Latte: Improving the Latency of Transiently Consistent Network Update Schedules

Niels Christensen, Mark Glavind, Jiří Srba, Stefan Schmid

ABSTRACT

Emerging software-defined and programmable networking technologies enable more adaptive communication infrastructures. However, leveraging these flexibilities and operating networks more adaptively is challenging, as the underlying infrastructure remains a complex distributed system that is a subject to delays, and as consistency properties need to be preserved *transiently*, even during network reconfiguration. Motivated by these challenges, we propose *Latte*, an automated approach to minimize the latency of network update schedules by avoiding unnecessary waiting times and exploiting concurrency, while at the same time provably ensuring a wide range of fundamental consistency properties like waypoint enforcement. To enable automated reasoning about the performance and consistency of software-defined networks during an update, we introduce the model of timed-arc colored Petri nets: an extension of Petri nets which allows us to account for time aspects in asynchronous networks, including characteristic timing behaviors, modeled as timed and colored tokens. This novel formalism may be of independent interest. Latte relies on an efficient translation of specific network update problems into timed-arc colored Petri nets. We show that the constructed nets can be analyzed efficiently via their unfolding into existing timed-arc Petri nets. We integrate Latte into the state-of-the-art model checking tool TAPAAL, and find that in many cases, we are able to reduce the latency of network updates by 90% or more.

1 INTRODUCTION

Programmable and software-defined networks introduce great flexibilities in how communication networks can be operated and optimized over time. Especially, the possibility to quickly and programmatically update configurations and routes, received much attention over the last years [18]. Such reconfigurations can be used, e.g., to improve the performance of traffic engineering by dynamically adjusting routes to the current demand and workload; other use cases are related to an improved fault-tolerance, e.g., allowing for a fast reaction to failures or supporting maintenance work [12].

However, while programmability is an *enabler* of more adaptive or even "self-driving" [16] communication networks, supporting such reconfigurations is highly non-trivial, for three reasons. First, modern communication networks typically come with stringent dependability guarantees, requiring consistency properties at any time, i.e., *transiently* during reconfigurations. Second, although software-defined networks provide a simple logically centralized abstraction, the underlying network remains a complex distributed system, where individual configuration updates are communicated and realized asynchronously and may hence take effect at different times. Third, updates should not only be implemented consistently but also fast: a main promise of more adaptive networks. The major Internet outage in Japan in 2017, which was due to an incorrect routing table update [11], highlights the importance of transiently correct reconfigurations more generally.

Existing consistent network update mechanisms in the literature are often based on hand-crafted algorithms and either assume an overly pessimistic model where the underlying network may be arbitrarily asynchronous (e.g., [29, 31]), or an overly optimistic model where updates can be timed precisely (e.g., [33, 44]). The resulting update schedules are likely to be unnecessarily slow (in the pessimistic model) or may be infeasible without specific hardware (in the optimistic model). Yet another class of algorithms relies on the modification of packet headers (e.g., to carry state information), which however can introduce further overheads or incompatibilities with existing protocols [38].

Our paper is motivated by the desire to *automate* the process of designing consistent network update algorithms. In particular, we believe that the future self-driving communication networks envisioned by the networking community require mechanisms that provide formal correctness guarantees but also perform well, both in theory (analytically) as well as in practice (in the "average case"). We hence envision algorithms that automatically improve the latency of network updates by removing unnecessary waiting times in update schedules while accounting for possible differences in update processing times: different packet types, such as *VoIP*, *SSH*, or *VPN*, entail different forwarding times at switches [8], hence requiring different waiting intervals; similarly, also the specific switch type effects forwarding times.

We develop a fully automated approach to optimize the performance of network update schedulers accounting for such possible differences in the update time characteristics. In particular, we aim to maximize update concurrency and minimize waiting times, while ensuring transient consistency. Minimizing the update duration is critical because the network may experience irregular behavior during the update procedure. On the other hand, the computation of the optimal update schedule is less time-critical as it occurs ahead of time and does not influence the reliability of the network operation. In our work, we support a wide range of per-packet consistency properties such as:

- **Waypoint enforcement:** Each packet traverses a specific waypoint (e.g., implementing a security-critical network function such as a firewall or intrusion detection system) during the update. We also support the traversal of a *sequence* of (ordered) waypoints as well as *sets* of (unordered) waypoints: a relevant scenario in the context of service chaining [40].
- **Loop freedom:** A packet will never end up in a loop during the update. We support both strong and weak loop-freedom [28].
- **Blacklist enforcement:** A packet never traverses certain blacklisted parts (see e.g. [21]) of the network.
- **Blackhole freedom:** A packet never encounters a blackhole [31] where no forwarding is currently defined.

Our approach to synthesize such time-optimized and provably correct update schedules relies on a novel application of automata theory, and in particular *Petri nets*: a well-known formal framework to model and reason about distributed and concurrent systems. However, in order to account for timing aspects and in particular, differences in update and processing times, we extend the traditional Petri nets and introduce the notion of *timed-arc colored Petri nets (TACPN)*: Petri nets that account for timing issues and differences in timing behaviors (encoded as colors). By encoding time in tokens, TACPNs allow us to keep track of the time analytically.

We show that despite being more general and powerful, timed-arc colored Petri nets can be analyzed efficiently, and we present a reduction algorithm accordingly: we show how to unfold TACPNs into timed-arc Petri nets without colors where efficient verifications engines are already available [14]. The resulting solution can be used to efficiently synthesize optimized update schedules, providing correctness guarantees as well as significantly improved update latency in practice.

1.1 Our Contributions

Our main contribution is Latte (*Lat*ency-aware *t*ransiently correct updat*e*s), an automated optimization approach for the synthesis of minimal delays between switch updates in order to ensure network update schedules of minimal latency that provably maintain general transient consistency properties. We achieve this by introducing a novel notion of timed-arc colored Petri nets, for which we define a formal syntax and semantics. Then we present an efficient verification algorithm by unfolding our timed-arc colored Petri nets into existing timed-arc Petri nets, and prove that the two models are timed bisimilar and hence preserve among others the consistency (safety) properties of network updates. We integrate Latte into the leading model checker TAPAAL [13] and report on our simulation study based on real-world network topologies, which show that our approach indeed results in significantly faster schedules.

As an independent contribution to the research community and to ensure reproducibility, we make Latte publicly available as an open source tool, including the integration of the modeling formalism into the TAPAAL GUI in order to support the visualization and graphical modelling of timed-arc colored Petri nets.

1.2 Organization

The remainder of this paper is organized as follows. We present a formalization of the problem in Section 2 and introduce the notion of the timed-arc colored Petri nets approach in Section 3. In Section 4, we show how to efficiently solve the problem instances by reduction, and report on our prototype and simulation results in Section 5. After reviewing related work in Section 6, we conclude in Section 7. Due to space limitation, some technical details are deferred to the Appendix.

2 MODEL AND METRICS

In a nutshell, the network update problem asks for a schedule to update a route from its initial path to an updated (final) path. The routes are realized by the forwarding functions of switches (or synonymously here: routers). The update schedule is implemented by a (logically centralized) controller that communicates updates to the switches. In particular, we are interested in update schemes that do not require packet header rewriting. In order to improve performance, updates are sent out by the controller in *batches*: due to the asynchronous communication, the updates in one batch can take effect in any order. Once the switches involved in a batch finish their updates, the controller schedules the next batch of updates: either immediately, or after a certain delay, as it may be required to ensure consistency properties such as waypoint enforcement.

Ideally, in order to optimize update delays, the number of interactions with the controller should be minimized and a single batch with all updates scheduled at once. However, it is well-known that this approach can yield various transient inconsistencies such as loops, blackholes, or violations of waypoint enforcement policies [30, 31]. Accordingly, our goal is to automatically optimize the delays in update schedules and to bundle as many updates to be issued concurrently as possible, while guaranteeing consistency of the update.

More formally, we define the *network update problem* as a tuple $(\mathcal{S}, S_0, \mathit{initial}, \mathit{final}, X_1 \dots X_k)$ where

- $\mathcal{S}$ is a finite set of *switches*,
- $S_0 \in \mathcal{S}$ is an initial switch,
- $\mathit{initial} : \mathcal{S} \hookrightarrow \mathcal{S}$ is the initial partial forwarding function,
- $\mathit{final} : \mathcal{S} \hookrightarrow \mathcal{S}$ is the final partial forwarding function, and
- $X_1 \dots X_k \subseteq (2^{\mathcal{S}})^*$ is a sequence of nonempty groups of switches (batches) that are updated concurrently such that the sets of switches $X_1, \dots, X_k$ form a partitioning of the set $\mathcal{S}$, i.e. $\cup_{i=1}^{k} X_i = \mathcal{S}$ and $X_i \cap X_j = \emptyset$ for all i and j where $1 \le i < j \le k$.

A *partial update* $X \subseteq \mathcal{S}$ is a subset of switches that are already updated and respect the update sequence $X_1 \dots X_k$, meaning that there exists a j, $1 < j \le k$, such that $X = \cup_{i=1}^{j-1} X_i \cup Y$ where $Y \subseteq X_j$. The partial update hence models the effect of asynchrony where only a subset of switches from the current batch have been updated so far. Clearly, the empty set is a partial update (no switches are updated yet) and the set of all switches is also a partial update (once all switches are updated).

Any partial update X defines the corresponding *network trace* starting from the initial switch S_0 as the maximal sequence of switches $S_0 S_1 \dots S_m$ such that for every i, $0 \le i < m$, we have either $\mathit{initial}(S_i) = S_{i+1}$ for every $S_i \notin X$, or $\mathit{final}(S_i) = S_{i+1}$ for every $S_i \in X$. By maximality we mean that if $S_m \notin X$ then $\mathit{initial}(S_m)$ is undefined and if $S_m \in X$ then $\mathit{final}(S_m)$ is undefined.

By *Traces* we denote the set of all network traces for all possible partial updates. We can now ask several properties about the set *Traces* such as:

- **Waypoint enforcement.** Given an end switch S_{end} and a waypoint switch S, we want to guarantee that every trace from *Traces* that starts in S_0 and ends in S_{end} contains also the waypoint switch S (or alternatively contains a subsequence of a priory given ordered or unordered list of waypoint switches).
- **Loop freedom.** For strong loop-freedom, we require that any switch appears at most once in any trace from *Traces*. In case of weak loop-freedom, we demand this property only for traces that end in an a priory selected end switch.
- **Blacklist enforcement.** Given a list of blacklisted switches, we want to ensure that all traces from *Traces* never contain any blacklisted switch.

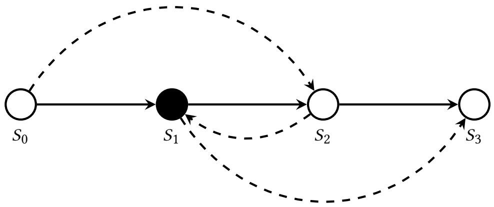


Figure 1: Basic example of a network update problem

- **Blackhole freedom.** Given an end switch S_{end}, we want to guarantee that every trace from *Traces* always ends with the switch S_{end}.

Example 2.1. Let us formalize a basic network update problem from the literature [28]: in this example we consider the network update instance $(\{S_0, S_1, S_2, S_3\},\ S_0,\ \mathit{initial},\ \mathit{final},\ \{S_1\} \cdot \{S_0, S_2\} \cdot \{S_3\})$ where $\mathit{initial}(S_0) = S_1$, $\mathit{initial}(S_1) = S_2$, $\mathit{initial}(S_2) = S_3$ and $\mathit{final}(S_0) = S_2$, $\mathit{final}(S_1) = S_3$, $\mathit{final}(S_2) = S_1$. The update problem is graphically depicted in Figure 1 where the solid lines correspond to the *initial* forwarding function and the dashed lines to the *final* function. The desired waypointing property is that each packet that starts at S_0 and leaves at S_3 must visit the waypoint S_1 (denoted by a filled circle in our figure). A possible partial update is $X = \{S_1, S_2\}$ which gives the network trace $S_0S_1S_3$. Another partial update is $X = \{S_0, S_1\}$ and the corresponding network trace is $S_0S_2S_3$. The set of all network traces can now be constructed by enumerating all partial updates and realizing that $\mathit{Traces} = \{S_0S_1S_2S_3,\ S_0S_1S_3,\ S_0S_2S_3,\ S_0S_1S_3,\ S_0S_2S_1S_3\}$. Clearly, the waypoint enforcement property is violated: the set of traces contains the sequence $S_0S_2S_3$ that forwards a packet from S_0 to S_3 without visiting the waypoint S_1.

Let us now consider that we want to perform a network update according to the group of switches $X_1 \ldots X_k$. In practical scenarios, we first update all switches from the set X_1 (without requiring any specific order of updates) and then we wait for a sufficiently long time before we start updating the switches from the group X_2 (and so on until all groups of switches are updated). Clearly, we must guarantee that all switches from X_1 finished their update before we start updating switches from X_2. This can be achieved by waiting for the maximum update time of any switch from X_1 plus the maximum time a packet can travel in the network. After this delay, it is now safe to update the switches from the group X_2 and so on.

There exist different timing behaviors in software-defined networks which, if accounted for, may significantly improve the latency of updates. However, the interaction between the timings of packet forwarding and switch updates can be quite intricate, and hence, in order to provide rigorous safety guarantees that enforce the absence of certain undesirable traces during the network update, we need to provide a formal framework accordingly. In particular, in the following sections we provide a method for minimizing the latency of a network update by using model checking techniques.

3 TIMED-ARC COLORED PETRI NETS

In order to automate the generation of fast network update schedules, and in order to account for specific timing behavior (e.g., related to packet processing or switch update time estimation), we suggest a novel extension of the classic Petri nets model that is introduced in this section.

Let us first introduce some preliminaries. The configuration (marking) of a Petri net [36] is generally determined by its *tokens* located at *places*. In the timed-arc colored Petri net model that we introduce in this section, *tokens* contain both the *color* information as well as the *timing* information (their *age* taken from the domain of nonnegative reals). We shall first define the P/T net and then add both color and timing features. Let $\mathbb{N}_0$ be the set of natural numbers including 0 and let $\mathbb{N}_0^\infty$ be the set $\mathbb{N}_0$ together with the special infinity symbol ∞ such that $n < \infty$ for any number $n \in \mathbb{N}_0$.

A *Petri net* is a tuple $N = (P, T, W, W_I)$ where P is a finite set of *places*, T is a finite set of *transitions* such that $P \cap T = \emptyset$, $W : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}_0$ is the *weight function* for the *input arcs* from places to transitions and *output arcs* from transitions to places and $W_I : P \times T \rightarrow \mathbb{N}_0^\infty$ is the *inhibitor weight function* that assigns weights to *inhibitor arcs* from places to transitions.

A *marking* on a Petri net N is a function $M : P \rightarrow \mathbb{N}_0$ such that $M(p)$ denotes the number of *tokens* in the place $p \in P$. A transition $t \in T$ is *enabled* in a marking M if $W(p,t) \leq M(p)$ and $W_I(p,t) > M(p)$ for all places $p \in P$. In other words, each input arc must have enough tokens that can be consumed from the connected place and all inhibitor arcs must be disabled. If a transition t is enabled in M, it can *fire* and produce a marking M' such that for all places $p \in P$ we have $M'(p) = M(p) - W(p,t) + W(t,p)$. We notice that inhibitor arcs only influence the enabledness of a transition but they do not participate in transition firing.

3.1 Color and Time Extension

We shall now describe the color and time extension of the basic Petri net model that we use for modelling of network updates. We assume a given (possibly infinite) set of *colors* C such that every place p in the net is assigned a finite subset of C called the *color type* of p. We also assume that with each place p there is an associated *age invariant* of the form $c \leq I_c$, where $I_c \in \mathbb{N}_0^\infty$, for each color c from the color type of p.

A *token* in a timed-arc colored Petri net (TACPN) is then a triple (p, x, c) where p is the location of the token, c is a color from the color type of p and x is the age of the token (initially all tokens are of age 0) such that $x \leq I_c$ where I_c is the upper bound of the age invariant for the color c. A *marking* in TACPN is a multiset of tokens. Transition enabledness in a given marking is now conditioned also on the colors and ages of tokens in a marking, in addition to the requirement on a number of tokens, as in the classical Petri net model introduced above. We assume that each input arc from a place p to a transition t is annotated (for each color c from the color type of p) with the expressions $c \rightarrow [a, b]$ where $[a, b]$ is a time interval such that $a, b \in \mathbb{N}_0^\infty$ and $a \leq b$. The intuition is that the age of each token of color c that will be consumed during the transition firing must belong to the interval $[a, b]$. Moreover, every such input arc is assigned an *arc expression* of the form $n_1'(\tau_1) + \ldots + n_k'(\tau_k)$ where $n_1, \ldots, n_k \in \mathbb{N}_0$ is the number of tokens of the *color expressions*

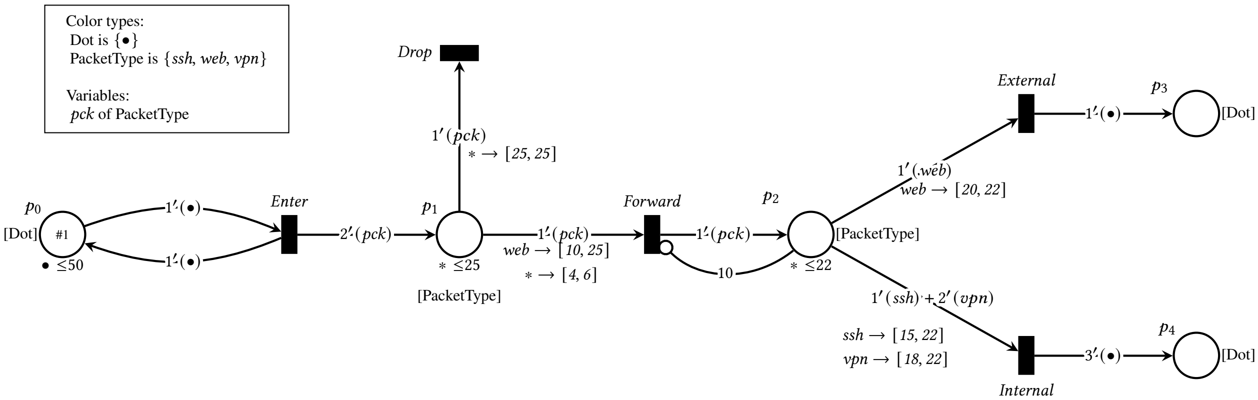


Figure 2: Example of a Timed-Arc Colored Petri Net (TACPN)

$\tau_1, \ldots, \tau_k$ to be consumed during the transition firing. For the purpose of this paper, we assume that τ_i, $1 \le i \le k$, is either a concrete color c or a *variable* that allows for a *binding* to any color from the color type of p. Similarly, each output arc from a transition to a place is also assigned an arc expression; however, output arcs do not contain any time intervals as the age of newly produced tokens by transition firing is reset to 0.

REMARK 1. *In the general TACPN model we allow for more complex color expressions that include also color products as well as basic operations for color manipulation like e.g. color successor and predecessor in case of cyclic color types. Moreover transitions can contain guards that further restrict the transition firing. Similarly, it is possible to preserve the age of tokens during transition firing by means of transport arcs and to define urgent transitions that restrict time delay whenever enabled. These features are not necessary for modelling of the network updates and in order to simplify the presentation, we do not define them in the present paper.*

Example. Figure 2 shows a graphical representation of an TACPN example. The net has five places $p_0, \ldots, p_4$ denoted by circles and each place has an associated color type in square parenthesis together with the age invariants. For example the place p_0 is of color type Dot (classical Petri net token) and the age invariant says that a token can reach age at most 50, after which the time cannot progress anymore. Similarly, the color type of p_1 is PacketType containing three elements *ssh*, *web* and *vpn*. Each color element should have defined its own age invariant, however, we abbreviate by the star notation $* \le 25$ the fact that all three colors share the same age invariant. Places that do not have listed any age invariant, like p_3 and p_4, assume the default invariant $* \le \infty$ that does not restrict the possible time delays. Transitions in the net are denoted by rectangles and represent events in the net (a packet entering a network, dropping a packet, forwarding a packet to p_2 and routing the packet: external or internal traffic). Places and transitions are connected by arcs that move tokens during transition firing. For example the transition *Enter* will consume one token from p_0, return the token back (while resetting its age to 0) and producing two tokens with color type PacketType into the place p_1. Depending on the binding of the variable *pck* to one of the three different colors in PacketType, the two produced tokens can be of three different colors. Again, their age is reset to 0. The input arc to the transition *Enter* does not contain any interval, assuming the default interval $[0, \infty]$ that does not restrict the age of the consumed token in any way. On the other hand, the arc from p_1 to *Forward* can only consume a token with color *web* of age between 10 and 25 or with color *ssh* or *vpn* of age between 4 and 6 (here we again use the star notation). An arc can also consume more than one token, like for example the arc from p_2 to *Internal* that requires one *ssh* token and two *vpn* tokens. The firing of *Internal* consumes three such tokens of corresponding ages (*ssh* between 15 and 22 and *vpn* between 18 and 22) and produces three fresh tokens of type Dot and with age 0. Finally, the arc with the circle-tip from p_2 to *Forward* is an inhibitor arc of weight 10 which means that as soon as the place p_2 contains 10 or more tokens, it disables the firing of the transition *Forward*.

We shall now define the behavior of TACPN that consists of a nondeterministic choice between firing one of the currently enabled transitions and a time delay where all tokens age by the same delay (unless the delay is disabled by some age invariant). An example of transition firings from the initial marking $\{(p_0, 0, \bullet)\}$ is shown below

$$\{(p_0, 0, \bullet)\} \xrightarrow{Enter}$$

$$\{(p_0, 0, \bullet), (p_1, 0, web), (p_1, 0, web)\} \xrightarrow{delay\ 20}$$

$$\{(p_0, 20, \bullet), (p_1, 20, web), (p_1, 20, web)\} \xrightarrow{Forward}$$

$$\{(p_0, 20, \bullet), (p_1, 20, web), (p_2, 0, web)\} \xrightarrow{delay\ 5}$$

$$\{(p_0, 25, \bullet), (p_1, 25, web), (p_2, 5, web)\} \xrightarrow{Drop}$$

$$\{(p_0, 25, \bullet), (p_2, 5, web)\} \xrightarrow{delay\ 16}$$

$$\{(p_0, 41, \bullet), (p_2, 21, web)\} \xrightarrow{External}$$

$$\{(p_0, 41, \bullet), (p_3, 0, \bullet)\} \xrightarrow{Enter} \ldots$$

where both when firing the transition *Enter* and *Forward* we use the binding *pck* = *web*. We notice that once there is a token $(p_1, 25, web)$ in the net, the age invariant in place p_1 disables the possibility of time delay and transition firing becomes urgent (in our example we decided to drop the token).

3.2 Tool Support for TACPN

The time-arc colored Petri nets underlying our consistent network update framework need strong tool support in order to be applied on real-world scenarios. For that purpose, we implemented and integrated our model of TACPN in the GUI of the open source tool TAPAAL [13]. This allows us to graphically draw the nets as well as to answer reachability and CTL queries with atomic propositions that consist of upper and lower bounds on the number of tokens in different places of the net, and their Boolean combinations. We also implemented unfolding of TACPN nets into plain timed-arc Petri nets (where the only color type is Dot = $\{\bullet\}$) by expanding the number of places in order to model tokens of different colors. The unfolding relies on the classical approach where color domains are expanded into multiple places and we had to further extend this unfolding technique to deal with the timing information and with inhibitor arcs.

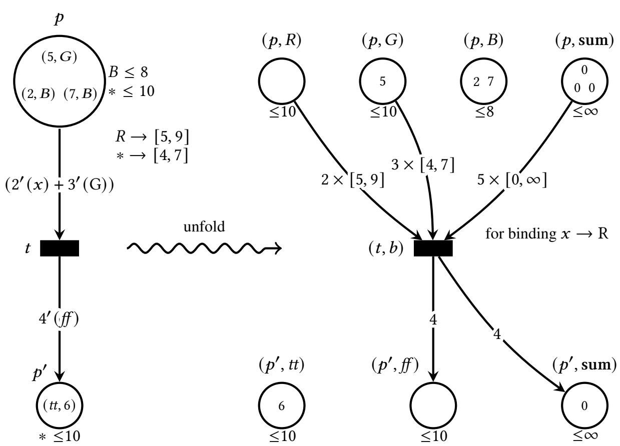


Figure 3: Unfolding example

An example of the unfolding process is given in Figure 3 where the color type of the place p is $\{R,G,B\}$ and it is unfolded into three places (p,R), (p,G) and (p,B). The tokens of age and color $(5,G)$, $(2,B)$ and $(7,B)$ are then placed in the corresponding unfolded places while preserving their age. Similarly, the color type of the place p' is $\{tt, ff\}$. The unfolding is given for the binding of the variable x to the color R and we also add a special place $(p, \mathbf{sum})$ where we keep the accumulated number of tokens in the original place p (for the purpose of inhibitor arc tests). The resulting timed-arc Petri net model can be proved to be timed bisimilar to the original net with colors (for the full proof consult the appendix), hence preserving the answers to (among others) reachability queries that we need for our application. The unfolded net and query can then be verified using existing verification engines of TAPAAL, both for the discrete [7] as well as continuous [14] time.

4 TIME OPTIMAL SCHEDULE GENERATION

Given the concepts introduced above, we can now present our approach for generating fast update schedules, ensuring transient consistency properties. In a nutshell, Latte translates a given network update problem into a timed-arc colored Petri net in order to compute how the delay between switch updates can be automatically minimized, and updates batched, hence optimizing the overall network update time. For ease of presentation, we demonstrate Latte on the waypoint enforcement property in the following; at the end of this section, we discuss how our approach can be generalized to other types of safety properties.

4.1 Overview of Reduction to TACPN

Let us assume a given instance of a network update problem $(\mathcal{S}, S_0, \mathit{initial}, \mathit{final}, X_1 \dots X_k)$. The input is automatically processed by our translation algorithm that creates different types of net components as well as a query for the verification. A conceptual overview of the translation is displayed in Figure 4. The translation algorithm works as follows.

1) We create time constraints based on packet input types and for each packet type we create a color in the color type *PckType*.
2) We create the start and the end component representing the start and end switches of the route.
3) For all switches, we create a separate switch component. A special component is created for a waypoint switch that remembers whether a packet passed through the switch.
4) For each switch we create an update component that uses a timing interval for the duration of the update.
5) We create an initialization sequence for switch updates and use the constants $C_1, C_2, \dots$ as the waiting delays before the next switch update is initialized.
6) We create a TACPN query for verification and use the bisection algorithm to minimize the constants $C_1, C_2, \dots$ while still preserving the waypoint enforcement property.

The different components that we create share places. This is denoted by the dotted circle around the place and the idea is that all such shared places with the same name are merged together. We use the colors to model the different timing behaviors of different packet types, allowing us to use the timing information to calculate a safe update delay given by the following expression:

$$\begin{gathered}\mathit{slowest\text{-}switch\text{-}update}+\\ \mathit{\#switches\text{-}on\text{-}the\text{-}route} \times \mathit{slowest\text{-}hop}\end{gathered} \tag{1}$$

In this safe delay estimate we include the time for the slowest switch update in order to make sure that all forwarding rules for all switches from a given batch are fully updated before we proceed to the next batch. But at the same time, we have to make sure that all packets that can still be in transit (and could potentially use some of the old forwarding rules), left the network. The second part hence approximates the latency of the routing path by multiplying the number of switches involved in the routing with the slowest packet forwarding time.

4.2 Examples of Switch and Next-Hop Timing

While our formalization and approach is more general, in the following, we will discuss some specific examples of packet and switch update times. These examples will also serve us as case studies in the evaluation.

Packet processing time. Different types of packets can occur with different processing times. For example, the resulting latency can be

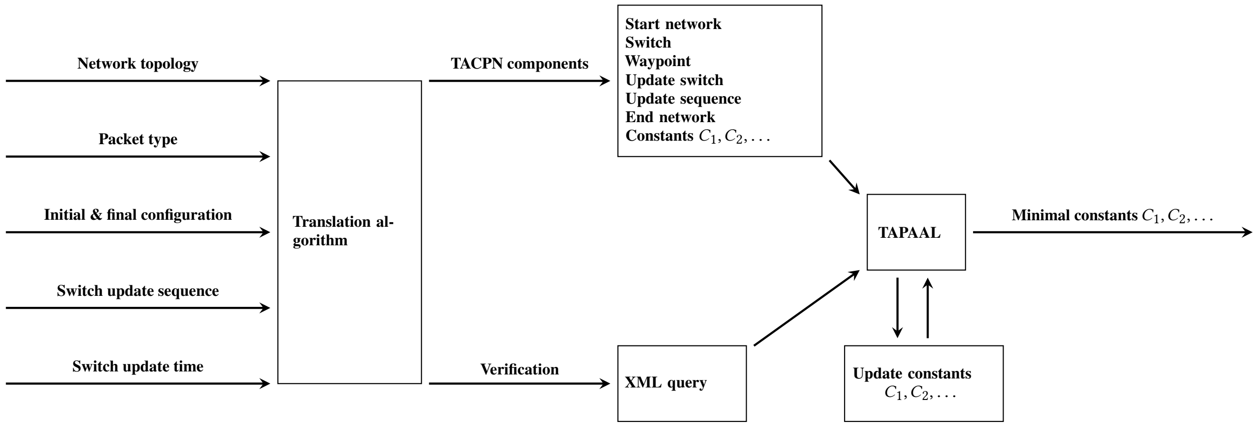


Figure 4: Overview of the translation algorithm

Packet type	**Size** [Byte]	**Latency** [$\mu s \pm 0.5$]	**Interval**[μs]
VoIP [3]	218	2.2	[1, 3]
SSH [2]	312	2.1	[1, 3]
VPN [41]	1300	3.2	[2, 4]

Table 1: Network packet types and next-hop latency

Algorithm	**Min** [ms]	**Max** [ms]	**Interval**[μs]
Optimal	0	2.5	[0, 250000]
Batch-ready	0.2	2.5	[20000, 250000]
No scheduler	0.5	2.5	[50000, 250000]

Table 2: Switch update times in FatTree topology

a function of packet size, as shown by Bauer et al. [8]: the authors find that the latency for, e.g., a Pica8 P3297 switch [1] can vary by 0.5 μs (difference between an upper and lower bound, see Figure 3 in [8]). In Table 1, we summarize the latency for some typical packet types on Pica8 P3297 switch and show the latency interval that we use in our experiments.

Switch update time estimation. The cumulative time for the switch update (installation of a new set of forwarding rules) can also differ significantly depending on the hardware and the scheduler that performs the update. For example, in Figure 5 from [35], the authors show the cumulative distribution function of flow installation for 1000 flows on a FatTree topology. In Table 2 we show the update completion times for the different scheduling algorithms when installing a batch of new forwarding rules and approximate them by intervals. As an example, we use the timing interval with no scheduler in our translation and experiments.

4.3 Translation Algorithm

We can now proceed to define the net components that are created during the translation of a network update instance to a corresponding TACPN.

Figure 5a creates an initial component that at any moment allows to inject a packet into the network by firing the transition **T0**. Depending on the binding of the variable *pck* to either *VPN*, *SSH* or *VoIP*, three different types of packets can be created and placed into the place $\mathbf{S_0}$ that corresponds to activation of the initial switch S_0. The place $\mathbf{S_0}$ is a shared place, meaning that it is the same place as the initial place for the component corresponding to the switch S_0 from Figure 5c.

Figure 5b shows a component representing the update of a given switch. One such component is created for each switch S. The update for the switch S is initialized by placing a token to the place **StartUpdateS** and the duration of the update is determined by the update interval from Table 2 on the input arc to **T1** and it is enforced by the invariant $* \leq Max$. Once **T1** is fired, it removes the token from $\mathbf{S_{InitialEnf}}$ and creates a token in $\mathbf{S_{FinalEnf}}$. These are shared places that are used in the switch component to determine whether the forwarding should be done according to the function *initial* (in case the token is in $\mathbf{S_{InitialEnf}}$) or to the function *final* (in case the token is moved to $\mathbf{S_{FinalEnf}}$).

Figure 5c is a component that executes the packet forwarding of a given switch S. Once a packet arrives to the shared place **S**, it is forwarded either to the place $\mathbf{S_{Initial}}$ assuming that there is a token in $\mathbf{S_{InitialEnf}}$ and $initial(S) = S_{Initial}$, or to the place $\mathbf{S_{Final}}$ in case that the token is in $\mathbf{S_{InitialEnf}}$ and $final(S) = S_{Final}$. The duration of such packet forwarding is determined by its color (packet type) and the associated forwarding interval from Table 1, while the age invariants ensure that a packet cannot stay at a switch for more than the upper bound of the forwarding interval.

Figure 5d models the execution of an update sequence of the network according to the switch update groups $X_1, \ldots, X_k$. We assume that the ordering of the switches $S_1, \ldots, S_n$ respects this update sequence such that the switches from the group X_i always come before the switches from the group X_{i+1}. The update sequence can start at any time by firing the transition **T1**, which initiates the update of the switch S_1 by placing a token to the place $\mathbf{StartUpdateS_1}$. At the same time a token of age 0 is placed into the place $\mathbf{Step_1}$ ensuring that the update of the switch S_2 starts exactly after C_1 units of time. The remaining switch updates are chained in a similar way such that the constants $C_1, \ldots, C_{n-1}$ determine the respective delays between switch updates are initiated. The constants that separate switch updates from the same update group will be set to 0, meaning that all such updates are started at the same time. Switch updates

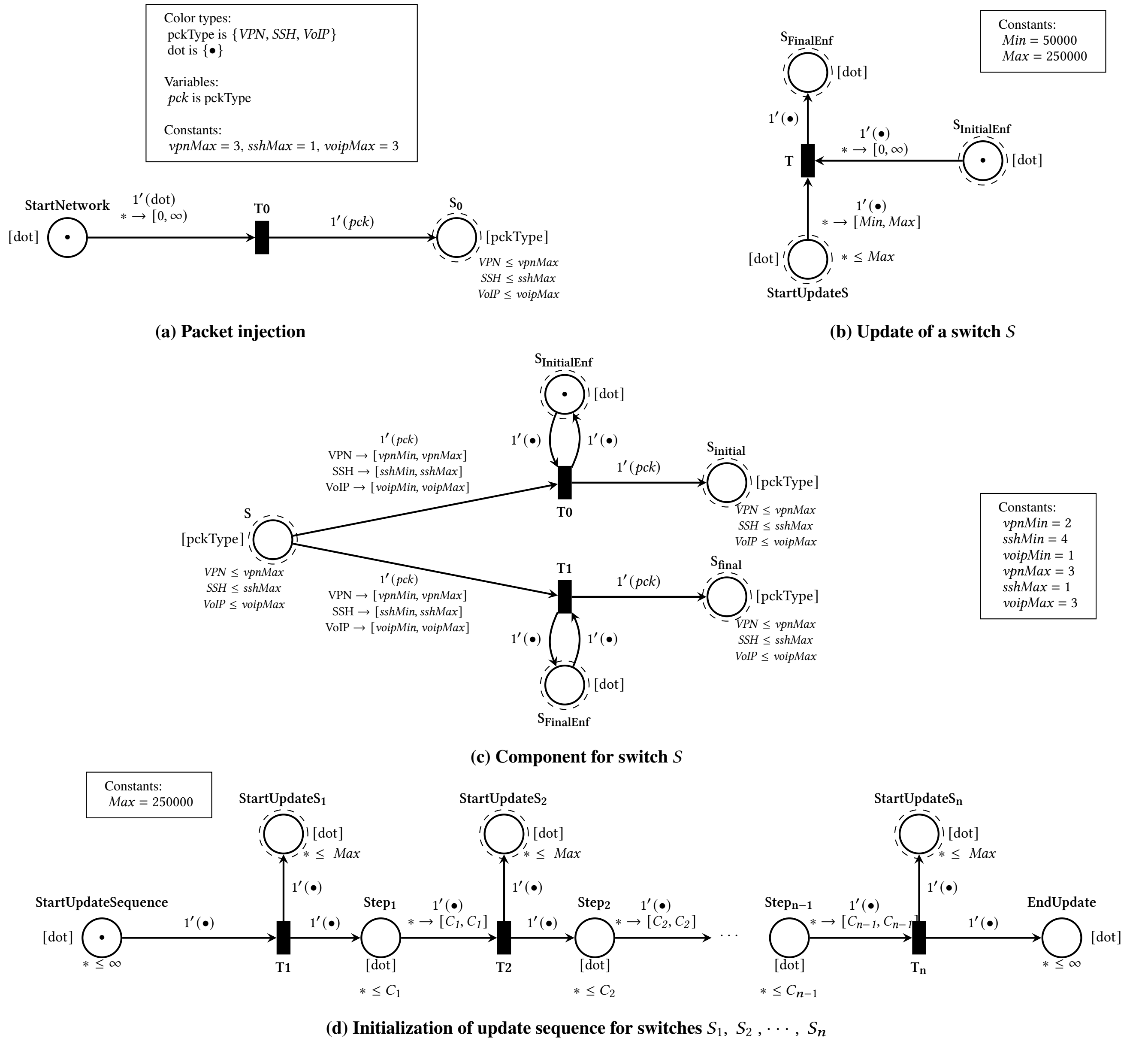


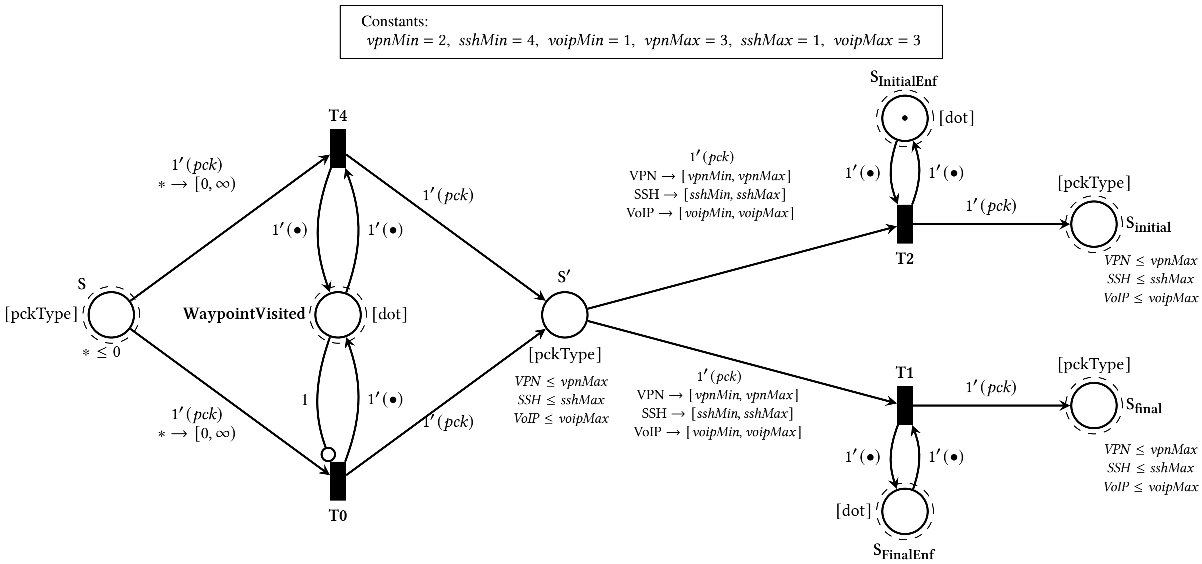
Figure 5: Translation of update synthesis into TACPN

from different update groups must be separated by a delay that is sufficiently long in order to guarantee that all previous switch updates finished and there is no packet in the network that followed some of the outdated forwarding rules—the delays are initially set according to Equation 1.

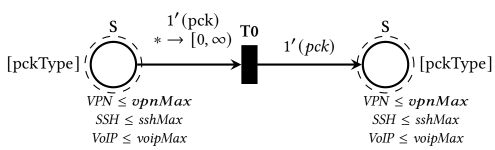
Figure 6a shows a waypoint component for a switch S. The forwarding part of a waypoint switch is the same as for ordinary switches; however, it is prefixed with firing either the transition **T4** or **T0**. The purpose is to place a token to a place **WaypointVisited** the first time the switch is used. This is enforced by the fact that **T4** is disabled as long as **WaypointVisited** has no tokens. On the other hand, once it contains a token, the transition **T0** is now disabled because of the inhibitor arc and we have to necessarily fire **T4** that keeps the token in **WaypointVisited**. This construction ensures that our Petri net remains bounded even in case of cyclic behavior.

The general switch component can be simplified in case that both the initial and final function return the same next-hop switch, as shown in Figure 6b. Finally, in Figure 6c we add the component for ending the packet forwarding once the last switch *SwitchEnd* in the network routing is reached. Due to the invariant $* \leq 0$ we enforce that the firing of the transition **T0** is urgent and the place **EndNetwork** gets marked without any time delay.

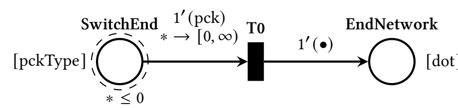
The general construction of the TACPN net that models the behavior of a network update problem is now finished. Once the update



(a) Component for a waypoint switch S



(b) Component for a switch S with no updates



(c) Component for finished packet forwarding

Figure 6: Continuation of update synthesis reduction

sequence is initiated, the switches are then updated with the delays determined by the constants $C_1, \ldots, C_{n-1}$. At any moment a packet (token) can be injected into the network and we execute a network trace according to the current status of switch updates. Once a waypoint is visited, we record this by placing a token into **WaypointVisited** and we must guarantee that this place is marked before the packet reaches the end switch and the routing is terminated by placing a token into the place **EndNetwork**. Hence the waypoint enforcement is expressed by the following reachability query

$$AG\ (\textbf{EndNetwork} = 0 \ \lor\ \textbf{WaypointVisited} \geq 1)$$

claiming that during any execution of the net either the place **EndNetwork** is still not marked (contains 0 tokens) or if this is not the case then the place **WaypointVisited** must contain at least one token.

Such a query can be automatically verified using our prototype implementation in the tool TAPAAL [13] that loads the network topology with forwarding tables and an update sequence and automatically generates the corresponding timed-arc colored Petri net on which it verifies the above mentioned query.

4.4 Minimization of Delay Points

In case the waypoint enforcement is satisfied, we are interested in minimizing the delays given by the constants $C_1, \ldots, C_{n-1}$, without breaking the waypointing property. We achieve this by sequentially minimizing the constants (using the bisection method) until we find the minimal constants that still satisfy waypoint enforcement. In order to speedup the identification of updates that can be performed concurrently, we start the bisection method by first setting each constant to 0. As it is often the case that a large degree of concurrency is possible during the updates, the bisection method hence becomes computationally cheap as it only needs to perform the repeated bisection between the switch updates where a delay is necessary for preserving the waypointing (typically less than two of such delay points are necessary). As we demonstrate by the experiments in the next section, this method scales even for larger update sequences and allows us to significantly reduce the total update time on several realistic network topologies. We conjecture that the sequential optimization of the delays in our application actually produces the shortest possible update sequence, however, the formal proof of this claim is beyond the scope of this paper.

4.5 Other Consistency Properties

For the sake of presentation, we formally described the translation to TACPN for the waypoint enforcement property. However, other consistency properties can be easily verified by small modifications of the translation.

- In order to optimize the update delays that preserve loop freedom, we use for *each switch* the component for waypoint switch as given in Figure 6a where the weight of the

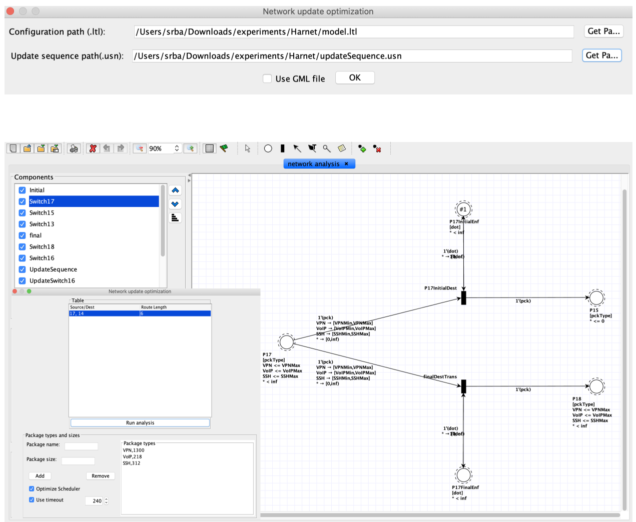


Figure 7: Screenshot from Latte Plugin for TAPAAL

inhibitor arc is 2 and the weights of the arcs connecting the **WaypointVisited** place with **T4** have weight 2 as well. To verify strong loop freedom, we now ask the AG query (for all reachable markings) requiring invariantly that the place **WaypointVisited** for each switch S in the network contains at most one token. In order to verify weak loop freedom, we require this property to hold only for execution traces that include also the end switch.

- For blacklist enforcement, we simply put a token to a newly created place each time a blacklist switch is visited and we verify that invariantly this place contains no tokens.
- Blackhole freedom can be also verified by marking a newly created place whenever a switch with undefined next-hop is traversed and asking a query that makes sure that this place is invariantly empty.

In practical applications, we are often interested in the combination of a number of consistency properties that should be invariantly preserved in conjunction during the network update. As the consistency properties can be, as argued above, expressed by the formulae $AG\varphi_1, \ldots, AG\varphi_n$ that invarianly postutale that the corresponding properties $\varphi_1, \ldots, \varphi_n$ hold at any moment during the network update, we can easily verify also the conjunction of these properties as the formula $AG(\varphi_1 \wedge \ldots \wedge \varphi_n)$. The advantage is that there is only a slight overhead when more properties are checked at the same time, as we are exploring the state space of the Petri net only once, while verifying all properties during this single search.

5 EVALUATION

In order to evaluate the performance achievable with our approach and compare it to the state-of-the-art, we implemented a prototype of Latte. In the following, we present our empirical results and discuss our findings in a variety of network update instances.

5.1 Prototype and Experimental Setup

The novel model of TACPN defined earlier is fully integrated as a plugin into the leading model checking tool TAPAAL (also in the GUI) and we provide an efficient C++ implementation of the unfolding algorithm for verification of TACPN. Our tool is now part of the TAPAAL model checking suite and available as a beta-release at http://tapaal.net/download/ in "Other Downloads", together with an archive including all experiments needed to reproduce the results presented in this section. The workflow of verifying network updates is fully automated and depicted in the screenshots in Figure 7. First, the user calls the Latte plugin in Tools menu, enters the paths to the network topology file and update sequence file and the tool automatically parses these files and produces the TACPN components. Then it opens a dialog where the user can specify the packet types together with their sizes in bytes the analysis should consider. The verification is then initialized by executing "Run analysis".

In our evaluation, we use network topologies from the Topology Zoo [26]. The initial and final configurations of the network as well as the update sequence are generated by the tool NetSynth [32]. The tool takes a network topology and creates one or more source and destination pairs in the given topology. It also creates an initial and final configuration and an update sequence that guarantees consistency. In our evaluation, we focus on waypoint enforcement as a case study. The update sequence generated by NetSynth contains the symbol # that requires a sufficiently long delay before the next switch gets updated. NetSynth does not identify updates that can be performed concurently, assuming that a safe delay point is inserted between any switch updates. As explained earlier, our task is to minimize these delays while still preserving the waypointing property. The sizes of network topologies range from tens to a hundred of switches (with the average network size of about 35 switches), however, we do not report these sizes in the results as our verification algorithm is only marginally dependant on the topology size. We instead report as the scaling parameter the length of the update route as this is the parameter that has the main influence on the performance of our method.

The experiments are run on a 64-bit Ubuntu 18.04 laptop with 16 GB RAM and Intel Core i7-7700HQ CPU @ 2.80GHz x 8 with a 10 minute timeout.

5.2 Results

We are primarily interested in two metrics in our evaluation: the runtime of our algorithm and the latency of the generated update schedules. Our results are summarized in Table 3. The size of each instance here is scaled by the route length, which is the sum of the lengths of the packet routing before and after the update. Verification time shows the total time needed to find the optimal delay constants that separate switch updates. The default update time is computed by replacing each delay symbol # produced by NetSynth with the safe delay constant as computed by Equation 1. The optimized update time is the sum of all delay constants computed by our algorithm as described in Section 4.4.

We can see that within the 10 minute timeout, we are able to compute the optimal update times for route lengths up to 16: over 90% improvement compared to the default update times. For the last three instances our algorithm times out, meaning that the bisection

Network	Route length	Verification time[s]	Default update time [s]	Optimized update time [s]	Improvement [%]
TLex	4	0.74	3.58	0.25	92.30%
HiberniaIreland	5	1.02	6.05	0.28	95.50%
Harnet	6	1.42	9.08	0.28	96.97%
UniC	7	1.49	12.65	0.28	97.83%
Oxford	8	2.02	16.78	0.28	98.36%
Xeex	10	5.86	26.68	0.28	98.97%
Sunet	11	10.23	32.45	0.28	99.15%
SwitchL3	12	18.88	38.78	0.28	99.29%
Psinet	14	89.67	53.01	0.28	99.48%
Uunet	15	211.86	61.05	0.28	99.55%
Renater2010	16	480.52	69.58	0.28	99.60%
Missouri	25	timeout	171.05	67.10	60.77%
Syringa	35	timeout	336.05	295.35	12.11%
VtlWavenet2011	35	timeout	336.06	295.35	12.11%

Table 3: Experiments with update sequences generated by NetSynth

algorithm did not manage to find the optimal constants, however, still achieving an improvement in the total update time. The reason for the timeout is that the update sequences produced by NetSynth actually allow to run all updates concurrently, meaning that all delay constants can be set to 0. This creates a large number of switches that update concurrently and we have to consider all (exponentially many) interleavings of the updates in order to guarantee the waypoint enforcement. On the other hand, as the tool NetSynth produces disjoint update sequences, the fact that all updates can be concurrent can be determined by exact static methods without the need of running the actual verification. In the future work we will explore the possibility of combining our method with static analysis in order to further improve the performance in case of a large number of concurrent updates.

We also explore (manually created) update sequences where a concurrent update of all switches is not safe and some minimum delays are necessary in order to guarantee waypointing. The results are summarized in Table 4. We can observe the optimal update times decrease but are still over 90% more efficient compared to the default update times. Moreover, the concurrency is reduced significantly and this is reflected by the improved verification times. The networks like Missouri that has 67 switches and update sequence of length 10 can still be verified in a matter of seconds, due to the reduced concurrency in the update batches.

In summary, we find that the proposed method of optimizing the network update time while preserving waypoint enforcement is feasible for a standard benchmark of network topologies and for up to 16 concurrent switch updates. Even in the situations where the state space explodes for a higher number of concurrent updates, we are still able to reduce the total update time while preserving consistency properties. We would like to emphasize that the critical factor here is the actual optimized update time for the whole network, which we often reduce below one second. The actual verification time that computes the optimized update sequence ranges from seconds to several minutes, however, as this is a pre-computation performed offline, it is less critical and does not influence the network performance: during the precomputation the network is stable as it is still forwarding using the previously loaded configurations.

6 RELATED WORK

Motivated by the advent of software-defined and hence more adaptive communication networks, the consistent network update problem has received much attention over the last years, see the recent survey [18] on the topic. The seminal work by Reitblatt et al. [38], and many followup works (e.g., [9, 10, 19, 23, 24, 27, 34]), rely on packet versions, ensuring a strong per-packet consistency. Mahajan and Wattenhofer [31] initiated the study of fast network update algorithms which do not require packet header rewriting, but which rather update switches in batches to ensure basic consistency properties. Their approach has been refined in several followup works, which presented various more efficient scheduling algorithms for different properties, including loop-freedom [20, 29], waypoint enforcement [28, 30], and beyond [5, 15]. These approaches have in common that they rely on clever algorithms developed for the specific problem. In contrast, we consider a more automated formal method approach to optimize update schedules, with a main focus on the timing aspects. In this regard, our paper is close in spirit to the work by McClurg et al. [32] who consider the synthesis of update schedules. Their work is on the synthesis of consistent network updates and they introduce the command *wait* that represents a delay that guarantees a safe flush of all packets that might follow the outdated forwarding rules. The authors suggest a conservative computation of such a delay based on the maximum hop count (similarly as in our Equation 1), however, contrary to the main focus of our work, they do not further study any optimization of such delays.

Existing work can be further classified regarding the assumptions made regarding the synchronization model. While all approaches above revolve around solutions for asynchronous communication networks where updates can take arbitrary time, there is also interesting work on technologies that assume exact time updates in software-defined networks [33, 44]. Our work is positioned in-between: we exploit specific timing behaviors with uncertainty (represented by time intervals) in order to reduce the update schedule while providing guarantees on consistency of the update. To this end, we do not only avoid unnecessary waiting times but also support concurrent updates whenever safe.

We are not the first to consider the application of Petri nets in the context of software-defined networking: [39] presents a model which

Name	Route length	Verification time[s]	Default update time [s]	Optimized update time [s]	Improvement [%]
HiberniaIreland	6	4.37	4.68	0.45	90.70%
Oxford	12	4.71	7.99	0.45	94.42%
SwitchL3	8	4.67	5.78	0.47	91.95%
Psinet	16	4.67	10.18	0.45	95.63%
Renater2010	7	4.23	5.23	0.45	91.48%
Missouri	10	5.14	6.88	0.45	93.53%
Ans	13	5.73	8.52	0.43	94.90%
Bics	13	6.20	12.65	0.44	96.56%
Globalcenter	14	7.63	17.88	0.45	97.51%
Geant2009	13	11.72	16.78	0.45	97.35%

Table 4: Experiments with update sequences that require nonzero delays

allows for performance prediction using queuing Petri nets, [6] studies fault-tolerant aspects, and [43] security aspects. These works hence have a different focus. To the best of our knowledge, the only work considering Petri nets for network updates is the parallel work by Finkbeiner [17]. However, while their approach relies on a powerful logic, it is different from ours in that it focuses on an asynchronous model, and does not account for timing aspects. Furthermore, the approach also supports the testing of update schedules, not the synthesis of improved schedules. Conceptually, the paper is also different from us in that it relies on classic Petri net theory, while for our use case, we had to develop a novel extension of the Petri net.

Around the same time as the work by McClurg et al. [32], Zhou et al. [45] presented a customizable approach to provide consistency properties in software-defined networks. The authors develop an uncertainty network model and apply a greedy algorithm that for each arriving update rule verifies if it can possibly break the consistency of the network: if this is not the case then the update is applied immediately, otherwise the rule is put on hold and processed at some later time after some predefined delay. The unresolved updates are usually handled using some fallback mechanism (like two-phase update) and the experiments document a considerable speed up (up to three times) in the duration of network update. Our work, on the other hand, provides an exact (provably optimal) solution of minimum switch update delays for a given update sequence and models a high timing precision both for the switch updates as well as packet transmission. We are not aware of other tools that allow to compute the exact minimum delays between switch updates.

Finally, it remains to point out that there exists much work on other notions of Petri nets accounting for time, most notably *timed* Petri nets [37, 46]. However in these nets, timing is fixed to transitions, while in our proposed timed-arc colored Petri nets, timing is related to tokens, which enables us to keep track of time for all (dynamically created) tokens in the net. As a result, the modelling capabilities of the two models are incomparable and in particular the timed Petri net model does not allow us to keep track of the ages of tokens (representing packets in our application)—a feature that is essential for modelling of network updates. The most related model of *interval timed colored nets* [42] associates, similarly to our model, tokens with both time and color infomation. However, the model uses an eager semantics that introduces priorities among transition firings (transitions with smallest enabling times fire first) whereas our model uses relative timing and allows for multiple enabledness of transitions that is essential for our application domain. We are not aware of any other work on Petri nets that combine both timing associated to tokens where arcs contain timing intervals restricting the ages of tokens they can consume (a feature essential for modeling of network updates) together with colored information (that allows us to account for multiple variants of packets in the network at the same time). We believe that our Petri net model of TACPN is of independent interest because other existing extensions of Petri nets with time and color relay on radically different semantics.

7 CONCLUSION

Motivated by the emerging more adaptive communication networks, we presented an automated approach to improve and speed up network update schedules, while ensuring rigorous transient correctness guarantees for a wide range of properties. Our approach relies on formal methods and in particular, a novel generalization of Petri nets which supports reasoning about different timing behaviors. We introduced an efficient algorithm to construct and solve our timed-arc colored Petri nets, presented an implementation in a state-of-the-art model checking tool, and reported on experimental results. For network topologies with up to 16 concurrent swich updates, we were able to reduce the network update time from about a minute to a fraction of a second and hence to significantly reduce the time of possible routing irregularities during the network update. The computation time needed to achieve this gain ranges from seconds to minutes, which is very reasonable given the high complexity of the task. Moreover, the network routing is not affected during the computation of the update delays, and hence it is only the network update duration that is critical for the network performance.

We understand our work as a first step and believe that it opens several interesting directions for future research. In particular, it will be interesting to generalize the synthesis algorithm further, supporting the synthesis of arbitrary update schedules from scratch. It will also be interesting to explore the use of our developed timed-arc colored Petri nets in other application domains as well: we believe that TACPN may be of independent interest and of use in other contexts where different timing behaviors occur, e.g., in transportation systems. In order to facilitate future research and ensure reproducibility, we share our implementation as part of the open source tool TAPAAL.

REFERENCES

[1] Pica8 p3297, 2014. - Data sheet. https://www.pica8.com/wp-content/uploads/pica8-datasheet-48x1gbe-p3297.pdf (2014).

[2] Trisul network analytics, 2017. https://www.trisul.org/blog/analysing-ssh/post.html.

[3] Dev io, 2018. https://dev.to/onmyway133/how-to-calculate-packet-size-in-voip–54ac.

[4] L. Aceto, A. Ingolfsdottir, K.G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[5] Saeed Akhoondian Amiri, Szymon Dudycz, Stefan Schmid, and Sebastian Wiederrecht. Congestion-free rerouting of flows on dags. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[6] Wael Hosny Fouad Aly and Yehia Kotb. Towards sdn fault tolerance using Petri-nets. In *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–3. IEEE, 2018.

[7] M. Andersen, H.G. Larsen, J. Srba, M.G. Sørensen, and J.H. Taankvist. Verification of liveness properties on closed timed-arc Petri nets. In *Proceedings of the 8th Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'12)*, volume 7721 of *LNCS*, pages 69–81. Springer-Verlag, 2013.

[8] Simon Bauer, Daniel Raumer, Paul Emmerich, and Georg Carle. Behind the scenes: what device benchmarks can tell us. In *Proceedings of the Applied Networking Research Workshop*, pages 58–65. ACM, 2018.

[9] Sebastian Brandt, Klaus-Tycho Förster, and Roger Wattenhofer. On consistent migration of flows in sdns. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[10] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed and robust sdn control plane for transactional network updates. In *2015 IEEE conference on computer communications (INFOCOM)*, pages 190–198. IEEE, 2015.

[11] Richard Chirgwin. Google routing blunder sent japan's internet dark on friday. In *https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/*, 2017.

[12] David Clark, Jennifer Rexford, and Amin Vahdat. A purpose-built global network: Google's move to sdn. *Communications of the ACM*, 2016.

[13] A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 492–497, 2012.

[14] A. David, L. Jacobsen, M. Jacobsen, and J. Srba. A forward reachability algorithm for bounded timed-arc Petri nets. In *Proceedings of the 7th International Conference on Systems Software Verification (SSV'12)*, volume 102 of *EPTCS*, pages 125–140. Open Publishing Association, 2012.

[15] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't touch this: Consistent network updates for multiple policies. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 133–143. IEEE, 2016.

[16] Nick Feamster and Jennifer Rexford. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583*, 2017.

[17] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. Model checking data flows in concurrent network updates (full version). *arXiv preprint arXiv:1907.11061*, 2019.

[18] K. Foerster, S. Schmid, and S. Vissicchio. Survey of consistent software-defined network updates. *IEEE Communications Surveys Tutorials*, 21(2):1435–1461, 2019.

[19] Klaus-Tycho Foerster. On the consistent migration of unsplittable flows: Upper and lower complexity bounds. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–4. IEEE, 2017.

[20] Klaus-Tycho Foerster, Thomas Luedi, Jochen Seidel, and Roger Wattenhofer. Local checkability, no strings attached:(a) cyclicity, reachability, loop free updates in sdns. *Theoretical Computer Science*, 709:48–63, 2018.

[21] Jesper Stenbjerg Jensen, Troels Beck Krogh, Jonas Sand Madsen, Stefan Schmid, Jiri Srba, and Marc Tom Thorgersen. P-rex: Fast verification of mpls networks with multiple link failures. In *Proc. 14th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[22] Kurt Jensen. Coloured Petri nets and the invariant-method. *Theoretical computer science*, 14(3):317–336, 1981.

[23] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 539–550. ACM, 2014.

[24] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 99–111, 2013.

[25] Andreas H Klostergaard. Efficient unfolding and approximation of colored Petri nets with inhibitor arcs, 2018. Master Thesis, Department of Computer Science, Aalborg University.

[26] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[27] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 411–422. ACM, 2013.

[28] Arne Ludwig, Szymon Dudycz, Matthias Rost, and Stefan Schmid. Transiently secure network updates. *ACM SIGMETRICS Performance Evaluation Review*, 44(1):273–284, 2016.

[29] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Scheduling loop-free network updates: It's good to relax! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 13–22. ACM, 2015.

[30] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proc. 13th ACM Workshop on Hot Topics in Networks (HotNets)*, page 15. ACM, 2014.

[31] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *Proc. 12th ACM Workshop on Hot Topics in Networks (HotNets)*, page 20. ACM, 2013.

[32] Jedidiah McClurg, Hossein Hojjat, Pavol Černỳ, and Nate Foster. Efficient synthesis of network updates. In *Acm Sigplan Notices*, volume 50, pages 196–207. ACM, 2015.

[33] Tal Mizrahi and Yoram Moses. Time4: Time for sdn. *IEEE Transactions on Network and Service Management*, 13(3):433–446, 2016.

[34] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 1–13, 2013.

[35] Peter Pereíni, Maciej Kuzniar, Marco Canini, and Dejan Kostić. Espres: transparent sdn update scheduling. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 73–78. ACM, 2014.

[36] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[37] Chander Ramchandani. *Analysis of asynchronous concurrent systems by timed petri nets*. PhD thesis, Massachusetts Institute of Technology, 1973.

[38] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, 2012.

[39] Piotr Rygielski, Marian Seliuchenko, and Samuel Kounev. Modeling and prediction of software-defined networks performance using queueing petri nets. In *Proc. 9th EAI International Conference on Simulation Tools and Techniques*, pages 66–75. ICST (Institute for Computer Sciences, Social-Informatics and ..., 2016.

[40] Sahel Sahhaf, Wouter Tavernier, Matthias Rost, Stefan Schmid, Didier Colle, Mario Pickavet, and Piet Demeester. Network service chaining with optimized network function embedding supporting service decompositions. *Computer Networks*, 93:492–505, 2015.

[41] Rishi Sinha, Christos Papadopoulos, and John Heidemann. Internet packet size distributions: Some observations. Technical Report ISI-TR-2007-643, USC/Information Sciences Institute, May 2007. Orignally released October 2005 as web page http://netweb.usc.edu/%7ersinha/pkt-sizes/.

[42] W. M. P. van der Aalst. Interval timed coloured Petri nets and their analysis. In *Application and Theory of Petri Nets (APTN'93)*, volume 691 of *LNCS*, pages 453–472. Springer, 1993.

[43] Linyuan Yao, Ping Dong, Tao Zheng, Hongke Zhang, Xiaojiang Du, and Mohsen Guizani. Network security analyzing and modeling based on petri net and attack tree for sdn. In *2016 International Conference on Computing, Networking and Communications (ICNC)*, pages 1–5. IEEE, 2016.

[44] Jiaqi Zheng, Guihai Chen, Stefan Schmid, Haipeng Dai, and Jie Wu. Chronus: Consistent data plane updates in timed sdns. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 319–327. IEEE, 2017.

[45] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In *Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 73–85, 2015.

[46] Wlodek M Zuberek. Timed Petri nets definitions, properties, and applications. *Microelectronics Reliability*, 31(4):627–644, 1991.

Appendix

A FORMAL DEFINITIONS OF TACPN

In the following, we formally define time-arc colored Petri nets, an extension of the colored Petri net (CPN) [22] with timed-arcs. The following definitions are based on [25]. We will start with some preliminary definitions.

A *finite* multiset over a set S is a collection of elements where a *finite* number of those elements occur a *finite* number of times in the multiset, formally a multiset b over the set S is a function $b : S \to \mathbb{N}^0$. Here $b(s)$ is the number of occurrences of the element s in the multiset b.

The common representation of the multiset b is by a formal sum:

$$\sum_{s \in S} b(s)'(s) \ .$$

We denote the *empty multiset* by $\emptyset$ and the *set of all finite multisets* over S by $\mathcal{B}(S)$. To ease the definition of multiset subtraction, we define the function *non-minus* as:

$$\textit{non-minus(x, y)} = \begin{cases} x - y & \text{if } x - y \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

For multisets we define the following operations where $b, b_1, b_2 \in \mathcal{B}(S)$, $s \in S$, and $n \in \mathbb{N}_0$

- $s \in b$ iff $b(s) > 0$
- $b_1 \uplus b_2 = \sum_{s \in S} (b_1(s) + b_2(s))'(s)$ (summation)
- $n * b = \sum_{s \in S} (n \cdot b(s))'(s)$ (scalar-multiplication)
- $b_1 \subseteq b_2$ iff $\forall s \in S : b_1(s) \leq b_2(s)$
- $b_1 = b_2$ iff $b_1 \subseteq b_2 \wedge b_2 \subseteq b_1$
- $|b| = \sum_{s \in S} b(s)$
- $b_1 \setminus b_2 = \sum_{s \in S} (\textit{non-minus}(b_1(s), b_2(s)))'(s)$

A.1 Colors

In a colored Petri net, every token is valued with a color. The set of all colors is defined as $\mathbb{C}$. Every color is an element of a color type. The set of all color types is defined as $\Sigma \subseteq 2^{\mathbb{C}}$. The colors of each color type are distinct, i.e. $\forall \sigma_1, \sigma_2 \in \Sigma.\ \sigma_1 \cap \sigma_2 = \emptyset$. The color type that a color belongs to is given by the function $Type_{\mathbb{C}} : \mathbb{C} \to \Sigma$.

Each color type is classified as one of the following:

Dots
The set containing only the dot color. It is denoted by $\{\bullet\}$. The dot color corresponds to tokens in a regular Petri net.

Cyclic enumerations
Is a set of elements represented as a sequence of non-repeating elements, where the sequence determines successor and predecessor elements.

Integers
Is a subset of Integer that contains integers. Any set of consecutive integers, together with a successor function and predecessor function. Is a cyclic enumeration only containing integers.

Product type
The Cartesian product of several color types.

A *cyclic enumeration* is a sequence, with length n, of non-repeating elements, i.e $S = (e_0, e_1, \ldots, e_n)$. The successor function and the predecessor function of a cyclic enumeration yields the successor and predecessor of a given element in the enumeration and is defined as:

$$Succ(e_i) = \begin{cases} e_0 & \text{if } i = n \\ e_{(i+1)} & \text{otherwise} \end{cases}$$

$$Pred(e_i) = \begin{cases} e_n & \text{if } i = 0 \\ e_{(i-1)} & \text{otherwise} \end{cases}$$

Note that the enumeration is cyclic, meaning that the successor of the last element in the sequence is the first element, i.e. $Succ(e_n) = e_0$. Likewise, the predecessor of the first element in the sequence is the last element, i.e., $Pred(e_0) = e_n$.

A.2 Variables, Types and Bindings

Before describing expressions, we must first give a notion of variables, types and bindings. Variables are used to represent colors and the set of all variables is denoted Var. Like with colors, each variable has an associated color type. We define this with the type function, $Type_{Var} : Var \to \Sigma$, that maps each variable to the color type of that variable. Finally, variables can have a color bound to them. A binding, $b : Var \to \mathbb{C}$, binds each variable to a concrete color such that $\forall v \in Var.\ b(v) \in Type_{Var}(v)$, or in other words the bounded color is in the color type of the variable. We denote a binding b as $\langle v_0 = c_0, v_1 = c_1, \ldots, v_k = c_k \rangle$, if for all $i \in [0, k]$ we have $b(v_i) = c_i$. The set of all bindings is denoted by $\mathbb{B}$.

A.3 Color Expressions

The color expression $\tau \in \mathbb{T}$ is defined as:
$\tau ::= \mu \mid (\tau, \ldots \tau)$
$\mu ::= \bullet \mid c \mid var \mid \mu{+}{+} \mid \mu{-}{-}$

where $c \in \mathbb{C}$ and $var \in Var$.

A.3.1 Types. Color expressions also have a type, similar to variables, that is given by the function $Type_{\mathbb{T}} : \mathbb{T} \to \Sigma$. The type of a color expression corresponds to the color type of the constants and variables within the expression. The $Type_{\mathbb{T}}$[1] function is defined as:

$$Type_{\mathbb{T}}((\tau_1, \tau_2, \ldots, \tau_n)) =$$
$$Type_{\mathbb{T}}(\tau_1) \times Type_{\mathbb{T}}(\tau_2) \times \cdots \times Type_{\mathbb{T}}(\tau_n)$$
$$Type_{\mathbb{T}}(\bullet) = \{\bullet\}$$
$$Type_{\mathbb{T}}(c) = Type_{\mathbb{C}}(c)$$
$$Type_{\mathbb{T}}(var) = Type_{Var}(var)$$
$$Type_{\mathbb{T}}(\mu{+}{+}) = Type_{\mathbb{T}}(\mu) = Type_{\mathbb{T}}(\mu{-}{-})$$

[1] As a remark, we in our type system do not allow integers from two different ranges to be compared. This however is allowed in our implementation.

A.3.2 Semantics. In order to evaluate the color expressions with a given binding we define the function $[\![]\!] : \mathbb{T} \times \mathbb{B} \rightarrow \mathbb{C}$.

Definition 1. ***(Color semantics)***
$[\![(\bullet)\langle b\rangle]\!] = \bullet$ *(neutral-color)*
$[\![(c)\langle b\rangle]\!] = c$ *(constant)*
$[\![(var)\langle \ldots, var = c, \ldots\rangle]\!] = c$ *(variable)*
$[\![(\mu{+}{+})\langle b\rangle]\!] = Succ([\![(\mu)\langle b\rangle]\!])$ *(successor)*
$[\![(\mu{-}{-})\langle b\rangle]\!] = Pred([\![(\mu)\langle b\rangle]\!])$ *(predecessor)*
$[\![((\tau, \ldots, \tau))\langle b\rangle]\!] = ([\![(\tau)\langle b\rangle]\!], \ldots, [\![(\tau)\langle b\rangle]\!])$ *(product)*

An example of a color expression could be $(x, y++)$ which denotes a product type of the variables x and y. Since both x and y are variables we need to get the type for both of them which we use the function $Type_{Var}(x) = [1, 3]$ and $Type_{Var}(y) = [5, 7]$. Then the color function color expression can be evaluated under the binding $[\![(x, y)\langle x = 1, y = 6\rangle]\!] = (1, 7)$.

A.4 Guard Expressions

The set of all guard expressions is defined as Γ and has the following syntax:

$\gamma ::= true \mid false \mid \neg\gamma \mid \gamma_1 \wedge \gamma_2 \mid \gamma_1 \vee \gamma_2 \mid \gamma_1 \rightarrow \gamma_2 \mid \gamma_1 \leftrightarrow \gamma_2 \mid \gamma_1$ xor $\gamma_2 \mid \tau_1 \bowtie \tau_2$
where $Type_{\mathbb{T}}(\tau_1) = Type_{\mathbb{T}}(\tau_2)$ and $\bowtie$ denotes the comparison operators $<, \leq, >, \geq, =,$ and $\neq$.

A.4.1 Semantics. All guard expressions with a binding will evaluate to a Boolean value either *true* or *false*. To evaluate the guard expressions the function $[\![]\!] : \Gamma \times \mathbb{B} \rightarrow \{true, false\}$

Definition 2. ***(Guard semantics)***
$[\![(\neg\gamma)\langle b\rangle]\!] = \neg[\![(\gamma)\langle b\rangle]\!]$ *(negation)*
$[\![(\gamma_1 \vee \gamma_2)\langle b\rangle]\!] = [\![(\gamma_1)\langle b\rangle]\!] \vee [\![(\gamma_2)\langle b\rangle]\!]$ *(disjunction)*
$[\![(\gamma_1 \wedge \gamma_2)\langle b\rangle]\!] = [\![(\gamma_1)\langle b\rangle]\!] \wedge [\![(\gamma_2)\langle b\rangle]\!]$ *(conjunction)*
$[\![(\gamma_1 \rightarrow \gamma_2)\langle b\rangle]\!] = [\![(\gamma_1)\langle b\rangle]\!] \rightarrow [\![(\gamma_2)\langle b\rangle]\!]$ *(implication)*
$[\![(\gamma_1 \leftrightarrow \gamma_2)\langle b\rangle]\!] = [\![(\gamma_1)\langle b\rangle]\!] \leftrightarrow [\![(\gamma_2)\langle b\rangle]\!]$ *(biconditional)*
$[\![(\gamma_1\ xor\ \gamma_2)\langle b\rangle]\!] = [\![(\gamma_1)\langle b\rangle]\!]\ xor\ [\![(\gamma_2)\langle b\rangle]\!]$ *(exclusive-or)*
$[\![(\tau_1 \bowtie \tau_2)\langle b\rangle]\!] = [\![(\tau_1)\langle b\rangle]\!] \bowtie [\![(\tau_2)\langle b\rangle]\!]$ *(comparison)*

An example of a guard expression γ is $(a \leq b \wedge b{+}{+} < c)$. If $Type_{Var}(a) = Type_{Var}(b) = Type_{Var}(c) = [1, 5]$ then this guard expression can be evaluated with a binding $[\![\gamma\langle a = 1,\ b = 2,\ c = 4\rangle]\!] = true.$

A.5 Arc Expressions

The set of all arc expressions is defined as Δ and has the following syntax:

$\delta ::= n'(\tau) \mid n'(\sigma.all) \mid \delta_1 \uplus \delta_2 \mid \delta_1 \setminus \delta_2 \mid n * \delta$

where $n \in \mathbb{N}$, τ is a color expression, σ is a color type and $Type_\Delta(\delta_1) = Type_\Delta(\delta_2)$ as defined next.

A.5.1 Types. Arc expressions, like color expressions, can also be associated with a type, given by the function $Type_\Delta : \Delta \rightarrow \Sigma$. The type of an arc expression corresponds to the color type of the colors within the expression. The $Type_\Delta$ function is defined as follows:

$$Type_\Delta(n'(\tau)) = Type_{\mathbb{T}}(\tau)$$
$$Type_\Delta(n'(\sigma.all)) = \sigma$$
$$Type_\Delta(\delta_1 \uplus \delta_2)\ if\ Type_\Delta(\delta_1) = Type_\Delta(\delta_2)$$
$$Type_\Delta(\delta_1 \setminus \delta_2)\ if\ Type_\Delta(\delta_1) = Type_\Delta(\delta_2)$$
$$Type_\Delta(n * \delta) = Type_\Delta(\delta)$$

A.5.2 Semantics. In order to evaluate the arc expressions we define the function: $[\![]\!] : \Delta \times \mathbb{B} \rightarrow \mathbb{C}_{MS}$

Definition 3. ***(Arc semantics)***
$[\![(n'(\tau))\langle b\rangle]\!] = n'([\![(\tau)\langle b\rangle]\!])$ *(number-of)*
$[\![(n'(\sigma.all))\langle b\rangle = \sum_{c \in \sigma}[\![(n'(c))\langle b\rangle]\!]$ *(all)*
$[\![(\delta_1 \uplus \delta_2)\langle b\rangle]\!] = [\![(\delta_1)\langle b\rangle]\!] \uplus [\![(\delta_2)\langle b\rangle]\!]$ *(addition)*
$[\![(\delta_1 \setminus \delta_2)\langle b\rangle]\!] = [\![(\delta_1)\langle b\rangle]\!] \setminus [\![(\delta_2)\langle b\rangle]\!]$ *(subtraction)*
$[\![(n * \delta)\langle b\rangle]\!] = n * [\![(\delta)\langle b\rangle]\!]$ *(scalar)*

As an example, an arc expression δ is defined as $2'(x) + 1'(2)$. The type of the variable is given as $Type_{Var}(x) = [1, 3]$, then it can be evaluated under a binding where $[\![\delta\langle x = 3\rangle]\!] = 2'(3) + 1'(2)$.

Before introducing the TACPN, we will first define time intervals.

Definition 4. ***(Time intervals)***
We define the set of well-formed closed time intervals as:

$$\mathcal{I} \stackrel{\text{def}}{=} \{[a, b] \mid a \in \mathbb{N}_0,\ b \in \mathbb{N}_0^\infty,\ a \leq b\}$$

and its subset $\mathcal{I}^{inv}$ used in assigning the age invariant defined as:

$$\mathcal{I}^{inv} \stackrel{\text{def}}{=} \{[0, b] \mid b \in \mathbb{N}_0^\infty\}$$

A.6 TACPN Definition

We will now formally define TACPNs.

Definition 5. ***(Timed-Arc Colored Petri Net)***
A Timed-Arc Colored Petri Net (TACPN) is a 15-tuple $(P, T, T_{urg}, IA, OA, INA, TA, \Sigma, C, CG, W, W_I, W_T, TG, I)$ where

(1) *P is a finite set of* ***places****,*
(2) *T is a finite set of* ***transitions*** *such that $P \cap T = \emptyset$,*
(3) *$T_{urg} \subseteq T$ is a finite set of* ***urgent transitions****,*
(4) *$IA \subseteq P \times T$ is a finite set of* ***input arcs****,*
(5) *$OA \subseteq T \times P$ is a finite set of* ***output arcs****,*
(6) *$INA \subseteq P \times T$ is a finite set of* ***inhibitor arcs****,*
(7) *$TA \subseteq P \times T \times P$ is a finite set of* ***transport arcs*** *such that $(p, t, p') \in TA \Rightarrow (p, t) \notin IA \wedge (t, p') \notin OA$, $\forall(p, t, p'), (p, t, p'') \in TA.\ p' = p''$ and $\forall(p', t, p), (p'', t, p) \in TA.\ p' = p''$,*
(8) *Σ is a finite set of* ***color sets****,*
(9) *$C : P \rightarrow \Sigma$ is a* ***color function****,*
(10) *$CG : T \rightarrow \Gamma$ is a* ***color guard****.*
(11) *$W : IA \cup OA \rightarrow \Delta$ is a finite set of* ***arc expressions*** *such that $Type_\Delta(W((p, t))) = Type_\Delta(W((t, p))) = C(p)$,*

(12) $W_I : INA \to \mathbb{N}$ *is a function assigning* ***inhibitor weights*** *to inhibitor arcs*
(13) $W_T : TA \to \mathbb{N} \times \mathbb{T} \times \mathbb{T}$ *is a function assigning* ***transport weights*** *to transport arcs that specifies a numeric weight, an input color and an output color in that order, such that* $W_T((p,t,p')) = (n,\tau,\tau') \Rightarrow (Type_{\mathbb{T}}(\tau) = C(p) \wedge Type_{\mathbb{T}}(\tau') = C(p'))$,
(14) $TG : IA \cup TA \to (C \to \mathcal{I})$ *is a* ***timed guard*** *such that* $\forall t \in T_{urg},\ \forall c \in C.\ TG(p,\ t)(c) = TG(p,t,p')(c) = [0,\infty]$,
(15) $I : P \to (C \to \mathcal{I}^{inv})$ *is a function assigning* ***age invariants*** *to each color of a place.*

In point 14) we use the notation $(C \to \mathcal{I})$ to express that each color of a color type got its own time guard, and in 15) we use $(C \to \mathcal{I}^{inv})$ to express that each color got its own invariant. Since we are able to express time guards and invariants for all colors, we use * as a graphical notation to express that if a color does not have a specified time guard or interval, it will have the one expressed with *.

Before we present the formal semantics for the model we introduce some notation.
Let $N = (P, T, T_{urg}, IA, OA, INA, TA, \Sigma, C, CG, W, W_I, W_T, TG, I)$ be a TACPN. We define

$$^{\bullet}y \stackrel{\text{def}}{=} \{z \in P \cup T \mid\ (z,y) \in IA \cup OA \vee (z,y,\_) \in TA \vee (\_,z,y) \in TA\}$$

as the preset of a transition or place y. The postset of y is defined as:

$$y^{\bullet} \stackrel{\text{def}}{=} \{z \in P \cup T \mid\ (y,z) \in IA \cup OA \vee (y,z,\_) \in TA \vee (\_,y,z) \in TA\}$$

We will also define the set of bindings that satisfy the color guard of a given transition t as

$$B(t) \stackrel{\text{def}}{=} \{b \in \mathbb{B} \mid CG(t)\langle b\rangle\}$$

A.7 Markings

Markings decorate a Petri-net with tokens. Let N be a TACPN. A marking M on N is a function $M : P \to \mathcal{B}(\mathbb{R}_{\geq 0} \times \mathbb{C})$, such that $\forall p.$ if $(d,c) \in M(p)$ then $d \in I(p)(c)$. We write $(p,\ x,\ c)$ to denote a token at a place p, with age $x \in \mathbb{R}_{\geq 0}$ and color $c \in \mathbb{C}$. The set of all markings in a net N is denoted $\mathcal{M}(N)$. We define the size of a marking as $|M| = \sum_{p \in P} |M(p)|$. A marked TACPN (N, M_0) is then defined as a TACPN together with an initial marking M_0 where $\forall (p,\ x,\ c) \in M_0.\ x = 0$.

To ease further definitions, we define how to strip either time or colors from our markings. We define two functions denoted with subscript, one function *nc*, that strips colors from a marking M, defined as:

$$M_{nc}(p)(x) = \sum_{c\ \in\ C(p)} M(p)(x,\ c)$$

and a second function *nt*, that strips time from a marking M, defined as:

$$M_{nt}(p)(c) = \sum_{x\ \in\ \mathbb{R}_{\geq 0}} M(p)(x,\ c)$$

A.8 Enabledness & Semantics

With markings defined, we can now define enabledness and transition firing. We will first define enabledness and transition firing for colored timed-arc Petri nets.

Definition 6. ***(Enabledness)***
A transition $t \in T$ is ***enabled*** *under binding $b \in B(t)$ in a marking M by the markings In and Out, denoted by $M \vdash t$, if the following conditions are satisfied:*

In is a sub-marking of M i.e.

$$\forall p \in P.\ In(p) \subseteq M(p) \tag{a}$$

In only has tokens in the preset of t, while Out only has tokens in the postset of t i.e.

$$\forall p \notin {}^{\bullet}t.\ In(p) = \emptyset \tag{b}$$

$$\forall p \notin t^{\bullet}.\ Out(p) = \emptyset \tag{c}$$

For all input arcs expect the inhibitor arcs, the colors of the tokens from In satisfy the arc expression evaluated under binding b.

$$\forall (p,\ t) \in IA.\ In_{nt}(p) = W(p,\ t)\langle b\rangle \tag{d}$$

Similarly, the output tokens of an output arc have colors corresponding to the expression of that arc evaluated under binding b.

$$\forall (t,\ p) \in OA.\ Out_{nt}(p) = W(t,\ p)\langle b\rangle \tag{e}$$

The input tokens of a transport arc must have the same color as the input color of the transport weight evaluated under b, while the color of the output tokens must match the output color. Additionally, the number of input tokens must match the number of output tokens which also matches the numeric weight of the transport weight. This is captured by the following rule:

$$W_T((p,\ t,\ p')) = \\ (n,\ \tau,\ \tau') \Rightarrow (In_{nt}(p) = n(\tau)\langle b\rangle \wedge Out_{nt}(p') = n(\tau')\langle b\rangle) \tag{f}$$

For all input arcs and transition arcs, for each color of all tokens in In have to satisfy the time guard for each color of the arc i.e.

$$\forall (p,\ x,\ c) \in In.\ (p,\ t) \in IA \Rightarrow x \in TG((p,\ t)(c)) \tag{g}$$

$$\forall (p,\ x,\ c) \in In.\ \forall p' \in t^{\bullet}.\ (p,\ t,\ p') \in TA \Rightarrow x \in TG((p,\ t,\ p')(c)) \tag{h}$$

All colors of all output tokens must satisfy the color invariants of the output place i.e.

$$\forall (p,\ x,\ c) \in Out.\ x \in I(p)(c) \tag{i}$$

For all output arcs, the age of the output token is 0 i.e.

$$\forall (p,\ x,\ c) \in Out.\ (t,\ p) \in OA \Rightarrow x = 0 \tag{j}$$

For all transport arcs, the ages of the input tokens matches the ages of the output tokens i.e.

$$\forall (p,\ t,\ p') \in TA.\ In_{nc}(p) = Out_{nc}(p') \tag{k}$$

For all inhibitor arcs from place p to transition t, the number of tokens in p have to be less than the inhibitor weight of the arc i.e.

$$\forall p \in {}^{\bullet}t.\ (p,\ t) \in INA \Rightarrow |M(p)| < W_I(p,\ t) \tag{l}$$

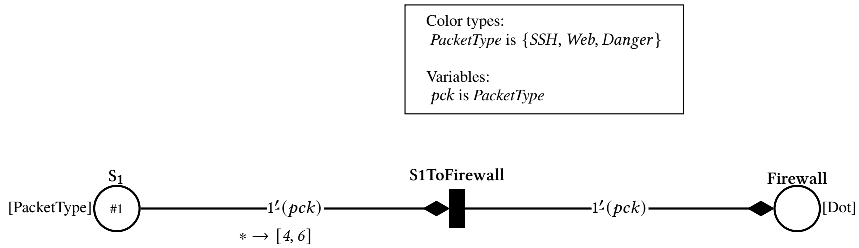


Figure 8: Enabledness example of Figure 2

To give an example of a marked TACPN we have isolated the part where the $\mathbf{S_1}$ sends packets to **Firewall** of Figure 2 as shown on Figure 8, where we have the marking

$$M = \{(S_1, 5, Web)\}$$

here transition is enabled under the binding $b = \langle pck = Web\rangle$ by the markings

$$In = \{(S_1, 5, Web)\}$$
$$Out = \{(Firewall, 5, Web)\}$$

DEFINITION 7. ***Timed Transition System*** *The semantics of a marked TACPN (N, M_0) is defined as a timed transition system $[\![N]\!]_{sem} = (\mathcal{M}(N), M_0, \to)$, where $\to : \mathcal{M}(N) \times T \cup \mathbb{R}_{\geq 0} \times \mathcal{M}(N)$ is the least transition relation generated by the following two rules:*

Transition firing
If a transition $t \in T$ under binding $b \in B(t)$ is enabled in a marking M_1 by the multisets of tokens In and Out, it may fire, changing the marking M_1 to M_2, where M_2 is defined as:

$$M_2 = (M_1 \setminus In) \uplus Out$$

This gives rise to the transition relation $(M_1, t, M_2) \in \to$, denoted by $M_1 \xrightarrow{t} M_2$

Time delay
A marking M can be delayed by a time duration $d \in \mathbb{R}_0$ if the following two conditions hold:

- *The delayed tokens all satisfy the invariants of their respective places, i.e.*
  $\forall (p, x, c) \in M.\ x + d \in I(p)(c)$
- *The duration is 0, if any urgent transitions are enabled, i.e.*
  $\forall t \in T_{urg}.\ M \vdash t \Rightarrow d = 0$

Delaying a marking M_1 by a duration d results in a new marking M_2 defined as:

$$M_2(p) = \{(p, x + d, c) \mid (p, x, c) \in M_1(p)\}$$

This gives rise to the transition relation $(M_1, d, M_2) \in \to$, denoted by $M_1 \xrightarrow{d} M_2$.

With the formal semantics defined, we with TACPN are able to model networks, where we with the notation of colors are able to represent the different packets, and with the notation of time, are able to model the calculation time for each router, packet delays for each packet type, the speed of each link in the network for each packet type etc. In order to utilize existing techniques for TAPNs, we provide a algorithm to *unfold* a given TACPN into a TAPN, while preserving the behavior of the model. In this section we will describe this unfolding following with a theorem of strongly timed bisimulation, and lastly a proof of this theorem.

A.9 Timed-Arc Petri Net

A TAPN is a TACPN without colors. More specifically it is a TACPN where the color sets only has the dots color type, defined as follows:

DEFINITION 8. ***(Timed-Arc Petri Net)*** *Let $N = (P, T, T_{urg}, IA, OA, INA, TA, \Sigma, C, CG, W, W_I, W_T, TG, I)$ be a TACPN. The TACPN N is a TAPN iff $\Sigma = \{\{\bullet\}\}$.*
Because a TAPN has no colors, the components of the tuple related to color can be simplified or removed as follows:

- *Color sets, Σ, can be removed, since it is already defined.*
- *The color function, C, can be removed, since it always yields the same color type.*
- *Color guards, CG, become trivial and can be removed, because only one binding can ever occur.*
- *Arc expressions, W, can only yield a number of tokens of the same color, and can therefore be simplified to just give that number.*
- *Transport weights, W_T, no longer needs an input color and an output color and can be reduced to just the numeric weight.*
- *Since arc expressions, and transport weights now have the same target set, they can be combined to a single weight function, W, assigning numeric weights arcs.*

A TAPN is therefore uniquely defined by the 11-tuple: $TAPN = (P, T, T_{urg}, IA, OA, INA, TA, W, W_I, TG, I)$.

A.10 Unfolding TACPN to Timed-Arc PN

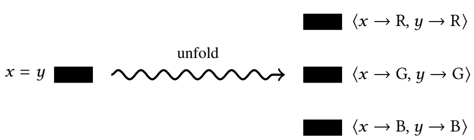
In order to unfold a TACPN into TAPN that preserves the behavior of the original net, we must be able to express the features of colors, without actually having them. This subsection will explain how this is achieved.

A.10.1 Places and tokens. In a TACPN, all places have an associated color type. Additionally each token in the marking of those places has a color. In order to distinguish between tokens of different colors in the unfolded net, we need to make separate places for each possible color, such that tokens of different colors are in different places. We also add an extra place for each original place that keeps track of the sum of tokens across all colors. This **sum** place initially gets one token for each token at the place in the original net. The number of tokens in the sum place will then invariantly be the sum of tokens across the colors, i.e.

$$|M((p, \mathbf{sum}))| = \sum_{c \in C(p)} |M((p,c))|$$

As for time, the new places inherits the invariant corresponding to the color invariant of the color they represent, except the **sum** which has an $[0,\infty)$ invariant. If an invariant is not specified, the new place will get the default invariant represented by the $*$. The age of the tokens in **sum** is initially 0.

A.10.2 Transitions. In a TACPN, firing a transition can have different outcomes depending on the given binding. In the unfolded net, we separate these possible outcomes into individual transitions, one for each possible binding that satisfy the color guard in the original net.



A.10.3 Input arcs and output arcs. In a TACPN, arcs are decorated with arc expressions that when evaluated yields a multiset of colors. In the unfolded net, we decompose the resulting multiset into its individual colors and spread these as weights across multiple arcs, one for each color present in the multiset. Additionally, we add an arc to the sum place with a summed weight of the other arcs. This ensures that the tokens in the sum gets updated accordingly whenever a transition is fired. As for time, all arcs, except the one connecting sum, inherit the time guard for the given color, with similar reasoning as with invariants.

A.10.4 Transport arcs and inhibitor arcs. In a TACPN, transport arcs transfer tokens between places, possibly changing their color along the way. This translates well in the unfolded net, as we simply add a transport arc between the specified colors. Akin to the regular arcs, we also add regular arcs to the sum, that ensures it stays updated. In a TACPN, an inhibitor arc could count the number of tokens across colors, since they were all in the same place. However, in the unfolded net, we chose to spread the tokens across multiple places. This is why we need the sum place, as connecting the inhibitor arc to this place allows it to use the sum across the colors and function as in the original net.

Given the above reasoning, we will now formally present the unfolding.

DEFINITION 9. ***Unfolded TAPN of a TACPN***
Let $N = (P, T, T_{urg}, IA, OA, INA, TA, \Sigma, C, CG, W, W_I, W_T, TG, I)$ be a TACPN. The unfolded TAPN of a given TACPN N, is a 11-tuple, $N' = (P', T', T'_{urg}, IA', OA', INA', TA', W', W'_I, TG'I')$ where

(1) $P' = \{(p,c) \mid p \in P \wedge c \in C(p)\} \cup \{(p, \mathbf{sum}) \mid p \in P\}$
(2) $T' = \{(t,b) \mid t \in T \wedge b \in B(t)\}$
(3) $T'_{urg} = \left\{(t,b) \mid t \in T_{urg} \wedge (t,b) \in T'\right\}$
(4) $W'((p,c),(t,b)) = W((p,t))\langle b\rangle(c)$
(5) $W'((p,\mathbf{sum}),(t,b)) = \begin{cases} |W((p,t))\langle b\rangle| & \mathit{if}\ (p,t) \in IA \\ n \ \mathit{where}\ W_T((p,t,p')) = (n,\tau,\tau') & \mathit{if}\ (p,t,p') \in TA \end{cases}$
(6) $W'((t,b),(p,c)) = W((t,p))\langle b\rangle(c)$
(7) $W'((t,b),(p,\mathbf{sum})) = \begin{cases} |W((t,p))\langle b\rangle| & \mathit{if}\ (t,p) \in OA \\ n \ \mathit{where}\ W_T((p',t,p)) = (n,\tau,\tau') & \mathit{if}\ (p',t,p) \in TA \end{cases}$
(8) $W'((p,c),(t,b),(p',c')) = n \ \mathit{where}\ W_T((p,t,p')) = (n,\tau,\tau')$
(9) $W'_I((p,\mathbf{sum}),(t,b)) = W_I((p,t))\langle b\rangle(c)$
(10) $IA' = \{((p,c),(t,b)) \mid (p,t) \in IA \wedge (p,c) \in P' \wedge (t,b) \in T' \wedge W'(((p,c),(t,b))) > 0\} \cup \{((p,\mathbf{sum}),(t,b)) \mid (p,t,p') \in TA \wedge (t,b) \in T'\}$
(11) $OA' = \{((t,b),(p,c)) \mid (t,p) \in OA \wedge (p,c) \in P' \wedge (t,b) \in T' \wedge W'(((t,b),(p,c))) > 0\} \cup \{((t,b),(p,\mathbf{sum})) \mid (p',t,p) \in TA \wedge (t,b) \in T'\}$
(12) $INA' = \{((p,\mathbf{sum}),(t,b)) \mid (p,t) \in INA \wedge (t,b) \in T'\}$
(13) $TA' = \{((p,c),(t,b),(p',c')) \mid (p,t,p') \in TA \wedge (t,b) \in T' \wedge W_T((p,t,p'))\langle b\rangle = (n,c,c')\}$
(14) $TG'((p,c),(t,b)) = TG((p,t)(c))$
(15) $TG'((p,\mathbf{sum}),(t,b)) = [0,\infty]$
(16) $TG'((p,c),(t,b),(p',c')) = TG((p,t,p')(c))$
(17) $I'((p,c)) = I(p)(c)$
(18) $I'((p,\mathbf{sum})) = [0,\infty]$

To ease understanding of Definition 9 we use Figure 9 and 10 to explain. P' on Figure 9 contaions the four places (p,R), (p,G) (p,B), and $(p,\mathbf{sum})$ where T' is the new transition. The weight of the arcs is transferred, so since the binding of x on Figure 9 is R we have weight 2 on on the arc from (p,R) to transition $(t,\ b)$, whereas the arc from $(p,\ \mathbf{sum})$ will get the sum of all the weights.

Input and output arcs are only created if we have a binding so on Figure 9 we do not have a binding of B, and therefore we do not create a input arc form $(p,\ B)$ to the transition, or an output arc to $(p,\ tt)$.

Time guards and invariant are transferred from the colors to the places represented the colors, in the unfolded net, expect for the **sum** place which always gets a $[0,\infty]$ time guard and $\leq \infty$ invariant.

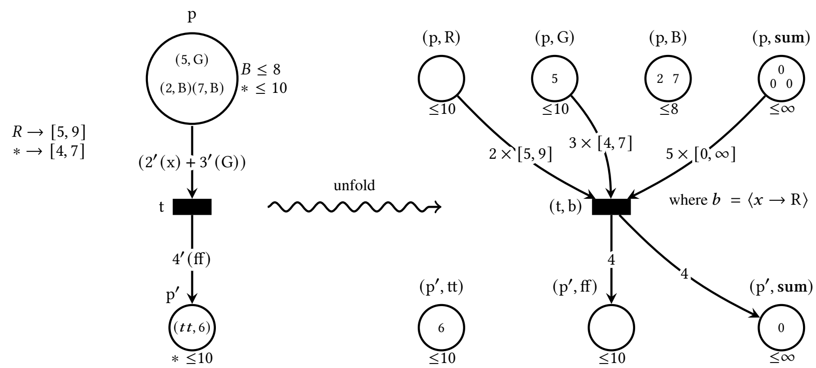


Figure 9: Example of unfolding of a timed-arc colored Petri net

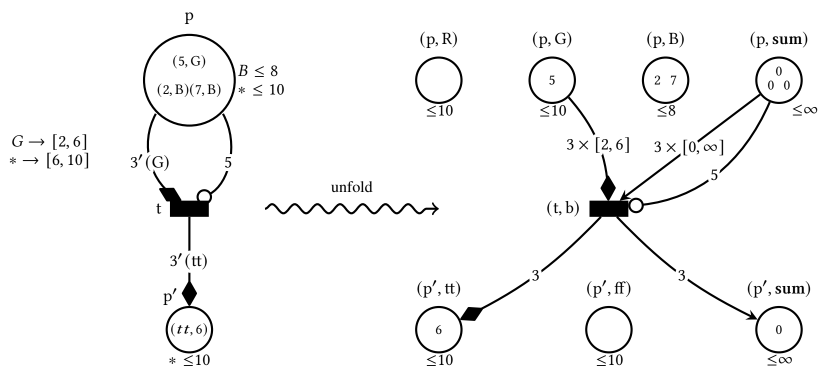


Figure 10: Timed-arc colored Petri net unfolding example with transport and inhibitor arc

We will now define the marking unfolding between the folded TACPN and the unfolded TAPN.

Definition 10. ***Marking unfolding***
Let M be a marking in a TACPN N and let N' be the corresponding unfolded TAPN. The corresponding unfolded marking is given by the function unfold : $\mathcal{M}(N) \to \mathcal{M}(N')$ *defined as:*

$$unfold(M)((p, c))(x) = \begin{cases} |M(p)| & \text{if } c = \mathbf{sum} \land x = 0 \\ 0 & \text{if } c = \mathbf{sum} \land x \neq 0 \\ M(p)((x, c)) & \text{otherwise} \end{cases}$$

The initial marking M_0 of the unfolded net is defined as $unfold(M_0)$ since we for each place $(p,\ c)$ in the unfolded net creates a token for each token with color c in place p in the folded net. In place $(p,\ \mathbf{sum})$ we as mentioned creates a token for each token at the place in the original net.

A.11 Unfolding equivalence

Before we show the equivalence between the folded TACPN and the unfolded TAPN, we first define an equivalence between unfolded TAPNs. As a consequence of our unfolding, the ages of the tokens in the sum places can never be exposed to a time guard or an invariant, rendering these ages obsolete. Therefore, if two markings only differ in the age of the tokens in the sum places, then these markings are bisimilar. We will now define strong timed bisimulation which is based upon definition from [4].

Definition 11. ***Strong timed bisimulation***
A binary relation $\mathcal{R}$ over the set of markings of a Petri net is a strong timed bisimulation iff whenever $M_1 \mathcal{R} M_2$ where a is a action and d is a time delay:

- $M_1 \xrightarrow{a} M_1'$ *then* $\exists\, M_2 \xrightarrow{a} M_2'$ *such that* $M_1' \mathcal{R} M_2'$;
- $M_2 \xrightarrow{a} M_2'$ *then* $\exists\, M_1 \xrightarrow{a} M_1'$ *such that* $M_1' \mathcal{R} M_2'$;
- $M_1 \xrightarrow{d} M_1'$ *then* $\exists\, M_2 \xrightarrow{d} M_2'$ *such that* $M_1' \mathcal{R} M_2'$;
- $M_2 \xrightarrow{d} M_2'$ *then* $\exists\, M_1 \xrightarrow{d} M_1'$ *such that* $M_1' \mathcal{R} M_2'$;

Two markings M and M' are strongly timed bisimilar written $M \sim M'$ iff there is a strong timed bisimulation that relates them.

As a consequence of our unfolding, the ages of tokens in of the **sum** place will grow, but never be exposed to a time guard or invariant. We therefore present Lemma A.1 that if two markings only differ in the ages of tokens in place **sum** then they are up to strong timed bisimular.

LEMMA A.1. *Let M and M' be two markings in the unfolded net. If $M(p, c) = M'(p, c)$ for all p, and c where $c \neq$ **sum** and $|M(p, \mathbf{sum})| = |M'(p, \mathbf{sum})|$ then $M \sim M'$.*

Finally, we finish with our main theorem stating that the original and unfolded nets are strongly timed bisimilar.

THEOREM A.2. *Let M be a marking in a TACPN N. The corresponding unfolded marking $unfold(M)$ is strongly timed bisimilar with M, i.e. $unfold(M) \sim M$.*

A.12 Proof of Theorem A.2

To prove that M and $unfold(M)$ are strongly timed bisimilar, we need to prove the following four statements:

(1) $M_1 \xrightarrow{t} M_2$ under binding b implies $unfold(M_1) \xrightarrow{(t,b)} unfold(M_2)$.
(2) $unfold(M_1) \xrightarrow{(t,b)} M'$ implies $M_1 \xrightarrow{t} M_2$ under binding b where $M' = unfold(M_2)$.
(3) $M_1 \xrightarrow{d} M_2$ implies $unfold(M_1) \xrightarrow{d} M'$ such that $M' \sim unfold(M_2)$.
(4) $unfold(M_1) \xrightarrow{d} M'$ implies $M_1 \xrightarrow{d} M_2$ where $M' \sim unfold(M_2)$.

For statement 3) and 4) both M' and $unfold(M_2)$ are makring in the unfolded net, and they are up to bisimilar by Lemma A.1 if they only differ in the age of tokens in place **sum**. The remainder of this proof will work through each of these statements.

A.12.1 $M_1 \xrightarrow{t} M_2$ *under binding* b *implies* $unfold(M_1) \xrightarrow{(t,\ b)} unfold(M_2)$. By the definition of transition firing, we have that M_1 enables t under binding b by the markings In and Out that satisfy Definition 6. We will show that $unfold(M_1)$ enables $(t,\ b)$ by the markings In' and Out' which is defined as:

$$In' \stackrel{\text{def}}{=} unfold(In)$$

$$Out' \stackrel{\text{def}}{=} unfold(Out)$$

From the definition of enabledness we have twelve conditions, (a) through (l), and for each condition we will show that if In and Out satisfy the condition, then In' and Out' must also satisfy the condition in the unfolded net N'.

Condition a) By definition $In'(p,\ c)(x) = unfold(In)(p,\ c)(x)$. We want to show that In' is a sub-marking of $unfold(M_1)$. Assume that for all $p \in P.\ In(p) \subseteq M_1(p)$. We want to show that for any $(p,\ c) \in P'.In'((p,\ c),(x)) \leq unfold(M_1)((p,\ c)(x))$ for all x. To show this we have three cases.

- Let $c = \textbf{sum} \wedge x = 0$
  $unfold(In)(p,\ c)(x) =$ by Definition 10 case 1

  $|In(p)| \leq$ by $In(p) \subseteq M_1(p)$

  $|M_1(p)| =$ by Definition 10 case 1

  $unfold(M_1)((p,\ c))(x)$

- Let $c = \textbf{sum} \wedge x \neq 0$
  $unfold(In)(p,\ c)(x) =$ by Definition 10 case 2

  $0 =$ by Definition 10 case 2

  $unfold(M_1)((p,\ c))(x)$

- Otherwise $c \neq \textbf{sum}$
  $unfold(In)(p,\ c)(x) =$ by Definition 10 case 3

  $In(p)(x,\ c) \leq$ by $In(p) \subseteq M_1(p)$

  $M_1(p)(x,\ c) =$ by Definition 10 case 3

  $unfold(M_1)((p,\ c))(x)$

Condition b) We want to show that In' only got tokens of the preset of t. Assume that for all $p \notin {}^\bullet t.\ In(p) = \emptyset$. We want to show that for any $(p,\ c) \notin {}^\bullet(t,\ b).\ In'((p,\ c))(x) = 0$ for all x. To show this we have three cases.

- Let $c = \textbf{sum} \wedge x = 0$

$unfold(In)(p,\ c)(x) =$ by Definition 10 case 1

$|In(p)| =$ by Definition 9 case 10 & 4 and $p \notin {}^\bullet t.\ In(p) = \emptyset$

0

- Let $c = \mathbf{sum} \wedge x \neq 0$
  $unfold(In)(p,\ c)(x) =$ by Definition 10 case 2

  0

- Otherwise $c \neq \mathbf{sum}$
  $unfold(In)(p,\ c)(x) =$ by Definition 10 case 3

  $In(p)(x,\ c) =$ by Definition 9 case 10 & 4 and by $p \notin {}^\bullet t.\ In(p) = \emptyset$

  0

Condition c) By definition $Out'(p,\ c)(x) = unfold(Out)(p,\ c)(x)$. We want to show that Out' only got tokens of the postset of t. Assume that for all $p \notin t^\bullet.\ Out(p) = \emptyset$. We want to show that for any $(p,\ c) \notin (t,\ b)^\bullet.\ Out'((p,\ c))(x) = 0$ for all x. To show this we have three cases.

- Let $c = \mathbf{sum} \wedge x = 0$
  $unfold(Out)(p,\ c)(x) =$ by Definition 10 case 1

  $|Out(p)| =$ by Definition 9 case 11 & 6 and $p \notin t^\bullet.\ Out(p) = \emptyset$

  0

- Let $c = \mathbf{sum} \wedge x \neq 0$
  $unfold(Out)(p,\ c)(x) =$ by Definition 10 case 2

  0

- Otherwise $c \neq \mathbf{sum}$
  $unfold(Out)(p,\ c)(x) =$ by Definition 10 case 3

  $Out(p)(x,\ c) =$ by Definition 9 case 11 & 6 and $p \notin t^\bullet.\ Out(p) = \emptyset$

  0

Condition d) For all input arcs all tokens in In' have to satisfy the arc expression evaluated under the binding. By definition $In'_{nt}(p,\ c)(x) = unfold(In_{nt})(p,\ c)(x)$. Assume that $\forall (p,\ t) \in IA.\ In_{nt}(p) = W(p,\ t)\langle b\rangle$. We want to show that $\forall ((p,\ c), (t,\ b)) \in IA'.\ In'_{nt}((p,\ c)) = W'((p,\ c), (t,\ b))$ for all $c \in C$. To show this we have two cases.

- Let $c = \mathbf{sum} \wedge x = 0$
  $(unfold(In))_{nt}(p,\ c) =$ by definition of function nt

  $\sum_{x \in \mathbb{R}_{\geq 0}} unfold(In)(p,\ c)(x) =$ by Definition 10 case 2

  $unfold(In(p,\ c))(0) =$ by Definition 10 case 1

  $|In(p)| =$ by $\forall (p,\ t) \in IA.\ In_{nt}(p) = W(p,\ t)\langle b\rangle$

  $|W(p,\ t)\langle b\rangle| =$ by Definition 9 case 5

  $W'((p,\ c), (t,\ b))$

- Otherwise $c \neq \mathbf{sum}$

$(unfold(In))_{nt}(p,\ c) =$ by definition of function nt

$\sum_{x \in \mathbb{R}_{\geq 0}} unfold(In)(p,\ c)(x) =$ by Definition 10 case 3

$\sum_{x \in \mathbb{R}_{\geq 0}} In(p)(x,\ c) =$ by definition of function nt

$In_{nt}(p)(c) =$ by $\forall(p,\ t) \in IA.\ In_{nt}(p) = W(p,\ t)\langle b\rangle$

$W((p,\ t))\langle b\rangle(c) =$ by Definition 9 case 4

$W'((p,\ c),(t,\ b))$

Condition e) For all output arcs all tokens in Out' have to satisfy the arc expression evaluated under the binding. By definition $Out'_{nt}(p,\ c)(x) = unfold(Out_{nt})(p,\ c)(x)$. Assume that $\forall(t,\ p) \in OA.\ Out_{nt}(p) = W(t,\ p)\langle b\rangle$. We want to show that $\forall((t,\ b),(p,\ c)) \in OA'.\ Out'_{nt}((p,\ c)) = W'((t,\ b),(p,\ c))$ for all $c \in C$. To show this we have two cases.

- Let $c = \mathbf{sum} \wedge x = 0$

  $(unfold(Out))_{nt}(p,\ c) =$ by definition of function nt

  $\sum_{x \in \mathbb{R}_{\geq 0}} unfold(Out)(p,\ c)(x) =$ by Definition 10 case 2

  $unfold(Out(p,\ c))(0) =$ by Definition 10 case 1

  $|Out(p)| =$ by $\forall(t,\ p) \in OA.\ Out_{nt}(p) = W(t,\ p)\langle b\rangle$

  $|W((p,\ t))\langle b\rangle| =$ by Definition 9 case 7

  $W'((t,\ b),(p,\ c))$

- Otherwise $c \neq \mathbf{sum}$

  $(unfold(Out))_{nt}(p,\ c) =$ by definition of function nt

  $\sum_{x \in \mathbb{R}_{\geq 0}} unfold(Out)(p,\ c)(x) =$ by Definition 10 case 3

  $\sum_{x \in \mathbb{R}_{\geq 0}} Out(p)(x,\ c) =$ by definition of function nt

  $Out_{nt}(p)(c) =$ by $\forall(p,\ t) \in IA.\ Out_{nt}(p) = W(p,\ t)\langle b\rangle$

  $W((t,\ p))\langle b\rangle(c) =$ by Definition 9 case 6

  $W'((t,\ b),(p,\ c))$

Condition f) The number of tokens in In' have to match the number of tokens in Out' and have the same numeric weight of the transport weight. Assume that $W_T((p,\ t,\ p')) = (n,\ \tau,\ \tau') \Rightarrow (In_{nt}(p) = n(\tau)\langle b\rangle \wedge Out_{nt}(p') = n(\tau')\langle b\rangle)$. We want to show $W'((p,\ c),\ (t,\ b),\ (p',\ c')) = n \Rightarrow (In'_{nt}((p,\ c)) = n \wedge Out'_{nt}((p',\ c')) = n)$ for all $c \in C$. To show this we have two cases.

- Let $c = \mathbf{sum} \wedge x = 0$.
  Let $W'((p,\ c),(t,\ b),(p',\ c')) = n$. By Definition 9 case 8 we know that $W_T((p,\ t,\ p') = (n,\ \tau,\ \tau')$ which by our assumption we know that $In_{nt}(p) = n(\tau)\langle b\rangle$ and $Out_{nt}(p') = n(\tau')\langle b\rangle$.
  Now we show that $In'_{nt}(p,\ c) = n$.

  $In'_{nt}(p,\ c) =$ by definition of *unfold*

  $(unfold(In))_{nt}(p,\ c) =$ by definition of function nt

  $\sum_{x \in \mathbb{R}_{\geq 0}} unfold(In)(p,\ c)(x) =$ by Definition 10 case 2

$unfold(In(p,\ c))(0) =$ by Definition 10 case 1

$|In(p)| =$ by $In_{nt}(p) = n(\tau)\langle b\rangle$

n

Now we show that $Out'_{nt}(p',\ c') = n$.

$Out'_{nt}(p',\ c') =$ by definition of *unfold*

$(unfold(Out))_{nt}(p',\ c') =$ by definition of function *nt*

$\sum_{x\,\in\,\mathbb{R}_{\geq 0}} unfold(Out)(p',\ c')(x) =$ by Definition 10 case 2

$unfold(Out(p',\ c'))(0) =$ by Definition 10 case 1

$|Out(p')| =$ by $Out_{nt}(p') = n(\tau')\langle b\rangle$

n

- Otherwise let $c \neq$ **sum**

  Now we show that $In'_{nt}(p,\ c) = n$.

  $In'_{nt}(p,\ c) =$ by definition of *unfold*

  $(unfold(In))_{nt}(p,\ c) =$ by definition of function *nt*

  $\sum_{x\,\in\,\mathbb{R}_{\geq 0}} unfold(In)(p,\ c)(x) =$ by Definition 10 case 3

  $\sum_{x\,\in\,\mathbb{R}_{\geq 0}} In(p)(c) =$ by definition of function *nt*

  $In_{nt}(p)(c) =$ by $In_{nt}(p) = n(\tau)\langle b\rangle$

  n

  Now we show that $Out'_{nt}(p',\ c') = n$.

  $Out'_{nt}(p',\ c') =$ by definition of *unfold*

  $(unfold(Out))_{nt}(p',\ c') =$ by definition of function *nt*

  $\sum_{x\,\in\,\mathbb{R}_{\geq 0}} unfold(Out)(p',\ c')(x) =$ by Definition 10 case 3

  $\sum_{x\,\in\,\mathbb{R}_{\geq 0}} Out(p')(c') =$ by definition of function *nt*

  $Out_{nt}(p')(c') =$ by $Out_{nt}(p') = n(\tau')\langle b\rangle$

  n

Condition g) For all input arcs all tokens in In' have to satisfy the time guard on the arc.
Assume that $\forall(p,\ x,\ c) \in In.\ (p,\ t) \in IA \Rightarrow x \in TG((p,\ t)(c))$. We want to show $\forall((p,\ c),\ x) \in In'.\ ((p,\ c),\ (t,\ b)) \in IA' \Rightarrow x \in TG'((p,\ c),(t,\ b))$ for all x. To show this we have two cases.

- Let $c =$ **sum**
  By Definition 9 case 15 $TG'((p,\ c),(t,\ b)) = [0,\infty)$ therefore $x \in TG'((p,\ c),(t,\ b))$ for any $x \in \mathbb{R}_{\geq 0}$.

- Otherwise let $c \neq$ **sum**

Assume $x \in TG((p,\ t)(c))$.

By Definition 9 case 14 $TG'((p,\ c),(t,\ b)) = TG((p,\ t)(c))$ therefore $x \in TG'((p,\ c),(t,\ b))$ for any $x \in \mathbb{R}_{\geq 0}$.

Since the conditions **h-k** describes the age of the tokens, time guards, or invariants of places we have only shown **Condition g)**, since the age of tokens, the time guards, or the invariants will be overtaken from TACPN.

Condition l) For all inhibitor arcs from place p to transition t, the number of tokens in place $(p,\ \mathbf{sum})$ has to be less than the weight of the inhibitor arc.
Assume $\forall p \in {}^\bullet t.\ (p,\ t) \in INA \Rightarrow |M(p)| < W_I(p,\ t)$.
We want to show $\forall (p,\ c) \in {}^\bullet(t,\ b).\ ((p,\ \mathbf{sum}),\ (t,\ b)) \in INA' \Rightarrow |unfold(M(p,\ \mathbf{sum}))| < W_I'((p,\ \mathbf{sum}),(t,\ b))$.

Let $((p,\ \mathbf{sum}),(t,\ b)) \in INA'$. By Definition 9 case 11 there is $(p,\ t) \in INA$.

By assumption then $|M(p)| < W_I(p,\ t)$.

By Definition 9 case 9 then $W_I'((p,\ \mathbf{sum}),(t,\ b)) = W_I((p,\ t))\langle b\rangle(c)$.

By Definition 10 case 1 then $|unfold(M(p,\ \mathbf{sum}))| = |M(p)|$

By assumption then $|M(p)| < W_I(p,\ t)$

therefore $|unfold(M(p,\ \mathbf{sum}))| < W_I'((p,\ \mathbf{sum}),(t,\ b))$.

Now we have shown that transition $(t,\ b)$ is enabled in $unfold(M_1)$ by the markings In' and Out', but we still need to show that firing transition $(t,\ b)$ yields $unfold(M_2)$. To do this, we notice that the function $unfold$ preserves the multiset operations e.g. $unfold(M_1 \uplus M_2) = unfold(M_1) \uplus unfold(M_2)$ and $unfold(M_1 \setminus M_2) = unfold(M_1) \setminus unfold(M_2)$.

From the definition of transition firing we have that $M_2 = (M_1 \setminus In) \uplus Out$. Since $unfold$ preserves the multiset operations from Definition 10 we get that $unfold(M_2) = (unfold(M_1) \setminus In') \uplus Out'$. Therefore firing a transition (t, b) in N' will change the marking from $unfold(M_1)$ to $unfold(M_2)$ where:

$$unfold(M_2) = (unfold(M_1) \setminus In') \uplus Out' \tag{1}$$

Therefore we have shown that firing transition $(t,\ b)$ yields $unfold(M_2)$ by the markings In' and Out'.

A.12.2 $unfold(M_1) \xrightarrow{(t,b)} M'$ *implies* $M_1 \xrightarrow{t} M_2$ *under binding* b *where* $M' = unfold(M_2)$. By the definition of transition firing, we have that $unfold(M_1)$ enables $(t,\ b)$ by the markings In' and Out' that satisfy Definition 6. We will show that M_1 enables t by the markings In and Out. We will do this is in the same way as the previous statement.

Condition a) Assume that for all $(p,\ c) \in P'.\ In'((p,\ c),(x)) \leq unfold(M_1(p,\ c)(x))$. We want to show that for any $p \in P.\ In(p) \subseteq M_1(p)$ for all $(x,\ c)$.
Let $(x,\ c) \in In(p)$. We want to show $(x,\ c) \in M_1(p)$ for all $(x,\ c)$.

Let $(x,\ c) \in In(p)$ then $0 < In(p)(x,\ c)$

$0 < In(p)(x,\ c) =$ by Definition 10 case 3

$unfold(In)(p,\ c)(x) \leq$ by $In'((p,\ c),(x)) \leq unfold(M_1(p,\ c)(x))$

$unfold(M_1)(p,\ c)(x) =$ by Definition 10 case 3

$M_1(p)(x,\ c)$ then $(x,\ c) \in M_1(p)$

Condition b-l) Can be done in same manner as **Condition a)** and is therefore not shown.

A.12.3 $M_1 \xrightarrow{d} M_2$ *implies* $unfold(M_1) \xrightarrow{d} M'$ *such that* $M' \sim unfold(M_2)$. From the definition of time delay we have that M_1 can be delayed by a time duration $d \in \mathbb{R}_{\geq 0}$ if the following two conditions hold:

- The delayed tokens all satisfy the invariants of their respective places, i.e.
  $\forall (p, x, c) \in M_1.\ x + d \in I(p)(c)$
- The duration is 0, if any urgent transitions are enabled, i.e.
  $\forall t \in T_{urg}.\ M_1 \vdash t \Rightarrow d = 0$

Since the unfolding creates tokens with preserved ages at places with preserved invariants, the first condition is also preserved except for **sum**. From Definition 9, case 18, we know that the invariant for place **sum** is $[0,\ \infty]$.

When proving the first statement, it is shown that $unfold(M_1)$ only enables a transition if M_1 does. Therefore $unfold(M_1)$ could not enable any urgent transition while M_1 does not. If however M_1 does enable an urgent transition, then the duration must be 0. Thus, the second condition is also preserved. Now we have shown that if M_1 can delay by d, then $unfold(M_1)$ can delay by d, but we still need to show that this yields $unfold(M_2)$. This can be achieved by showing that the unfolding preserves delaying, i.e. delaying before unfolding yields the same result as delaying after unfolding. By Lemma A.1 we know that the markings $unfold(M_1)$, $unfoldM_2$ in the same unfolded net are strong timed bisimilar, thus proves that $M_1 \xrightarrow{d} M_2$ implies $unfold(M_1) \xrightarrow{d} M'$ such that $M' \sim unfold(M_2)$.

A.12.4 $unfold(M_1) \xrightarrow{d} M'$ *implies* $M_1 \xrightarrow{d} M_2$ *where* $M' \sim unfold(M_2)$. This can be proven in a similar fashion to the previous statement. ■