Learning Workflow Petri Nets

Javier Esparza, Martin Leucker, and Maximilian Schlund

Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
{esparza,leucker,schlund}@in.tum.de

Abstract. Workflow mining is the task of automatically producing a workflow model from a set of event logs recording sequences of workflow events; each sequence corresponds to a use case or workflow instance. Formal approaches to workflow mining assume that the event log is complete (contains enough information to infer the workflow) which is often not the case. We present a learning approach that relaxes this assumption: if the event log is incomplete, our learning algorithm automatically derives queries about the executability of some event sequences. If a teacher answers these queries, the algorithm is guaranteed to terminate with a correct model. We provide matching upper and lower bounds on the number of queries required by the algorithm, and report on the application of an implementation to some examples.

1 Introduction

Modern workflow management systems offer modelling capabilities to support business processes [vdAvH04]. However, constructing a formal or semi-formal workflow model of an existing business process is a non-trivial task, and for this reason *workflow mining* has been extensively studied (see [vdAvDH$^+$03] for a survey). In this approach, information about the business processes is gathered in the form of logs recording sequences of workflow events, where each sequence corresponds to a use case. The logs are then used to extract a formal model. Workflow mining techniques have been implemented in several systems, most prominently in the ProM tool [vdAvDG$^+$07], and successfully applied.

Most approaches to process mining use a combination of heuristics and formal techniques, like machine learning or neural networks, and do not offer any kind of guarantee about the relationship between the business process and the mined model. Formal approaches have been studied using *workflow graphs* [AGL98] and *workflow nets* [vdA98, BDLM07] as formalisms. These approaches assume that the logs provide enough information to infer the model, i.e., that there is one single model compatible with them. In this case we the call the logs *complete*. This is a strong assumption, which often fails to hold, for two reasons: first, the number of use cases may grow exponentially in the number of tasks of the process, and so may the size of a complete set of logs. Second, many processes have "corner cases": unusual process instances that rarely happen. A complete set of logs must contain at least one instance of each corner case.

In this paper we propose a learning technique to relax the completeness assumption on the set of logs. In this approach the model is produced by a *Learner* that may ask questions to a *Teacher*. The Learner can have initial knowledge in the form of an initial set of logs; if the log contains enough information to infer the model, the Learner produces it. If not, it iteratively produces *membership queries* of the form: Does the business process have an instance (a use case) starting with a given sequence of tasks? For instance, in the standard example of complaint processing (see Figure 1 and [vdA98]), a membership query could have the form “Is there a use case in which first the complaint is registered and then immediately rejected?” The Teacher would answer no, because a decision on acceptance or rejection is made only after the customer has been sent a questionnaire.

Notice that the Learner does not guess the queries, they are automatically constructed by the learning algorithm. Under the assumption that the Teacher provides correct answers, the learning process is guaranteed to terminate with a correct model: a model whose executions coincide with the possible event sequences of the business process. In other words, we provide a formal framework with a correctness and completeness guarantee which only assumes the existence of the Teacher.

It could be objected that if a Teacher exists, then a workflow model must already exist, and there is no need to produce it. To see the flaw in this argument, observe that the Teachers can be employees, databases of client records, etc, that have knowledge about the process, but usually lack the modelling expertise required to produce a formal model. Our learning algorithm only requires from the Teacher the low-level ability to recognize a given sequence of process actions as the initial sequence of process actions of some use case.

It is useful to draw an analogy. Witnesses of a crime can usually answer questions about the physical appearance of the criminal, but they are very rarely able to draw the criminal’s portrait: this requires interaction with a police expert. This interaction can be seen as a learning process: the Teacher is the witness, and the Learner is the police expert. The teacher has knowledge about the criminal, but is unable to express it in the form of a portrait. The Learner has the expertise required to produce a portrait, but needs input from the Teacher. In the context of business processes,

Like [vdA98, KRS06, BDLM07, RGvdA^{+}07], we use *workflow nets*, introduced by van der Aalst, as formal model of business processes. Loosely speaking, a workflow net is a Petri net with a distinguished initial and final marking. Van der Aalst convincingly argues that well-formed business processes (an informal notion) correspond to *sound* workflow nets (a formal concept). A workflow net is *sound* [vdA98] if it is live and bounded. In this paper we follow van der Aalst’s ideas. Given a Teacher, we wish to learn a sound workflow net for the business process. It is easy to come up with a naive correct learning algorithm. However, a first naive complexity analysis yields that the number of queries necessary to learn a workflow net can be triple exponential in the number of tasks of the business process in the worst case. This seems to indicate that the approach is useless. However, we show how the special properties of sound workflow nets, together with a finer complexity analysis, lead to WNL, a new learning algorithm requiring a single exponential number of queries in the worst case. We also provide an exponential lower bound, showing that WNL is asymptotically optimal. Finally, in a number of experiments we show that despite the exponential worst-case complexity the algorithm is able to synthesize interesting workflows. Notice also that the complexity is analysed for the case in which no initial event log is provided, that is, the case in which all knowledge has to be extracted from the Teacher by asking membership queries.

Technically, the triple exponential complexity of the naive algorithm is a consequence of the following three facts:

(a) the size of a deterministic finite automaton (DFA) recognizing the language of a net with n transitions can be a priori double exponential in n;
(b) learning such a DFA using only membership queries requires exponentially many queries in the size of the DFA (follows from [Ang87] and [Vas73, Cho78]); and
(c) the algorithms of Darondeau et al. for synthesis of Petri nets from regular languages [BBD95] are exponential in the size of the DFA.

In the paper we solve (a) by proving that the size of the DFA is only single exponential; we solve (b) by exhibiting a better learning algorithm for sound workflow nets requiring only polynomially many queries; finally, we solve (c) by showing that for sound workflow nets the algorithms for synthesis of Petri nets from regular languages can be replaced by the algorithms for synthesis of bounded nets from minimal DFA, which are of polynomial complexity. Notice that our solution very much profits from the restriction to sound workflow nets, but that this restriction is given by the application domain: that sound workflow nets are an adequate formalization of well-formed business processes has been proved by the large success of the model in both the workflow modelling and Petri net communities.

Outline In the next section, we fix the notation of automata, recall the notion of Petri nets and workflow nets, and cite results on synthesis of Petri nets from automata. Our learning algorithm WNL is elaborated in Section 3. Section 4 reports on our implementation and experimental results. Finally, we sum up our contribution in the conclusion.

2 Preliminaries

We assume that the reader is familiar with elementary notions of graphs, automata and net theory. In this section we fix some notations and define some less common notions.

Automata and Languages A deterministic finite automaton (DFA) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of *states*, Σ is a finite *alphabet*, $q_0 \in Q$ is the *initial state*, $\delta \colon Q \times \Sigma \to Q$ is the (partial) *transition function* and

$F \subseteq Q$ is the set of *final states*. We denote by $\widehat{\delta}$ the function $\widehat{\delta}\colon Q \times \Sigma^* \to Q$ inductively defined by $\widehat{\delta}(q, \epsilon) = q$ and $\widehat{\delta}(q, wa) = \delta(\widehat{\delta}(q, w), a)$. The language $\mathcal{L}(q)$ of some state $q \in Q$ is the set of words $w \in \Sigma^*$ such that $\widehat{\delta}(q, w) \in F$. The language recognized by a DFA A is defined as $\mathcal{L}(A) := \mathcal{L}(q_0)$. A language is *regular* if it is accepted by some DFA.

Myhill-Nerode's theorem and minimal DFAs Given a language $L \subseteq \Sigma^*$, we say two words $w, w' \in \Sigma^*$ are *L-equivalent*, denoted by $w \sim_L w'$, if $wv \in L \Leftrightarrow w'v \in L$ for every $v \in \Sigma^*$. The language L is regular iff L-equivalence partitions Σ^* into a finite number of equivalence classes. Given a regular language L, there exists a unique DFA A up to isomorphism with a minimal number of states such that $\mathcal{L}(A) = L$; this automaton A is called the *minimal DFA* for L. The number of states of this automaton recognizing is equal to the number of equivalence classes.

Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$, we say two states $q, q' \in Q$ are *A-equivalent* if $\mathcal{L}(q) = \mathcal{L}(q')$. We can quotient A with respect to this equivalence relation. The states of the quotient DFA are the equivalence classes of $\sim_A$. The transitions are defined by "lifting" the transitions of A: for every transition $q \xrightarrow{a} q'$, add $[q] \xrightarrow{a} [q']$ to the transitions of the quotient DFA, where $[q]$ and $[q']$ denote as the equivalence classes of q and q'. The initial state is $[q_0]$, and the final states are $\{[q] \mid q \in F\}$. The quotient DFA recognizes the same language as A, and is isomorphic to the minimal DFA recognizing L.

It is easy to see that the minimal automaton for a prefix-closed regular language has a unique non-final state (a trap state). For simplicity, we sometimes identify this automaton with the one obtained by removing the trap state together with its ingoing and outgoing transitions.

Petri Nets A (marked) Petri net is a 5-tuple $N = (P, T, F, W, m_0)$ where P is a set of *places*, T is a set of *transitions* with $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*, $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a *weight function* satisfying $W(x, y) > 0$ iff $(x, y) \in F$, and $m_0 : P \to \mathbb{N}$ is a mapping called the *initial marking*.

For each transition or place x we call the set ${}^\bullet x := \{y \in P \cup T :\ (y, x) \in F\}$ the *preset* of x. Analogously we call $x^\bullet := \{y \in P \cup T :\ (x, y) \in F\}$ the *postset* of x. A net is *pure* if no transition belongs to both the pre- and postsets of some place.

Given an arbitrary but fixed numbering of P and T, the *incidence matrix* of N is the $|P| \times |T|$-matrix C given by: $C(p_i, t_j) = W(t_j, p_i) - W(p_i, t_j)$.

A transition $t \in T$ is *enabled* at a marking m, if $\forall p \in {}^\bullet t : m(p) \geq W(p, t)$. If a transition t is enabled it can *fire* to produce the new marking m', written as $m \xrightarrow{t} m'$.

$$m'(p) := m(p) + \sum_{p' \in P} C(p', t)$$

Given $w = t_1 \cdots t_n \in T^*$ (i.e. $t_i \in T$), we write $m_0 \xrightarrow{w} m$ if there exist markings $m_1, \ldots, m_{n-1}$ such that $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \ldots m_{n-1} \xrightarrow{t_n} m$. Then, we say that m is *reachable*. The set of reachable markings of N is denoted by $\mathcal{M}(N)$ and defined by $\mathcal{M}(N) = \{m : \exists w \in T^*.\ m_0 \xrightarrow{w} m\}$. It is well-known that if $m_0 \xrightarrow{w} m$, then $m = m_0 + C \cdot P(w)$, where $P(w)$, the *Parikh vector* of w, is the vector of dimension $|T|$ having as i-th component the number of times that t_i occurs in w. We call this equality the *marking equation*.

A net N is *k-bounded* if $m(p) \leq k$ for every reachable marking m and every place p of N, and *bounded* if it is k-bounded for some $k \geq 0$. A 1-bounded net is also called *safe*. A net is *reversible* if for every firing sequence $m_0 \xrightarrow{w} m$ there is a sequence v_w leading back to the initial state, i.e. $m \xrightarrow{v_w} m_0$. N is *live* if every transition can fire eventually at every marking, i.e. $\forall m \in \mathcal{M}(N) \exists w_m . m \xrightarrow{w_m t} m'$ for some m'.

The *reachability graph* of a net $N = (P, T, F, W, m_0)$ is the directed graph $G = (V, E)$ with $V = \mathcal{M}(N)$ and $(x, y) \in E$ iff $x \xrightarrow{t} y$ for some $t \in T$. If G is finite, then the five-tuple $A(N) = (Q, \Sigma, \delta, q_0, F)$ given by $Q = \mathcal{M}(N)$, $\Sigma = T$, $q_0 = m_0$, $F = Q$ and $\delta(m, t) := m'$ if m $\xrightarrow{t} m'$ is a DFA, and undefined otherwise. (Note that δ is well-defined, because if $m \xrightarrow{t} m'$ and $m \xrightarrow{t} m''$ then $m' = m''$.) We call it the *marking-DFA* of N. The *language* of N, denoted by $\mathcal{L}(N)$, is defined as the language of $A(N)$.

Workflow nets Loosely speaking, a workflow net is a Petri net with two distinguished input and output places without input and output transitions respectively, and such that the addition of a "reset" transition leading back from the output to the input place makes the net strongly connected (see Figure 1, for example). Formally, a net $N = (P, T, F, W, m_0)$ is a *workflow net* if there exist places $i, o \in P$ such that ${}^\bullet i = \emptyset = o^\bullet$, $m_0(p) = 1$ for $p = i$ and $m_0(p) = 0$, otherwise, and the net $\tilde{N} = (P, T \cup \{\mathbf{r}\}, F \cup \{(o, \mathbf{r}), (\mathbf{r}, i)\}, W \cup \{(o, \mathbf{r}) \mapsto 1, (\mathbf{r}, i) \mapsto 1\}, m_0)$, where $\mathbf{r} \notin T$, is strongly connected.

A firing sequence w of a workflow net N is a *run* if $m_0 \xrightarrow{w\mathbf{r}} m_0$ in $\tilde{N}$. The runs of N are the formalization of the use cases of the business process modelled by the workflow net. A workflow net N is *sound* if $\tilde{N}$ is live and bounded. It is argued in [vdA98] that a well-formed business process can be modelled by a sound workflow net (at a certain level of abstraction). The workflow net in Figure 1 is a very simple model for processing complaints (a slightly altered example, taken from [vdAvH04])

The following lemma characterizes soundness. In the paper we work with this characterization as definition.

Lemma 1. *A workflow net N is sound iff $\tilde{N}$ is bounded, reversible, and for every transition t there is a reachable marking m such that m enables t.*

Proof. Let $N = (P, T, F, W, m_0)$ be workflow net.
($\Rightarrow$): Assume N is sound. Then $\tilde{N}$ is bounded and live. We show $\tilde{N}$ is reversible. Let m be an arbitrary reachable marking of $\tilde{N}$. Then $m_0 \xrightarrow{w} m$ for some $w \in (T \cup \{\mathbf{r}\})^*$. Since $\tilde{N}$ is live, there is a firing sequence w such that $m \xrightarrow{w\mathbf{r}} m'$ for some marking m'. We claim $m' = m_0$. Assume $m' \neq m_0$. Then, since $m'(i) > 0$,

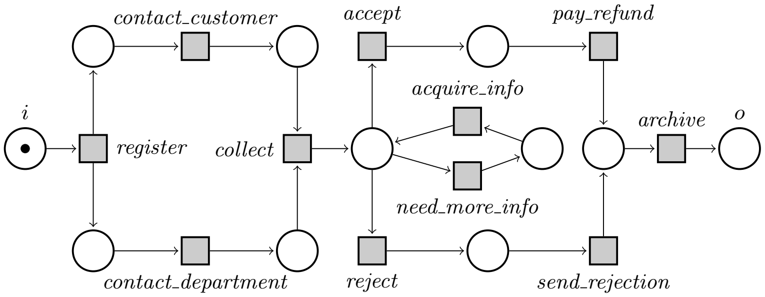


Fig. 1. An example for a sound workflow net (drawn without the reset transition $\mathbf{r}$)

we have $m'(p) \geq m_0(p)$ for every place p, and $m'(p) > m_0(p)$ for some p. So m' strictly covers m_0, and so N is not bounded.

($\Leftarrow$): Assume $\tilde{N}$ is bounded, reversible and every transition is enabled at some reachable marking. We show that $\tilde{N}$ is live, which implies that N is sound. Let m be an arbitrary reachable marking of $\tilde{N}$, and let $t \in T \cup \{\mathbf{r}\}$. Since $\tilde{N}$ is reversible, $m \xrightarrow{w} m_0$ for some $w \in (T \cup \{\mathbf{r}\})^*$, and since t occurs in some firing sequence $m_0 \xrightarrow{vt} m'$ for some $v \in (T \cup \{\mathbf{r}\})^*$ and some m'. So $\tilde{N}$ is live (and bounded by assumption) and therefore N is sound.

Synthesis of Petri nets from Languages and from Automata In [BBD95], Darondeau et al. address two synthesis problems of Petri nets from a minimal DFA A over an alphabet T:

(S1) Decide if there is a bounded net N with T as set of transitions such that $\mathcal{L}(N) = \mathcal{L}(A)$, and if so return one. We call this problem *synthesis up to language equivalence.*

(S2) Decide if there is a bounded net N with T as set of transitions such that the reachability graph of N is isomorphic to A, and if so return one. We call this problem *synthesis up to isomorphism.*

The algorithm of [BBD95] for synthesis up to language equivalence works in two phases: firstly, A is transformed into an equivalent automaton A' in a certain normal form. In the worst case, A' can be exponentially larger than A. The second phase constructs the net N, if it exists, in polynomial time in A'. The algorithm requires *exponential time* in A. The algorithm of [BBD95] for synthesis up to isomorphism, on the contrary, needs only *polynomial time* in A. Notice that, in general, if one knows the language $\mathcal{L}(N)$ of a net, one does not know yet its reachability graph. In particular, the minimal automaton recognizing $\mathcal{L}(N)$ may not be the reachability graph of any net. The basic algorithm in [BBD95] can only handle pure nets, but there is also a generalization to non-pure nets to be found in [BDBM96].

Hints on how to obtain nets that are more "visually appealing" (i.e. have few arcs, no redundant places, etc.) than those generated by standard synthesis algorithms can be found in [BDKM08], where net synthesis was applied to process mining from event logs.

3 A Learning Algorithm for Sound Workflow Nets

Our goal is to develop a learning algorithm for sound workflow nets which is guaranteed to terminate, and in which a teacher only needs to answer membership queries.

The precise learning setting is as follows. We have a *Leaner* and a *Teacher*. The Learner is given a set T of transitions, where each transition corresponds to a dedicated *task* (in the sense of [vdA98]) of the business process. The Learner repeatedly asks the Teacher *workflow membership queries*. A query is a sequence $\sigma \in T^*$, and is answered by the Teacher as follows: if σ can be extended to a use case (i.e., a sequence corresponding to a complete instance of the business process), then the Teacher returns this use case in the form of a transition sequence $\sigma\tau\mathbf{r}$, where $\tau \in T^*$. Otherwise, the Teacher answers "no". In our running example the Learner is given the set of transitions of the net of Figure 1, and the Teacher's answers are compatible with this net, i.e., acts as if it knew the net. Note that in practice, this only means that the Teacher can either extend the query to a use case of the net to learn or can reject the query. Two possible queries are

register contact_customer contact_department
register contact_customer collect

A possible answer to the first query is the run

register contact_customer contact_department collect accept pay_refund archive

while the answer to the second query is "no".

Assuming that the Teacher's answers are compatible with a k-bounded and reversible net N, the goal of the Learner is to produce a net N' such that $\mathcal{L}(N) = \mathcal{L}(N')$. It is easy to see that a (very inefficient) learning algorithm exists:

(1) A net with n transitions has at most $c_1 := 2^{(n+1)}$ places, because a place is determined by its pre- and post-sets of transitions.

(2) By (1), N has at most $c_2 := (k+1)^{c_1}$ reachable markings. Therefore, there exists a minimal DFA A with at most c_2 states such that $\mathcal{L}(N) = \mathcal{L}(A)$.

(3) Since any two prefix-closed minimal DFAs with c_2 states differ in some word of length c_2, the automaton A can be learned by querying all words over T of length $2c_2$, i.e., after at most $c_3 := n^{2c_2}$ queries.
This follows easily from Myhill-Nerode's theorem. The DFA A can be constructed from the answers to the queries as follows. The states of A are the equivalence classes of words of $\mathcal{L}(N)$ of length up to c_2, where two words w, v are equivalent if for every word u of length up to c_2 either wu and vu belong to $\mathcal{L}(N)$, or none of them does [Vas73, Cho78].) The initial state is the equivalence class of the empty word, and all states are final. There is a transition $[w] \xrightarrow{a} [wa]$ for every word w of length at most c_2.

(4) The net N is obtained from A by means of the algorithm of [BBD95] for synthesis up to language equivalence (see problem (S1) in Section 2). The algorithm runs in $2^{\mathcal{O}(p(c_2))}$ time for some polynomial p.

The query complexity of this naive algorithm, i.e. the number of queries it needs to ask, is triple exponential in the number n of transitions. In this section we prove a series of results ending in an improved algorithm with single exponential query and time complexity (notice that single exponential time complexity implies single exponential query complexity, but not vice versa).

3.1 An Upper Bound on the Number of Reachable Markings

We show that the naive bound on the number of states of A obtained in (2) above, which is double exponential in n, can be improved to a single exponential bound.

Given a net $N = (P, T, F, W, m_0)$ with incidence matrix C, we denote by $C(p)$ the vector $(C(p, t_1), \ldots, C(p, t_{|T|}))$. We say that a place p is a *linear combination* of the places $p_1, \ldots, p_k$ if there are real numbers $\lambda_1, \ldots, \lambda_k$ such that $C(p) = \sum_{i=1}^{k} \lambda_i \cdot C(p_i)$.

The following lemma is well known.

Lemma 2. *Let $N = (P, T, F, W, m_0)$ be a net with incidence matrix C, and let $C(p) = \sum_{i=1}^{k} \lambda_i C(p_i)$. Then for every reachable marking m: $\forall p \in P.\ m(p) = m_0(p) + \sum_{i=1}^{k} \lambda_i (m(p_i) - m_0(p_i))$.*

Proof. Since m is reachable, there is $w \in T^*$ such that $m_0 \xrightarrow{w} m$. By the marking equation $m = m_0 + C \cdot P(w)$, and so in particular $m(p) = m_0(p) + C(p) \cdot P(w)$, and $m(p_i) = m_0(p_i) + C(p_i) \cdot P(w)$ for every $1 \leq i \leq k$. So $m(p) = m_0(p) + \sum_{i=1}^{k} \lambda_i C(p_i) \cdot P(w) = m_0(p) + \sum_{i=1}^{k} \lambda_i (m(p_i) - m_0(p_i))$

Theorem 1. *Let $N = (P, T, F, W, m_0)$ be a k-bounded net with n transitions. Then N has at most $(k+1)^n$ reachable markings.*

Proof. The incidence matrix C has $|P|$ rows and n columns, and so it has rank at most n. So there are l places $p_1, \ldots, p_l$, $l \leq n$, such that $C(p_1), \ldots, C(p_l)$ are linearly independent. So every place p is a linear combination of $p_1, \ldots, p_l$. It follows from Lemma 2 that for every two reachable markings m, m', if $m(p_i) = m'(p_i)$ for every $1 \leq i \leq l$, then $m(p) = m'(p)$ for every place p. In other words, if two markings coincide on all of $p_1, \ldots, p_l$, they are equal. Since for every reachable marking m we have $0 \leq m(p_i) \leq k$, the number of projections of the reachable markings onto the places $p_1, \ldots, p_l$ is at most $(k+1)^l \leq (k+1)^n$. So N has at most $(k+1)^n$ reachable markings.

3.2 Minimality of the marking-DFA

We show that the marking-DFA of a bounded and reversible net is minimal. Since our goal is to construct a bounded and reversible net model N of the business process, after we learn the minimal DFA A with $\mathcal{L}(A) = \mathcal{L}(N)$ in step (3), we can can synthesize N by applying the algorithm of [BBD95] for synthesis up to isomorphism (Problem (S2)), instead of the algorithm for synthesis up to language equivalence (Problem (S1)). This eliminates one exponential from step (4) of the naive algorithm.

The proof is based on Lemma 3 below. Readers familiar with Myhill-Nerode's theorem (see also Section 2) will probably need no proof, but we include one for completeness. Recall that we identify a DFA with a single trap state with the one obtained by removing the trap state together with its ingoing and outgoing transitions.

Lemma 3. *A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is minimal iff the following two conditions hold:*

(1) every state lies in a path leading from q_0 to some state of F, and
(2) $\mathcal{L}(q) \neq \mathcal{L}(q')$ for every two distinct states $q, q' \in Q$.

Proof. ($\Rightarrow$): We prove the contrapositive. For (1), if some state q does not lie in any path from q_0 to some final state, then it can be removed without changing the language, and so A is not minimal. For (2), if two distinct states q, q' of A satisfy $\mathcal{L}(q) = \mathcal{L}(q')$, then $[q] = [q']$, and so the quotient automaton has fewer states than A. So A is not minimal.

($\Leftarrow$): Assume (1) and (2) hold. We prove that for every state q the language of the words w such that $\delta(q_0, w) = q$ is an equivalence class of L-equivalence. It follows that the number of states of A is at most as large as the number of equivalence classes of L-equivalence, which implies that A is the minimal DFA for L.

It suffices to show:

- If $\widehat{\delta}(q_0, w) = q = \widehat{\delta}(q_0, v)$, then $w \sim_L v$.
  This follows immediately from the definition of L-equivalence.
- If $\widehat{\delta}(q_0, w) = q$ and $\widehat{\delta}(q_0, v) = q'$ for some $q' \neq q$, then $w \not\sim_L v$.
  Since $\mathcal{L}(q) \neq \mathcal{L}(q')$, w.l.o.g. there is a word $u \in \mathcal{L}(q) \setminus \mathcal{L}(q')$. So $wu \in L$ and $vu \notin L$, which implies $w \not\sim_L v$.

Theorem 2. *Let $N = (P, T, F, W, m_0)$ be a bounded and reversible Petri net. The marking-DFA $A(N)$ of N is a minimal DFA.*

Proof. Assume that $A(N)$ is not minimal. Since every state of $A(N)$ is final, by Lemma 3 there are two states of $A(N)$, i.e., two reachable markings $m_1 \neq m_2$ of N, such that $\mathcal{L}(m_1) = \mathcal{L}(m_2)$. As $m_1 \neq m_2$ there exists $p \in P$ with $m_1(p) \neq m_2(p)$. Assume w.l.o.g. $m_1(p) < m_2(p)$. Let m be a reachable marking such that $m(p)$ is minimal, i.e. there is no other reachable marking m' s.t. $m'(p) < m(p)$. Since m is reachable and N is reversible, there is $w \in T^*$ such that $m_2 \xrightarrow{w} m$. Since $\mathcal{L}(m_1) = \mathcal{L}(m_2)$, there is a reachable marking m' such that $m_1 \xrightarrow{w} m'$. It follows

$$m'(p) = m_1(p) + C(p) \cdot P(w) < m_2(p) + C(p) \cdot P(w) = m(p)$$

contradicting the minimality of $m(p)$.

3.3 Learning the reachability graph by Exploration

The final step towards a single exponential learning algorithm consists of improving the naive algorithm of step (3) for learning the minimal DFA A. Recall that we assume that the Teacher's answers are compatible with a k-bounded and reversible net N. If n and r are the number of transitions and reachable markings of N, then the naive algorithm requires n^r membership queries. We present a new algorithm that requires only $\mathcal{O}(n \cdot r^2)$ queries.

Recall the standard search approach for constructing the reachability graph of a net *if the net is known*. We maintain a queue of markings, initially containing the initial marking, and two sets of already visited markings and transitions (transitions between markings). While the queue is non-empty, we take the marking m at the top of the queue, and check for each transition a whether a is enabled at m. If so, we compute the marking m' such that $m \xrightarrow{a} m'$, and proceed as follows: if m' has been already visited, we add $m \xrightarrow{a} m'$ to the set of visited transitions; if m' had not been visited yet, we add m' to the set of visited markings and to the queue, and add $m \xrightarrow{a} m'$ to the set of visited transitions.

Our learning algorithm closely mimics this behaviour, but works with firing sequences of N instead of reachable markings (the Learner does not know the markings of the net, it does not even know its places). We maintain a queue of firing sequences, initially containing the empty sequence, and two sets of already visited firing sequences and transitions. While the queue is non-empty, we take the firing sequence $w \in (T \cup \{\mathbf{r}\})^*$ at the top of the queue, and ask the Teacher for each transition a whether wa is also a firing sequence of N. If so, we proceed as follows. We first determine whether each already visited firing sequence u leads to the same marking as wa. Notice that it is not obvious how to do this—this is the key of the learning algorithm. If some firing sequence u leads to the same marking as wa, then we add $w \xrightarrow{a} u$ to the set of visited transitions; otherwise, we add wa to the set of visited firing sequences and to the queue, and add $w \xrightarrow{a} wa$ to the set of visited transitions. The algorithm in pseudo code can be found below (Algorithm 1), where $\text{Equiv}(u, v)$ denotes that there is a marking m such that $m_0 \xrightarrow{u} m$ and $m_0 \xrightarrow{v} m$.

The correctness of the algorithm is immediate: we just simulate a search algorithm for the construction of the reachability graph, using a firing sequence u to represent the marking m such that $m_0 \xrightarrow{u} m$. The check $\text{Equiv}(u, wa)$ guarantees that each marking gets exactly one representative.

The problem is to implement $\text{Equiv}(u, wa)$ using only membership queries. In general this is no easy task, but in the case of reversible nets it can be easily done as follows. When checking $\text{Equiv}(u, wa)$ the word u has been already added to V, and so the Learner has established that $u \in \mathcal{L}(N)$. So in particular the Teacher has answered positively a query about u and, due to the structure of workflow membership queries, it *has returned a run* uu_c, where $u_c\mathbf{r}$ is a transition sequence leading back to the initial marking.

We prove that $\text{Equiv}(x, y)$ holds if and only if the sequence xy_c is a run of N:

Algorithm 1: Learning the reachability graph

```
Output: graph $(V, E)$ isomorphic to the reachability graph of $N$
$V \longleftarrow \emptyset$; $E \longleftarrow \emptyset$
$F \longleftarrow \{\epsilon\}$      // queue of firing sequences
while not $F$.empty() do
    $w \longleftarrow F$.dequeue()
    forall $a \in T$ do
        if $wa$ is accepted by the Teacher then
            /* This means $wa \in \mathcal{L}(N)$ */
            $\sigma \leftarrow wa$
            forall $u \in V$ do
                if Equiv($u, wa$) then $\sigma \leftarrow u$
            end
            if $\sigma = wa$ then $F$.enqueue($wa$)
            add $\sigma$ to $V$ and $w \xrightarrow{a} \sigma$ to $E$
        end
    end
end
```

Proposition 1. *In Algorithm 1, Equiv(u, wa) =* **true** *if and only if uw_c is a run of N, where waw_c is the run reported by the Teacher when positively answering the query about $wa \in \mathcal{L}(N)$.*

Proof. If Equiv(u, wa) = **true**, then there is a marking m such that $m_0 \xrightarrow{u} m$ and $m_0 \xrightarrow{wa} m$. Because $m_0 \xrightarrow{wa} m \xrightarrow{w_c\mathbf{r}} m_0$, we have $m_0 \xrightarrow{u} m \xrightarrow{w_c\mathbf{r}} m_0$, which implies that uw_c is a run.

If $u \cdot w_c$ is a run, then we have $m_0 \xrightarrow{waw_c\mathbf{r}} m_0$ and $m_0 \xrightarrow{uw_c\mathbf{r}} m_0$. Let m be the marking such that $m_0 \xrightarrow{wa} m$. We then have $m \xrightarrow{w_c\mathbf{r}} m_0$. Moreover, m is the only marking such that $m \xrightarrow{w_c\mathbf{r}} m_0$ (Petri nets are backward deterministic: given a firing sequence and its target marking, the source marking is uniquely determined). Since $m_0 \xrightarrow{uw_c\mathbf{r}} m_0$, we then necessarily have $m_0 \xrightarrow{u} m \xrightarrow{w_c\mathbf{r}} m_0$, and so in particular $m_0 \xrightarrow{u} m$. So both wa and u lead to the same marking m, and we have Equiv(u, wa) = **true**.

We can now easily show that checking Equiv(u, wa) reduces to one single membership query.

Proposition 2. *The check Equiv(u, wa) can be performed by querying whether $uw_c \in \mathcal{L}(N)$: Equiv(u, wa) holds if and only if the Teacher answers positively and returns the sequence uw_c itself as a run.*

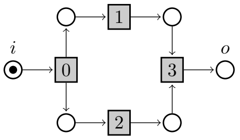
Proof. There are three possible cases:

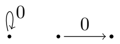
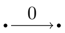
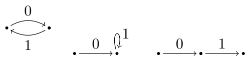
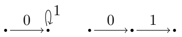
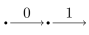
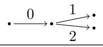
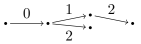
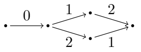
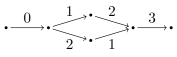
- The answer is negative.
  Then $uw_c \notin \mathcal{L}(N)$, and so in particular it is not a run of N. So Equiv(u, wa) = **false**.

- The answer is positive *and* the Teacher returns uw_c as run.
  Then Equiv(u, wa) = **true** by Proposition 1.
- The answer is positive, but the Teacher returns uw_cv for some $v \neq \epsilon$ as run. Since the Teacher returns a run uw_cv such that no proper prefix uw_cv' is a run, we have in particular by taking $v' = \epsilon$ that uw_c is not a run. By Proposition 1 we have Equiv(u, wa) = **false**.

Remark 1. In anticipation to the experiments described in Section 4, let us mention that in many cases the queries for uw_c do not even have to be submitted to the teacher. (recall that wa labels the potentially new state and u labels a known state). Often we can deduce that uw_c is not fireable by observing that $uw_c \notin L(A)$ where A is the part of the DFA that is already known. If we would query wau_c instead (which would also tell us if Equiv(u, wa) = **true**) we would not be able to discard any query because the neighbourhood of wa has not yet been explored. This is one of the reasons why this algorithm is so efficient in practice (cf. Section 4).

Example 1. We now provide an example run of our algorithm, applied to the first part of the net in figure 1. To simplify presentation we grouped together some queries which correspond to the interesting stages of the algorithm ($w \cdot A$ are all queries wa with $a \in A$).



#	Query	Answer	Possible Automata
1-4	$\varepsilon \cdot \{0,1,2,3\}$	0(123)	
5	Equiv(ε, 0)? $\rightsquigarrow$ $\varepsilon \cdot 123$	no	
6-8	$0 \cdot \{0,1,3\}$	01(23)	3 Possibilities: 
9	Equiv(ε, 01)? $\rightsquigarrow$ $\varepsilon \cdot 23$	no	
10	Equiv(0, 01)? $\rightsquigarrow$ $0 \cdot 23$	no	
11	02	02(13)	(4 Possiblities)
12	Equiv(ε, 02)? $\rightsquigarrow$ $\varepsilon \cdot 13$	no	(3 Poss.)
13	Equiv(0, 02)? $\rightsquigarrow$ $0 \cdot 13$	no	(2 Poss.)
14	Equiv(01, 02)? $\rightsquigarrow$ $01 \cdot 13$	no	
15-18	$01 \cdot \{0,1,2,3\}$	012(3)	(5 Poss.)
19-22	Equiv? ($\{\varepsilon,0,01,02\}$, 012)	no	
23-26	$02 \cdot \{0,1,2,3\}$	021(3)	(6 Poss. - naive)
27	[Equiv(021,012)?]	yes	
28-31	$012 \cdot \{0,1,2,3\}$	0123(ε)	(7 Poss. - naive)
32	[Equiv-Queries]	no	

The “Answer”-column contains the run ww_c returned by the teacher, if $w \in \mathcal{L}(N)$ - we put the continuations w_c in brackets. As observed in Remark **??**, many queries (like ”23” in # 9) do not really have to be asked - either because we already asked a prefix of the query that was rejected, or because the query is a prefix of a run supplied by the teacher and therefore we already know that is is accepted. We also do not need to ask query # 27 because 021 and 012 have the same Parikh vector and therefore must lead to the same marking.

There is a technical issue that should be mentioned at this point. The algorithm delivers a net N' such that the reachability graphs of N and N' are isomorphic. It follows that N' is reversible and bounded. However, we cannot guarantee that N' has the same bound as N. We consider this a minor problem, since N' and N are for behavioural purposes equivalent models.

Complexity It follows clearly from the description of Algorithm 1 that the number of firing sequences added to the queue is equal to the number of reachable markings r of N. For the i-th sequence taken from the queue, say w, and for each transition, say a, we perform at most i membership queries: one to check if $wa \in \mathcal{L}(N)$, and at most $(i-1)$ for checks $\mathrm{Equiv}(u, wa)$, because at that point V contains at most $i-1$ elements. So the algorithm performs at most $\sum_{i=1}^{r} n \cdot i = nr(r+1)/2 \in \mathcal{O}(n \cdot r^2)$ queries.

The following theorem sums up the results of the section.

Theorem 3 (Learning by Exploration). *We can learn a k-bounded and reversible net N with a number of workflow membership queries and a running time that are single exponential in the number of transitions of N.*

The proof follows easily form our discussion. The overall algorithm, that we call WNL, uses the learning technique of Section 3.3 to learn a minimal DFA A such that $\mathcal{L}(A) = \mathcal{L}(N)$. Section 3.1 shows that A is single exponential in the number of transitions of N, and so it can be learned with a single exponential number of queries. Section 3.2 shows that this minimal DFA is (isomorphic to) the reachability graph of N. We can then apply the polynomial algorithm of [BBD95] for synthesis up to isomorphism (S2).

A final question is what happens if the Teacher's answers are not compatible with any k-bounded and reversible net N. In this case there are two possibilities: they are not compatible with any minimal DFA having at most $(k+1)^n$ states, or they are compatible with some such DFA, but this DFA is not the marking-DFA of any net. In the first case the algorithm can stop when the number of generated states exceeds $(k+1)^n$. In the second case, the algorithm terminates and produces a DFA, but the synthesis algorithm of [BBD95] does not return a net.

3.4 Mixing process mining and learning

The algorithm we have just presented does not assume the existence of an event log: the Learner only gets information from membership queries. However, as explained in the introduction, we consider our learning approach as a way of complementing log-based process mining. In this section we explain how to modify the algorithm accordingly.

We assume the existence of an event log consisting of use cases. Given the set of tasks T of the business process, we can think of each use case as a word $w \in T^*$, such that w corresponds to a run of the reversible net to be learnt. The event log then corresponds to a language $L \subseteq T^*$.

In a first step we construct the minimal DFA for the language L. This can be done space-efficiently in a number of ways. For instance, we can divide the set of runs in two halves L_1, L_2, recursively compute minimal DFAs A_1, A_2 recognizing

L_1 and L_2, and then compute the minimal DFA for L from A_1, A_2 using an algorithm very similar to the one that computes the union of two binary decision diagrams [And99]. Once this is done, we easily get the minimal DFA A for the language of prefixes of $(L\mathbf{r})^*$ (this requires to add one extra state and make all states final).

Once A is computed, we assign to each state q of A a word w_q such that $q_0 \xrightarrow{w_q} q$. For every two states q_1, q_2, we check whether the states correspond to the same reachable marking by calling $\mathrm{Equiv}(w_{q_1}, w_{q_2})$. After this step we are in the same situation we would have reached if the algorithm would have queried all the words w_q.

From a practical point of view, notice that it is very inefficient to ask the Teacher for each pair of states q_1, q_2 whether $\mathrm{Equiv}(w_{q_1}, w_{q_2})$. A better procedure is to ask the Teacher, given a sequence w, which are the letters a such that wa can be extended to a use case. We call them the *possible extensions* of w. The test $\mathrm{Equiv}(w_{q_1}, w_{q_2})$ need only be carried out for sequences w_{q_1}, w_{q_2} having the same set of extensions. Note that the teacher does not have to provide full runs for any of these possible extensions so this is quite a simple task.

We can even more reduce the number of calls to Equiv() by first merging states for which we can already deduce that they have to be equivalent. Some criteria, which are easy properties of Petri nets, and can be directly used to trim a DFA that was generated from event logs are:

- The DFA is backward deterministic: if $m_1 \xrightarrow{a} m_3$ and $m_2 \xrightarrow{a} m_3$ for some $a \in T$ then $m_1 = m_2$
- If two words w_1,w_2 only differ in the order of their letters (i.e. their Parikh vectors coincide $P(w_1) = P(w_2)$) then they lead to the same state
- Given a k-bounded net N, if $vw^{k+1} \in \mathcal{L}(N)$ for some words v, w then w describes a cycle in the reachability graph of N

A further criterion *for pure nets* is the “diamond property”: We can add transitions that have to be present due to basic Petri net properties. A *diamond* is a subgraph in the reachability graph of a net with four states that are connected in the following way: $m_1 \xrightarrow{a} m_2$, $m_1 \xrightarrow{b} m_3$, $m_2 \xrightarrow{b} m_4$, $m_3 \xrightarrow{a} m_4$. A diamond is *incomplete* if it is missing exactly one transition (see Figure 2). One can easily see that incomplete diamonds can always be completed with the missing transition (in the case of pure nets), i.e., if an incomplete diamond is found in the DFA, we can add the missing transition. This *diamond property* can also be used to merge states as indicated in Figure 2.

3.5 A Lower Bound for Petri Net Learning

We now show that we cannot in general solve the learning problem in subexponential time, by providing a hard-to-learn instance. We will show with the help of an adversary argument that any learning algorithm has to ask at least $\Omega(2^n)$ membership queries to derive the correct net, where n is the number of transitions.

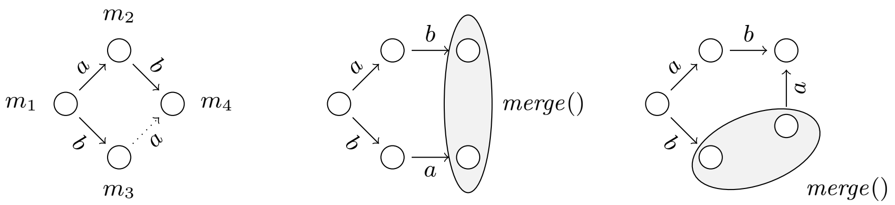


Fig. 2. Incomplete diamond (left), states merged because of equal parikh-vectors (middle) or by using the diamond property (middle and right)

Consider the following set $\mathcal{N}$ of workflow nets. All the nets in $\mathcal{N}$ have the same number $n+3$ of transitions: two transitions *init* and *final*, transitions called $a_1, \ldots, a_n$, and a transition t (see Figure 3). The pre- and postsets of all transitions but t, which are identical for all nets of $\mathcal{N}$, are shown in the figure. The postset of t is always the place o. The preset of t always contains for each i exactly one of the places p_i or q_i, and the only difference between two nets in $\mathcal{N}$ is their choice of p_i or q_i. Clearly, the set $\mathcal{N}$ contains 2^n workflow nets, all of them sound.

For each net $N \in \mathcal{N}$ there is exactly one subset of $\{a_1, \ldots, a_n\}$ such that t can fire after the transitions of the set have fired. We call this subset S_N. Notice that if we know S_N then we can infer ${}^\bullet t$.

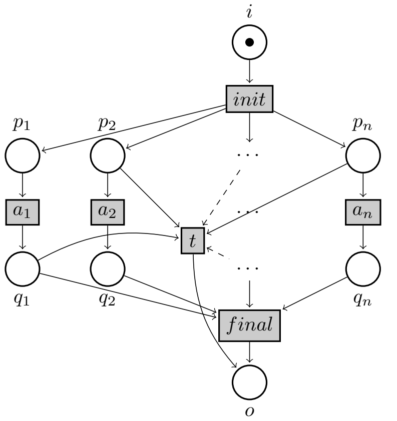


Fig. 3. Hard-to-learn instance for Petri net learning

We ease the task for the Learner by assuming she knows that the net to be learned belongs to $\mathcal{N}$. Her task consists only of finding out ${}^\bullet t$, or, equivalently, the set S_N.

A query of an optimal Learner has always the form $a_{i_1} a_{i_2} \cdots a_{i_k} t$, because querying any a_i after t does not provide the Learner with any information. Furthermore the order of the a_i is not important—all these transitions are independent and the Learner already knows this. So we can view a query just as a subset S of the set of all transitions. A negative answer to a query S always rules out *exactly one* of the nets of $\mathcal{N}$, namely the one in which ${}^\bullet t = S$. The worst case appears when the Learner ask queries "in the worst possible order", eliminating all nets of $\mathcal{N}$ but the right one. This requires $2^n - 1$ queries.

4 Practical experiences

To get insights in the practical feasibility of the derived algorithm WNL, we have developed a prototype learning and synthesis tool for workflow nets and examined its practical performance on a number of examples.

Implementation Our prototype is written in C++ with approximately 3,000 lines of code and uses libALF for dealing with automata. libALF is part of the automata learning factory currently developed jointly at RWTH Aachen and TU München[1].

The synthesis algorithm (S2) of [BBD95] is implemented using the lp_solve[2] framework to efficiently solve the linear programs needed for computing the places of the net. Furthermore lp_solve is used for eliminating redundant places after the net has been synthesized to reduce its size and to make it look more appealing. The implementation is currently not tailored to user interaction but consults pre-existing workflow nets for queries. Outputs are given in form of dot-files that can be visualized using the graphviz toolkit.

Experimental Results We tested our implementation on various examples of pure, safe and reversible nets. The examples range from existing sound workflow nets obtained in case studies performed by [Ver04] to more standard examples like mutual exclusion between processes and an n-cell buffer with 2^n reachable markings. The latter example is especially suitable to understand scalability issues of the algorithms. The ”absence” workflow is loosely modelled after an example from [SAP01], the ”complaint” workflow is the example presented in our background section (Figure 1).

We applied our implementation once without any event logs as initial knowledge and then again with randomly generated logs as input and counted the number of queries needed to learn the model. Besides counting the queries needed for Equiv(), we only count queries answered positively by the Teacher, as these correspond to runs supplied by him, and thus reflect the actual work to be done by an expert in an adequate manner.

To illustrate this, consider the task of learning the sequence of calendar months: instead of asking twelve questions of the form “Does January, February, . . . come after July?” (we call these ”small-step” queries) we would just ask “Which month comes after July?”. So we count every continuation provided by the teacher as one query. In the situation of Figure 4 we would count 2 workflow membership queries compared to 3 “small-step” queries. We have also included the number of “small-step” queries in the table below for comparison.

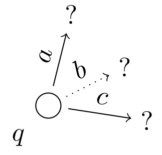


Fig. 4. Querying extensions at state q, possible extensions: solid arrows

[1] http://libalf.informatik.rwth-aachen.de/

[2] http://lpsolve.sourceforge.net/

Model	$\lvert T\rvert$	$\lvert RG\rvert$	ssq	WNL
buf_2	3	4	19	12
buf_3	4	8	52	32
buf_4	5	16	137	85
buf_5	6	32	344	216
buf_6	7	64	842	538
buf_7	8	128	2008	1304
buf_8	9	256	4707	3107
mutex_2	6	8	74	40
mutex_3	9	20	300	168
mutex_4	12	48	1026	594
order_simp	9	7	77	23
absence	11	8	109	32
complaint	12	11	155	37
transit1	25	77	2256	474

Fig. 5. Membership queries needed by WNL *without any event logs*; ssq = number of "small-step" queries, RG = reachability graph

We have first collected the number of membership queries needed by WNL when learning a model "from scratch" with respect to the size of the alphabet and the reachability graph, see Figure 5. On the chosen examples, the number of membership queries ranges between 12 and 3100. The series of the n-cell buffer examples from $n = 2$ to $n = 8$ suggests that the practical performance of WNL is even better than quadratic in the number of reachable markings.

Next, we studied the effect of learning workflow nets in the presence of existing logs. To this end, we used our tool to generate random event logs containing a varying number of runs (see Figure 6 for an example log). The runs in the generated log-files are not unique—runs that are more likely will probably appear multiple times, which is also the case for real-world event logs. For the random logs we calculated the average number of queries over 100 executions.

```
.a.b.a.c.d.b.c.a.d.b.c.d.
.a.b.c.d.
.a.b.c.a.b.d.c.a.d.b.c.d.
.a.b.a.c.d.b.c.d.
.a.b.a.c.b.a.d.c.d.b.c.d.
.a.b.c.a.d.b.a.c.d.b.c.d.
.a.b.c.a.b.d.c.d.
.a.b.a.c.b.d.a.c.d.b.c.d.
.a.b.c.d.
.a.b.c.d.
```

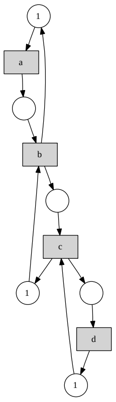


Fig. 6. Example event log for 3-cell buffer

We found out that for tiny models like the buffer with two cells or the "complaint" workflow a very small number of runs (< 10) suffices to already construct the model. The Teacher does not have to supply additional runs for these. Clearly, for larger models, we can only expect that the Teacher's work is reduced but not completely eliminated when logs are given. To illustrate the impact of event logs on the learning process we show how

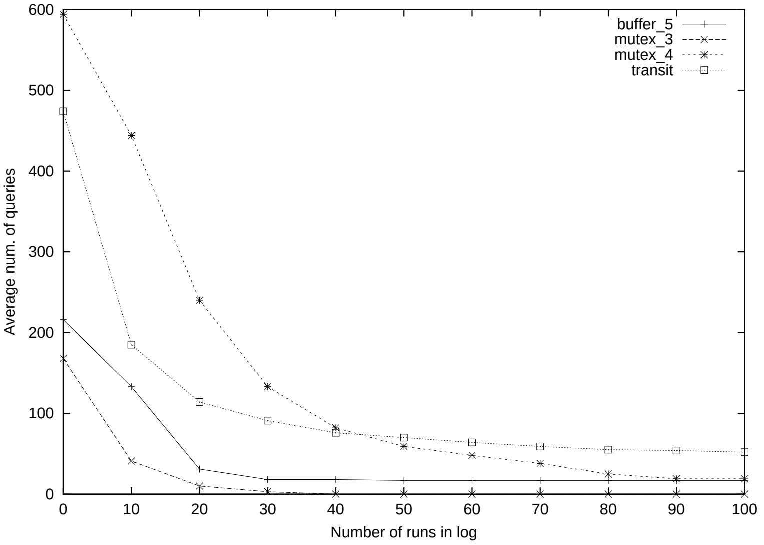


Fig. 7. Average number of queries needed by WNL applied to event logs of different sizes

the number of queries behaves for some of the larger models with logs of different sizes (see Figure 7).[3] We observe that already quite small logs drastically reduce the number of queries to be answered. At the same time, because our logs may not contain unique but many identical entries, larger logs contribute less and less new knowledge. This reflects the situation for real-life logs, which mostly contain common executions of a workflow but lack less common runs. In other words, it seems most promising for practical applications, to combine knowledge from (small) logs with that of Teachers responsible for "corner cases" to actually learn the workflow net in question efficiently.

The time needed for learning the nets in an applied setting is of course dominated by the number of queries a user has to answer. Synthesizing the resulting Petri net using the method proposed by Darondeau et al. (see Section 2) together with some post-processing to remove redundant places needs just a few seconds in the worst case and is therefore negligible.

The results depicted in Figures 5 and 7 suggest that, despite the seemingly intimidating result in Section 3.5, learning of workflow models is quite feasible for practical applications.

[3] Also larger examples behave in the same way, yet, we depicted models requiring a number of queries in the same order of magnitude to optimize the figure.

5 Conclusion

We have presented a new approach for mining workflow nets based on learning techniques. The approach palliates the problem of incompleteness of event logs: if a log is incomplete, our algorithm derives membership queries identifying the missing knowledge. The queries can be passed to an expert, whose answers allow to produce a model.

We have shown the correctness and completeness of our approach under the assumption of a teacher answering workflow membership queries. Starting with general combinatorial arguments showing that workflow models can in principle be learned, we have derived a learning algorithm requiring a single exponential number of queries in the worst case, and we have given a matching lower bound. We have also shown experimental evidence indicating that the combination of an event log, even of small size, and a Teacher responsible for providing information about "corner cases" allows to efficiently produce models in practically relevant cases.

There are several promising paths for further research. One aspect is the application of learning to the *design* of workflows. In this approach an expert on business processes and a modelling expert (or an adequate software) cooperate. The modelling expert asks queries about how the workflow should behave, which are answered by the Teacher, until a model accepted by the business process expert is produced. We expect to transfer ideas from the field of learning models of software systems [BKKL09] to workflow systems, and develop "teaching assistants" that filter the queries, automatically answering those for which the answer can be deduced from current information (for instance because it is known that two tasks must be concurrent), and only passing to the expert the remaining ones. Here we expect to profit from related work by Desel, Lorenz and others [BDML09]. An important point for process mining and even more for process design is designing fault tolerance techniques allowing to cope with false answers by the Teacher. Finally, learning more general classes of Petri nets, and applications to modelling/reconstruction of distributed systems, or biological/chemical processes, are also promising paths for future work.

References

[AGL98] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. In *EDBT*, volume 1377 of *LNCS*, pages 469–483. Springer, 1998.

[And99] Henrik Reif Andersen. An introduction to binary decision diagrams. Technical report, 1999, http://www.itu.dk/people/hra/bdd-eap.pdf

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[BBD95] Eric Badouel, Luca Bernardinello, and Philippe Darondeau. Polynomial algorithms for the synthesis of bounded nets. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 364–378, London, UK, 1995. Springer-Verlag.

[BDBM96] Eric Badouel and Philippe Darondeau. On the synthesis of general petri nets. Technical report, INRIA, 1996.

[BDKM08] Robin Bergenthum, Jörg Desel, Christian Kölbl, and Sebastian Mauser. Experimental results on process mining based on regions of languages. In *CHINA 2008, workshop at the Applications and theory of Petri nets : 29th international conference*, 2008.

[BDLM07] Robin Bergenthum, Jörg Desel, Robert Lorenz, and Sebastian Mauser. Process mining based on regions of languages. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *LNCS*, pages 375–383. Springer, 2007.

[BDML09] Robin Bergenthum, Jörg Desel, Sebastian Mauser, and Robert Lorenz. Construction of process models from example runs. *T. Petri Nets and Other Models of Concurrency*, 2:243–259, 2009.

[BKKL09] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning communicating automata from MSCs. *IEEE Transactions on Software Engineering (TSE)*, 2009. in press.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *TSE*, 4(3):178–187, May 1978. Special collection based on COMPSAC.

[KRS06] Ekkart Kindler, Vladimir Rubin, and Wilhelm Schäfer. Process mining and petri net synthesis. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, volume 4103 of *LNCS*, pages 105–116. Springer, 2006.

[RGvdA$^+$07] Vladimir Rubin, Christian W. Günther, Wil M. P. van der Aalst, Ekkart Kindler, Boudewijn F. van Dongen, and Wilhelm Schäfer. Process mining framework for software processes. In Qing Wang, Dietmar Pfahl, and David M. Raffo, editors, *ICSP*, volume 4470 of *LNCS*, pages 169–181. Springer, 2007.

[SAP01] SAP AG. *SAP Business Workflow Demo Examples (BC-BMT-WFM)*, 2001.

[Vas73] M. P. Vasilevski. Failure diagnosis of automata. *Cybernetic*, 9(4):653–665, 1973.

[vdA98] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[vdAvDG$^+$07] Wil M. P. van der Aalst, Boudewijn F. van Dongen, Christian W. Günther, R. S. Mans, Ana Karla Alves de Medeiros, Anne Rozinat, Vladimir Rubin, Minseok Song, H. M. W. (Eric) Verbeek, and A. J. M. M. Weijters. Prom 4.0: Comprehensive support for *eal* process analysis. In Jetty Kleijn and Alexandre Yakovlev, editors, *ICATPN*, volume 4546 of *LNCS*, pages 484–494. Springer, 2007.

[vdAvDH$^+$03] Wil M. P. van der Aalst, Boudewijn F. van Dongen, Joachim Herbst, Laura Maruster, Guido Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.

[vdAvH04] Wil van der Aalst and Kees van Hee. *Workflow Management. Models, Methods, and Systems.* MIT Press, 2004.

[Ver04] Henricus M.W. Verbeek. *Verification of WF-nets.* PhD thesis, Technische Universiteit Eindhoven, 2004.