PetriFlow: A Petri Net Based Framework for Modelling and Control of Workflow Processes

Martin Riesz, Martin Sečkár, and Gabriel Juhás

Faculty of Electrical Engineering and Information Technology,
Slovak University of Technology,
Ilkovičova 3, 812 19 Bratislava, Slovak Republic
{martin.riesz,xseckar,gabriel.juhas}@stuba.sk

Abstract. The paper presents an architecture of a Petri net based framework for modelling and control of workflow processes. It focuses on the PNEditor module and briefly discusses workflow engine based on the models designed using the PNEditor. Then the paper describes method of synthesis *Separating feasible places* and an algorithm for reducing the number of places in the resulting Petri net.

Key words: PetriFlow, PNEditor, workflow management, synthesis

1 Introduction

There are many different Petri net tools and Petri net based tools for modelling workflow processes, such as CPN ([1]), Viptool ([2], [3]), Yasper ([4]) or ProM ([5]), to mention just some of them. The question arises why to implement another one. Most of them are determined to create models and some of them to analyze the models. However, models are only the first step in a business process management systems (BPMS). The main advantage of BPMS is that the models can be used to control the workflow process according to the designed model using a workflow engine.

The problem of the most existing editors (let us just mention Viptool as a prominent example), is that they were implemented with different aims, mostly to analyze the model, but they do not provide all the information needed for the generation of a deployable application, such as resource management.

Another problem is with the case generation. Usually, a model obtained via a Petri net modelling tool can be understood as a general definition of a model of a process, while the single cases can be understood as instances of the process. Using an analogy with object-oriented programming, a model can be understood as a class, while single cases can be understood as objects of that class. In Petri net based modelling tools, the realization of cases is often done using coloured Petri nets ([6]). But in such tools, colours are used both for distinguishing cases from each other as well as for modelling the data of the cases.

Another important feature for a successful application of models is a hierarchy, which enables not only to model on different level of abstraction but also to deploy reusable components. This is important for a practical use, as the business consultants, which design the models often do not offer them of sufficient level of details. Many tools offer different kinds of hierarchical nets, but because they also mostly implement a semantical framework with aim to preserve some properties, they are mostly too complex and/or too restrictive to generate a deployable code.

2 PetriFlow: A Brief Introduction

For the above mentioned reason, we develop a new framework for modelling and control of workflow processes based on Petri nets. Presently, it consists of two modules, PNEditor and PNEngine.

PNEditor enables to design a model, which is basically a place/transition nets enriched with the feature for distinguishing between static and dynamic places ([7]), where the static places correspond to static variables (they exists once for a process), while the dynamic places are constructed for each case. Moreover PNEditor enables modelling with subnets, where the subnets represents only a visual tool for designer, i.e. on semantical level, PNEditor works with a flat place/transition nets. Another important feature of PNEditor is that it provides a resource management on a very simple and intuitive way. It enables to create roles, where a role is basically a subset of the set of transitions, determining which transitions (which tasks) can be performed by users having the role. Further is PNEditor able to synthesize Petri nets from process logs.

The development version of the PetriFlow PNEditor can be downloaded via: `http://pneditor.matmas.net/`

PNEngine is a light version of a workflow engine based on the models provided by PNEditor. User registered in PNEngine is able to upload processes and thus becoming their owner. The owner of the process can then assign other users to roles defined in the process. Only user assigned to a given role is able to fire transitions contained in the role. Further, PNEngine enables to create new cases for a given process and to control the cases processing according to the business logic given by the Petri net modelling the process. The business logic layer is implemented in J2EE, the connection with the database and persistence of the cases is realized using Hibernate. The user interface is realized via Java Server Pages. The PNEngine is running on a Tomcat server. It enables the users to perform an activity from the task list for actually processed cases, i.e. to fire enabled transitions of the correspondent copy of the net, via a web browser. After a user performs an activity from the task list for a given case, the corresponding transition is fired, the new marking is computed and the task list is updated. Different cases of the same process are able to communicate over static places.

3 PetriFlow PNEditor

PNEditor offers usual features of a graphical editor for designing place/transition nets. It enables to draw place/transition nets, i.e. labeled places, transitions, weighted arcs and markings with multiple tokens per place. The further functionality is saving the net to a file, the definition of roles, subnets (nested nets), saving of predefined subnets to files and their reuse as reusable components, replacement of subnets, definition of static places, which exists once per process, and other standard features, such as unlimited undo-redo actions.

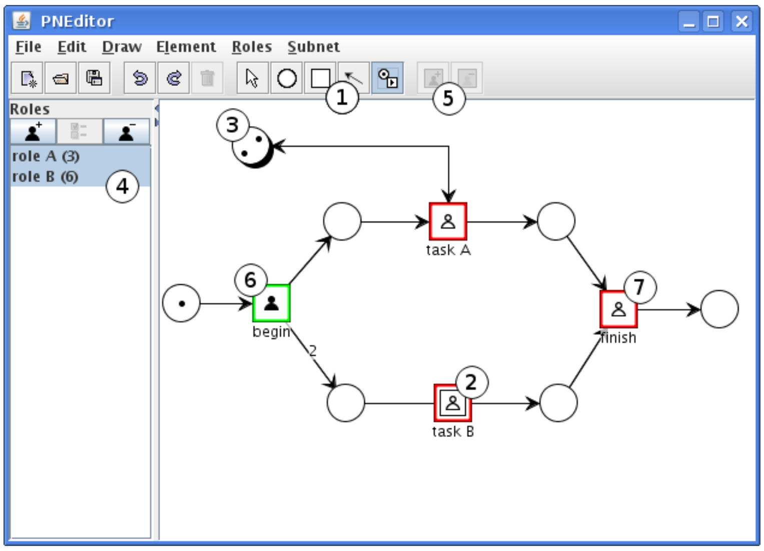

Fig. 1. Illustration of the key features of the PetriFlow PNEditor.

For a better illustration, Fig. 1 gives an overview of the functionality of the PetriFlow PNEditor module:

1. Drawing tool selection: from left: object selection tool, inserting places, inserting transitions, arc drawing, adding/removing tokens/transition firing
2. Square with double border represents a subnet
3. Place with shadow represents a static place
4. Panel with roles: buttons from left: add a role, edit role properties, delete a role.
   – role A contains set of transitions: begin, task A, finish (total of 3)
   – role B contains set of transitions: begin and all nested transitions in subnet task B (total of 6)
5. Buttons for adding or removing transitions from the currently selected roles
6. Both roles are selected so the information icons are displayed on top of the transitions in the diagram: black person icon on the transition describes the situation in which all the selected roles contain this transition
7. White person icon on the transition describes the situation in which only some selected roles contain this transition

4 Subnets in PetriFlow PNEditor

Usually, business modeller models a task as atomic transitions. On a more detailed level, typically a task can be started, finished, paused, continued or cancelled. Each of such tasks can be illustrated with a part of Petri net given in Fig. 2, which can be understood as a subnet on a more detailed level and should be used as a reusable component.

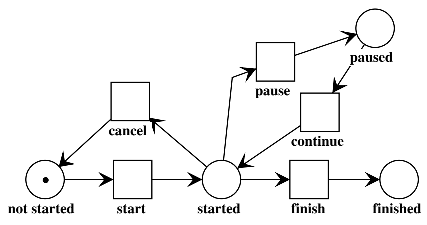

Fig. 2.

The place "not started" in Fig. 2 describes a state, in which the task has not yet been started and "finished" describes the state reached after the task is finished.

It would be very practical to model workflow processes using such reusable components and in general to give modeller an option to design and use his own reusable components possibly without any semantical restriction.

Another situation in which reusable components represents a desirable feature is in case of complex processes assembled from a relatively independent units with exactly defined inputs and outputs. The PNEditor supports subnets allowing each subnet to have more nested subnets so that the whole hierarchy can be build up in a place/transition net.

In case the user creates a workflow model where the tasks will be represented by transitions, PNEditor gives a choice of replacing a transition with a subnet. The subnets can be replaced by existing stored subnets. In this way, individual transitions can be converted to custom subnets representing reusable components.

In order to explain the concept of subnets, we have to recall some basic definitions of place/transition nets (for more details see e.g. [8]). Given a place/transition net $N = (P, T, F, W)$, where P is a finite set of places, T is a finite number of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs (i.e. the flow relation) and $W : F \to \mathbb{N}_0$ is the weight function ($\mathbb{N}_0$ denoting nonnegative integers). We say that a subnet of N is any net $N' = (P', T', F', W')$ where $P' \subseteq P$, $T \subseteq T$, the flow relation $F' = F \cap ((P' \times T') \cup (T' \times P'))$ and $W' = W|F'$. Moreover, we consider only proper subnets, i.e. subnets satisfying the following condition for each $p \in P$ and each $t \in T'$: $((p, t) \in F \vee (t, p) \in F) \Rightarrow p \in P'$. For clarity of the text let us define the interface of a proper subnet as the set of places $p \in P'$ which are connected with a transition which does not belong to

T'. In the PNEditor, the interface places are graphically expressed using dashed places. On one abstraction level, a subnet is visualized via interface places connected with a square with double border via reference edges. These edges can have one of two appearances:

1. dotted edge - the interface place is not connected with any transition in the subnet.
2. dashed edge - the interface place is connected in the subnet with one or more transitions

In case the interface place is connected in the subnet with exactly one transition, the reference edge takes the direction of the arc. Otherwise the reference edge is undirected, i.e. it is displayed without an arrow.

Neighbourhood of a place $p \in P$ w.r.t. the net N is a subnet $N_p = (\{p\}, T_p, F_p, W_p)$ of N with the set of places formed by the place itself and the set of transitions $T_p = \{t \in T | (t, p) \in F \vee (p, t) \in F\}$ formed by the the union of the preset and the postset of the place p in the net N, i.e. by surrounding transitions of p in net N.

Recall that two place/transition nets are isomorphic, when there exists a bijective mapping between the sets of places and a bijective mapping between the sets of transitions, which preserves arcs and their weights. We say that a place p in a place/transition net is said unique place of the net, if there is no place p' in the net with the isomorphic neighbourhood.

In the PNEditor, identities of the interface places are not saved when storing a subnet to make it a reusable component, i.e. a subnet is stored just as an ordinary net with an additional information which places form its interface. When replacing one subnet by another stored subnet, the interface places of the replaced subnet are identified with the interface places of the stored replacing net according to the following rules:

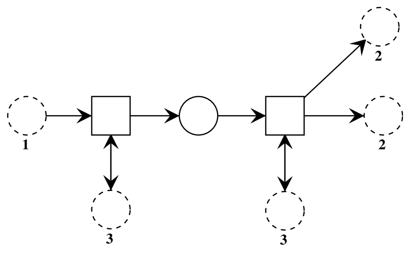

Fig. 3. Interface places labelled with the same number have isomorphic neighbourhood. Only the interface place labelled with number 1 is unique.

1. In the first place, only unique interface places are identified: A unique interface place p of the replaced subnet is considered to be equal to a unique

interface place p' of the stored replacing subnet, if the neighbourhood of p w.r.t. the replaced subnet is isomorphic to the neighbourhood of p' w.r.t. the replacing stored net.
2. In the second step, if there exists exactly one unique interface place of the replaced subnet and exactly one unique interface place of the replacing stored net satisfying that their neighbourhoods w.r.t. the respective subnets are not isomorphic, then these interface places are considered to be equal. This correspond to a predicate, that if it is unambiguously possible, then the interface places should be identified.
3. Remaining interface places of replacing subnet replacing stored net are changed to ordinary places and remaining interface places of replaced net become interface places of the replacing net. It means that it is left to the user to identify manually by further editing which remaining interface places of the replaced subnet equal to the remaining interface places of the replacing net.

An example of the use of the subnet concept in the PNEditor is illustrated in Fig. 4:

1. The transition is created and converted to subnet
2. Visualization of the inside of the subnet
3. The subnet is modified and saved to a file
4. New subnet is created, selected command for replacing subnet
5. The result of 2 identical subnets

Thus, behind the visualization of a hierarchical process model in the PNEditor using the subnet concept is a single flat place/transition net.

5 Synthesis in PetriFlow PNEditor

We could design process models manually – which can be tedious and error-prone. There is also the possibility to collect logs from real-time processes and let an algorithm do the work for us. Workflow management systems such as the PNEngine can also be used to collect the logs. We just need simple p/t net with all transitions always enabled that will represent expecting activities.

There are multiple methods of Petri net synthesis already invented [9]. In the PNEditor we implemented a region based method *Separating feasible places* as described in [10].

5.1 Preliminaries

As usual we use the following notations. For details see [10].

An *alphabet* is a finite set A. The set of all *strings (words)* over an alphabet A is denoted by A^*. The *empty word* is denoted by λ. A subset $L \subseteq A^*$ is called *language over* A. For a word $w \in A^*$, $|w|$ denotes the *length of* w and $|w|_a$ denotes the number of occurrences of $a \in A$ in w. Given two words v, w, we

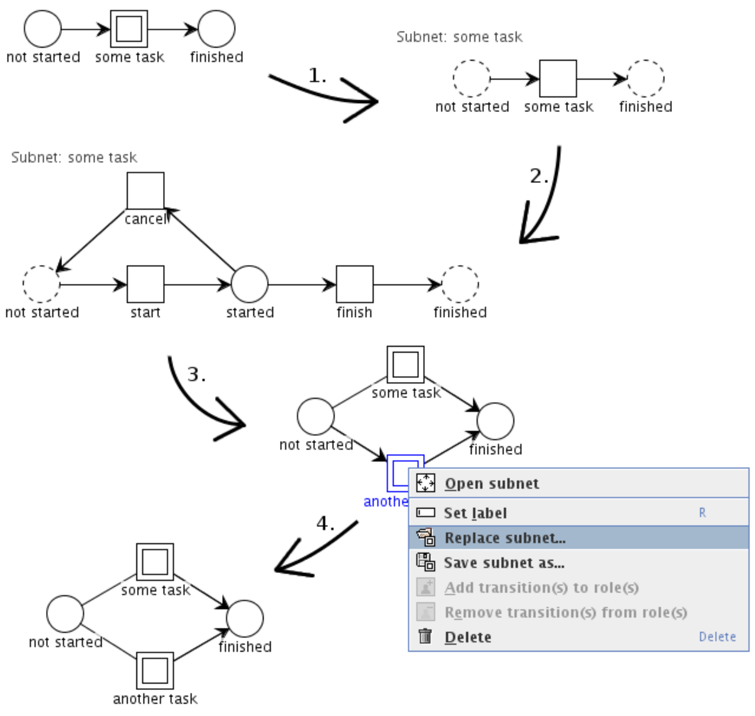

Fig. 4. Illustration of the subnet concept in PetriFlow PNEditor

call v *prefix* of w if there exists a word u such that $vu = w$. A language L is *prefix-closed*, if for every $w \in L$ each prefix of w also belongs to L.

Let T be a finite set of *activities* and C be a finite set of *cases*. And *event* is an element of $T \times C$. And *event log* is an element of $(T \times C)^*$.

Given a case $c \in C$ we define the function $p_c : T \times C \to T$ by $p_c(t, c') = t$ if $c = c'$ and $p_c(t, c') = \lambda$ else. Given an event log $\sigma = e_1 \dots e_n \in (T \times C)^*$ we define the *process language* $L(\sigma)$ of σ by $L(\sigma) = \{p_c(e_1) \dots p_c(e_i) | i \leq n, c \in C\} \subseteq T^*$.

A net is a triple $N = (P, T, F)$, where P is a set of *places*, T is a finite set of *transitions* satisfying $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation.* Let $x \in P \cup T$ be an element. The *preset* $\bullet x$ is the set $\{y \in P \cup T | (y, x) \in F\}$, and the *post-set* $x\bullet$ is the set $\{t \in P \cup T | (x, y) \in F\}$.

A *marking* of a p/t net $N = (P, T, F, W)$ is a function $m : P \to \mathbb{N}_0$ assigning $m(p)$ tokens to a place $p \in P$. A *marked p/t net* is a pair (N, m_0), where N is a p/t net, and m_0 is a marking of N, called *initial marking.*

A transition $t \in T$ is *enabled to occur* in a marking m of a p/t net N if $m(p) \geq W(p, t)$ for every place $p \in \bullet t$. If transition t is enabled to occur in a marking m, then its *occurrence* leads to the new marking m' defined by $m'(p) = m(p) - W(p, t) + W(t, p)$ for every place $p \in P$. That means t consumes $W(p, t)$ tokens from p and *produces* $W(t, p)$ tokens in p. We write $m \xrightarrow{t} m'$ to denote that t is enabled to occur in m and that its occurrence leads to m'. A finite sequence of transitions $w = t_1 \dots t_n$, $n \in \mathbb{N}$ is called an *occurrence sequence enabled in m and leading to m_n* if there exists a sequence of markings $m_1, \dots, m_n$ such that $m \xrightarrow{t_1} m_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} m_n$. The set of all occurrence sequences enabled in the initial marking m_0 of a marked p/t net (N, m_0) forms a language over T and is denoted by $L(N, m_0)$.

Let (N, m_p), $N = (\{p\}, T, F_p, W_p)$ be a marked p/t net with only one place p (F_p, W_p, m_p are defined according to the definition of p). The place p is called *feasible* (w.r.t. $L(\sigma)$), if $L(\sigma) \subseteq L(N, m_p)$, otherwise *non-feasible.*

Denoting $T = \{t_1, \dots, t_n\}$, a *region* of $L(\sigma)$ is a tuple $\mathbf{r} = (r_0, \dots, r_{2n}) \in \mathbb{N}^{2n+1}$ satisfying for every $ct \in L(\sigma)$ ($c \in L(\sigma)$, $t \in T$):

$$r_0 + \sum_{i=1}^{n} (|c|_{t_i} \cdot r_i - |ct|_{t_i} \cdot r_{n+i}) \geq 0. \tag{1}$$

Every region $\mathbf{r}$ of $L(\sigma)$ defines a place p_r via $m_0(p_r) := r_0$, $W(t_i, p_r) := r_i$ and $W(p_r, t_i) := r_{n+i}$ for $1 \leqslant i \leqslant n$. The place p_r is called corresponding place to $\mathbf{r}$.

Given language L over T, $WC(L) = \{w \in L, t \in T : wt \notin L\}$ is called a set of wrong continuations of L over T.

Let $\mathbf{r}$ be a region of $L(\sigma)$ and let $WC \subseteq WC(L(\sigma))$ is a set of wrong continuations. The region $\mathbf{r}$ is a *separating region (w.r.t. WC)* if for every $wt \in WC$:

$$r_0 + \sum_{i=1}^{n} (|w|_{t_i} \cdot r_i - |wt|_{t_i} \cdot r_{n+i}) < 0. \tag{2}$$

A separating region $\mathbf{r}$ w.r.t. a set of wrong continuations $WC \subseteq WC(L(\sigma))$ can be calculated (if it exists) as a non-negative integer solution of a homogeneous linear inequation system with integer coefficients of the form

$$\mathbf{A}_{L(\sigma)} \cdot \mathbf{r} \geq \mathbf{0}$$
$$\mathbf{B}_{WC} \cdot \mathbf{r} < \mathbf{0}.$$

The matrix $\mathbf{A}_{L(\sigma)}$ consists of rows $\mathbf{a}_{ct} = (a_{ct,0}, \dots, a_{ct,2n})$ for all $ct \in L(\sigma)$, satisfying $\mathbf{a}_{ct} \cdot \mathbf{r} \geq \mathbf{0} \Leftrightarrow (1)$. This is achieved by setting for each $ct \in L(\sigma)$:

$$a_{ct,i} = \begin{cases} 1 & \text{for } i = 0, \\ |c|_{t_i} & \text{for } i = 1, \dots, n, \\ -|ct|_{t_{i-n}} & \text{for } i = n+1, \dots, 2n. \end{cases}$$

The matrix $\mathbf{B}_{WC}$ consists of rows $\mathbf{b}_{wt} = (b_{wt,0}, \dots, b_{wt,2n})$ for all $wt \in WC$, satisfying $\mathbf{b}_{wt} \cdot \mathbf{r} < \mathbf{0} \Leftrightarrow (2)$. This is achieved by setting for each $wt \in WC$:

$$b_{wt,i} = \begin{cases} 1 & \text{for } i = 0, \\ |w|_{t_i} & \text{for } i = 1, \dots, n, \\ -|wt|_{t_{i-n}} & \text{for } i = n+1, \dots, 2n. \end{cases}$$

The linear inequation system mentioned can be solved using linear programming ([11]) with linear objective function to minimize the resulting separating region, i.e. to generate minimal arc weights and a minimal initial marking.

5.2 Method of Separating Feasible Places

Given an event log σ with set of activities T we search for a preferably small finite marked p/t net (N, m_0) such that $L(\sigma) \subseteq L(N, m_0)$ and $L(N, m_0) \setminus L(\sigma)$ is small. According to the method of *Separating feasible places* we first create a p/t net with all transitions T but no arcs or places. This way, any occurrence sequence is enabled. Then we keep adding feasible places, until each wrong continuation of $WC(L(\sigma))$ is prohibited. Each feasible place is created according to one wrong continuation $wt \in WC(L(\sigma))$. We only calculate separating region w.r.t. $\{wt\}$ if wt is not already prohibited by already added places, because one place can prohibit multiple wrong continuations. For details see Algorithm 1.

5.3 Algorithm for Reducing the Number of Places

In some cases (see Fig. 5) we observed more than necessary number of places in the resulting net, so we created an algorithm for reducing the number of places in the resulting net. This algorithm is also implemented in the PNEditor.

Given a finite set A, the symbol $|A|$ denotes cardinality of A.

Our solution to this problem was to first identify which wrong continuations are prohibited by which places. Each place can prohibit multiple wrong continuations. This information is easy to get: we temporarily remove places $P' \subseteq P$ from the marked p/t net (N, m_0), $N = (P, T, F, W)$ and if the net permits given

```
input  : An event log $\sigma$
output: $(N, m_0)$, $N = (P, T, F, W)$ such that $L(\sigma) \subseteq L(N, m_0)$
$L(\sigma) \leftarrow$ process language of $\sigma$;
$A \leftarrow$ empty matrix;
$(P, T, F, W, m_0) \leftarrow (\emptyset,$ activities of $\sigma, \emptyset, \emptyset, \emptyset)$;
foreach $w \in L(\sigma)$ do
    add row $a_w$ to matrix $A$;
end
foreach $w \in WC(L(\sigma))$ do
    if $w \in L(N, m_0)$ then
        $\mathbf{r} \leftarrow$ integer solution of $\mathbf{A} \cdot \mathbf{r} \geq \mathbf{0}, \mathbf{b}_w \cdot \mathbf{r} < 0, \mathbf{r} \geq \mathbf{0}$ such that $\mathbf{r}$ is minimal;
        if such solution $\mathbf{r}$ exist then
            $p \leftarrow$ corresponding place to $\mathbf{r}$;
            $P \leftarrow P \cup \{p\}$;
        end
    end
end
$coveredWrongContinuations \leftarrow \{w \in WC(L(N, m_0))\}$;
foreach $p \in P$ do
    $P \leftarrow P \setminus \{p\}$;
    $undo \leftarrow False$;
    foreach $w \in coveredWrongContinuations$ do
        if $w \in L(N, m_0)$ then
            $undo \leftarrow True$;
            break;
        end
    end
    if $undo$ then
        $P \leftarrow P \cup \{p\}$;
    end
end
```

Algorithm 1: The method of *Separating feasible places*: We add places that permit all correct continuations and prohibit at least one wrong continuation. In case a given wrong continuation is already prohibited by an already added place we do not need to create new one for the wrong continuation – it would be unnecessary. We slightly modified the existing algorithm – we moved the cleaning of unnecessary places after computing initial marked p/t net. See the original algorithm in [10]. If all wrong continuations are prohibited without given place then the place is unnecessary.

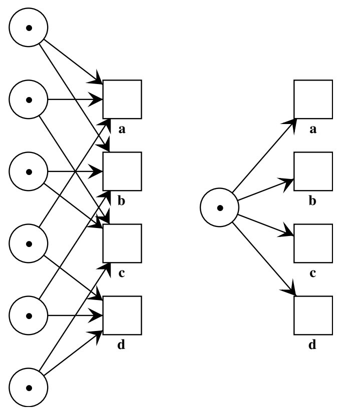

Fig. 5. On the left is a synthesized net using method of *Separating feasible places* and on the right is what we think an optimal solution.

wrong continuation $w \in WC(L(N, m_0))$ then we can say w is prohibited by the places P'.

The original method of *Separating feasible places* constructed each place from exactly one wrong continuation and all correct continuations. We will be constructing new places in the same way except we will be using one or more wrong continuations simultaneously.

First, we pick two places $p_1, p_2 \in P$. We determine which wrong continuations $WC_{p_1,p_2} \subseteq WC$ are prohibited by these places. Then we construct a new place p_3 from WC_{p_1,p_2} using the original method.

Now we compare what is "better": the two places p_1, p_2 or the one place p_3. We need to define what "better" actually is. We assumed that if the net has overall fewer places, fewer arcs then it is better. We decided to measure the complexity of the net $N = (P, T, F, W)$ as:

$$complexity(N) = |P| + \sum_{p \in P, t \in T} W(p,t) + W(t,p).$$

If the net has lower complexity without p_1, p_2 and with p_3, we replace p_1, p_2 with p_3, else we pick another pair of p_1, p_2 and repeat the cycle until we tried every combination.

When we make a replace, we run the algorithm again until no replace is made. The algorithm terminates because we have finite number of places.

We decided to test just two places at a time because we sought a fast algorithm. Testing every possible subset of the places would not be practical as it would have exponential time complexity according to the number of places.

Let N^* be a set of all possible p/t nets. Let *neighbour* be a function $N^* \times P \times P \rightarrow \{0, 1\}$. For a given p/t net $N = (P, T, F, W)$ and $p_1, p_2 \in P$ is $neighbour(N, p1, p2) = 1$ when $\exists t \in T : p_1 \in \bullet t \wedge p_2 \in \bullet t \vee p_1 \in t \bullet \wedge p_2 \in t \bullet$, otherwise $neighbour(N, p1, p2) = 0$.

Further we observed that each combination of places $p_1, p_2 \in P$, that were later merged to one place p_3, were in the same preset or post-set of some transition, i.e. $neighbour(N, p_1, p_2) = 1$. We used this observation to improve average case performance of the algorithm. Instead of testing each possible pair $p_1, p_2 \in P$, for each $p1 \in P$ we pick $p_2 \in P$ such that $neighbour(N, p_1, p_2) = 1$. For details see Algorithm 2.

Experimental results were positive (see Fig. 6 and Fig. 7).

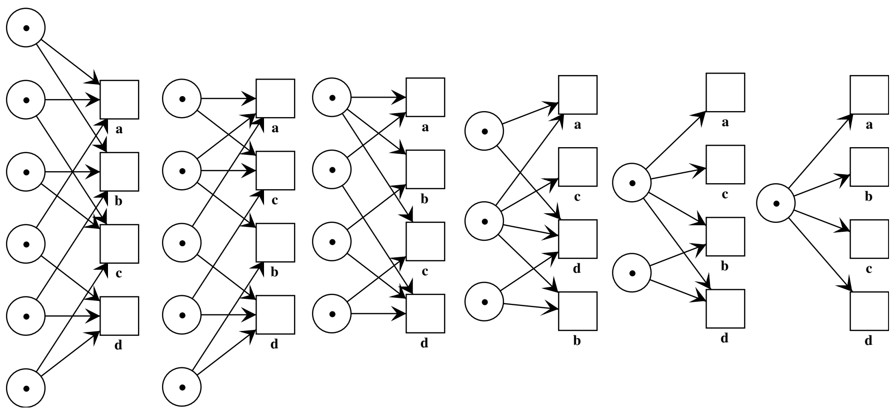

Fig. 6. Multiple steps of the algorithm for reducing complexity. On the left is input net and on the right is the output.

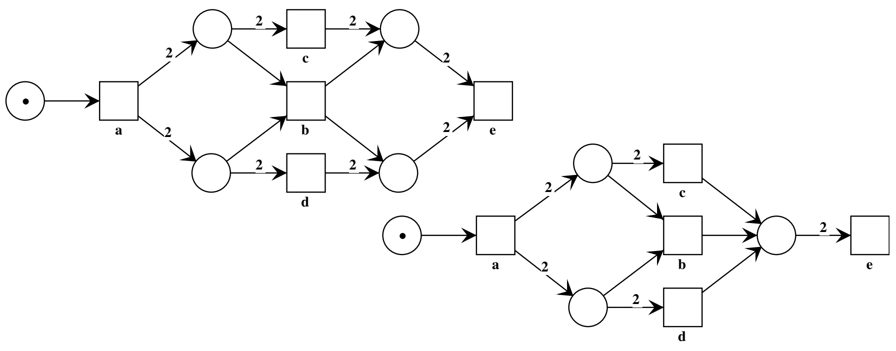

Fig. 7. On the left is synthesized net (N', m'_0) using method of *Separating feasible places* only. On the right is output (N, m_0) of our algorithm where input is the net on the left. Both nets have the same set of all occurrence sequences enabled in the initial marking, i. e. $L(N, m_0) = L(N', m'_0)$.

```
input  : Marked p/t net $(N', m'_0), N' = (P', T', F', W')$
output: Marked p/t net $(N, m_0), N = (P, T, F, W)$ such that
        $L(N, m_0) = L(N')$, $|P| \leq |P'|$ and $complexity(N) \leq complexity(N')$
$(P, T, F, W, m_0) \leftarrow (P', T', F', W', m'_0)$;
$A \leftarrow$ empty matrix;
foreach $w \in L(N, m_0)$ do
    add row $a_w$ to matrix $A$;
end
$oldNumPlaces \leftarrow |P|$;
while $True$ do
    foreach $p1, p2 \in P : p1 \neq p2 \wedge neighbour(N, p1, p2)$ do
        $coveredWC \leftarrow \emptyset$;
        foreach $p \in \{p_1, p_2\}$ do
            $P \leftarrow P \setminus \{p\}$;
            foreach $w \in WC(L(N, m_0))$ do
                if $w \in L(N, m_0)$ then
                    $coveredWC \leftarrow coveredWC \cup \{w\}$;
                end
            end
            $P \leftarrow P \cup \{p\}$;
        end
        $B \leftarrow$ empty matrix;
        foreach $w \in coveredWC$ do
            add row $b_w$ to matrix $B$;
        end
        $\mathbf{r} \leftarrow$ integer solution of $\mathbf{A} \cdot \mathbf{r} \geq \mathbf{0}, \mathbf{B} \cdot \mathbf{r} < 0, \mathbf{r} \geq \mathbf{0}$ such that $\mathbf{r}$ is minimal;
        if such solution $\mathbf{r}$ exist then
            $oldComplexity \leftarrow complexity(N)$;
            $p_3 \leftarrow$ corresponding place to $\mathbf{r}$;
            $P \leftarrow P \setminus \{p_1, p_2\} \cup \{p_3\}$;
            $newComplexity \leftarrow complexity(N)$;
            $P \leftarrow P \cup \{p_1, p_2\} \setminus \{p_3\}$;
            if $newComplexity < oldComplexity$ then
                $P \leftarrow P \setminus \{p_1, p_2\}$;
                if $\sum_{t \in T}(W(p_3, t) + W(t, p_3)) > 0$ then
                    $P \leftarrow P \cup \{p_3\}$;
                end
                break;
            end
        end
    end
    if $|P| = oldNumPlaces$ then
        break;
    end
    $oldNumPlaces \leftarrow |P|$;
end
```

Algorithm 2: Algorithm for reducing complexity.

6 Conclusion

Although there are many methods for synthesis of Petri nets from logs (sequences, languages, partial languages, etc.), the main drawback when used in practice remains: the obtained nets are still too complicated in comparison with human made models. There are different reasons. Remember, that a net without places enables all sequences of transitions, and each place restricts behaviour by removing some sequences. Often, one reason for getting compicated models is that not each valid sequence is presented in the logs and therefore synthetised net obtain too much places restricting too much behaviour. Obviously, such cases cannot be solved by optimizing algorithms as presented in this paper. However, in the case that the logs are complete, optimization is a crucial step for acceptance of process mining in practice. The presented algorithm provides a simple step towards this direction. However, much of the work still has to be done in this area to nd the right mixture between accuracy of the resulting nets and their readability.

References

1. M. Beaudouin-Lafon, W. E. Mackay, M. Jensen, P. Andersen, P. Janecek, M. Lassen, K. Lund, K. Mortensen, S. Munck, A. Ratzer, K. Ravn, S. Christensen, K. Jensen: CPN/Tools: A Tool for Editing and Simulating Coloured Petri Nets *ETAPS Tool Demonstration Related to TACAS* In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2031, pp. 574–577, Springer-Verlag, 2001.
2. R. Bergenthum, J. Desel, G. Juhs, R. Lorenz: Can I Execute my Scenario in Your Net? VipTool tells you! In *Application and Theory of Petri Nets and Other Models of Concurrency.* LNCS 4024, pp. 381–390, Springer-Verlag, 2006.
3. J. Desel, G. Juhs, R. Lorenz and C. Neumair: Modelling and Validation with VipTool. In: *Business Process Management 2003*, LNCS 2678, pp. 380–389, Springer-Verlag, 2003.
4. K.M. van Hee, J. Keiren, R. Post, N. Sidorova, J.M. van der Werf: Designing case handling systems. In *Transactions on Petri Nets and Other Models of Concurrency I*, LNCS 5100, pp. 119–133, Springer, Berlin. 2008.
5. B. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005, volume 3536 of Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
6. C.W. Gunther, W.M.P. van der Aalst: *Modeling the Case Handling Principles with Colored Petri Nets.* Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, October 2005, Department of Computer Science, University of Aarhus, PB-576, 211–230.
7. K. van Hee, N. Sidorova, M. Voorhoeve: *Resource-Constrained Workow Nets.* Fundamenta Informaticae, 2006.
8. W. Reisig: Petri nets: An introduction. Springer, 1985.
9. R. Bergenthum, J. Desel, S. Mauser: *Comparison of Different Algorithms to Synthesize a Petri Net from a Partial Language.* In: Lecture Notes in Computer Science: Transactions on Petri Nets and Other Models of Concurrency, pp. 216–243, 2009.

10. R. Bergenthum, J. Desel, R. Lorenz, S. Mauser: *Process Mining Based on Regions of Languages.* In: Lecture Notes in Computer Science: Business Process Management, pp. 375–383, 2007.
11. R. J. Vanderbei: *Linear Programming: Foundations and Extensions.* Kluwer Academic Publishers, 1996.