Token Trail Semantics – Modeling Behavior of Petri Nets with Labeled Petri Nets

Robin Bergenthum[1(✉)], Sabine Folz-Weinstein[2], and Jakub Kovář[3]

[1] Fakultät für Mathematik und Informatik, FernUniversität in Hagen, Hagen, Germany
robin.bergenthum@fernuni-hagen.de

[2] Lehrgebiet Softwaretechnik und Theorie der Programmierung, FernUniversität in Hagen, Hagen, Germany
sabine.folz-weinstein@fernuni-hagen.de

[3] Lehrgebiet Programmiersysteme, FernUniversität in Hagen, Hagen, Germany
jakub.kovar@fernuni-hagen.de

Abstract. There are different semantics for Petri nets. The behavior of a Petri net is either its set of enabled firing sequences, the reachability graph, a set of process nets, a valid partial language, its branching process, or any other known semantics taken from the literature. Every semantics has different advantages in different applications. Some focus on the set of reachable states and can model conflicts well. Other focus on the control flow of actions and can directly specify concurrency. Yet, every semantics has its drawbacks. State graphs explode in size when there is concurrency. Sequential and partial languages explode in size if there is conflict. Furthermore, all semantics use different concepts, definitions, graphical representations, and related algorithms. In this paper, we introduce token trails to define whether a labeled Petri net is in the language of another Petri net. Using labeled Petri nets as a specification language, we show how to faithfully model behavior including conflict and concurrency. Furthermore, we prove that token trail semantics faithfully covers all other semantics of Petri nets and, thus, serves as a kind of meta semantics.

Keywords: Petri nets · Labeled nets · Token trails · Semantics · Compact tokenflows · Modeling behavior · Conflict · Concurrency

1 Introduction

Petri nets [1, 9, 10, 18, 20] have formal semantics, an intuitive graphical representation, and can express conflict and concurrency among the occurrences of actions of a system. Petri nets model actions by transitions, local states by places, and the relations between actions and local states by arcs. We model a state of a Petri net by putting tokens in places. A marked Petri net can change its state by firing a transition. A transition can fire if every place in its pre-set is marked. If a transition fires, it consumes tokens from its pre-set and produces tokens in its post-set. This firing rule is very intuitive and easy to formalize. This surely is a big part of the success of Petri nets as a modeling language and why we love Petri nets.

The firing rule is the core of every Petri net semantics. Although the firing rule itself is simple, there are a lot of different semantics for Petri nets in the literature. There is the sequential language of a Petri net, there are state graphs, partially ordered runs, process nets, branching processes, prime event structures, and so on and so forth. Every semantics has its own advantages and disadvantages in different applications. On the one hand, every semantics is specialized, and we can choose the best fit for every application. On the other hand, it is a mess of different definitions to choose from and even if we choose the correct semantics there are drawbacks inherent to that choice.

For example, repeatedly processing the firing rule creates so called firing sequences. The set of enabled firing sequences is the language of a marked Petri net. This language is very easy to handle but it is not able to specify concurrency. It is easy to come up with a Petri net where we can fire two transitions in any order, but not concurrently. Furthermore, a firing sequence cannot directly specify conflict. When there is conflict, we need one firing sequence for every combination of options in every conflict. Thus, even for a simple Petri net, the size of the language may be huge.

The set of all reachable states, together with the set of all transitions from one state to another, is called the reachability graph of a Petri net. This semantics is still relatively easy to handle. In contrast to firing sequences, these state graphs can very conveniently express conflict, merging, and looping of sequences of actions. But again, state graphs cannot express concurrency. Even worse, if there is concurrency there is the so-called state space explosion where the number of global states grows exponentially in the number of local states.

There are step sequences and state graphs based on multisets of transitions. These semantics can specify concurrent sets of transitions but still, the number of global states explodes just like in every other state graph. Furthermore, sequences of steps are rather technical. Thus, it is neither easy nor intuitive to specify behavior using combinations of sequences of steps.

To model concurrency, there are partially ordered runs [7, 15, 19, 22]. A run is a firing sequence, but the sequence is a partial, not a total order. Thus, it is not sufficient to a have sequences of global markings enabling transitions anymore. We need partially ordered sets of local markings. These sets of markings are called compact tokenflows [3]. Using runs we can easily model concurrent behavior, but just like for firing sequences, it is not possible to directly specify conflict. Again, if there is conflict, we need a run for every possible combination of options.

Compact tokenflows in runs abstract from the history of tokens. We can consider labeled partial orders and regular tokenflows to include the history [16]. We can use so called process nets to include the history and even distinguish individual tokens. Yet, labeled partial orders and process nets have the same disadvantages as runs and compact tokenflows.

We can extend runs and process nets with an additional conflict relation and get prime event structures and branching processes [22]. These semantics can specify concurrency but also merge identical prefixes of runs or process nets. Remark, we can branch but not merge so that these structures fan-out and it is hard to keep track of the relations between the different conflict-free sets of partially ordered nodes. Furthermore, they are not able to directly define looping behavior.

Altogether, we identify two major problems: Firstly, although the definition of a Petri net is easy and clean, the different semantics are all over the place. Just to give one example, the concepts of valid regular tokenflows, enabled cuts in runs, sets of enabled step sequences, process nets, valid compact tokenflows, valid prime event structures, and branching processes all define the same partial language for every Petri net. Yet, there are these different definitions, proofs of their equivalence, different graphical representations, and different algorithms in the literature. Wouldn't it be nice to have some easy to understand meta semantic covering them all? Secondly, it is still not possible to come up with an intuitive and compact graphical representation of the behavior of a Petri net if there is conflict and concurrency.

In this paper, to tackle these problems, we refer the reader to the first sentence of this section. If there is conflict and concurrency, use Petri nets. We introduce token trail semantics for Petri nets. Using token trails, we define whether a labeled Petri net is in the language of another Petri net. To show that this is a valid and useful definition, we prove that if a labeled net models a firing sequence, a state graph, or a run, the labeled net is in the net language of a Petri net if and only if, the firing sequence, the state graph, or the run is in the language of this Petri net as well. Thus, the language defined by token trails will respect and cover all the above-mentioned semantics. Furthermore, we prove that every Petri net is in its own net language. We show examples of how to faithfully model behavior truly specifying conflict and concurrency generating readable graphical representations of executions using general labeled nets. We show how to calculate token trails and introduce a web-tool to demonstrate that token trails are a simple yet very powerful semantics for Petri nets.

2 Preliminaries

Let $\mathbb{N}$ be the non-negative integers. 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 $m\colon A \to \mathbb{N}$ a multiset and write $m = \sum_{a \in A} m(a) \cdot a$ to denote multiplicities of elements in m. Let $m' : \mathrm{A} \to \mathbb{N}$ be another multiset. We write $m \leq m'$ if $\forall a \in A : m(a) \leq m'(a)$ 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, $\left(V, <^\diamond\right)$ is called its Hasse diagram.

We model distributed systems by Petri nets [5, 9, 18, 20].

Definition 1. A Petri 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 $W\colon (P \times T) \cup (T \times P) \to \mathbb{N}$ is a multiset of arcs. A marking of (P, T, W) is a multiset $m\colon P \to \mathbb{N}$. Let m_0 be a marking, we call $N = (P, T, W, m_0)$ a marked Petri net and m_0 the initial marking of N.

Figure 1 depicts a marked Petri net. We show transitions as rectangles, places as circles, the multiset of arcs as a set of weighted arcs, and the initial marking as a set of black dots called tokens.

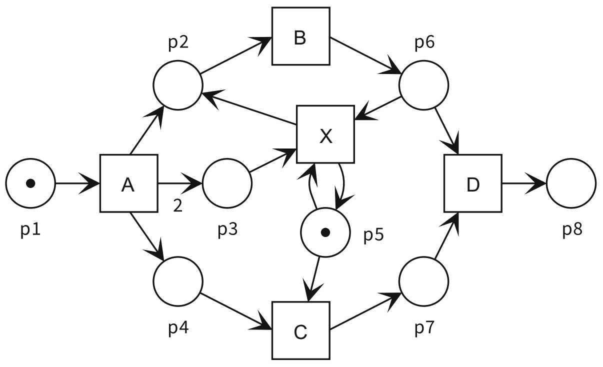

Fig. 1. A marked Petri net.

For Petri nets there is a firing rule. Let t be a transition of a marked Petri net (P,T,W,m_0). We denote $^\circ t = \sum_{p\in P} W(p,t)\cdot p$ the weighted pre-set of t. We denote $t^\circ = \sum_{p\in P} W(t,p)\cdot p$ the weighted post-set of t. A transition t can fire in marking m if $m \geq {}^\circ t$ holds. Once transition t fires, the marking of the Petri net changes from m to $m' = m - {}^\circ t + t^\circ$.

In our example marked Petri net, transition A can fire in the initial marking. If A fires, this removes one token from p_1. Additionally, firing A produces a new token in p_2, two new tokens in p_3, and a new token in p_4. In this new marking transitions B and C can fire. A is not enabled anymore, because there are no more tokens in p_1. Firing transition B will enable transition X and transition D. Firing transition C will disable transition X.

Repeatedly processing the firing rule of a Petri net produces so-called firing sequences. These firing sequences are the most basic behavioral model of Petri nets. For example, the sequence $ABXCBD$ is enabled in the marked Petri net of Figure 1. The sequence $ACBD$ is another example. Let N be a marked Petri net, the set of all enabled firing sequences of N is the sequential language of N.

Another formalism to model the behavior of a Petri net is the reachability graph. A marking is reachable if there is a firing sequence that produces this marking. The reachability graph of a marked Petri net $N = (P,T,W,m_0)$ is a tuple (R,T,X) where R is the set of reachable markings of N, T is the set of transitions of N, and X (called transitions as well) is a set of triples in $R\times T\times R$ so that (m,t,m') is in X if and only if t is enabled in (P,T,W,m), and firing t in m leads to the marking m'.

We call a tuple (R',T',X',i) a state graph enabled in N if there is an injective function $g\colon R' \to R$, $g(i) = m_0$, $T' \subseteq T$, $\forall(m,t,m') \in X' : (g(m),t,g(m')) \in X$, and for every $m' \in R'$ there is a directed path from i to m' using the elements of X' as arcs. Roughly speaking, every node of a state graph relates to a reachable state, we don't have to include all transitions and states if every node can be reached from the initial node. Thus, a state graph is kind of a prefix of a reachability graph. We call the set of enabled state graphs the state language of N.

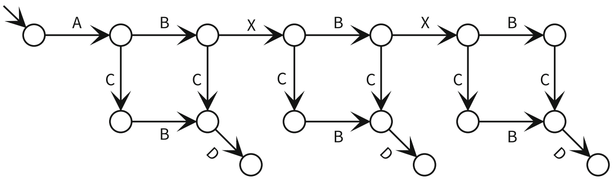

Fig. 2. A state graph of the Petri net of Figure 1.

Figure 2 depicts a state graph modeling the behavior of the marked Petri net depicted in Figure 1. The state graph has 16 states and 18 transitions labeled with transitions of the Petri net. The state graph describes the Petri nets behavior as follows. At first, we must fire transition A. Then, we have some choices. We can execute the loop $BXBXB$ until place p_3 of the Petri net is empty, or we can fire transition C at any time during this loop. As soon as we fire C, we disable transition X by removing a token from place p_5. If there is an occurrence of transition B after the last occurrence of transition X, we can fire transition D once.

The state language includes firing sequences as the set of all paths through the graphs. In this sense, state graphs can merge firing sequences on shared states and can contain loops. Yet, these graphs are not able to directly express concurrency.

Firing A in the initial marking depicted in Figure 1 leads to the marking $p_2 + 2 \cdot p_3 + p_4$. In this marking, transitions B and C can fire concurrently because they don't share tokens. Neither firing sequences nor state graphs can express this concurrency. Two transitions can occur in any order but cannot be executed at the same time. Therefore, there are additional semantics of Petri nets in the literature, able to explicitly express concurrency. There are step semantics of Petri nets [14], process net semantics of Petri nets [13], tokenflow semantics of Petri nets [16], and compact tokenflow semantics of Petri nets [3]. Fortunately, these semantics are equivalent [3, 16, 17, 21] and all define the same partial language. In a partial language, every so-called run is a partially ordered set of events. Obviously, runs can express concurrency and are a very intuitive approach to model behavior of a distributed system.

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.

Using tokenflow semantics or compact tokenflow semantics, we can decide if a run is in the partial language of a Petri net in polynomial time. Tokenflows, just like branching processes, track the history of tokens. Compact tokenflows define a partially ordered set of local states and thus, abstract from this history. Compact tokenflows are more efficient [4]. Roughly speaking, a compact tokenflow is a distribution of tokens on the arcs of a run so that every event receives enough tokens, no event must pass too many tokens, and all events share tokens from the initial marking.

Definition 3. Let $N = (P, T, W, m_0)$ be a marked Petri net and $run = (V, <, l)$ be a run so that $l(V) \subseteq T$ holds. A compact tokenflow is a function $x : (V \cup <) \to \mathbb{N}$. Let $v \in V$ be an event. We denote $in(v) := x(v) + \sum_{v' < v} x(v', v)$ the inflow of v, and $out(v) = \sum_{v < v'} x(v, v')$ the outflow of v. We define, x is valid for $p \in P$ iff the following conditions hold:

(i) $\forall v \in V : in(v) \geq \; W(p, l(v))$,
(ii) $\forall v \in V : out(v) \leq in(v) + W(l(v), p) - W(p, l(v))$, and
(iii) $\sum_{v \in V} x(v) \leq m_0(p)$.

run is enabled in N iff there is a valid compact tokenflow for every $p \in P$. The set of all enabled runs of N is the partial language of N.

Figure 3 depicts three different runs modeling the behavior of the marked Petri net depicted in Figure 1. Every run starts with executing transition A. The first run models the concurrent execution of transitions B and C before firing D. The second run models the execution of the loop BXB concurrently to transition C. But transition C can only occur after X because of place p_5. The third run models two times the loop, and again modeling that there is no occurrence of X after the occurrence of C.

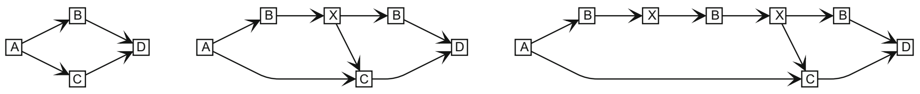

Fig. 3. Three runs of the Petri net of Figure 1.

Figure 4 depicts the branching process of the Petri net of Figure 1. For formal definitions we refer the reader to [13, 20]. Just note, that the maximal conflict-free sets of events of Figure 4 are the three runs of Figure 3. Using the branching process, we merge identical prefixes of the three runs and directly model conflict. But we can only branch and not merge, which is weird when we model a sequence of choices. Only the first choice will be modeled directly, any other following choice will be copied and distributed over the consistency sets of the branching process. Adding such upwards-closed conflict relation comes at a high cost of readability of the modeled behavior.

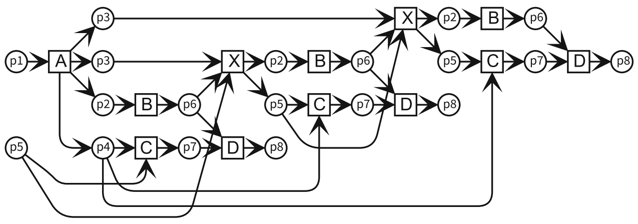

Fig. 4. The branching process of the Petri net of Figure 1.

Figure 2, Figure 3, and Figure 4 all model the behavior of the Petri net of Figure 1. Figure 2 is unable to express the concurrency of transitions B and C. Figure 3 needs three separate runs, because runs cannot contain conflict. Figure 4 is troublesome to read because it cannot merge states as state graphs can. Thus, there will always be some tradeoff choosing one semantics over the others.

3 Token Trails

In this section, we introduce token trails for Petri nets. Using token trails, we define whether a labeled Petri net is in the language of another Petri net.

We define the rise of a transition as the difference between the number of tokens in the pre-set and the number of tokens in the post-set of a transition. Whenever there are arc weights, the rise is the difference between the weighted sums.

Definition 4. Let $\mathrm{N} = (P, T, W, m)$ be a marked Petri net, let $t \in T$ be a transition. We denote the weighted sum of tokens $t^{\blacktriangle}$ as $t^{\blacktriangle} := \sum_{(p,t)\in W} W(p,t) \cdot m(p)$, the weighted sum of tokens $t^{\blacktriangle}$ as $t^{\blacktriangle} := \sum_{(p,t)\in W} W(p,t) \cdot m(p)$ and define the rise as $t^{\triangle}$ of transition t as $t^{\triangle} := t^{\blacktriangle} - t^{\blacktriangle}$.

We model behavior by labeled nets. A labeled net is just a regular Petri net but there is an additional set of actions, and every transition is labeled by one of them. We call a labeled net a plain marked labeled net if the labeled net is marked, every place is carrying at most one token in the initial marking, and there are no arc weights.

Definition 5. A labeled net is a tuple (C, E, F, A, l) where (C, E, F) is a Petri net, A is a finite set of actions, and $l : E \rightarrow A$ is an injective labeling function. A marking m of (C, E, F, A, l) is a marking of (C, E, F).
We call (C, E, F, A, l, m) a plain marked labeled net if (C, E, F, A, l, m) is a marked labeled net, $F \leq \sum_{f\in((C\times E)\cup(E\times C))} f$, and $m \leq \sum_{p\in C} p$ holds.

If a labeled net models behavior, similarly to process nets and branching processes, every transition models an event and every place models a condition. Thus, we also call them events and conditions. The arcs between events and conditions form the control flow of the behavior. Now, we need to define when a labeled net is in the language of a Petri net. We follow the ideas of compact tokenflows and model valid distributions of tokens between the events of our specification. But this time the relations between the events are given by a set of conditions and arcs, not by the later-than relation of the partial order. Still, every valid distribution must respect the firing rule, so that every event must receive enough tokens, every event must consume and produce the right number of tokens, and tokens from the initial marking can be freely distributed over a set of initial local states. To model the set of initial local states, we use an initial marking of the labeled net. Roughly speaking, tokens from the initial marking of the Petri net can be distributed to the initially marked places of the specification.

We call a valid distribution of tokens over the local states of a labeled net a token trail. Such a distribution is a multiset of conditions. Thus, it is straight forward to just formalize token trails as markings of a marked labeled net.

Definition 6. Let $S = (C, E, F, A, l, m_x)$ be a marked labeled net and let m be a marking of S. Let $\mathrm{N} = (P, T, W, m_0)$ be a marked Petri net, $A \subseteq T$, and $p \in P$ be a place. The marking m is a token trail for p iff

(I) $\forall e \in E: e^{\blacktriangle} \geq W(p, l(\mathrm{e}))$,
(II) $\forall e \in E : e^{\vartriangle} = W(l(e), p) - W(p, l(e))$, and
(III) $\sum_{c \in C} m_x(c) \cdot m(c) = m_0(p)$.

S is enabled in N iff there is a token trail for every $p \in P$. The set of all enabled labeled nets of N is the net language of N.

We just kind of brute-forced the definition of a token trail and the definition of the net language of a Petri net. Remark, although conditions (I) and (II) look just like the regular firing rule of Petri nets, they are not. They are derived from conditions (i) and (ii) of compact tokenflows. For example, in the firing rule, we require that there are enough tokens in every place in the pre-set of a transition. Fix a place p and a transition t, we need at least $W(p, t)$ tokens in p. In Definition 6, t needs $W(p, l(t))$ tokens as well, but these tokens can arrive at t over different paths through the labeled net. Tokens could have been produced earlier and then be distributed over the dependencies defined by the conditions of the specification until they arrive at t. Thus, the sum of all tokens distributed over all conditions in the prefix of t count towards the number $W(p, l(t))$ of tokens needed for $l(t)$ to be enabled. A token trail does not model a state of the Petri net, it models a distribution of tokens where tokens can travel using conditions of the specification. Obviously, this idea is directly taken from compact tokenflows, where tokens travel along the defined later-than relation.

In this paper, we lift the idea of compact tokenflows to arbitrary labeled nets. Tokenflows only travel in one direction, can synchronize, but not loop nor merge. Token trails can utilize every condition of a specification, thus, can split, branch, merge, synchronize, and loop. In the remainder of the paper, we show examples and prove that Definition 6 is well-defined. We argue that the net language of a Petri net covers, unifies, and extends existing semantics for Petri nets faithfully.

4 Token Trails for Transition Systems and Partial Languages

In this section, we prove that the net language covers the state language and the partial language of a Petri net. If we model state graphs and runs as marked labeled nets, we only need plain nets.

Let e be a transition of a plain marked labeled net. We denote ${}^\bullet e$ and $e^\bullet$ the pre-set and the post-set of e. Using plain nets only, we can simply calculate $e^{\blacktriangle} = \sum_{c \in {}^\bullet e} m(c)$ and $e^{\blacktriangle} = \sum_{c \in e^\bullet} m(c)$. Furthermore, in a plain marked labeled net, the initial marking is one-bound. We simplify condition (III) of Definition 6 to $\sum_{c \in m_x} m(c) = m_0(p)$.

Token Trails for Firing Sequences. A firing sequence specifies sequential behavior of a Petri net. The biggest application dealing with this type of semantics is of course process mining [2]. In process mining, observations of a running workflow system are recorded in (sequential) event logs. This behavioral specification is used to mine, evaluate, operate, and optimize business processes.

It is easy to model a firing sequence as a labeled net. Every event relates to exactly one transition of the firing sequence. Every condition relates to one element of the total order relation. Arcs connect events and conditions to form a sequence. Consequently, there is one place with an empty pre-set, and we mark this place as the starting point by one token in the initial marking of the labeled net.

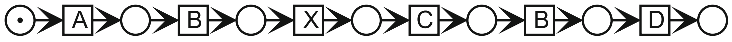

Fig. 5. A plain marked labeled net modeling the firing sequence *ABXCBD*.

Figure 5 depicts a marked labeled net modeling a sequence of transitions of a firing sequence. In figures of labeled nets, we show the labels of the transitions, not their name. In this example, there are two different transitions in the labeled net both labeled *B*. The token marks the first condition as the initial local state of the sequence.

Definition 8. We call a plain marked labeled net a **sequence net** if there is exactly one place *i* with an empty prefix, the initial marking is *i*, there is exactly one place *o* with an empty postfix, every other place and transition has exactly one predecessor and one successor, and there is a path from *i* to *o* visiting all transitions and places.

To construct a token trail for p_1 of Figure 1 in the labeled net depicted in Figure 5, we start by looking at condition (III). There is a token in the initial marking of p_1 of Figure 1 and therefore, we must put this token in the first condition of Figure 5. According to condition (I), the event labeled *A* gets enough tokens and, according to condition (II), this event must consume this token because its rise needs to be -1. Now, every other condition is unmarked, because, according to condition (II), the rise for all other labels must be 0. Remark, we mentioned that the difference between the regular firing rule and the conditions of a token trail is that we sum and weight all tokens in the prefix of an event. In a sequence net there are no weights and there is only one condition in every prefix, thus, there is no actual difference between the regular firing rule and the rules of token trails.

Figure 6 depicts eight copies of the sequence net of Figure 5. The first copy shows the token trail for place $p1$ of Figure 1. The second copy shows the token trail for $p2$ of Figure 1 and so forth. There is a token trail for every place of Figure 1 and thus, the sequence net depicted in Figure 5 is in the net language of the Petri net of Figure 1.

We look at the columns of conditions in Figure 6 to directly see the one-to-one relation between a set of token trails of a sequence net and the set of markings enabling a firing sequence. This comes down to the fact that transitions neither branch nor merge, so we can prove the following theorem.

Theorem 1. Let *S* be a sequence net. *S* models a firing sequence of a Petri net *N* iff there is a token trail in *S* for every place of *N*. Furthermore, the token trail in *S* for a place *p* is the *p*-component of markings generated by the firing sequence.

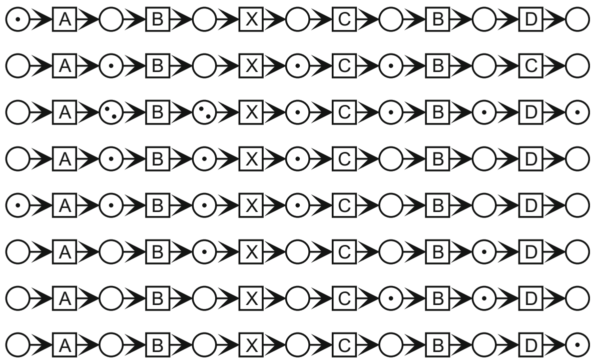

Fig. 6. Eight token trails for the places of Figure 1 in copies of the labeled net of Figure 5.

Proof. Let $t_0 t_1 \ldots t_n$ be a firing sequence of $N = (P, T, W, m_0)$. There is a unique sequence of markings $m_1 m_2 \ldots m_{n+1}$ so that t_i is enabled in m_i, and firing t_i in m_i yields m_{i+1}. If a marked sequence net $S = (C, E, F, A, l, m_x)$ models the firing sequence $t_0 t_1 \ldots t_n$, we can rename the elements of the net as follows: There is one place c_0 with an empty prefix carrying one token, transitions $e_0, e_1, \ldots, e_n$ labeled $l(e_i) = t_i$ and places $c_1, c_2, \ldots, c_{n+1}$ so that the multiset of arcs is $\sum_i ((c_i, e_i) + (e_i, c_{i+1}))$, $A = T$, and $m_x = c_0$.

Fix a place $p \in P$. The marking $m = \sum_i m_i(p) \cdot c_i$ of S is the token trail for p because $\sum_{c \in m_x} m(c) = m(c_0) = m_0(p)$ and thus condition (III) holds. For all e_i, firing t_i in m_i yields m_{i+1}. Thus, $m_{i+1} = m_i - {}^\circ t_i + t_i^\circ$ holds. We look at the p-component of this equation to get $m_{i+1}(p) = m_i(p) - W(p, t_i) + W(t_i, p)$ and thus, $e_i^\Delta = m(c_{i+1}) - m(c_i) = m_{i+1}(p) - m_i(p) = W(l(e_i), p) - W(p, l(e_i))$. This is condition (II). For all t_i, t_i is enabled in m_i. Thus, $m_i \geq {}^\circ t_i$ holds. Again, $m_i(p) \geq W(p, t_i)$ and $\sum_{c \in {}^\bullet e_i} m(c) = m(c_i) = m_i(p) \geq W(p, l(e_i))$ and thus condition (I) holds as well. Furthermore, for every p, m is completely defined by the sequence of markings $m_1 m_2 \ldots m_{n+1}$.

Let S be a sequence net and for every $p \in P$ let m_p be a token trail for p. Conditions (I), (II), and (III) hold and we use the same arguments as above backwards to construct the p-components of a sequence of markings $m_1 m_2 \ldots m_{n+1}$ enabling $t_0 t_1 \ldots t_n$ in m_0. ■

Theorem 1 shows that token trails respect the definitions of firing sequences. A firing sequence is in the sequential language of a Petri net if and only if the related sequence net is in the net language.

Token Trails for Transition Systems. A firing sequence models behavior as a sequence of actions. We use transition systems to model behavior focused on states. The biggest application dealing with this type of semantics is asynchronous circuit design [8]. In circuit design, we specify behavior as a transition system and use region-based approaches to synthesize a Petri net to be implemented.

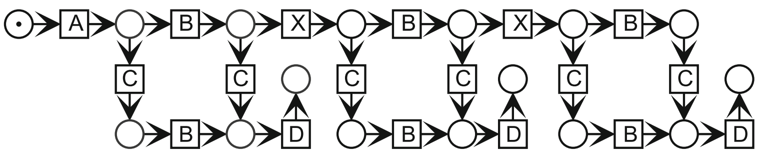

Fig. 7. A plain marked labeled net modeling the state graph of Figure 2.

Again, it is very easy to model a transition system in terms of labeled nets. Every state is a condition, and every transition of the state graph is an event of the labeled net connecting two conditions with one ingoing and one outgoing arc.

Definition 9. We call a plain marked labeled net a **state graph net** if there is exactly one place i so that there is a path from i to any other place, the initial marking is i, and every transition has exactly one predecessor and one successor.

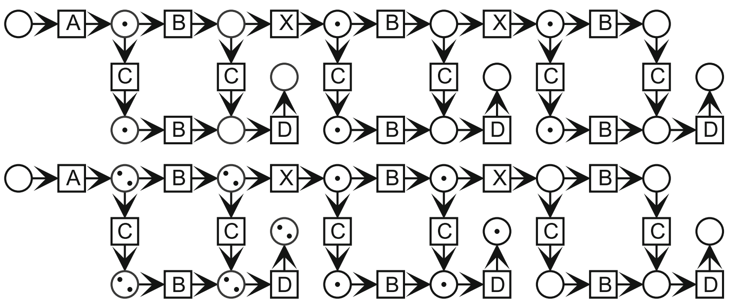

Fig. 8. The token trails for the place p_2 and p_3 of Figure 1 in the labeled net of Figure 7.

Figure 8 depicts two copies of the state graph net of Figure 7 with two token trails. The marking of the first state graph net is a token trail for p_2 of Figure 1. The initial marking of p_2 is 0, thus, the marking of the initial place of Figure 8 must be 0 as well. The rise of the event labeled A is 1, all events labeled B have a rise of -1, all events labeled C or D have a rise of 0, and all events labeled X have a rise of 1. The marking of the second state graph net of Figure 8 is a token trail for place p_3 of Figure 1.

Like in sequence nets, events of a state graph net do not branch. Thus, there is just one place in the pre-set of every event where tokens can arrive, and there is only one place where an event can pass the tokens to. We build a token trail as follows: we put the number of initial tokens in the initial place of the labeled net. Then we start a breadth-first search at the initial place. For every visited event, we have the number of ingoing tokens and just calculate the tokens to put in the successor place, using the rise of the label. This will construct a token trail if merging paths agree on the number of tokens and if no marking must be negative. Obviously, this is the case if the state graph net models reachable states of the original Petri net, only considering the place p. Like for sequence nets, there is a one-to-one relation between a set of token trails of a state graph net and the set of markings of a state graph in the language of the Petri net.

Theorem 2. Let S be a state graph net. S models a state graph of a Petri net N iff there is a token trail in S for every place of N. Furthermore, the token trail in S for a place p is the p-component of every state of the state graph.

Proof. Let $G = (R, T, X, i)$ be a state graph of $N = (P, T, W, m_0)$. For every transition $(m', t, m'') \in X$, t is enabled in m' and firing t in m' yields m''. If a marked state graph net $S = (C, E, F, A, l, m_x)$ models G, we can rename this net as follows: there is a transition $e_{(m',t,m'')}$ labeled $l\big(e_{(m',t,m'')}\big) = t$ for every transition, and a places p_r for every state $r \in R$ so that the multiset of arcs is $\sum_{(m',t,m'') \in X} ((p_{m'}, t) + (t, p_{m''}))$, $A = T$, and $m_x = p_i$.
For every state s of G there is a cycle free path of transitions leading from the initial state to s. This path is a firing sequence. $S = (C, E, F, A, l, m_x)$ models G and all transitions of S have exactly one predecessor and exactly one successor. The path in G relates to a subnet in S so that this net is a sequence net. Fix a place $p \in P$, we apply Theorem 1 to get $m = \sum_{r \in R} r(p) \cdot p_r$ is the only candidate for a token trail in S for p. Again, there is a one-to-one relation between the p-component of every state of the state graph and the token trail in S for p. Even if a transition (m', t, m'') in G is part of a cycle, $e_{(m',t,m'')}$ is unbranched so that, $e^{\triangle}_{(m',t,m'')} = m_{m''}(p) - m_{m'}(p) = m''(p) - m'(p) = W(t, p) - W(p, t)$ and $\sum_{c \in {}^{\bullet}e_{(m',t,m'')}} m(c) = m_{m'}(p) = m'(p) \geq W(p, t)$ holds. m is a token trail for p. Again, the p-components of the set of states define the token trail for every place. Using the same arguments backwards, we also get the other direction. ∎

Theorem 2 shows that token trails respect the definitions of state graphs. A transition system is a state graph of a Petri net if and only if the related state graph net is in the net language.

Token Trails for Partial Languages. A partial language models behavior as partially ordered sequences of actions. Using partial languages, we can model concurrency of action occurrences. The biggest application dealing with this kind of semantics is business process management [12]. Modern business processes, like for example Order-to-Cash, Quote-to-Order workflow processes, are distributed over different people, departments, and systems of a company, so that we model these processes using partially ordered runs.

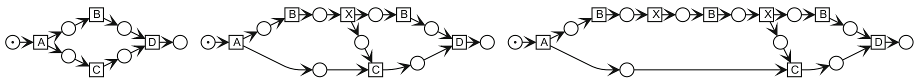

Fig. 9. Three labeled nets modeling the partially ordered sequences of actions of Figure 3.

Modeling a partial order of events in terms of labeled nets is easy. Every event is a labeled transition, and we model the skeleton of the partial order by a set of conditions. We add a condition between two transitions whenever there is a later-than relation between the two related events. Thus, every condition has at most one ingoing and one outgoing arc. There is concurrency but no conflict. Figure 9 depicts three labeled nets modeling three runs.

Definition 10. We call a plain marked labeled net a **partial order net** if the net is acyclic, every transition has at least one ingoing and at least one outgoing arc, every place has at most one ingoing and at most one outgoing arc, and the initial marking is the sum of places with an empty prefix.

Figure 10 depicts three copies of the three partial order nets of Figure 9 with token trail markings. For partial order nets, for the first time in this paper, we actually have to sum-up ingoing and outgoing tokens in order to calculate the rise of a transition. For example, the rise of transition A is 1 in the first row of examples, 2 in the second row, and 0 in the last row.

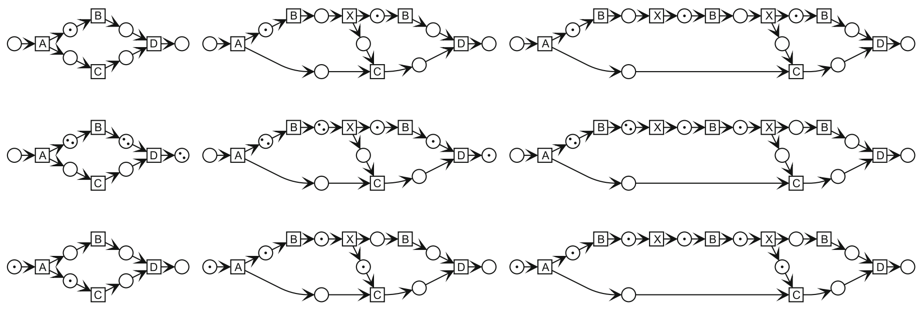

Fig. 10. Token trails for places p_2, p_3, and p_5 of Figure 1 in the labeled nets of Figure 9.

In Figure 10, the token trails of the first row all relate to place p_2 of Figure 1. The token trails of the second row relate to place p_3, and the token trails of the last row relate to p_5. Remark, in partial order nets there is not a one-to-one relation between the rises of events to token trails anymore. For example, the token trail in the middle of Figure 10 relates to place p_3. One token travels from X to the final marking. This token could also travel via C and thus lead to another token trail related to p_3. For partial order nets, there is a one-to-one relation between a token trail in the partial order net and a compact tokenflow in the labeled Hasse diagram.

Theorem 3. Let S be a partial order net. S models a run of a Petri net N iff there is a token trail in S for every place of N. Furthermore, there is a one-to-one relation between token trails in S and compact tokenflows in the run.

Proof. Let $run = (V, <, l)$ be a run of $N = (P, T, W, m_0)$. For every $p \in P$ there is a compact tokenflow x in run for p. If a marked partial order net $S = (C, E, F, A, l, m_x)$ models run, we can rename this net as follows: for every event $v \in V$ there is a transition $e_v \in E$ with $l(e_v) = l(v)$, for every arc $(v, v') \in <$ there is a place $c_{(v,v')} \in C$ so that ${}^\bullet c_{(v,v')} = \{e_v\}$ and $c_{(v,v')}{}^\bullet = \{e_{v'}\}$. For every $v \in V$ with an empty prefix there is

a place c_v^i in the pre-set of v, for every $v \in V$ with an empty postfix there is a place c_v^f in the post-set of v, $A = T$, and $m_x = \sum_{c \in C, {}^\bullet c = \emptyset} c$.
Fix a place $p \in P$ and its compact tokenflow x. Conditions (i), (ii), (iii) hold. The main idea is to construct another valid compact tokenflow x so that (b) and (c) hold as well.

(b) $\forall v \in V, v^\bullet \neq \emptyset : out(v) = in(v) + W(l(v), p) - W(p, l(v))$,
(c) $\sum_{v \in V, {}^\bullet v \neq \emptyset} x(c) = m_0(p)$.

As long as there is an event v with a non-empty prefix so that its outflow is not yet as big as the inflow plus $W(l(v), p) - W(p, l(v))$, there is a path from v to an event with an empty post-set. We can add the missing tokens to every arc of this path to fix the outflow of v. For every event on this path, inflow and outflow will be increased by the same number, for the last event only the inflow will be increased so that (i), (ii), (iii) still hold. We repeat until (b) holds. If there is an event v with a non-empty prefix so that $x(v) > 0$, there is a path from an event v' with an empty prefix to v. Again, we add $x(v)$ to $x(v')$, add $x(v)$ tokens to every event on the path and set $x(v)$ to 0 to move tokens consumed from the initial marking to the initial events, only increasing inflow and outflow of every event by the same amount. Now, the value of the compact tokenflow is 0 on non-initial events. If the sum of this tokenflow is not yet as big as the number of tokens in the initial marking, we add tokens on a path from an event with an empty prefix to an event with an empty postfix. Altogether, we construct a compact tokenflow x so that (i), (ii), (iii), (b), and (c) hold.
We construct a token trail m in S for p as follows. $m := \sum_{(v,v') \in <} x(v, v') \cdot c_{(v,v')} + \sum_{v \in V, {}^\bullet v = \emptyset} x(v) \cdot c_v^i + \sum_{v \in V, v^\bullet = \emptyset} (in(v) + W(l(v), p) - W(p, l(v))) \cdot c_v^f$.
m is a token trail for p because the only difference between x and m is that we moved tokens from the initial marking to the initial places and added the right number of tokens to the final places so that (b) implies (II). Obviously, (i) implies (I), and (c) implies (III) because all initial places of S are marked with one token in m_x.
This time the other direction is even simpler because we build a valid compact tokenflow from a token trail by just ignoring tokens in the final places and moving tokens from the initial places to the initial events. Now, (I), (II), and (III) imply (i), (ii), and (iii) directly without even adapting tokenflow on the edges of *run*. ■

Theorem 3 shows that token trails cover the definitions of compact tokenflows. A run is in the partial language of a Petri net if and only if the related partial order net is in the net language.

Like in the previous subsection, going from firing sequences to state graphs, we can add conflict to a run of a Petri net. To add conflict, we use the formalism of labeled prime event structures of a Petri net. The main idea is that there is an additional conflict relation so that sets of conflict-free events are runs. The conflict relation is upwards closed so that runs branch, but do not merge. Thus, every prime event structure is covered by runs and is enabled in a Petri net if every run of the prime event structure is enabled with a compact token flow so that the flows match on shared prefixes of the prime event structure. We add this idea to the proof of Theorem 3, like going from proof of Theorem 1 to the proof of Theorem 2, to get that there are also matching token trails.

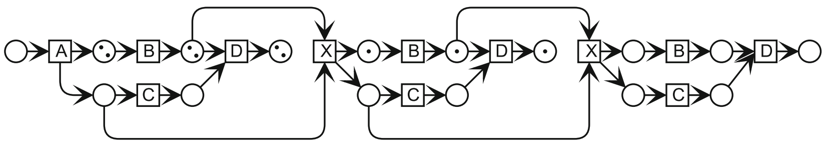

Fig. 11. Labeled net modeling the prime event structure for the three runs of Figure 9 marked with a token trail for place p_3 of Figure 1.

Figure 11 depicts a labeled net modeling a prime event structure. The conflict free sets of events of this net are the runs of Figure 9. In this prime event structure net we don't show its initial marking (just the first condition), but depict a token trail for place p_3 of Figure 1 as a composition of the three token trails for p_3 depicted in the second row of Figure 10.

A labeled net is a branching process of a Petri net if there is an additional one-to-one relation between tokens in the Petri net and tokens in the token trails. This must be examined in future work. Up to this point, the net language covers runs, as well as prime event structures of a Petri net.

5 Token Trails for Labeled Petri Nets

In Section 4, we proved that token trails cover firing sequences, state graphs, and partial languages using plain marked labeled nets only. In this section, we show that the token trail semantics is a well-defined generalization of existing semantics, and we show how to model behavior using general marked labeled nets. We show examples of labeled nets and Petri nets to argue that some labeled nets model behavior of these Petri nets and some do not.

Token Trails for Arbitrary Initial Markings. In the definition of a token trail, we consider arbitrary initial markings. Up to this point, only modeling firing sequences, state graphs, and partial languages there is no need to put multiple tokens in conditions.

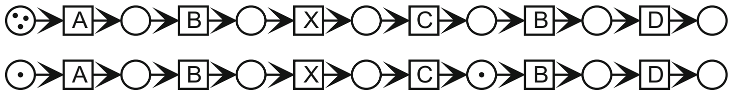

Fig. 12. Two marked labeled nets.

The first labeled net in Figure 12 depicts our running example sequentially ordered net, but this time with an initial marking of three tokens in the first place. Thus, it is neither a plain marked labeled net, nor a sequence net anymore. However, if we consider the token trails of Figure 6, trails number 2, 3, 4, 6, 7, and 8 are still token trails for the related places p_2, p_3, p_4, p_6, p_7, and p_8 of Figure 1. If we change the initial marking of Figure 1 to $3 \cdot p_1 + 3 \cdot p_5$, then token trails 1 and 5 would relate to p_1 and p_5. The first labeled net in Figure 12 is only in the net language of a Petri net if the sequence *ABXCBD* can be executed concurrently three times to itself. This perfectly matches our intuition looking at the first net of Figure 12.

The second labeled net in Figure 12 depicts an even more sophisticated initial marking. We specify that we start the sequence at the beginning and simultaneously before the last two actions *B* and *D*. In this example, the first token trail of Figure 6 is still a trail for p_1 of Figure 1 because the fifth place of the sequence is not marked so that the sum defined by condition (III) is 1 and thus, is the initial marking of p_1 in Figure 1. Similarly, the token trails 4, 5, 6, and 8 still hold for their related places. If we consider places p_2 and p_7, we must add one token each to the initial marking of the Petri net of Figure 1 to fix trails number 2 and 7. This is consistent with the intuition behind the specification in Figure 12 because the marked Petri net should be able to execute *BD* from the initial marking. However, something is strange about the third token trail of Figure 6 because p_3 is not connected to transitions *B* and *D* but still, we must add one token to the initial marking of p_3 so that the third token trail of Figure 6 is a token trail for p_3. But if we really think about the behavior we specify, we see that in Figure 12, using the first initial token, the sequence of events *ABXCB* reaches the same condition as using the second initial token and just execute the second event labeled *B*. In other words, we specify that we want to have a Petri net so that firing *ABXCB* and firing *B* leads to the same state. This is only satisfied if we add one additional token to p_3.

We can add an initial marking to any kind of labeled net to model a multiset of local states as starting points. This is the main idea of condition (III) of Definition 6.

Token Trails for Merged Local States. Figure 13 depicts labeled nets modeling the behavior of our running example in terms of marked labeled nets. There is conflict, a merge, a split, and synchronization. Obviously, they are neither a state graph net nor a partial order net. The upper net of Figure 13 specifies behavior where we start by an action *A*. Then there is a split. In the top part of the split, we have the choice between the sequence just *B*, or the sequence of actions *BXB*. Executing either of these two sequences will lead to the merge at the local state c_6. In parallel to this choice, we have action *C* followed by a synchronization using action *D*. This kind of control flow seems to fit the Petri net depicted in Figure 1.

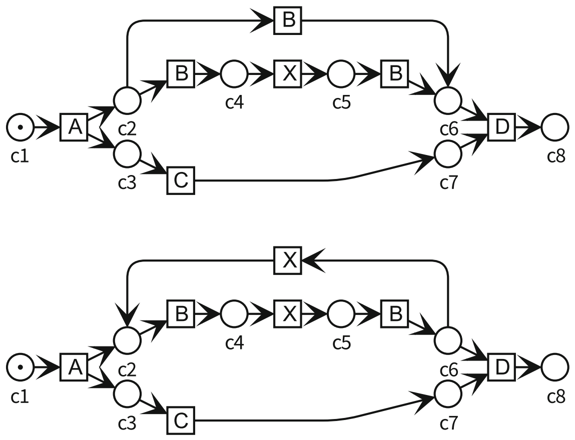

Fig. 13. A marked labeled net with a merge and a marked labeled net with a loop.

Now, we check if the upper labeled net of Figure 13 is in the net language of the Petri net of Figure 1 by constructing a token trail for every place. Figure 14 depicts the token trails for places p_1, p_2, p_6, p_4, p_7, and p_8.

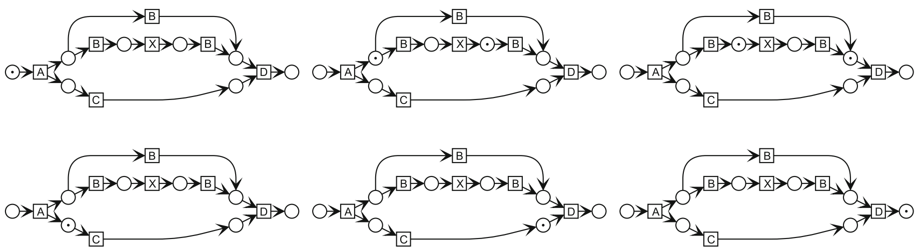

Fig. 14. Token trails in copies of the labeled net of Figure 13 for some places of Figure 1.

We still miss a token trail for the other places. For the place p_3, the rise of X must be -1 and thus, we need at least one token in c_4. Assume there are n tokens in c_4. The rise of B must be 0 and thus, there must be n tokens in c_2 as well. With the same argument there must be n tokens in c_6, and there must be n tokens in c_5, all because B must have a rise of 0. Consequently, the number of tokens in c_4 is the number of tokens in c_5, therefore, the rise of X cannot be -1. There is no token trail for the place p_3 in the upper labeled net of Figure 13. Remark, this is perfectly fine with our intuition! In the first labeled net of Figure 13, we specify that executing the loop, and not executing the loop, leads to the same local state c_6. Thus, if the upper net of Figure 13 is in the net language of some Petri net, executing or skipping the loop must lead to the same state. This holds for places p_1, p_2, p_6, p_4, p_7, and p_8, but not for place p_3. If we skip the loop, there are two tokens left in p_3. If we execute the loop once, there is one token left in p_3. Altogether, the upper labeled net in Figure 13 specifies that counting iterations is not allowed. It is only in the net language of Figure 1 if we were to delete p_3 and thus, allow for arbitrary iterations of B and X.

If we want to change the specification and model that counting should be possible, we must split the local state called c_6 in the upper net of Figure 13 into two separate states. Thus, we would end up with a branching process. Here, token trail semantic perfectly handles merging of local states.

We are still missing a token trail for place p_5. This place is initially marked, therefore, c_1 must be initially marked as well. The rise of A must be 0 so that the initial token can either go to c_2 or to c_3. The rise of C is -1 so that we must mark c_3 and there is no token left for c_2. Thus, there is no token for c_4 and X is not enabled according to (I). Again, there is no token trail for p_5. The reason is that p_5 ensures that C can only be executed after the execution of X. The upper net of Figure 13 models C and X as independent. Thus, it is correct that there is no token trail for p_5.

Summing up the first labeled net of Figure 13, if we delete p_3 and p_5 from Figure 1, the upper net of Figure 13 is in the net language of Figure 1. If we keep p_5 and add a condition from X to C, as is the case in the partial order nets of Figure 9, the initial token can go to X and another token from X to C. This labeled net would be in the net language of Figure 1 without p_3. If we want to count loops, we must split c_6 to allow a branched local state. These examples highlight how token trails deal with merging of alternative executions and shared local states.

Token Trails for Loops. We just indirectly specified looping behavior using a shared local state. What if the specification directly models a loop? The second labeled net of Figure 13 depicts an example where we specify a similar behavior but this time using a loop. Again, there is a loop of B and X in parallel to the action C. Here, we directly specify that there is at least one execution of X and at least two executions of B before another X can return to the local state c_2. With the same arguments as before, we can't construct a token trail for place p_3 where X (or B) is producing, and D is consuming tokens because we must always be able to go back to c_2. Again, it is not possible to count Bs and Xs.

We can copy the token trails of Figure 14 into the second labeled net of Figure 13 to directly see that there are token trails for the places p_1, p_2, p_6, p_4, p_7, and p_8. There is no token trail for p_3 because of the merged local state c_2 and there is no token trail for p_5 because C and X are modeled as independent again. Thus, again token trails match our intuition and handle Figure 13 correctly, although it is a specification with a loop.

To really put token trails to the test we introduce the first Petri net of Figure 15. This example might look strange at first, but we want to have a bounded Petri net where we must execute the loop of our running example at least once. Here, after the first execution of B the place p_5 is not marked yet. We execute X to mark p_5 and enable B again. After firing B a second time p_5 and p_6 are marked, enabling D. We can fire Y to move a token from p_5 to p_3 to reset the marking to the state already visited after firing the first B.

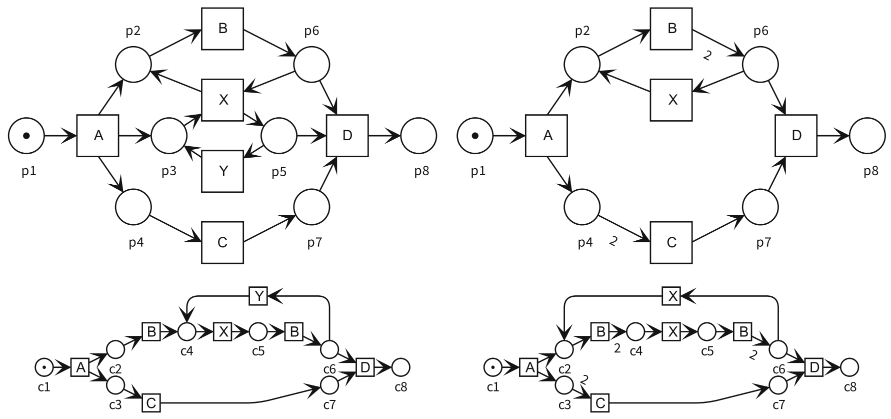

Fig. 15. Two Petri nets and two labeled nets of their languages.

In Figure 15, the first labeled net is in the net language of the first Petri net. Intuitively, Figure 15 specifies that neither the local states c_2 and c_5 nor the local states c_4 and c_6 are merged. Using an additional action Y, we can distinguish entering and reentering the loop. Here, the token trail for place p_5 is simply $c_5 + c_6$. The token trail for place p_3 is simply $c_2 + c_4$. Token trails faithfully handle merges and loops.

Token Trails for Weighted Arcs. The second Petri net of Figure 15 depicts our running example net with additional arc weights. The second labeled net of the same figure is in its net language. We need condition c_3 twice to execute the event labeled C. Whenever we execute an event labeled B, we produce condition c_4 or c_6 twice. The marking c_3 in the labeled net is a token trail of p_4. The rise of the event labeled A is still 1, the rise of the event labeled C is -2. The marking $c_4 + c_6$ is a token trail for p_6. Remark, in the second Petri net of Figure 15, C is not enabled after firing A in the initial marking, but the same holds for the depicted labeled net. Token trails just respect condition c_3 and the related arc weights.

At the end of this section, we show one more strong argument highlighting that the net language is well-defined. We prove that every labeled net without duplicate labels is in its own net language.

Theorem 4. Let S be a marked labeled net without duplicate labels, so that S models a Petri net N. S is in the net language of N.

Proof. Let $N = (P, T, W, m_0)$ be a Petri net and $S = (C, E, F, A, l, m_x)$ be a labeled net without duplicate labels. S models N, so we rename all elements of S so that $S = (P, T, W, T, id, m_0)$ holds. Fix a place $p \in P$ in N, $m = p$ is a token trail for p in S because $m(c)$ is only 1 for $c = p$ and 0 for any other condition.
Conditions (I), (II), and (III) hold because $\forall e \in T : \sum_{c \in {}^\bullet e} W(c, e) \cdot\ m(c) = W(p, t)$, $\forall e \in T : \sum_{c \in P} (W(e, c) - W(c, e)) \cdot m(c) = (W(e, p) - W(p, e)) \cdot m(p) = W(e, p) - W(p, e)$, and $\sum_{c \in C} m_0(c) \cdot m(c) = m_0(p)$. ■

Every Petri net is in its own net language. Token trails work perfectly fine for any kind of state-based or event-based specification, as well as for general labeled nets with loops, initial markings, and arc weights.

Calculating Token Trails. We implemented token trails as a new module of the I ♥ Petri Nets website. The website is available at www.fernuni-hagen.de/ilovepetrinets/. The 🦊 module implements the conditions of Definition 6 as a simple Integer Linear Program. We can drag a labeled net and a Petri net to the related Buttons and see if there are token trails for every place of the Petri net. Click on some place to see an example token trail in the labeled net. At www.fernuni-hagen.de/ilovepetrinets/fox/ please find the webtool and all the examples used in this paper.

6 Conclusion

In this paper, we introduced token trails. A token trail is a distribution of tokens on the set of conditions of a labeled net respecting the consumption and production of tokens of the labels of the events. Whenever there is conflict, a token trail must agree on the number of tokens in every condition of a labeled net. Whenever there is concurrency, a token trail can distribute tokens to local states. We have proven that token trails cover firing sequences, state graphs, and partial languages of Petri nets. Furthermore, they faithfully extend Petri net semantics to labeled nets. Token trails have a very intuitive graphical representation and are very easy to calculate. In addition to Petri nets, we only need the additional concept of labels to directly define token trail semantics. For all these reasons, we see token trails as a kind of meta semantics for Petri nets.

Besides the rather formal stuff, we see a lot of applications for token trails. In future work, we will define synthesis based on token trails. The goal is to come up with a framework so that we specify behavior of a system in terms of labeled nets and get the resulting Petri net for free. Using token trails, we can unify the definitions of state-based and event-based regions. We refer the reader to the workshop paper [6] for a first glimpse at this new region definition. Roughly speaking, we will specify labeled nets and calculate a set of places for the set of labels so that for every place there is a token trail using the ILP defined in Definition 6. We use the concept of minimal token trails (i.e., minimal number of tokens) to get a finite result. A prototype of the approach is already implemented in the 🐴 module of the I ♥ Petri Nets website.

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., Carmona, J.: Process Mining Handbook. Springer (2022). https://doi.org/10.1007/978-3-031-08848-3
3. Bergenthum, R., Lorenz, R.: Verification of Scenarios in Petri Nets Using Compact Tokenflows. In: Fundamenta Informaticae, vol. 137, no. 1, pp. 117–142. IOS Press (2015)
4. Bergenthum, R.: Firing Partial Orders in a Petri Net. In: Buchs, D., Carmona, J. (eds.) PETRI NETS 2021. LNCS, vol. 12734, pp. 399–419. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76983-3_20
5. Bergenthum, R.: Petrinetze: Grundlagen der Formalen Prozessanalyse. In: Prozessmanagement und Process-Mining, De Gruyter Studium, pp. 125–152. De Gruyter (2021)
6. Bergenthum, R., Kovar, J.: A First Glimpse at Petri Net Regions. In: Proceedings of Application and Theory of Petri Nets 2022, CEUR Workshop Proceedings 3167, pp. 60–68 (2022)
7. Best, E., Devillers, R.: Sequential and Concurrent Behaviour in Petri Net Theory. In: Theoretical Computer Science 55, nr. 1, pp. 87–136. Elsevier (1987)
8. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Hardware and Petri Nets Application to Asynchronous Circuit Design. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 1–15. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44988-4_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. Desel, J., Juhás, G.: What is a Petri Net? In: Ehrig, H., Juhás, G., Padberg, J., Rozenberg, G. (eds.) Unifying Petri Nets, Advances in Petri Nets, LNCS 2128, pp. 1–25. Springer, Cham (2001). https://doi.org/10.1007/3-540-45541-8_1

11. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM Framework: A New Era in Process Mining Tool Support. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 444–454. Springer, Heidelberg (2005). https://doi.org/10.1007/11494744_25
12. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33143-5
13. Goltz, U., Reisig, W.: Processes of Place/Transition-Nets. In: Diaz, J. (eds.) Automata Languages and Programming, vol. 154, pp. 264–277. Springer, Heidelberg (1983). https://doi.org/10.1007/BFb0036914
14. Grabowski, J.: On Partial Languages. In: Fundamenta Informaticae, vol. 4, no. 2, pp. 427–498. IOS Press (1981)
15. Janicki, R., Koutny, M.: Structure of Concurrency. In: Theoretical Computer Science 112, no. 1, pp. 5–52. Elsevier (1993)
16. Juhás, G., Lorenz, R., Desel, J.: Can I Execute My Scenario in Your Net? In: Ciardo, G., Darondeau, P. (eds.) Proceedings of Application and Theory of Petri Nets 2005, LNCS 3536, pp. 289–308. Springer, Heidelberg (2005). https://doi.org/10.1007/11494744_17
17. Kiehn, A.: On the Interrelation Between Synchronized and Non-Synchronized Behavior of Petri Nets. In: Elektronische Informationsverarbeitung und Kybernetik, vol. 24, no. 1–2, pp. 3–18 (1988)
18. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs (1981)
19. Pratt, V.: Modelling Concurrency with Partial Orders. In: International Journal of ParallelProgramming 15, pp. 33–71 (1986)
20. Reisig, W.: Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33278-4
21. 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
22. Winskel, G.: Event Structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) ACPN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-17906-2_31