Firing Partial Orders in a Petri Net

Robin Bergenthum(✉)

Faculty of Mathematics and Computer Science, FernUniversität in Hagen, Hagen, Germany
robin.bergenthum@fernuni-hagen.de

Abstract. Petri nets have the simple firing rule that a transition is enabled to fire if its preset of places is marked. The occurrence of a transition is called an event. To check whether a sequence of events is enabled, we simply try to fire the sequence from 'start' to 'end' in the initial marking of the net. It is a bit of a stretch to call this an algorithm, but its runtime complexity is in $O(|P| \cdot |V|)$, where P is the set of places and V is the set of events.

Petri nets model distributed systems. An execution of a distributed system is a partial order of events rather than a sequence. Compact tokenflows are tailored to an efficient algorithm that decides if a partial order of events is enabled in a Petri net. Yet, the runtime complexity of this algorithm is in $O(|P| \cdot |V|^3)$.

In practical applications dealing with a huge amount of behavioral data, the gap between *just firing* a sequence and *deciding* if a partial order is enabled, makes a big difference.

In this paper, we present an approach to *just firing* a partial order of events in a Petri net. By firing a partial order, we obtain a lot of information about whether or not the partial order is enabled. We show that *just firing* is often enough if done correctly.

1 Introduction

Petri nets model distributed systems. They have formal semantics, an intuitive graphical representation, and are able to express concurrency among the occurrence of events [1,2,8,9,21,22]. Petri nets have the simple firing rule that a transition can fire if its prefix is marked. This definition implies so-called firing sequences, i.e., sequences of subsequently enabled transitions. In many practical applications, we equate the set of firing sequences with the behavior of the net. Thus, it is very easy to check if specified or recorded behavior is 'in' a Petri net model. We simply fire the sequence from 'start' to 'end' and immediately obtain a result.

Then again, the behavior of a concurrent system is often defined as a set of scenarios [4,7,10–13] expressing causal dependencies and concurrency among the events of the systems behavior. Obviously, such scenarios cannot be modeled by sequences, only by partially ordered sets of events.

Even though partially ordered sets of events are a very intuitive approach to modeling the behavior of a distributed system, checking if such order is 'in' a

D. Buchs and J. Carmona (Eds.): PETRI NETS 2021, LNCS 12734, pp. 399–419, 2021.
https://doi.org/10.1007/978-3-030-76983-3_20

Petri net model is not trivial. There are even different semantics which all define the same partial language of a Petri net. In this sense, the notion of executing a partially ordered set of events in a Petri net is ambiguous.

(i) Step semantics of Petri nets [15]: A partially ordered set of events is in a Petri net model if and only if each maximal set of unordered events of the partial order is enabled after the occurrence of its prefix.
(ii) Process net semantics of Petri net [16]: A partially ordered set of events is in a Petri net model if and only if there is a process net (occurrence net) of the Petri net, so that the run extends the order relation between events of this process.
(iii) Tokenflow semantics of Petri net [4,17]: A partially ordered set of events is in a Petri net model if and only if there is a valid distribution of tokens between events only using the relations specified by the partial order.

These three semantics are equivalent [17,19,23], i.e., they (fortunately) define the same partial language. However, each semantic implies a different algorithm deciding if a partially ordered set of events is enabled. Utilizing step or process net semantics, the number of process nets and the number of maximal sets of unordered events grow exponentially with the size of the partial order, producing slow algorithms. Only algorithms utilizing tokenflow semantics run in polynomial time [4].

A tokenflow is a distribution of tokens between events along the relations of a partial order. A tokenflow is valid if every event receives enough tokens to occur from its prefix, and no event has to produce more tokens than it is able to. If there is a valid tokenflow for every place of a Petri net, the partial order is enabled. We test if there is a valid tokenflow for a place by solving a related flow optimization problem [14]. There a many highly specialized flow optimization algorithms pushing the boundaries of their worst-case runtime [3]. For practical applications, however, the famous pre-flow-push algorithm [18] is easy to implement and has a very good runtime for most examples. The pre-flow-push is in $O(n^3)$, where n is the number of nodes of the flow network. Thus, deciding if a set of partially ordered events is enabled in a Petri net is in $O(|P| \cdot |V|^3)$, where P is the set of places and V is the set of events.

In the area of process mining, some of the recently developed algorithms exploit the idea that firing sequences of events is really cheap. For example, the eST-miner [20] records huge sets of data observing the behavior of a business process and tries to automatically generate a fitting process model. The eST-miner fires the observed sequences over and over again in some initial Petri net to generate more fitting places to complete the model. However, increasingly more process mining papers state that the observed log files are actually partial orders, not only sequences. Thus, there is a clear need to also *fire* partial orders fast.

In this paper, we revisit the problem of deciding if a partially ordered set of events is enabled in a Petri net and decompose the problem along the set of places. We *brute force* fire the partial order in the net once to build a first

tokenflow for every place. If the size of the Petri net is fixed, this is possible in linear runtime. The constructed tokenflow will be valid for a subset of places. Experimental results will show that this set is actually quite large for most practical applications. During the firing of the partial order, we, furthermore, present how to easily check if there are alternative distributions of tokens. If there is only one possibility to distribute tokens and our first firing fails, the partial order is not enabled. Only if enabledness is not decided yet, we tackle the subset of *not yet*-decided places by a dedicated compact tokenflow algorithm. In the remainder of the paper, we present the new algorithm, discuss its runtime, and show experimental results using models taken from practical applications.

2 Preliminaries

Let f be a function and B be a subset of the domain of f. We write $f|_B$ to denote the restriction of f to B. As usual, we call a function $m : A \to \mathbb{N}$ a multiset and write $m = \sum_{a \in A} m(a) \cdot a$ to denote multiplicities of elements in m. Let $m' : A \to \mathbb{N}$ be another multiset. We write $m \geq m'$ if $\forall a \in A : m(a) \geq m'(a)$ holds. We denote the transitive closure of an acyclic and finite relation $<$ by $<^*$. We denote the skeleton of $<$ by $<^\diamond$. The skeleton of $<$ is the smallest relation $\lhd$, so that $\lhd^* = <^*$ holds. Let $(V, <)$ be some acyclic and finite graph, $(V, <^\diamond)$ is called its Hasse diagram. We model distributed systems by place/transition nets [9,21,22].

Definition 1. *A place/transition net (p/t-net) is a tuple (P, T, W) where P is a finite set of places, T is a finite set of transitions so that $P \cap T = \emptyset$ holds, and $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a multiset of arcs. A marking of (P, T, W) is a multiset $m : P \to \mathbb{N}$. Let m_0 be a marking, we call the tuple $N = (P, T, W, m_0)$ a marked p/t-net and m_0 the initial marking of N.*

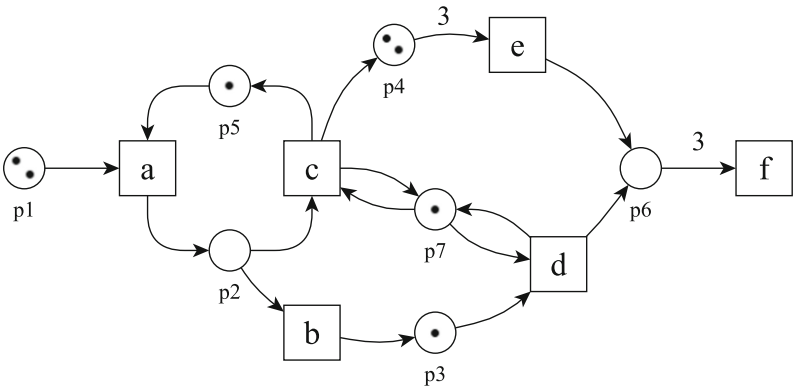


Fig. 1. A marked p/t-net.

Figure 1 depicts a marked p/t-net. Transitions are rectangles, places are circles, the multiset of arcs is represented by weighted arcs, and the initial marking is represented by black dots called tokens.

There is a simple firing rule for transitions of a p/t-net. Let t be a transition of a marked p/t-net (P, T, W, m_0). We denote $\circ t = \sum_{p \in P} W(p,t) \cdot p$ the weighted preset of t. We denote $t\circ = \sum_{p \in P} W(t,p) \cdot p$ the weighted postset of t. A transition t is enabled (can fire) at marking m if $m \geq \circ t$ holds. Once transition t fires, the marking of the p/t-net changes from m to $m' = m - \circ t + t\circ$.

In our exemplary marked p/t-net, transitions a and d can fire at the initial marking. If a fires, this removes one token from $p1$ and the token from $p5$. Additionally, firing a will produce a new token in $p2$. In this new marking, transitions c and d can fire. a is not enabled anymore because there is no more token in $p5$. Firing transition c will enable transition a again and enable transition e. Transition e needs three tokens in $p4$ to be enabled.

Repeatedly processing the firing rule produces so-called firing sequences. These firing sequences are the most basic behavioral model of Petri nets. For example, the sequence $a\ d\ c\ a\ b\ d\ e\ f$ is enabled in the marked p/t-net of Fig. 1. Let N be a marked p/t-net, the set of all enabled firing sequences of N is the (sequential) language of N.

Petri nets are able to express concurrency between events. For example, transitions a and d can fire independently from one another. Roughly speaking, they can fire in any order while not sharing tokens. If we fire transition a, transitions c and d can fire in any order but not concurrently because they share the token in $p7$.

To specify concurrency between events, we formalize executions of a p/t-net by means of labeled partial orders.

Definition 2. *Let T be a set of labels. A labeled partial order is a triple $(V, \ll, l)$ where V is a finite set of events, $\ll \subseteq V \times V$ is a transitive and irreflexive relation, and the labeling function $l : V \to T$ assigns a label to every event. A run is a triple $(V, <, l)$ iff $(V, <^*, l)$ is a labeled partial order. A run $(V, <, l)$ is also called a labeled Hasse diagram iff $<^\diamond = <$ holds.*

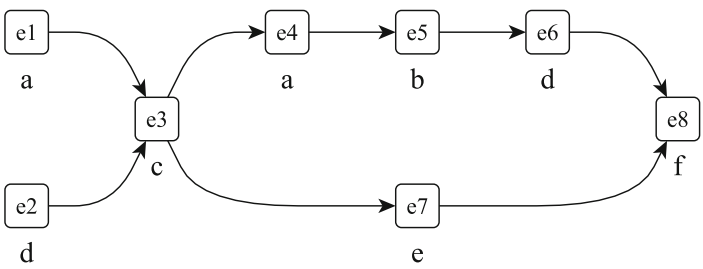


Fig. 2. A run.

Just like a firing sequence, a run can be enabled in a marked p/t-net. A run is enabled if we can replay the order by firing transitions where unordered parts of the partial order can fire concurrently. As stated in the introduction, there are different semantics to formally define whether a run is enabled, but the compact tokenflow semantic is the most efficient [4].

A compact tokenflow is a distribution of tokens along the relations and nodes of a run. A run is in the partial language of a p/t-net if there is a compact tokenflow distributing tokens so that three conditions hold: first, every event receives enough tokens, second, no event has to pass too many tokens, and third, the initial marking is not exceeded. Tokens must be received from the particular presets of events. Thus, we ensure that consumed tokens are available before the actual event occurs. If a transition produces tokens, the related events are allowed to produce tokenflow in the run and pass these tokens to their particular postsets. If an event receives tokens, it consumes the tokenflow needed and passes the redundant tokenflow to later events. Tokens of the initial marking are free for all, i.e., any event can consume or pass tokens from the initial marking.

Definition 3. *Let $N = (P, T, W, m_0)$ be a marked p/t-net and run $= (V, <, l)$ be a run so that $l(V) \subseteq T$ holds. A compact tokenflow is a function $x : (V \cup <) \to \mathbb{N}$. x is valid for $p \in P$ iff the following conditions hold:*

(i) $\forall\, v \in V\colon x(v) + \sum_{v'<v} x(v', v) \geq W(p, l(v))$,

(ii) $\forall\, v \in V\colon \sum_{v<v'} x(v, v') \leq x(v) + \sum_{v'<v} x(v', v) - W(p, l(v)) + W(l(v), p)$,

(iii) $\sum_{v \in V} x(v) \leq m_0(p)$.

run is valid for N iff there is a compact valid tokenflow for every $p \in P$.

Figure 3, Fig. 4, and Fig. 5 depict three compact tokenflows for three different places of the marked p/t-net of Fig. 1 and the run of Fig. 2 (integer 0 is not shown).

Figure 3 depicts a valid compact tokenflow for the place $p2$ of Fig. 1. The transitions related to the two events labeled b and c need to receive one token in $p2$. The transition related to the events labeled a can produce one token in $p2$. Initially, there are no tokens in this place but no event consumes tokens from the initial marking. Thus, this is a valid tokenflow for $p2$.

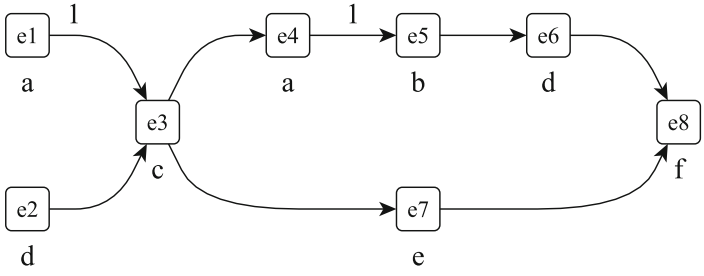


Fig. 3. Valid compact tokenflow for $p2$ of Fig. 1.

Figure 4 depicts a valid compact tokenflow for the place $p6$ of Fig. 1. The event labeled f needs to receive three tokens. All three events labeled d or e can produce one token. All other events just receive and push tokens to later events. Again, we do not need any token from the initial marking. Thus, this is a valid tokenflow for $p6$.

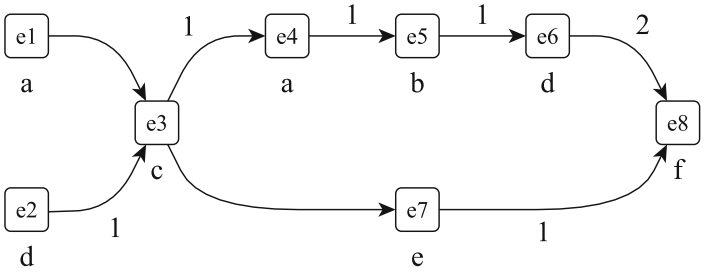


Fig. 4. Valid compact tokenflow for $p6$ of Fig. 1.

Figure 5 depicts a valid compact tokenflow for the place $p7$ of Fig. 1. The events labeled c or d need to receive one token. Because of the short loops at place $p7$, these events can also produce one token in $p6$. Obviously, an event is not allowed to consume the tokens it produces itself. This is why event $e2$ consumes a token from the initial marking before pushing its own token to $e3$. $e3$ consumes the token and pushes a new token to $e6$. This is a valid tokenflow for $p7$.

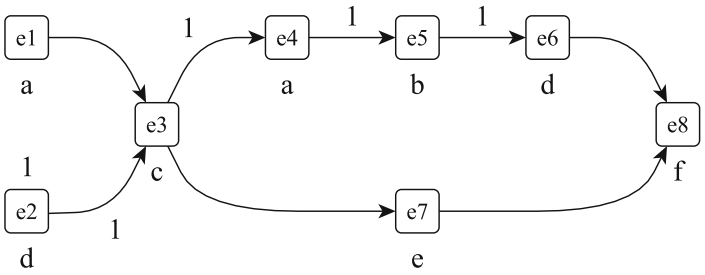


Fig. 5. Valid compact tokenflow for $p7$ of Fig. 1.

If there is a valid tokenflow for every place of a marked p/t-net, the run is valid. The set of valid runs coincides with the (partial) language of a p/t-net. Here, we refer the reader to [4,5] and state the following theorem.

Theorem 1. *The language of a marked p/t-net is well-defined by the set of valid runs [4].*

3 Deciding Enabledness and Firing Runs

In this section, we decide if a run is enabled in a marked p/t-net. In the first subsection, we recap the algorithm that decides if a run is enabled using compact tokenflows in polynomial runtime, originally introduced in [5]. In the second subsection, we present an approach to firing a run in a p/t-net to decide if the run is enabled for a subset of the places of the p/t-net in linear runtime. In the last subsection, we present the idea of firing backwards and combine all approaches to obtain a new and faster algorithm to decide enabledness for runs in p/t-nets.

3.1 Tokenflows and Flow Networks

We decide if a run is enabled in a marked p/t-net by constructing a flow network and a maximal flow for every place. A flow network (see for example [3]) is a directed graph with two specific nodes: A source, the only node having no ingoing arcs, and a sink, the only node having no outgoing arcs. Each arc has a capacity, and a flow is a function from the arcs to the non-negative integers assigning a value of flow to each arc. This flow function needs to respect the capacity of each arc and the so-called flow conservation. The flow conservation states that the sum of flow reaching a node is equal to the sum of flow leaving a node for every inner node of the flow network. Thus, flow is only generated at the source and flows along different paths till it reaches the sink. The value of a flow function in a flow network is the sum of flow reaching the sink. The maximal flow problem is to find a flow function that has a maximal value.

For a place of a p/t-net and a run, we construct the so-called associated flow network. The flow in the associated flow network directly coincides with a compact tokenflow in the related partial order. For each event, we create two nodes in the flow network: an *in-node* and an *out-node*. The flow at the in-node is the value of tokenflow received by the related event. This value has to be greater than the number of tokens needed by the related transition. We rout the number of tokens needed from the in-node to the sink representing tokens consumed by the occurrence of the transition. We distribute additional flow further through the network by adding an arc from every in-node to its out-node. The out-node will distribute flow to later events. The maximal amount of flow this node can push on is the amount of flow received from its in-node plus the number of tokens produced by the occurrence of the related transition. We rout the number of additionally produced tokens from the source to the out-node. In addition, all pairs of in-nodes and out-nodes are connected just like the partial order of events. Whenever there is an arc from one event to another, there is a related arc in the flow network connecting the out-node of the first event with the in-node of the second event. Finally, we add one additional node to the flow network to represent the initial marking.

Figure 6 depicts the associated flow network for Fig. 1, the place $p7$ of the same figure, and Fig. 2. We already saw a related compact tokenflow in Fig. 5. At the top of Fig. 6 is the source, at the bottom is the sink. The node furthest to the

left relates to the initial marking. The initial marking of $p7$ is 1; thus, this node is connected to the source with capacity 1. Events $e2$, $e3$, and $e6$ can produce a token each. Thus, the related out-nodes are connected to the source as well. Roughly speaking, one piece of flow, i.e., one token in the run, can enter the flow network at the initial marking and at all events labeled by c or d. Events $e2$, $e3$, and $e6$ need to receive a token each. Thus, the related in-nodes are connected to the sink. Again, one piece of flow, i.e., one token in the run, can leave the flow network at these nodes. The capacity of all inner arcs is not limited. Only looking at place $p7$, the run is enabled if the nodes related to events $e2$, $e3$, and $e6$ can consume a token each. Due to the construction of the associated flow network, there is a valid compact tokenflow if there is a flow saturating all arcs leading to the sink (value 3 in this example).

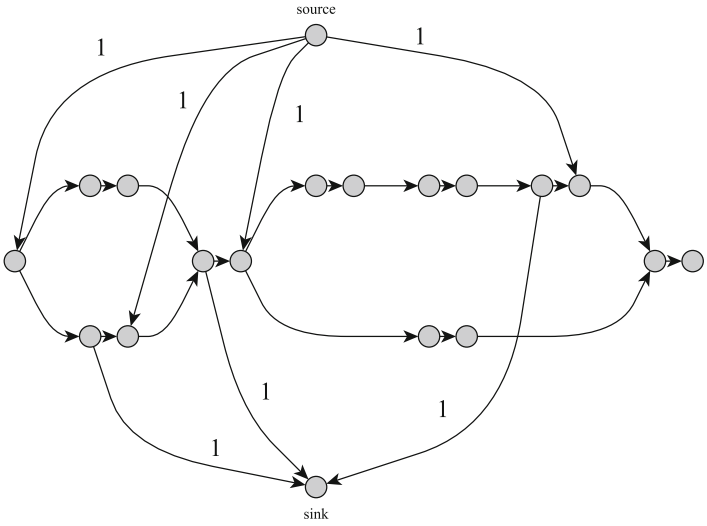


Fig. 6. Associated flow network for Fig. 1, the place $p7$ of the same figure, and Fig. 2.

Figure 7 depicts a maximal flow in the flow network of Fig. 6. This flow saturates all arcs going to the sink; thus, by construction, the flow directly relates to a valid compact tokenflow and the run is enabled. If the value of a maximal flow does not saturate all arcs leading to the sink, there is no valid compact tokenflow because for every distribution of tokens, at least one event cannot occur and the run is not enabled.

Constructing a maximal flow in a flow network is the well-known maximal flow problem (see, for example, [3]). There are various algorithms solving the maximal flow problem in polynomial time. For the application of calculating the value of a maximal flow in an associated flow network, we consider a pre-flow-push algorithm using a so-called gap heuristic. The worst-case time complexity of the pre-flow-push algorithm is in $O(n^3)$, where n is the number of nodes.

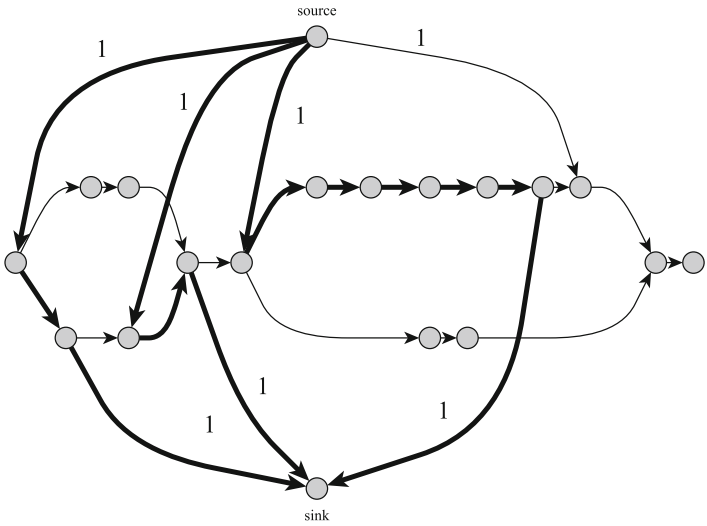


Fig. 7. A maximal flow in Fig. 6.

We recap the algorithm that decides if a run is enabled in a marked p/t-net using compact tokenflows. Additionally, Algorithm 1 computes the set of places that hinders the execution of the run. Obviously, the run is enabled if and only if this set of non-valid places is empty. We can simply stop the algorithm as soon as we find the first non-valid place, but in applications, it may be very helpful to know the set of all non-valid places to fix model or run.

Algorithm 1. *Calculates the set of non-valid places of a marked p/t-net for a run.*

1: **input**: marked p/t-net (P, T, W, m_0), run $(V, <, l)$
2: **for each** $p \in P$ **do**
3: $G \leftarrow$ associated flow network of (P, T, W, m_0), $(V, <, l)$, p
4: $x \leftarrow$ sum of capacities of arcs leading to the sink of G
5: $w \leftarrow$ pre-flow-push of G
6: **if** $(w < x)$ P_{nvalid} **add** p
7: **return** P_{nvalid}

The runtime of Algorithm 1 is in $O(|P| \cdot |V|^3)$.

3.2 Firing Runs

In this subsection, we introduce the concept of firing runs in a marked p/t-net. The initial marking is a multiset of places. We (randomly) distribute this marking to the set of minimal nodes of the run, creating a set of local marking at each event. We fire each event in its local marking and (randomly) push the resulting local markings to later events. The four conceptual differences to the

construction of a valid tokenflow are: (a) Instead of building a network for every place, we handle all places at once. (b) We want to fire in linear time; thus, we cannot redistribute markings or search for paths. We just randomly push local markings to later events to fire every event exactly once. (c) Valid compact tokenflows are tailored to a fast flow network algorithm. Every event only has to receive enough tokens to occur. Thus, conditions (i) and (ii) of Definition 3 are formulated as inequalities to keep the number of tokens small. When firing an event in a local marking, every event will produce its maximal number of tokens. Thus, no token is lost, and we also construct a final marking. (d) A local marking can be negative.

We fire a run in a p/t-net as a first step to decide if the run is enabled or not. The main idea is that the enabledness problem can easily be decomposed along the set of places. For some of the places, we need the maximal flow algorithm to distribute and re-distribute tokens to decide if a valid tokenflow exist. For other places, obviously highly depending on the specific run, a valid compact tokenflow may be very easy to construct. Thus, we tackle our problem in two steps: first *brute force* fire a run in a p/t-net, constructing a so-called multi-tokenflow describing a distribution of local markings. For some of the places, these markings will directly relate to valid compact tokenflows and we will not have to consider these places further. If the multi-tokenflow is not valid for some place, we will gain additional information about the existence of alternative token distributions to even see if a re-distribution is possible. If a re-distribution is possible, we redistribute using the algorithm presented in the previous subsection.

To fire a run in a marked p/t-net, constructing a multi-tokenflow as a distribution of (local) markings, we first extend the run by introducing two additional events, one initial and one final event.

Definition 4. *Let $N = (P, T, W, m_0)$ be a marked p/t-net and $run = (V, <, l)$ be a run so that $l(V) \subseteq T$ holds. We denote $V_{min} \subseteq V$ the set of events with an empty preset and $V_{max} \subseteq V$ the set of events with an empty postset. Let $v_i, v_f \notin V$ be two events and define an extended relation $\prec$ by $\prec := < \cup (v_i \times V_{min}) \cup (V_{max} \times v_f)$. We denote $run^+ = (V, v_i, v_f, \prec, l)$ the extended run of run. A function $X : \prec \rightarrow \mathbb{Z}^P$ is a multi-tokenflow for run iff the following conditions hold:*

(I) $\sum_{v_i \prec v'} X(v_i, v') = m_0$.

(II) $\forall\, v \in V\colon \sum_{v \prec v'} X(v, v') = \sum_{v' \prec v} X(v', v) - \circ l(v) + l(v) \circ$,

(III) $\forall\, v \in V \cup \{v_i\}\colon (\sum_{v \prec v'} X|_p(v, v') \geq 0 \Longrightarrow \forall\, v'' \in V\colon X|_p(v, v'') \geq 0)$.

We call $m_f := \sum_{v' \prec v_f} X(v', v_f)$ the final marking of X.

Note that local markings of a multi-tokenflow can be negative. Condition (I) distributes the initial marking to the minimal events of the run, condition (II) ensures that the local markings reflect the firing rule, and condition (III) ensures that tokens are distributed and not just appearing from nowhere by adding negative values to nearby arcs. Thus, a multi-tokenflow is a distribution of actually produced tokens whenever possible.

Figure 8 depicts a sequential run. Obviously, the concept of a multi-tokenflow for a sequential run is just the concept of markings of a firing sequence.

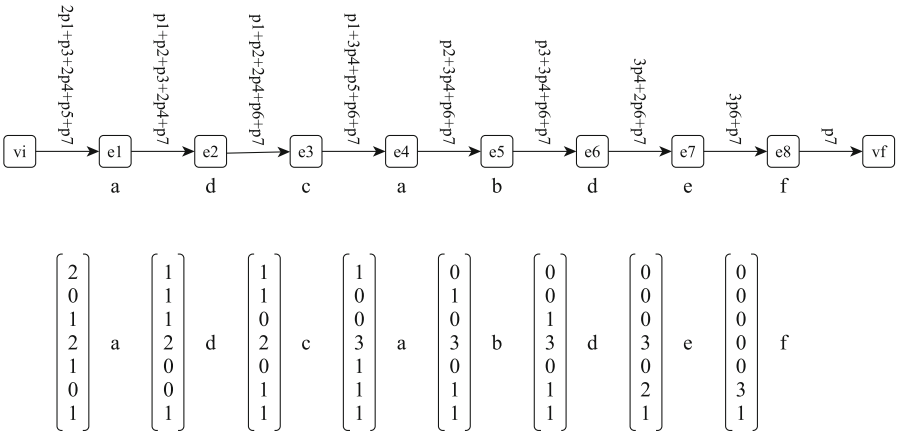


Fig. 8. A multi-tokenflow and markings of a firing sequence.

Figure 9 depicts a multi-tokenflow for the run depicted in Fig. 2. In comparison to Fig. 8, a multi-tokenflow implements the concept of local markings. Just like a tokenflow, the marking is distributed whenever the partial order branches. Every event receives a sum of local markings and fires to push the resulting local marking to later events. In contrast to tokenflows, every produced token has to be pushed until it is consumed or until it reaches the final marking. In contrast to markings, we allow negative values and distribute markings at every branch in the partial order.

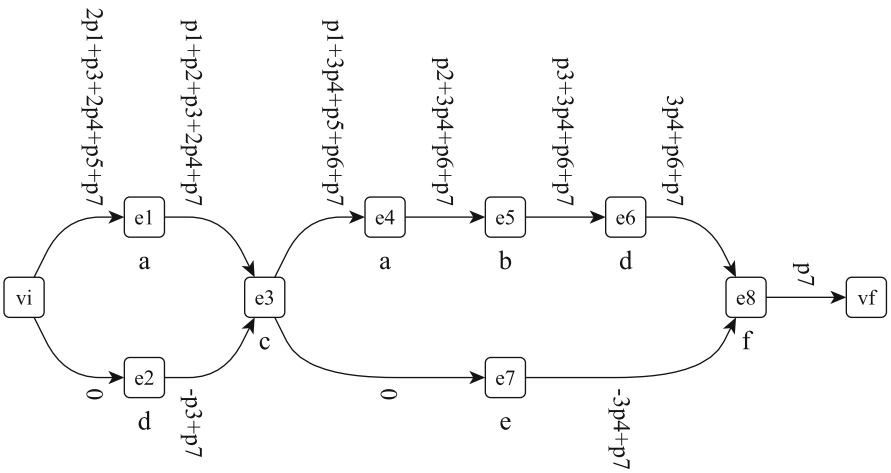


Fig. 9. A multi-tokenflow in the run of Fig. 2.

We formalize the relation between a tokenflow and a multi-tokenflow in the following theorem.

Theorem 2. *Let $N = (P, T, W, m_0)$ be a marked p/t-net, $run = (V, <, l)$ be a run so that $l(V) \subseteq T$ holds, $run^+ = (V, v_i, v_f, \prec, l)$ be the extended run. There is a valid compact tokenflow x in run for N and p, if and only if there is a multi-tokenflow X in run^+, so that $X|p$ enables every event in its sum of in-going local markings in N only considering p.*

Proof. If there is a valid compact tokenflow for a place p in *run*, we simply construct a multi-tokenflow enabling every event for place p. We can construct this p-component of a multi-tokenflow by copying the tokenflow to the extended run and moving the initial tokenflow to paths outgoing of the initial event. Whenever there is an event not producing its full number of tokens (i.e., (ii) holds, but not yet (II)), we find a path from this event to the final event. We can add tokenflow to this path without making conditions (i), (ii), and (iii) not valid until (II) holds. The same holds for the initial marking; we can go from (iii) to (I) by adding tokens on paths from the initial to the final event. Every valid tokenflow is non-negative, thus, (III) holds as well.

If there is a multi-tokenflow enabling every event for place p, there is a valid compact tokenflow for p. If all the events are enabled for p, we show that every local marking is positive for p. Assume there is some negative value. Without loss of generality, we choose an arc with a negative value so that there is no earlier arc with a negative value. This is always possible because the initial marking is non-negative. Because of (III), the start-event of a first negative arc has a negative sum of outgoing local markings for p but a non-negative sum of in-going local markings. We apply the firing rule to see that this event is not enabled for p in its in-going local markings. Thus, the multi-tokenflow is non-negative for component p. By copying component p of the multi-tokenflow from the extended run into the run and moving initial tokenflow to the minimal nodes of the run, we directly obtain a valid compact tokenflow, because if every event is enabled, (i) holds, (II) implies (ii), and (I) implies (iii). □

Thus, if we fix a run, we can construct a multi-tokenflow for a marked p/t-net, and if this flow enables all the events of the run for a subset of places, the run is enabled according to this set.

Furthermore, if there is only one possibility to construct a p-component of a multi-tokenflow, if the set of local markings does not enable every event for p, there is no valid compact tokenflow for p.

Lemma 1. *Let $N = (P, T, W, m_0)$ be a marked p/t-net, $run = (V, <, l)$ be a run so that $l(V) \subseteq T$ holds, $run^+ = (V, v_i, v_f, \prec, l)$ be the extended run. Let X be a multi-tokenflow in run^+. For every $p \in P$, where at least one event is not-enabled for p in its sum of in-going local markings, if $X|_p \leq 0$ for every event with multiple out-going arcs, there is no valid compact tokenflow in run for p and N.*

Proof. With the preconditions of this lemma: assume there is a valid compact tokenflow for p. We can construct another multi-tokenflow enabling all events for p, but $X|_p$ is unique. □

In the next lemma, we use the concept of a final marking of a multi-tokenflow. We show that this marking is actually unique and if it is negative for one p-component, the related run is not enabled.

Lemma 2. *Let $N = (P, T, W, m_0)$ be a marked p/t-net, $run = (V, <, l)$ be a run so that $l(V) \subseteq T$ holds, $run^+ = (V, v_i, v_f, \prec, l)$ be the extended run. Let X be a multi-tokenflow in run^+ and m_f be the final marking. If $m_f(p) < 0$ holds, there is no valid compact tokenflow for p in run.*

Proof. Due to the construction of the extended run, every event is on a path from the initial to the final event. For this reason and because of (II), a multi-tokenflow does not lose tokens and the final marking is $m_f = m_0 + (\sum_{v \in V} l(v) \circ - \circ l(v))$ for every multi-tokenflow. The final marking of a multi-tokenflow is independent from the distribution of tokens. If this final marking is negative for a component p, there is no valid compact tokenflow for p, because, if we assume *run* is enabled, then also every firing sequence respecting the order of run is enabled and leads to a negative local marking in p according to the (usual) firing rule for sequences. □

In Fig. 9, the depicted multi-tokenflow enables all events considering places $p1$, $p2$, $p5$, $p6$. Thus, with the help of Theorem 2 and only using one multi-tokenflow, we decide enabledness for four of the seven places. The multi-tokenflow branches for places $p3$, $p4$, and $p7$ at the initial event. Thus, although these three components do not enable all events, we cannot apply Lemma 1 because there are other possible distributions of tokens. We cannot apply Lemma 2, either, because the final marking is not negative. Thus, we have to decide places $p3$, $p4$, and $p7$ using Algorithm 1.

At the end of this subsection, we present the algorithm firing a run in a marked p/t-net, deciding enabledness and non-enabledness for a subset of places using Theorem 2, Lemma 1, and Lemma 2.

Algorithm 2. *Calculates a set of valid and a set of non-valid places of a marked p/t-net for a run.*

input: marked p/t-net (P, T, W, m_0), run $(V, <, l)$
$(V, v_i, v_f, \prec, l) \leftarrow$ extension of $(V, <, l)$.
(first successor of v_i).marking **add** m_0
for each $e \in V$ **in** $\prec$-order **do**
    $P_{fnvaild}$ **add** $\{p \in P | e.marking(p) < W(p, l(e))\}$
    $P_{fbranch}$ **add** $\{p \in P | e.marking(p) > 0, |e \bullet| > 1\}$
    (first successor of e).marking **add** $e.marking - \circ l(e) + l(e) \circ$
$P_{fnvaild}$ **add** $\{p \in P | v_f.marking(p) < 0\}$
$P_{valid} \leftarrow P \setminus P_{fnvaild}$
$P_{nvalid} \leftarrow P_{fnvaild} \cup (P \setminus P_{fbranch})$
return (P_{valid}, P_{nvalid})

To conclude this subsection, we take a look at the runtime of Algorithm 2. There is a problem in line 4, where we have to consider all events in some total order respecting $\prec$. We can very easily calculate such an order in a preprocessing step but not in linear time. If this order is part of the input, we only consider every event once. For every event, we only touch one outgoing arc. We store the sum of in-going arcs at events whenever we push local markings. Thus, we never have to iterate a set of arcs to calculate a sum of local markings. The runtime of Algorithm 2 is in $O(|P| \cdot |V|)$, or in $O(|V|^2 + |P| \cdot |V|)$ if we need to calculate a total order first.

3.3 Firing Backwards

The reason we have to tackle places $p3$, $p4$, and $p7$ from Fig. 1 deciding enabledness of 2 by Algorithm 2 is that they are marked at the two forward-branched events v_i and $e3$. The components of these three places do not enable all events, but we cannot apply Lemma 1 because there might be another valid distribution. The flow of $p2$ is unique because it is only positive at non-branching events. Yet, the multi-tokenflow for places $p1$ and $p5$ is positive at forward-branched events. However, since the multi-tokenflow components $p1$ and $p5$ already enable all events, we do not have to distribute further. In some sense, we were lucky to find these valid distributions for $p1$ and $p5$ at the first attempt. In this section, we will introduce the concept of firing backwards to offer a heuristic to find valid distributions more often.

We already mentioned that it is important not to re-distribute local markings or even look for paths to be able to fire in linear time. The most efficient strategy to construct a multi-tokenflow is to push the complete local marking of every event to its first subsequent event. Thus, the number of push operations is the number of events, not the number of arcs. We call this strategy the forward-strategy in the remainder of the paper. The multi-tokenflow depicted in Fig. 9 is produced by the forward-strategy, i.e., the complete initial marking is pushed to $e1$, the local marking of $e3$ is pushed to $e4$. Thus, the arcs $(v_i, e2)$ and $(e3, e7)$ are never touched. Obviously, the constructed multi-tokenflow randomly depends on the order of events.

Figure 10 depicts a typical structure of a run. In Fig. 10, all arcs leaving forward-branched events are depicted by dashed arcs. When firing this run in a p/t-net, we can actually decide enabledness in linear time using the forward-strategy of pushing local markings for every place which is not marked at the two highlighted events. If there is a local marking for a place at the branching events, we may be lucky to randomly construct a valid distribution. However, if we need to share the marking between different subsequent events, the forward-strategy will always fail. For this reason, it is not a good idea to fire the run forward again using some modified distribution strategy.

We *brute force* fire the run again but starting from the (unique) final marking and backwards. We already constructed this marking firing forward once. Starting from this final marking at the final event, we push this marking backwards to the first predecessor. We fire the event backwards to calculate the ingoing local

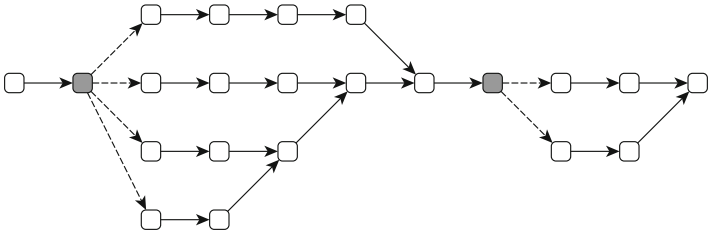


Fig. 10. The set of forward-branched events.

marking to this event. This local marking is pushed to the next first predecessor and so on. This will construct a multi-tokenflow as well.

Figure 11 depicts the run of Fig. 10 and highlights another set of arcs and events. When firing this run in a p/t-net, we can actually decide enabledness in linear time using the backward-strategy of pushing local markings from the final marking of a run for every place which is not marked at the four highlighted events.

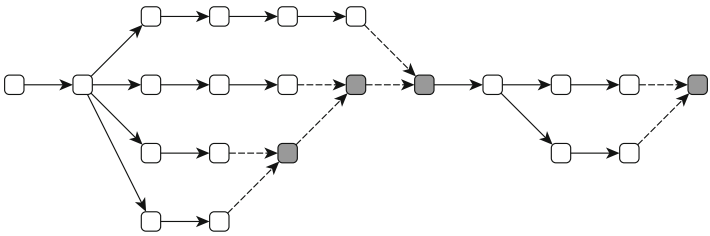


Fig. 11. The set of backward-branched events.

The main advantage of combining a forward-strategy with the backward-strategy in the example run of Fig. 10 is that the example does not contain any forward and backward-branched events. Thus, the set of difficult events is disjoint. If the forward-strategy is not able to decide enabledness, the backward-strategy only fails as well if the related place is also marked at some backward-branched event. If we think of typical workflow Petri nets, for example, that are relatively well-structured using workflow patterns like and/xor-splits and joins, these kinds of places are very rare. In the next section, we present experimental results on how many places can be decided in linear time using the combination of the forward-strategy and the backward-strategy for models taken from practical applications. Yet, before we move on, we combine the forward-, backward-, and flow network-strategies to present the new algorithm that decides if a run is enabled in a marked p/t-net.

Algorithm 3. *Calculates the set of non-valid places of a marked p/t-net for a run.*

1: **input**: marked p/t-net (P, T, W, m_0), run $(V, <, l)$
2: $(V, v_i, v_f, \prec, l) \leftarrow$ extension of $(V, <, l)$.
3: (first successor of v_i).marking **add** m_0
4: **for each** $e \in V$ **in** $\prec$-order **do**
5: $\quad P_{fnvaild}$ **add** $\{p \in P | e.marking(p) < W(p, l(e))\}$
6: $\quad P_{fbranch}$ **add** $\{p \in P | e.marking(p) > 0, |e \bullet| > 1\}$
7: $\quad$ (first successor of e).marking **add** $e.marking - \circ l(e) + l(e) \circ$
8: $P_{fnvaild}$ **add** $\{p \in P | v_f.marking(p) < 0\}$
9: $P_{valid} \leftarrow P \setminus P_{fnvaild}$
10: $P_{nvalid} \leftarrow P_{fnvaild} \cup (P \setminus P_{fbranch})$
11: $P' \leftarrow P \setminus (P_{vaild} \cup P_{nvalid})$
12: $(P, T, W, m_0) \leftarrow (P', T, W|_{P' \times P'}, m_0|_{P'})$
13: (first predecessor of v_f).marking2 **add** v_f.marking$|_P$
14: **for each** $e \in V$ **in** reverse $\prec$-order **do**
15: $\quad P_{bnvaild}$ **add** $\{p \in P | e.marking2(p) < W(l(e)), p)\}$
16: $\quad P_{bbranch}$ **add** $\{p \in P | e.marking2(p) + W(p, l(e)) - W(l(e), p) > 0, |\bullet e| > 1\}$
17: $\quad$ (first predecessor of e).marking2 **add** $e.marking2 + \circ l(e) - l(e) \circ$
18: P_{valid} **add** $P \setminus P_{bnvaild}$
19: P_{nvalid} **add** $P_{bnvaild} \cup (P \setminus P_{bbranch})$
20: $P' \leftarrow P \setminus (P_{vaild} \cup P_{nvalid})$
21: $(P, T, W, m_0) \leftarrow (P', T, W|_{P' \times P'}, m_0|_{P'})$
22: P_{nvalid} **add Algorithm 1** $(P, T, W, m_0), (V, <, l)$
23: **return** P_{nvalid}

Algorithm 3 has three parts: lines 1 to 10 implement the forward-strategy. The set $P_{fbranch}$ keeps track of places marked at forward-branching events. Line 8 implements Lemma 2. In lines 11 and 12, we remove the set of places that we do not have to tackle anymore. Again, if a total order of events is part of the input, this part of the algorithm runs in linear run-time. In line 13, we start the backward-strategy reusing the final marking calculated in the first part of the algorithm. The set $P_{bbranch}$ keeps track of places marked at backward-branching events. Line 16 implements a backward version of Lemma 1. Note that there is no backwards version of Lemma 2 because firing backwards from the final marking will reconstruct the initial marking. In lines 20 and 21, we remove the set of places that we do not have to tackle further. Again, if a total order of events is part of the input, this second part of the algorithm runs in linear time as well. In line 20, we only keep places that have to be handled in cubic runtime by Algorithm 1.

4 Comparison and Experimental Results

In this section, we compare the runtimes of Algorithm 1 and Algorithm 3. To compare run-time, we denote (P, T, W, m_0) a marked p/t-net, $(V, <, l)$ a run,

and assume a total order respecting $<$ is part of the input. In the remainder, we call all places we can check by brute force firing the run in a p/t-net *simple* places. We call the remaining places *complex* places. This is a bit misleading because whether a place is simple highly depends on the run as well. As stated above, dealing with sequential runs, every place is simple. Furthermore, whether a place is simple or complex does not only depend on the structure of a run, but also on the order of events. Figure 12 depicts a very simple p/t-net where a transition b can fire if transition a produces a token in $p1$ and a run with some kind of w-structure. Figure 12 depicts a distribution for tokens produced by events $e1$, $e2$, and $e3$ on solid arcs. The dashed arcs depict a redistribution of tokens redistributing all previous tokens adding the token from $e4$ to $e8$. This re-distribution is done by flow network algorithms looking for paths, and considering already produced flow as a possible step backwards. Although the net of Fig. 12 is very simple, it is a complex place.

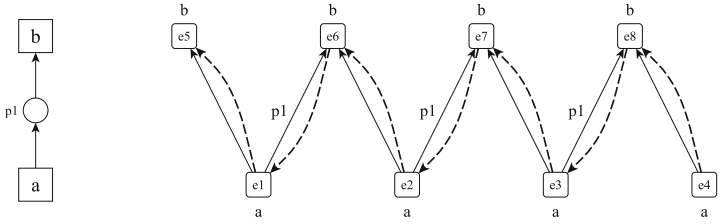


Fig. 12. Redistribution of tokens.

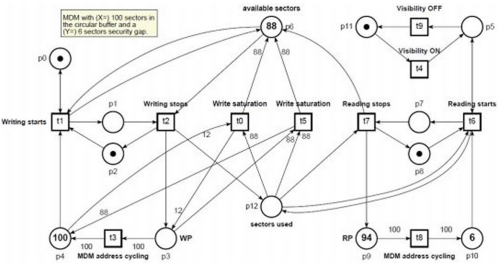
The worst-case runtimes of Algorithm 1 and Algorithm 3 is in $O(|P| \cdot |V|^3)$. If the set of simple-places is empty, the run-time of Algorithm 3 is, obviously, the runtime of Algorithm 1 plus the runtime of two times firing the run. If the set of complex places is empty, the runtime of Algorithm 3 is two times firing the run, i.e., in linear runtime.

In the remainder of this section, we will take a look at examples from practical applications to have a feeling for an average number of simple-places. In a first experiment, we take a look at the latest example net of the model checking contest (https://mcc.lip6.fr/2020/) to have a variety of different models. We calculate a set of 1000 runs of every net by simply randomly unfolding the net [6]. We implement Algorithm 3 and Algorithm 1 in Java to decide if the run is valid. Obviously, every run will be valid as a part of the unfolding. Yet, this will only increase the runtime of both algorithms. For a run of the language of a net, we have to calculate a distribution of tokens for every place. If a run is not valid, both algorithms can stop as soon as they find one place that is not valid. Roughly speaking, stopping early is an advantage for Algorithm 3, because the first two thirds of the algorithm are very fast. Using the examples, we compare the runtime of both algorithms and depict the number of simple places.

We perform the following two experiments on an Intel Core i5 3.30 GHz (4 CPUs) machine with 8 GB RAM running a Windows 10 operating system. The

implementation of both algorithms is available at https://www.fernuni-hagen.de/ilovepetrinets/.

Experiment 1. *We consider the most recent example called SatelliteMemory from the Model Checking Contest 2020. SatelliteMemory has two parameters X and Y defining the maximal number of tokens per place. We refer the reader to https://mcc.lip6.fr/2020/pdf/SatelliteMemory-form.pdf for a detailed description of the example and the parameters. Figure 13 depicts the structure of the p/t-nets, i.e., markings with many tokens, arc weights, short loops, and cyclic behavior. For each example, we randomly compute* 1000 *runs for every number* 100, 200, 300, *and* 400 *of events and decide enabledness using Algorithm 1 and Algorithm 3. Figure 13 depicts: (1) percentage of all places decided by firing once with the forward strategy, (2) percentage of all places decided by firing twice, once with the forward and once with the backward-strategy, (3) overall average run-time of Algorithm 3, (4) overall average run-time of Algorithm 1. We set the parameters to (a) X=*100 *Y=*3 *(*100 *tokens), (b) X=*1000 *Y=*32 *(*1000 *tokens), (c) X=*1500 *Y=*46 *(*1500 *tokens), (d) X=*3000 *Y=*94 *(*3000 *tokens), (e) X=*65535 *Y=*2048 *(*65535 *tokens).*



	100 events	200 events	300 events	400 events
a.	.34 .48 5ms 11ms	.31 .45 44ms 98ms	.28 .43 147ms 330ms	.26 .40 391ms 800ms
b.	.40 .55 4ms 15ms	.38 .54 34ms 115ms	.34 .50 115ms 310ms	.29 .41 287ms 608ms
c.	.39 .54 4ms 14ms	.38 .54 32ms 109ms	.36 .52 110ms 356ms	.35 .49 220ms 581ms
d.	.39 .54 4ms 14ms	.38 .53 32ms 105ms	.39 .54 108ms 352ms	.35 .53 267ms 936ms
e.	.39 .54 4ms 14ms	.38 .54 32ms 108ms	.38 .54 111ms 363ms	.36 .51 224ms 709ms

Fig. 13. Model and results of Experiment 1.

Experiment 1 considers a quite complex p/t-net model. The number of places in every combination of parameters is 13. We increase the number of tokens and the number of events. Experiment 1 shows that half of the places are simple-places in this example. The runtime of both algorithms grows quadratic with the size of the input, i.e., number of events, in this example. This fits perfectly with our considerations because, if tokens don't have to be redistributed often and the

number of places is fixed, the runtime of Algorithm 1 is in $O(|V|^2)$. Furthermore, if half of the places are simple, the run-time of Algorithm 3 is twice as fast but still in $O(|V|^2)$).

Experiment 2. *We consider the data set from the Process Discovery Contest 2020. The model has parameters defining the control-flow, i.e., dependent tasks, loops, or-constructs, routing constructs, optional tasks, and duplicate tasks, of the model. We refer the reader to https://icpmconference.org/2020/process-discovery-contest/data-set/ for a detailed description of the example and the parameters. The structure of the p/t-nets are typical workflow Petri nets with an initial and a final marking. For each example, we randomly compute* 1000 *runs from start to end and decide enabledness using Algorithm 1 and Algorithm 3. Figure 14 depicts: (1) file-name (2) average number of events per run (3) percentage of all places decided by firing once with the forward-strategy, (4) percentage of all places decided by firing twice, once with the forward and once with the backward-strategy, (5) overall average runtime of Algorithm 3, (6) overall average runtime of Algorithm 1.*

pdc_2020_0010000.pnml	15	.93	.99	0.043ms	0.284ms
pdc_2020_1000000.pnml	16	.95	1.0	0.036ms	0.296ms
pdc_2020_0001000.pnml	21	.87	.99	0.060ms	0.389ms
pdc_2020_0000000.pnml	21	.87	.99	0.051ms	0.412ms
pdc_2020_0000100.pnml	21	.87	1.0	0.041ms	0.419ms
pdc_2020_0000010.pnml	22	.87	.99	0.041ms	0.387ms
pdc_2020_1111110.pnml	25	.99	1.0	0.014ms	0.845ms
pdc_2020_1211110.pnml	37	.99	1.0	0.019ms	1.733ms
pdc_2020_1210110.pnml	37	.98	1.0	0.023ms	2.412ms
pdc_2020_0100000.pnml	50	.86	.97	0.208ms	2.432ms
pdc_2020_0200000.pnml	86	.82	.97	0.579ms	7.540ms

Fig. 14. Model and results of Experiment 2.

Experiment 2 considers a workflow p/t-net model with standard workflow patterns. All examples in this experiment have an initial and a final marking; thus, we cannot scale the size of the input as we did in Experiment 1. Only if the model contains loops, i.e., the second parameter in this example, the number of events of the randomly generated runs increases. Almost every place of this example is a simple place. In workflow models, most of the places are empty most of the time and can, thus, be handled by firing very easily. The runtime of Algorithm 1 grows quadratic with the size of the input. In Experiment 2, the overall runtime of Algorithm 1 is almost exactly $|V|^2/1000$ ms. The overall runtime of Algorithm 3 is much smaller and highly depends on the number of simple places. The algorithm is very fast for the examples in lines 7, 8, and 9 where almost every place is decided by the forward-strategy of the algorithm. Obviously, for those examples, the algorithm runs in linear time.

5 Conclusion and Future Work

This paper presents an approach to firing a partially ordered set of events in a Petri net model. The new approach also introduces the concept of local markings of a run and a marked p/t-net. With the help of this definition, it is possible to define a set of simple places and to decide enabledness fast.

The paper presents two experiments deciding enabledness of a run in models taken from two very different but well-known contests in the area of Petri nets. In the latest example of the Model Checking Contest, we deal with nets having cyclic behavior, complex net structure, and many tokens. In the latest example of the Process Discovery Contest, we deal with workflow nets with initial and final markings, workflow-patterns, control-flow structure, and only few tokens.

In both experiments, the new algorithm clearly outperforms the algorithm using compact tokenflows only. In that sense, there is never a disadvantage in trying to fire first. The new algorithm is especially fast in workflow-net-like p/t-nets. Here, we open the door to further applications in the area of business process modelling.

In future work, we would like to check if the set of simple places is a good indicator for the complexity of a process model. Let us say we want to discover or synthesize a p/t-net model from behavioral data recorded in terms of partial orders of events; maybe it is sufficient to only generate simple places to obtain a readable, well-structured process model.

References

1. van der Aalst, W.M.P., van Dongen, B.F.: Discovering petri nets from event logs. In: Jensen, K., van der Aalst, W.M.P., Balbo, G., Koutny, M., Wolf, K. (eds.) Transactions on Petri Nets and Other Models of Concurrency VII. LNCS, vol. 7480, pp. 372–422. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38143-0_10
2. van der Aalst, W.M.P.: The application of petri nets to workflow management. J. Circ. Syst. Comput. **8**(1), 21–66 (1998)
3. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs (1993)
4. Bergenthum, R., Lorenz, R.: Verification of scenarios in petri nets using compact tokenflows. Fundamenta Informaticae **137**, 117–142 (2015)
5. Bergenthum, R.: Faster verification of partially ordered runs in petri nets using compact tokenflows. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 330–348. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38697-8_18
6. Bergenthum, R., Lorenz, R., Mauser, S.: Faster unfolding of general petri nets based on token flows. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 13–32. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68746-7_6
7. Desel, J., Juhás, G., Lorenz, R., Neumair, C.: Modelling and validation with VipTool. Bus. Process Manag. **2003**, 380–389 (2003)

8. Desel, J., Juhás, G.: "What is a petri net?" informal answers for the informed reader. In: Ehrig, H., Padberg, J., Juhás, G., Rozenberg, G. (eds.) Unifying Petri Nets. LNCS, vol. 2128, pp. 1–25. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45541-8_1
9. Desel, J., Reisig, W.: Place/transition petri nets. In: Reisig, W., Rozenberg, G. (eds.) ACPN 1996. LNCS, vol. 1491, pp. 122–173. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-65306-6_15
10. Dumas, M., García-Bañuelos, L.: Process mining reloaded: event structures as a unified representation of process models and event logs. In: Devillers, R., Valmari, A. (eds.) PETRI NETS 2015. LNCS, vol. 9115, pp. 33–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19488-2_2
11. Desel, J., Erwin, T.: Quantitative Engineering of Business Processes with *VIPbusiness*. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 219–242. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40022-6_11
12. Fahland, D.: Scenario-based process modeling with Greta. BPM Demonstration Track 2010, CEUR 615 (2010)
13. Fahland, D.: Oclets – scenario-based modeling with petri nets. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 223–242. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02424-5_14
14. Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. Can. J. Math. **8**, 399–404 (1956)
15. Grabowski, J.: On partial languages. Fundamenta Informaticae **4**, 427–498 (1981)
16. Goltz, U., Reisig, W.: Processes of place/transition-nets. In: Diaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 264–277. Springer, Heidelberg (1983). https://doi.org/10.1007/BFb0036914
17. Juhás, G., Lorenz, R., Desel, J.: Can i execute my scenario in your net? In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 289–308. Springer, Heidelberg (2005). https://doi.org/10.1007/11494744_17
18. Karzanov, A.: Determining the maximal flow in a network by the method of preflows. Doklady Math. **15**, 434–437 (1974)
19. Kiehn, A.: On the interrelation between synchronized and non-synchronized behaviour of petri nets. Elektronische Informationsverarbeitung und Kybernetik **2**(1/2), 3–18 (1988)
20. Mannel, L.L., van der Aalst, W.M.P.: Finding complex process-structures by exploiting the token-game. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 258–278. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_15
21. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs (1981)
22. Reisig, W.: Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33278-4
23. Vogler, W. (ed.): Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55767-9