Learning Scheduling Models from Event Data

removed for review

Abstract

Solving scheduling problems requires two main ingredients: a model that captures the essence of the underlying system and an algorithm that provides optimal solutions based on the model. Constraint programming (CP) is a well-established paradigm for modeling and solving deterministic scheduling problems. It has been well-recognized that creating a suitable CP model is knowledge intensive task even when the underlying system is well-understood. In this work, we aim at automating the process of modeling scheduling problems by learning CP models from event data. To this end, we introduce a novel methodology that combines process mining, timed Petri nets and CP. As a first step, the approach mines timed Petri nets (TPNs) from event logs that contain executions of past schedules including information on activities, timestamps and resources. For the second step, we define a specialized TPN type, namely the seize-delay-release net with resources (SDRR net), which can be mapped into CP models. We provide a correct and complete algorithm for detecting whether a TPN is an SDRR net. Furthermore, we show a mapping from SDRR nets into CP models. Our approach provides an end-to-end solution to model learning, going from event logs to model-based optimal schedules without human intervention. To demonstrate the usefulness of the methodology we conduct a series of experiments in which we learn scheduling models from two types of data: (1) event logs generated from job-shop scheduling benchmarks and (2) real-world event logs that come from an outpatient hospital.

1 Introduction

Modern scheduling algorithms successfully solve problems with thousands of activities, complex temporal constraints, and scarce resources. The main two prerequisites to solving scheduling problems are (1) a mathematical model that describes the system and (2) an algorithm that solves the problem based on the aforementioned model.

In deterministic scheduling, Constraint Programming (CP) has been shown to be extremely effective in solving a wide range of problems (Baptiste, Pape, and Nuijten 2001, Chapter 6), (Lallouet et al. 2010). However, creating CP models often requires a high level of expertise, as one must specify suitable variables, constraints and objective functions that correspond to the underlying real-world problem (Lallouet et al. 2010).

In this work, we address the challenge of automating the process of modeling scheduling problems by learning CP models from data. Previous works that learn CP models assume that the data is tightly coupled with the CP formulation. For example, when learning from positive examples only, the data at hand must contain variables as they appear in CP models (Beldiceanu and Simonis 2012). In other works, it is assumed that the data must include both positive and negative labels for assignments that satisfy or violate constraints, respectively (Lallouet et al. 2010).

In practice, for many real-world problems existing data recordings do not relate to the CP formalism (e.g., does not include variables). Instead, the data comes in the form of event logs, which are logs that contain the activities that were executed, along with their start and completion times, and information on resource consumptions (van der Aalst 2011, Chapter 4). In this work, we address the challenge of learning scheduling models automatically from event logs without humans intervention. Figure 1 presents an overview of our approach. As our first step, we use existing techniques from process mining, a rapidly developing research field that aims at learning models of the underlying process from event logs (van der Aalst 2011). Specifically, we apply a model learning method that considers scheduled processes and their execution data (Senderovich et al. 2015a). The result of the process mining step is a timed Petri net (TPN), a well-known formalism for modeling dynamic schedule-driven systems (Van der Aalst 1996). Since the TPN is highly expressive and captures synchronization, resource consumption, and temporal constraints, it cannot be used efficiently for scheduling the system that it models (Van der Aalst 1996). Therefore, in the next step we transform the learned TPN into a CP model. However, due to differences in the expressive power of the two formalisms (TPN and CP), we must restrict the TPN to a family of models that can be transformed into CP formulations. To this end, we define a new type of

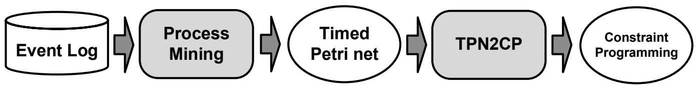

Figure 1: Our solution to learning scheduling models.

TPNs, namely Seize-Delay-Release nets with resources (SDRR nets). We show a correct and complete algorithm that given a TPN verifies that it is an SDRR net. Subsequently, if the TPN is an SDRR net, we provide a correct and complete mapping of the SDRR net into a CP model, thus completing our solution to learning CP models from event data.

The main contribution of our work is threefold:

- We provide an data-to-model solution that learns scheduling models from event logs.
- We introduce a specialized type of timed Petri nets, namely the SDRR nets, which enables a TPN to CP transformation.
- We show a correct and complete algorithm for verifying that a TPN is an SDRR net and provide a mapping of SDRR nets into CP models.

To demonstrate the usefulness of our approach, we provide a two-part experimental evaluation. In the first part, we learn models of job-shop scheduling benchmarks using synthetically generated data. In the second part of the evaluation, we learn scheduling models from real-world data coming from a large outpatient cancel hospital in the United States. Given the data, our algorithms generate the underlying appointment scheduling problem that we consequently solve using constraint programming.

2 Background

In this section we present the preliminaries to our approach. Firstly, we define timed Petri nets, a special case of stochastic Petri nets as defined in (Haas 2002, Chapter 2), having deterministic activity durations and routing. Secondly, we define our data model in the form of events logs and briefly outline existing approaches for learning TPN models from event logs. We conclude the section with a definition of a general family of scheduling problems, namely *basic scheduling problems* (BSP).

2.1 Timed Petri Nets

Petri nets are procedural models for analyzing discrete-event dynamic systems that exhibit parallelism, synchronization and resource consumption. Formally, a timed Petri net (TPN) is defined as follows:

Definition 1 (Timed Petri net (TPN); TPN System). *A timed petri net $\mathcal{N}$ is a 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,*
- *P being a finite set of places,*
- *$F \subseteq E \times P \cup P \times E$ being the flow relation of the Petri net, and,*
- *$\tau \in (E' \rightarrow \mathcal{T})$ being a function that maps deterministic durations to timed transitions;*

A TPN is fully characterized as a pair $(\mathcal{N}, m_0)$ with $\mathcal{N}$ being the net and $m_0 : P \rightarrow \mathbb{N}$ being an initial marking of the net, which is a function that maps each place to the number of tokens that it contains. We use Figure X to demonstrate a TPN and its dynamic behavior. In the figure, we observe two jobs waiting in the queue place.... **Insert simple (not too simple) TPN example here to explain the dynamics** We denote $\bullet p$ ($\bullet e$) the set of transitions (places) that precede a place p (a transition e), and by $p\bullet$ the set of transitions (places) that succeed a place p (a transition e). Furthermore, we define F_P to be the set of incoming and outgoing flows the corresponds to a set of places P, i.e., $F_P = \{(x, y) \in F \mid x \in P \ \vee \ y \in P\}$.

For example, in Figure X $\bullet p_3$... **complete here**

TPNs were previously applied to model scheduling problems (Van der Aalst 1996; Lee and DiCesare 1994). However, they provide an inefficient solution platform for solving scheduling problems, since TPNs require a global search over the entire state-space of the underlying problem (Lee and DiCesare 1994). Therefore, previous literature on scheduling with Petri nets mostly focuses on heuristic solutions (Lee and DiCesare 1994).

2.2 Mining Timed Petri nets from Event Logs

Process mining is a rapidly evolving research field that is centered around developing methodologies for learning models from data (van der Aalst 2011). The assumption is that the execution of processes is recorded into event logs, which can in turn be employed to learn models of the underlying system (e.g., in the form of a Petri net). The learning task is typically assumed to be unsupervised, since the learned model cannot be validated against ground truth.

To define process learning, we must first give an overview of our data model, namely the event log. Let $\mathcal{E}$ be the universe of events with $e \in \mathcal{E}$ having attributes $e.j$ for job identifier, $e.s$ for start of operation timestamp, $e.c$ for completion of operation timestamp, $e.R$ for resource and $e.A$ for event label (e.g., operation 3 start, operation 5 complete, operation 7 start). An event log L is a subset of $\mathcal{E}$. Figure Y presents an excerpt from a real-world event log of an outpatient cancer hospital. Furthermore, we let $\psi(L, M) \in [0, 1]$ be a learning quality function that given an event log L and a TPN

M evaluates the model with 0 (1) indicating low (high) quality model with respect to the log.

The task of a process learning function γ that maps an event log L onto a Petri net model M (in our case a TPN) such that $\psi(L, \gamma(L))$ is minimized (van der Aalst 2011). The measure ψ measures the distance between model and log indirectly, since we do not assume to have labeled pairs of logs and models. Many process learning algorithms were proposed in the past. See Chapter X in (van der Aalst 2011) for a survey.

In this work, we learn TPNs using the approach developed for scheduled processes (Senderovich et al. 2015b). Specifically, the method in (Senderovich et al. 2015b) learns the various components of the TPN, while guaranteeing maximal quality value, assuming that the event log was generated by a process that followed a pre-defined schedule. **Should we add more details here? no options between operations and loops are allowed**

3 Schedule-Driven Petri Nets

In this section, we provide our approach to solving the schedule learning problem. We assume that a TPN is learned from event data using existing approaches, e.g., via the approach presented in (Senderovich et al. 2015b). **we actually make a modification that enables that some of the activities can be performed by alternating sets of resources**.

3.1 Seize-Delay-Release Nets

The transformation of a learned TPN into a BSP, is based on the notion of Seize-Delay-Release nets with resources (SDRR nets).

To define SDRR nets, we first consider seize-delay-release constructs (SDCs), which are TPNs that consist of the following three components: (1) two transitions: one immediate transition that seizes a token and one timed transition that releases the token after a delay, (2) a place where the token is delayed, and (3) two flows that connect the two transitions to the delay place. Formally,

Definition 2 (Seize-Delay Construct (SDC))**.** *An SDC is a timed Petri net, $\mathcal{S} = \langle E, E', P, F, \tau \rangle)\rangle$, such that*

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

The gray parts in Figure 2 are the three SDC components of the TPN. The SDC that directly follows the start place in Figure 2 contains an immediate transition, which seizes the token in the start place. Then, the token is delayed for $\tau(e_{release})$, which corresponds to the duration of the 'Exam' transition. Lastly, 'Exam' releases the token into the subsequent place.

Given an SDC, $\mathcal{S}$, we denote $E_{\mathcal{S}}$ (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.,

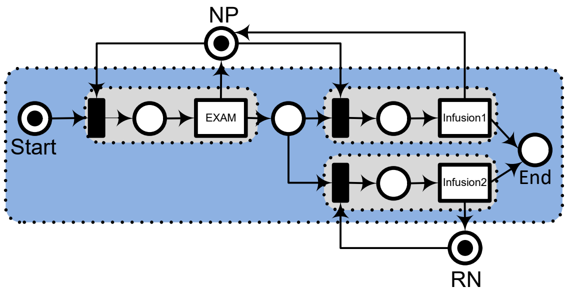

Figure 2: An SDRR Net of a Hospital Process.

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

Detecting the SDCs of a given TPN is linear in the number of transitions ($\mathcal{O}(|E| + |P|)$), since it involves traversing over the immediate transitions of the TPN and verifying that the transition is followed by a single place, which is in turn followed by a single timed transition (see Definition 2).

A TPN that comprises a set of SDCs is referred to as a seize-delay-release net (SDR net). To define an SDR net, we let $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$ be a set of SDCs. The set S can be partitioned into k sets denoted by $C = \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$, with each set $\mathcal{C}_j, j = 1, \ldots, k$ containing SDCs that have equal input and output places. It is easy to show that C always exists and that it is unique for a given TPN. To simplify notation, we denote $E_{\mathcal{C}_j}$ ($E'_{\mathcal{C}_j}$) the transitions (timed transitions) of the SDCs that are elements of $\mathcal{C}_j$. We are now ready to define seize-delay-release nets.

Definition 3 (Seize-Delay-Release Net)**.** *A seize-delay-release net is a timed Petri net, $\mathcal{N}_{sdr} = (E, E', P, F, \tau)$, which satisfies the following conditions:*

- *The set $\bigcup_{j=1}^{m} E_{\mathcal{S}_j}$ contains only transitions from the set of SDCs ($E' \subseteq E$ being the set of timed transitions),*
- *The set $P = \bigcup_{j=1}^{m} P_{\mathcal{S}_j} \cup P_c$ contains both the delay places of N 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 SDC flows and a set F_c such that*

$$\begin{aligned} F_c = &\{(p, e) \in P_c \times E \setminus E' \mid && (1)\\ &p = p_j \ \ \wedge \exists \mathcal{C}_j \in C(\ e \in E_{\mathcal{C}_j} \setminus E'_{\mathcal{C}_j})\} \cup \\ &\{(e, p) \in E' \times P_c \mid && (2)\\ &\exists \mathcal{C}_j \in C(e \in E'_{\mathcal{C}_j} \ \ \wedge \ \ p = p_{j+1})\}. && (3) \end{aligned}$$

An example for an SDR net can be found in Figure 2. The blue area corresponds to an SDR net that consists of three SDCs partitioned into two sets, namely an SDC that involves 'Exam' and two SDCs that correspond to Infusion (transitions Infusion1 and Infusion2 are part of the same set in the partition of SDCs, C). Furthermore, the place of connectors P_c contains three places: start place, end place and a place that connects 'Exam' to the two 'Infusion' SDCs. SDR nets are important building blocks of the Seize-Delay-Release nets with resources (SDRR nets), which we map into basic scheduling problems. Algorithm 1 verifies that a TPN is an SDR net.

Algorithm 1: Verifies that TPN is SDR net.

```
Input: Timed Petri net $\mathcal{N} = \langle E, E', P, F, \tau \rangle$
Output: $\langle \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}, P_c \rangle$ - Set of SDCs and
        set of connector places $P_c$
1  begin
2      $P_{in} \leftarrow \{p \in P \mid \bullet p = \emptyset\}$
3      $P_{out} \leftarrow \{p \in P \mid p\bullet = \emptyset\}$
4      if $|P_{in}| \neq 1 \ \vee |P_{out}| \neq 1$ then
5          return False
6      $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\} \leftarrow DetectSDC(\mathcal{N})$
7      if $\bigcup_{j=1}^{m} E_{\mathcal{S}_j} \neq E$ then
8          return False
9      $C \leftarrow \{\mathcal{C} \subseteq S \mid \forall \mathcal{S}_i, \mathcal{S}_j \in \mathcal{C} :$
         $\bullet E_{\mathcal{S}_i} = \bullet E_{\mathcal{S}_j} \ \wedge \ E_{\mathcal{S}_i}\bullet = E_{\mathcal{S}_j}\bullet\}$
10     foreach $\mathcal{C}_j \in C$ do
11         $\bullet\mathcal{C}_j = \bigcup_{\mathcal{S} \in \mathcal{C}_j} \bullet E_{\mathcal{S}}$
12         $\mathcal{C}_j\bullet = \bigcup_{\mathcal{S} \in \mathcal{C}_j} E_{\mathcal{S}}\bullet$
13         if $(\bullet\mathcal{C}_j = \mathcal{C}_j\bullet \ \vee \ |\bullet\mathcal{C}_j| \neq 1 \ \vee \ |\mathcal{C}_j\bullet| \neq 1$
            then
14             return False
15     end
16     if $\exists \mathcal{C}_i \neq \mathcal{C}_j \in C \ (\mathcal{C}_i\bullet = \mathcal{C}_j\bullet \ \vee \ \bullet\mathcal{C}_i = \bullet\mathcal{C}_j))$
         then
17         return False
18     $P_c \leftarrow \bigcup_{\mathcal{C}_j \in C}(\bullet\mathcal{C}_j \cup \mathcal{C}_j\bullet)$
19     $F_c \leftarrow \bigcup_{\mathcal{C}_j \in C}\{(x, y) \in F \mid (x \in \bullet\mathcal{C}_j \ \wedge \ y \in$
         $E_{\mathcal{C}_j} \setminus E'_{\mathcal{C}_j}) \ \vee \ (x \in E'_{\mathcal{C}_j} \ \wedge \ y \in \mathcal{C}_i\bullet)\}$
20     if $P_c \neq P \setminus \bigcup_{j=1}^{m} P_{\mathcal{S}_j} \ \vee \ F_c \neq F \setminus \bigcup_{j=1}^{m} F_{\mathcal{S}_j}$
         then
21         return False
22     return True
23 end
```

update explanation and proof

Below, we explain the algorithm line-by-line. Lines 2-5 verify that the TPN has unique input and output places. Line 6 returns the set of SDCs, S, that are present in the input TPN. Detecting SDCs is performed via the function $DetectSDC(\cdot)$, which uses a simple traversal over all immediate transitions $e \in E \setminus E'$ and checking whether the direct followers of e are a single place and a timed transition (as in Definition 2). Lines 7-8 make sure that the set of transitions E includes only transitions from S.

Line 9 partitions the SDCs into sets of SDCs, such that each set $\mathcal{C} \in C$ contains only SDCs that share input and output places.

Lines 10-15 traverse the sets in C and verify that each set $\mathcal{C}_j \in C$ has a unique input place and a unique output place ($|\bullet\mathcal{C}_j| = |\mathcal{C}_j\bullet| = 1$). Furthermore, Lines 10-13 ensure that the input and output places of $\mathcal{C}_j$ are not equal to each other (no loops), and that there does not exist an additional set $\mathcal{C}_i \neq \mathcal{C}_j$ for which one of its input or output places are equal to the corresponding input and output places of $\mathcal{C}_j$. Intuitively, this enforces a sequential execution of the sets of SDCs in C. Lines 14-17 check whether the TPN places can be either SDC places or connectors (P_c) and that the only flows allowed in the TPN are within SDCs or between connector places, P_c, and SDC constructs.

If the algorithm reaches Line 18 without breaking, it concludes that the TPN is an SDR net and returns its set of SDCs S and the set of connector places P_c. Proposition 1 states the correctness and completeness of Algorithm 1. **polish proof**

Proposition 1. *Algorithm 1 is correct and complete, i.e., the algorithm returns non-empty sets, if and only if, the input TPN is an SDR net.*

Proof. If: Given an input TPN that is an SDR net, the algorithm returns non-empty sets.
The input TPN is an SDR net. Hence, by Definition 3, it has a source and a sink. Furthermore, $E = \bigcup_{j=1}^{m} E_{\mathcal{S}_j}$ and therefore Lines 4 and 7 return True. In an SDR net, all flows are either in $\bigcup_{j=1}^{m} F_{\mathcal{S}_j}$ (being part of an SDC) or in F_c, i.e., connecting places in $P_c = \{p_1, \ldots, p_{k+1}\}$ to the set of SDCs in the partition set, C. Furthermore, from the definition of F_c each partition set $\mathcal{C}_j$ has exactly a single direct predecessor place $p_j \in P_c$ and a single direct successor place p_{j+1}. Moreover, no two sets in C can have the exact same predecessors and successors (otherwise, they would not be two different sets in C). Therefore, the condition in Line 13 holds for every $\mathcal{C}_j \in C$.

and only If: Given that the algorithm returns non-empty sets, the input net is an SDR net.
We prove the second direction by using the intermediate computations of Algorithm 1 to construct an SDR net $\mathcal{N} = (E, E', P, F, \tau)$ and showing that the net $\mathcal{N}$ is equal to the input TPN.

□

Having defined SDR nets and an algorithm that verifies whether a given TPN is an SDR net, we define the seize-delay-release net with resources (SDRR net), which is a timed Petri net that consists of a set of SDR nets and a set of places and flows that correspond to resources. Let $N_{sdr} = \{\mathcal{N}_1, \ldots, \mathcal{N}_n\}$ be a set of SDR nets with $\mathcal{N}_j = \langle E_j, E'_j, P_j, F_j, \tau_j \rangle, j = 1, \ldots, n$ and let $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$ be a set of SD constructs that participate in N_{sdr}. We are now ready to define the SDRR nets.

Definition 4 (Seize-Delay-Release Net with Resources (SDRR net))**.** *An SDR net with resources (SDRR net) is a timed Petri net, $\mathcal{N}_{sdrr} = (E, E', P, F, \tau)$, such that*

- *The set $E = \bigcup_{j=1}^{n} E_j$ contains only transitions from the SDNs ($E' \subseteq E$ being the set of timed transitions),*
- *The set $P = \bigcup_{j=1}^{m} P_j \cup P_r$ contains both places from N and a finite set of resource places P_r, and,*
- *The flow set $F = \bigcup_{i=1}^{m} F_i \cup F_r$ contains all SDN flows 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 resource flows F_r allows for resource tokens to be consumed only by immediate transitions and produced only by timed transitions. Furthermore, the property $Q(x,y)$ makes sure that if an immediate transition that consumes a resource token is part of an SDC, say $\mathcal{S}_i$, then there must also be a flow between the timed transition of $\mathcal{S}_i$ back to the same resource place. Similarly, if a timed transition of an SDC produces a resource token, there must be a flow between the resource place and the immediate transition of the same SDC. One can easily verify that Figure 2 is an SDRR net with a single SDR net, three SDCs and two resource places (NP for nurse practitioner and RN for registered nurse).

Next, we introduce Algorithm 2, which take a TPN as its input, and returns whether the TPN is an SDRR net.

Algorithm 2: Verifies that TPN is SDR net with resources (SDRR net).

```
Input: Timed Petri net $\mathcal{N} = \langle E, E', P, F, \tau \rangle$
Output: True or False
1 begin
2     $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\} \leftarrow DetectSDC(\mathcal{N})$
3     $P_r \leftarrow \{p \in P \mid \forall (x,y) \in F_{\{p\}}(Q(x,y,S))\}$
4     $\mathcal{N}' = (E, E', P \setminus P_r, F \setminus F_{P_r}, \tau)$
5     $\{\mathcal{N}_i\}_{i=1}^{n} \leftarrow ConnectedComponents(\mathcal{N}')$
6     if $\exists i \in [n]$ : VerifySDR($\mathcal{N}_i$) $= False$ then
7         return False
8     return True
9 end
```

update this part for the new algorithm We shall briefly go over the algorithm and explain its parts. Lines 2-7 serve us to identify a set of places P_r that are candidates to be the resource places of the resulting SDRR. In an SDRR, a necessary condition for every $p \in P_r$ is that when p is removed from the net (along with the incoming and outgoing flows $F_{\{p\}}$) there will still exist a path between SDR net sources to their corresponding sinks. Removing a non-resource place, i.e., a place from one of the SDR nets of the SDRR net, will result in a source without a path to its sink. Therefore, removing places from the input TPN and verifying that every source has a path to a sink (via the function *AllSourceToSink* that receives a TPN and checks whether every source has a path to a sink), enables us to find candidates for P_r. Then, in Line 8, we remove all places in P_r from the input TPN and store the resulting net $\mathcal{N}'$. In Line 9, we store the connected components of $\mathcal{N}'$ in a set of TPNs $\{\mathcal{N}_i\}_{i=1}^{n}$. If the input is an SDRR, this set will contain n SDR nets. Therefore, in Lines 12-18, we verify that the n connected components are SDR nets. Verifying whether $\mathcal{N}_i$ is an SDR net is performed by running a Breadth-First Search (BFS) algorithm that verifies the conditions in Definition 3. The procedure *VerifySDR*$(\cdot)$ returns $\emptyset$ if $\mathcal{N}_i$ is not an SDR (which will lead to an answer False in Algorithm 2 due to a violation of Definition 4); otherwise, it returns the set $P_{c,i}$, which is the connector set of the now verified SDR net $\mathcal{N}_i$, and S_i, which is the set of SDCs that $\mathcal{N}_i$ comprises. Lastly, Lines 19-20 check whether the set of candidate resource places, P_r, satisfies $Q(x,y,S)$ from Definition 4. If one of the flows into $p \in P_r$ violates $Q(x,y,S)$, the algorithm returns False. Otherwise, the algorithm returns True.

polish proof

Proposition 2. *Algorithm 2 is correct and complete, i.e., the algorithm returns* True, *if and only if, the input to the algorithm is an SDRR net.*

Proof. If: Given an input TPN that is an SDRR net, the algorithm returns True.
If $\mathcal{N}$ is an SDRR net, then the only places that would not interrupt the flow between every source and sink of its SDR nets in N_{sdr}, are places in P_r. Therefore, the set P_r computed by the algorithm will contain only resource places of the SDRR net. After removing the resource places and their corresponding flows (Line 8), we are remained with n connected components, with each component being an SDR net (Line 9). The algorithm will verify the following two conditions: (1) the n components are SDR nets (Lines 10-18), and (2) the set of flows F_r respects $Q(x,y,S)$ (since the TPN is an SDRR net). Therefore it will return True.

and only If: Given that the algorithm returns True, the input net is an SDRR net.
We prove the second direction by using the intermediate computations of Algorithm 1 to construct a Petri net $\mathcal{N} = (E, E', P, F, \tau)$ and showing that $\mathcal{N}$ corresponds to an SDRR net (Definition 4). We compose the SDRR net by connecting the TPNs $\mathcal{N}_i, i = 1, \ldots, n$ (which are verified to be SDR nets by Lines 10-18) to the set of places P_r using the flow set F_r. Since the flows F_r respect property $Q(x,y)$, the resulting net is an SDRR net according to Definition 4.

□

4 Transforming SDPN to CP Models

In this part, we present a transformation of schedule-driven Petri nets (SDPNs) to constraint programming (CP) formulations. To this end, we define the *basic scheduling constructs set* (BSCS). Subsequently, we show that the BSCS can be derived from the SDPN. Lastly, we construct CP models from a given BSCS, thus completing the TPN2CP phase of our solution.

4.1 The Basic Scheduling Constructs Set

The BSCS will contain the activities to be scheduled, the resources along with their capacities, precedence constraints, and activity-and-resource dependent durations. In essence, the BSCS contains a set of building blocks of many well-known scheduling problems such as the job-shop scheduling problem (JSSP), and the resource-constrained project scheduling problem (RCPSP) (including its multi-mode variation). To define BSCS, we follow the scheduling problem definition introduced by (Van der Aalst 1996).

Definition 5 (Basic Scheduling Constructs Set BSCS)**.** *Basic scheduling constructs are a set $B = \{\mathcal{A}, \mathcal{R}, \Pi, c, d\}$ over a over a finite time domain $\mathcal{T}$ with,*

- *$\mathcal{A}$ being the set of activities to be scheduled,*
- *$\mathcal{R}$ being the set of renewable resources,*
- *$\Pi \subseteq \mathcal{A} \times \mathcal{A}$ being the precedence relation between pairs of activities,*
- *$c : \mathcal{R} \to \mathbb{N}^+$ being the function that maps resources to their capacities, and,*
- *$d : \mathcal{A} \times 2^{\mathcal{R}} \nrightarrow \mathcal{T}$ being the duration partial function that maps pairs of activities and resource sets (that can execute these activities) to values in the time domain.*

A schedule s, which satisfies a BSCS, is an allocation of resource sets to activities over time, i.e., $s \in \mathcal{A} \to (2^{\mathcal{R}} \times \mathcal{T})$. A feasible schedule respects resource and precedence constraints. Without loss of generality, one often considers an objective function $\phi(s) \in \mathcal{R}^{+0}$ that assigns a real-valued number to a given schedule. In optimal scheduling one aims at finding a schedule that minimizes $\phi(s)$. In this work, we avoid learning the objective function of the underlying scheduling problem, and hence assume that ϕ is given.

A prominent example that can be constructed using the BSCS is the *job shop scheduling problem* (JSSP), where only a single resource can perform each activity (hence $d(a), a \in \mathcal{A}$ is sufficient to represent durations), and resource capacities are equal to 1. The objective function in JSSP is to minimize the makespan.

4.2 Deriving BSCS from SDPN

In this part, we provide a mapping from Seize-Delay-Release timed Petri nets with resources (SDRR nets) into basic scheduling problems (BSPs). Specifically, given an SDRR net, $\mathcal{N} = \langle E, E', P, F, \tau \rangle$, our we propose a mapping that creates the corresponding BSP tuple, $\langle \mathcal{A}, \mathcal{R}, \Pi, c, d \rangle$.

For conciseness, we refer to elements of the TPN (e.g., places, transitions, flows) when defining the BSP instead of labeling the TPN and then using these labels to define the BSP. **not sure that this needs to be said here**

For an SDRR net, we may reuse parts of Algorithms 1 and 2 to derive the following sets: (1) the set of SDCs $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$, (2) the set of resource places P_r, (3) the set of SDR nets that comprise the SDRR net $N_{sdr} = \{\mathcal{N}_i\}_{i=1}^n$ (4) the set of connector places $P_c = \{P_{c,i}\}_{i=1}^n$ for every SDR net that comprises the SDRR net, and (5) the partition of SDCs, $\{C_i\}_{i=1}^n$, which includes sets of SDCs with common input and output connectors in the ith SDR net.

Definition 6 (SDRR net to BSP Mapping)**.** *Given an SDRR net $(\mathcal{N} = (E, E', P, F, \tau), m_0)$ and the sets $S, P_r, N_{sdr}, P_{c,i}, C_i$, the BSP is constructed as follows:*

- *The resource set is given by the set of places, $\mathcal{R} = P_r$,*
- *The activity set corresponds to all connector places in P_c except the sinks, namely $\mathcal{A} = \{p \in P_c | p\bullet \neq \emptyset\}$,*
- *The precedence relation Π is constructed using the following:*

$$\Pi = \{(a, b) \in \mathcal{A} \times \mathcal{A} \mid a \rightsquigarrow b\},$$

  with $\rightsquigarrow$ indicating that there exists a path from a to b,
- *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 finally,*
- *the duration (partial) function d is computed as follows:*

$$\begin{aligned} d = &\{(a, R, d) \in \mathcal{A} \times 2^{\mathcal{R}} \times \mathcal{T} \mid \\ &\forall \mathcal{S} \in S (\forall e \in E_{\mathcal{S}} \setminus E'_{\mathcal{S}} (a \in \bullet e \ \wedge \ R \subseteq (\bullet e \cap \mathcal{R})) \\ &\wedge \forall e' \in E'_{\mathcal{S}} (d = \tau(e')))\} \end{aligned}$$

4.3 From BSCS to CP Models

Kyle, please insert BSP to CP here. Chris will add a part on the intuition for scheduling people

5 Evaluation

6 Related Work

7 Conclusion

References

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

[Beldiceanu and Simonis 2012] 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.

[Haas 2002] Haas, P. J. 2002. *Stochastic Petri Nets: Modelling, Stability, Simulation.* Springer.

[Lallouet et al. 2010] 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 and DiCesare 1994] 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.

[Motahari-Nezhad, Recker, and Weidlich 2015] Motahari-Nezhad, H. R.; Recker, J.; and Weidlich, M., eds. 2015. *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, volume 9253 of *Lecture Notes in Computer Science*. Springer.

[Senderovich et al. 2015a] Senderovich, A.; Rogge-Solti, A.; Gal, A.; Mendling, J.; Mandelbaum, A.; Kadish, S.; and Bunnell, C. A. 2015a. Data-driven performance analysis of scheduled processes. In Motahari-Nezhad et al. (2015), 35–52.

[2015b] Senderovich, A.; Rogge-Solti, A.; Gal, A.; Mendling, J.; Mandelbaum, A.; Kadish, S.; and Bunnell, C. A. 2015b. Data-driven performance analysis of scheduled processes. In Motahari-Nezhad et al. (2015), 35–52.

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

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