–78. Springer.

Popova-Zeugmann, L. 2013. Time petri nets. In *Time and Petri nets*. Springer. 31–137.

Raedt, L. D.; Dries, A.; Guns, T.; and Bessiere, C. 2016. Learning constraint satisfaction problems: An ILP perspective. In Bessiere, C.; Raedt, L. D.; Kotthoff, L.; Nijssen, S.; O'Sullivan, B.; and Pedreschi, D., eds., *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, volume 10101 of *Lecture Notes in Computer Science*. Springer. 96–112.

Rozinat, A., and Van der Aalst, W. M. 2005. Conformance testing: Measuring the fit and appropriateness of event logs and process models. In *International Conference on Business Process Management*, 163–176. Springer.

Senderovich, A.; Rogge-Solti, A.; Gal, A.; Mendling, J.; Mandelbaum, A.; Kadish, S.; and Bunnell, C. A. 2015. Data-driven performance analysis of scheduled processes. In Motahari-Nezhad, H. R.; Recker, J.; and Weidlich, M., eds., *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, volume 9253 of *Lecture Notes in Computer Science*, 35–52. Springer.

Senderovich, A.; Weidlich, M.; Yedidsion, L.; Gal, A.; Mandelbaum, A.; Kadish, S.; and Bunnell, C. A. 2016. Conformance checking and performance improvement in scheduled processes: A queueing-network perspective. *Information Systems* 62:185–206.

Silva, J. R., and del Foyo, P. M. 2012. Timed petri nets. In *Petri Nets-Manufacturing and Computer Science*. InTech.

Silva, M., and Valette, R. 1988. Petri nets and flexible manufacturing. In *European Workshop on Applications and Theory in Petri Nets*, 374–417. Springer.

Smith, B. 2006. Modelling. In Rossi, F.; van Beek, P.; and Walsh, T., eds., *Handbook of Constraint Programming*. Elsevier. chapter 11, 377–406.

van der Aalst, W. M.; Rubin, V.; van Dongen, B. F.; Kindler, E.; and Günther, C. W. 2006. Process mining: A two-step approach using transition systems and regions. *BPM Center Report BPM-06-30, BPMcenter. org* 6.

van der Aalst, W. M. P.; Weijters, T.; and Maruster, L. 2004. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Trans. Knowl. Data Eng.* 16(9):1128–1142.

van der Aalst, W. 1996. Petri net based scheduling. *Operations-Research-Spektrum* 18(4):219–229.

van der Aalst, W. M. P. 2011. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer.

Zhuo, H. H., and Kambhampati, S. 2017. Model-lite planning: Case-based vs. model-based approaches. *Artif. Intell.* 246:1–21.

Zuberek, W., and Kubiak, W. 1999. Timed petri nets in modeling and analysis of simple schedules for manufacturing cells. *Computers & Mathematics with Applications* 37(11-12):191–206.