Application of Timed-Arc Colored Petri Net for Network Update Synthesis

Mark Glavind & Niels Christensen

Department of Computer Science, Aalborg University, Denmark

{**mglavi14, nchri13**}**@student.aau.dk**

Abstract

A modern network is a complex distributed system and provides many challenges. One challenge is route updates which happen often and need to be performed consistently. This is a complex issue where solutions are still being developed. However, there has not been much focus on the time used for deploying a new configuration. In order to automate provably correct and optimize network updates, we present Timed-arc colored Petri net (TACPN) with formal semantics, a model where it is possible to model asynchronous concurrent systems such as a network. We propose an algorithm to unfold TACPN into a Timed-arc Petri net, where the unfolded model can then be verified by known verification engines. To create the models visually we have created a full implementation of TACPN into the tool TAPAAL. We make an encoding of networks in TACPN, where the networks have to hold the security policy Waypoint Enforcement (WPE). We present results from experiments where we create real-world topologies into TACPN and optimize the worst case update time between two configurations, while WPE still holds.

I. Introduction

A modern network is a complex distributed system often containing hundreds of switches and routers, where each device can have different performance characteristics and behaviours. Modern communication networks also need to be able to quickly adjust, e.g., to changes in the load, network policies, due to the migration of virtual machines etc. How to support such changes has received much attention in the literature: some have focused on data plane consistency [1] [2], to ensure that updates are completed with minimum interruption, while others have focused on optimizing the installation of updates [3]. With the introduction of Software-Defined Networks, it has been made easier to verify the flows and updates in networks. Previous work have introduced mechanisms such as two-phase update [4] while others have introduced efficient synthesis for updates [5]. In order solve these challenges formal methods can be a solution.

Using formal methods to solve some of the problems in networks is not new, but is still growing in network community [6]. But using formal methods also had challenges. As mentioned a network often consists of hundreds of switches a routers where each device may have different packet processing time since its based on the hardware [7].
In a network there are sent many different packets that all got different sizes. This provides a new challenge, since the processing delay of a packet also is different based upon the size of a packet [7].

To deal with these challenges we propose to use Petri net, a mathematical modeling language, introduced by Carl Adam Petri in 1962 [8]. Petri net is a suitable model, since it can deal with asynchronous concurrent systems well. Through out the years Petri nets have gotten many extensions, but in particular the *Coloured Petri net* [9] and *Timed Petri net* [10] are extensions we can use in order to model a network with the challenges presented. With the introduction of colors, tokens and places can now have multiple values, thereby it is possible to model all the different packet types. With the notation of time, we are able to model the timing aspects of a network including the delays from the

different packets, router and switch processing time, timeouts etc. But so far there is no literature that combines both the colors and timed Petri net that suits our needs.

Therefore in this thesis we propose *Timed-arc Colored Petri Net* (TACPN) an extension to Petri net, combing the Coloured Petri nets and Timed-arc Petri nets [11]. By using TACPN we want to minimize the updating time between two configurations in a network. An example of a update scheme that is safe but takes a long time, is the two-phase commit [4]. Here the waiting time between the two phases is at least the maximum delay across all the devices. By using TACPN we can minimize the update time between configurations, by starting for each device, with the maximum delay, and then systematically reducing the delay while still verifying that some properties, such as a dangerous packet will never enter the network.

We have six main contributions and they are as follows: *(i)* The new model TACPN with formal semantics. *(ii)* We propose a algorithm of how a TACPN can be unfolded into a Timed-arc Petri net (TAPN) [12], and why TACPN and TAPN are strongly timed bisimilar. *(iii)* We propose a proof of correctness by how our TACPN model can be expressed in an TAPN. *(iv)* We have created an implementation of TACPN into the tool TAPAAL [13] where it is now possible to visually create TACPNs. For this implementation, we have also created a unfolding engine. The engine takes a TACPN as input and creates the corresponding TAPN which then can be used to verify the model, through the usage of other verification engines. *(v)* We have created an encoding of the network safety property Waypoint Enforcement (WPE) into TACPN. *(vi)* We have lastly conducted a case study where we use TACPN to reduce the update time between an initial and final configuration of a network. To get the configurations we first used Netsynth [5] a tool that creates a initial and finial configuration of a network, from a network topology.The network topologies used in this thesis are from the *Topology Zoo* dataset [14]. At the last experiment we generated our own paths in the topologies and randomly generated an update sequence between the two configurations.

Our experiments shows that when using configurations from Netsynth we have optimized the update time between them, up to 99 %. This is due to the fact that when NetSynth creates the configurations, it creates them for the optimal routing throughout the network, and therefore the configurations do not intersect i.e. does not use the same switches in the routes. Because of this, all the updates can be done concurrently.

We have therefore manually generated routes to get configurations that intersect as much as possible. These experiments show that when the routes do intersect, it will not always be able to reduce the update time to the minimum, which also shows there are cases where our method is applicable.

The rest of the thesis is structured as follows. In Section II we give an informal Colored Petri net, whereas we in Section III introduce Timed-arc Colored Petri Net, and give a informal description. Section IV we propose algorithm of how a Timed-arc Colored Petri Net can be unfolded into a Timed-arc Petri net. Section V is a description of how we have made an implementation of TACPN in TAPAAL and create the unfold engine. In Section VI we introduce the input needed to model a network into a TACPN, while also showing how we modelled it. In Section VII we show the experiments we have conducted with TACPN. Lastly we have a conclusion and future work.

II. Background: Colored Petri net

In this section we will describe the colored Petri net (CPN) which was first introduced by K. Jensen in 1981 [9]. The definitions presented in this section are based on those presented in [15].

Before introducing definitions of CPN we show a small network. This network will be used as a running example throughout the thesis.

To introduce a small network consider Figure 1. Packets get sent from the internet to the first switch called **Enter Network**. Packets always comes in a stack of four. The **Enter Network** switch then forwards them to the **Firewall**. Based upon what type of the packet, the **Firewall** will either forward it to an **Internal Network**, forward it to an **External Network**, or if it detects it as a dangerous packet, it will drop it, and remove it from the network.

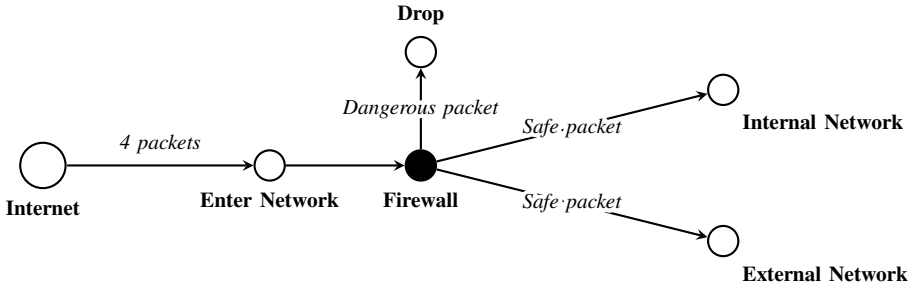


Fig. 1: Updating of a network from the solid to the dashed line that may break WPE

A. Multiset

Before we present the colored Petri net, we will first describe the preliminary definitions.

Definition 1: **(Multisets)**
A *finite* multiset is a collection over a set S of elements where a *finite* number of those elements occur a *finite* number of times in the multiset. Let b be a multiset over the set S then $b \in S \to \mathbb{N}^0$. If $s \in S$ then $b(s)$ is the number of occurrences of s in 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 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:

Definition 2: **(Multiset operations)**
S is a set, $b, b_1, b_2 \in \mathcal{B}(S)$, $s \in S$, 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)
- $n * b = \sum_{s \in S} (n * b(s))'(s)$ (scalar-multiplication)
- $b_1 \subseteq b_2$ iff $\forall s \in S : b_1(s) \leq b_2(s)$
- $b_1 = b_2$ iff $b_1 \subseteq b_2 \wedge b_2 \subseteq b_1$
- $|b| = \sum_{s \in S} b(s)$
- $b_1 \setminus b_2 = \sum_{s \in S} (\textit{non-minus}(b_1(s), b_2(s)))'(s)$

B. The colored Petri net

To ease understanding of CPN we have modelled the small network from Figure 1 as a CPN shown in Figure 2. First packets are entering the network from the internet, then they are sent to the first switch. From the first switch, the packet is send to a firewall. The network is accepting two types of packet otherwise, if the firewall detects that its a dangerous packet, it will be blocked. The firewall has a queue of size 10. If the packet has the correct type, the switch will send it to either an internal or external network.

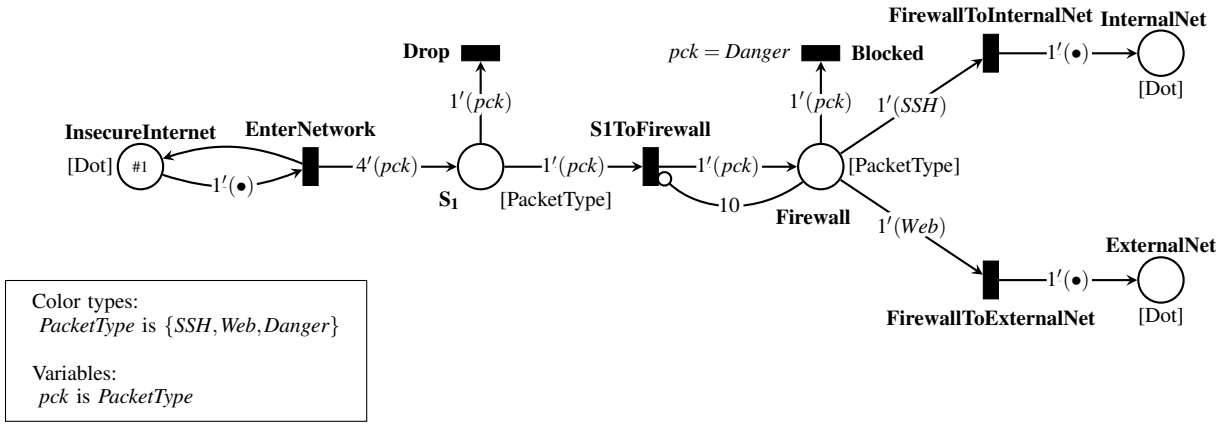


Fig. 2: Colored Petri net example

Every token is valued with a **color**. Each of these colors belong to a **color type** e.g. in the box of Figure 2 we have the color type *PacketType* that covers the colors *SSH*, *Web*, and *Danger*.
Each place has a color type, that indicates the possible colors of the tokens at that place, e.g. the place **Firewall** has the color type *Pck* and can therefore only have tokens valued with *SSH*, *Web*, or *Danger*. The reserved color type *Dot* only covers one color and is used to represent uncolored tokens.
The circles are **places**, the black rectangles are **transitions** and the arrows between the two are **arcs**. The state of the net is given as a **marking** that assigns a number of **tokens** to each place, e.g. in Figure 2, the number of packets that will enter the network is represented by the number of tokens at the place $\mathbf{S_1}$.

A transition can be fired, changing the marking of the net according to the expressions of its connected arcs. If **EnterNetwork** is fired, its connected **output arc**, with the expression $4'(pck)$, will add four tokens to $\mathbf{S_1}$. This expression has **variables**, as can be seen on the box on Figure 2 the variable *Pck* is of color type *PacketType*. Therefore when firing the transition, the four tokens will get one binding with a color from the *PacketType* color type. Note that the number of tokens in a place is always non-negative, and therefore **S1ToFirewall** will only be enabled, if $\mathbf{S_1}$ has enough tokens to satisfy its input arc.

In addition to regular arcs, we also have **inhibitor arcs** denoted with a circle instead of the usual arrow head e.g. the one connected to **S1ToFirewall**. Inhibitor arcs do not move any tokens, but will only allow their connected transition to fire if the number of tokens at the connected place is less than its expression, e.g. **S1ToFirewall** can only fire if there are less than 10 tokens at **Firewall**. This is to represent that the firewall got a queue of size 10.

The addition of colors allows us to create more expressive arc expressions, e. g. the arc, connecting the place **Firewall** with the transition **FirewallToExternalNet**, has the expression $1'(Web)$, which represents that the firewall will send web to an external net. For the transition **FirewallToExternalNet** to be enabled, a token in **Firewall** need to have the binding *Web*.

Finally transitions can also have **color guards** that restrict the bindings of the relevant variables, e.g. the color guard, $pck = Danger$, seen on the transition **Blocked** ensures that the firewall will block only dangerous traffic.

C. 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, e.i. $\forall \sigma_1, \sigma_2 \in \Sigma.\ \sigma_1 \cap \sigma_2 = \emptyset$. 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 contaioning 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 $S = (e_0, e_1, ..., e_n)$. 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:

$$Succ(e_i) = \begin{cases} e_0 & \text{if } i = n \\ e_{(i+1)} & \text{otherwise} \end{cases} \qquad Pred(e_i) = \begin{cases} e_n & \text{if } i = 0 \\ e_{(i-1)} & \text{otherwise} \end{cases}$$

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 $Pred(e_0) = e_n$.

D. 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 $\langle v_0 = c_0, v_1 = c_1, ..., v_k = c_k \rangle$, if $\forall i \in [0,k]\ b(v_i) = c_i$. The set of all bindings is denoted $\mathbb{B}$.

E. Color Expressions

The color expressions $\tau \in \mathbb{T}$ 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 $var \in Var$.

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:

$$Type_{\mathbb{T}}((\tau_1, \tau_2, \ldots, \tau_n)) = Type_{\mathbb{T}}(\tau_1) \times Type_{\mathbb{T}}(\tau_2) \times \cdots \times Type_{\mathbb{T}}(\tau_n)$$

[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.

$$Type_{\mathbb{T}}(\bullet) = \{\bullet\}$$
$$Type_{\mathbb{T}}(c) = Type_{\mathbb{C}}(c)$$
$$Type_{\mathbb{T}}(var) = Type_{Var}(var)$$
$$Type_{\mathbb{T}}(\mu{+}{+}) = Type_{\mathbb{T}}(\mu) = Type_{\mathbb{T}}(\mu{-}{-})$$

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

Definition 3: **(Color semantics)**
$[\![(\bullet)\langle b\rangle]\!] = \bullet$ (neutral-color)
$[\![(c)\langle b\rangle]\!] = c$ (constant)
$[\![(var)\langle \ldots, var = c, \ldots\rangle]\!] = c$ (variable)
$[\![(\mu{+}{+})\langle b\rangle]\!] = \mathsf{Succ}([\![(\mu)\langle b\rangle]\!])$ (successor)
$[\![(\mu{-}{-})\langle b\rangle]\!] = \mathsf{Pred}([\![(\mu)\langle b\rangle]\!])$ (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 $[\![(x,y)\langle x = 1, y = 6\rangle]\!] = (1,7)$.

F. Guard Expressions

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

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

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 $[\![]\!] : \Gamma \times \mathbb{B} \to \{true, false\}$

Definition 4: **(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)
$[\![(\gamma_1 \to \gamma_2)\langle b\rangle]\!] = [\![(\gamma_1)\langle b\rangle]\!] \to [\![(\gamma_2)\langle b\rangle]\!]$ (implication)
$[\![(\gamma_1 \leftrightarrow \gamma_2)\langle b\rangle]\!] = [\![(\gamma_1)\langle b\rangle]\!] \leftrightarrow [\![(\gamma_2)\langle b\rangle]\!]$ (biconditional)
$[\![(\gamma_1 \text{ xor } \gamma_2)\langle b\rangle]\!] = [\![(\gamma_1)\langle b\rangle]\!] \text{ xor } [\![(\gamma_2)\langle b\rangle]\!]$ (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 $[\![\gamma\langle a = 1,\ b = 2,\ c = 4\rangle]\!] = true$.

G. 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.

1) Types: Arc expressions, like color expressions, can also be associated a type, given by the function $Type_\Delta : \Delta \to \Sigma$. 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:

$$Type_\Delta(n'(\tau)) = Type_\mathbb{T}(\tau)$$

$$Type_\Delta(n'(\sigma.all)) = \sigma$$

$$Type_\Delta(\delta_1 \uplus \delta_2) \text{ if } Type_\Delta(\delta_1) = Type_\Delta(\delta_2)$$

$$Type_\Delta(\delta_1 \setminus \delta_2) \text{ if } Type_\Delta(\delta_1) = Type_\Delta(\delta_2)$$

$$Type_\Delta(n * \delta) = Type_\Delta(\delta)$$

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

Definition 5: **(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 $[\![\delta\langle x = 3 \rangle]\!] = 2'(3) + 1'(2)$.

III. Timed-Arc Colored Petri Nets

In this section we define timed-arc colored Petri nets (TACPN). The TACPN presented in this paper has also been inspired by the timed-arc Petri net (TAPN) in [16] and the colored Petri net (CPN) in [15]. With the notion of colors, we are able to model different packet types of a network into a Petri net, but we do not have a notion of time. In a network, routers may use different time to process a packet, calculate next-hop, some packet types can have a longer delay, or some links may be slower than others etc. In order to be able to model this, we need to combine both colors and time, and therefore define TACPN.

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

Definition 6: **(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:

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

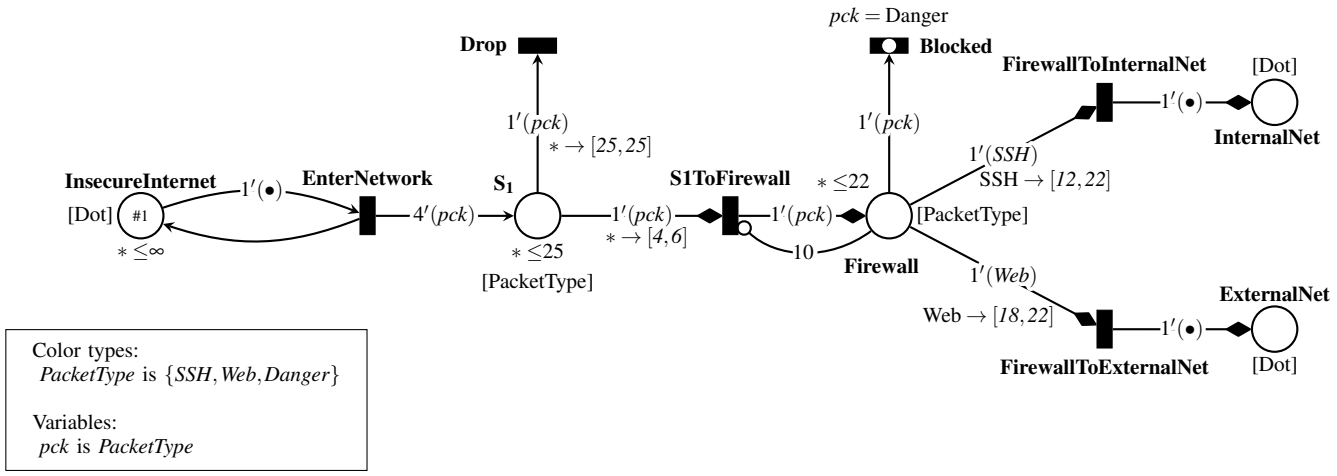


Fig. 3: Small network example modeled as a timed-arc colored Petri net

A. TACPN example

We will now give an informal description of TACPNs. In Section II-B, we presented Figure 2, a small network modeled as a CPN. This model failed to capture some aspects of a network, such as a packet could stay at a switch infinite, or when a packet reached $\mathbf{S_1}$ both the **Drop** and **S1ToFirewall** transitions are enabled, so packets may never reach the firewall. These problems are all resolved when we change it into a TACPN by adding timing to the model, as seen in Figure 3.

In a TACPN, all tokens also have an **age** which allows us to control how long tokens spend at places. Instead of just firing transitions, we can also delay, incrementing the age of all tokens in the net by some duration.

To ensure that we do not just delay forever, places can have a **time invariant** that limits the ages of the tokens at the place, e.g. $\mathbf{S_1}$ has the invariant ≤ 25 enforcing that packet will not be at the $\mathbf{S_1}$ more than 25 time unites.

Arcs now also have a **time guard** that restricts the ages of the tokens it can remove, e.g. the input arc connected to **Drop** has a guard that ensures that only tokens of at least and at most age 25 gets removed. Together with the invariant on $\mathbf{S_1}$, this makes a common pattern where a transition is fired if and only if a token reaches a certain age. In this case the packet will time out and therefore leave the network after waiting exactly 25 time units.

Usually, all new tokens added will have an age of 0, but the age of the tokens can be preserved with **transport arcs**, denoted with diamonds instead of the usual arrowheads, e.g. firing the transition **S1ToFirewall** will not change the time of how long a packet has been in the network. Note that transport arcs always come in pairs, one for input and one for output.

Finally we have **urgent transitions**, denoted with a small circle on the transition. These are urgent in the sense that we can not delay when they are enabled, i.e. **Blocked** is urgent, and therefore if the binding of the packet is *Danger* the firewall will block it immediately. Note that urgent transitions can not have any connected arcs with restrictive time guards.

B. TACPN Definition

We will now formally define TACPNs.

Definition 7: **(Timed-Arc Colored Petri Net)**
A Timed-Arc Colored Petri Net (TACPN)
is a 15-tuple $TACPN = (P, T, T_{urg}, IA, OA, INA, TA, \Sigma, C, CG, W, W_I, W_T, TG, I)$ 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) $C : P \to \Sigma$ *is a* **color function**,
10) $CG : T \to \Gamma$ *is a* **color guard**.
11) $W : IA \cup OA \to \Delta$ *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 *. An example of this notation can be see on Figure 3, where $\mathbf{S_1}$ got the invariant $* \leq 25$, denoting that *SSH*, *Web*, and *Danger* all got the invariant ≤ 25.

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

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

C. 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:

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

D. Enableness & Semantics

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

Definition 8: **(Enableness)**
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.

$$\forall p \in {}^\bullet t.\ (p,\ t) \in INA \Rightarrow |M(p)| < W_I(p,\ t) \tag{1}$$

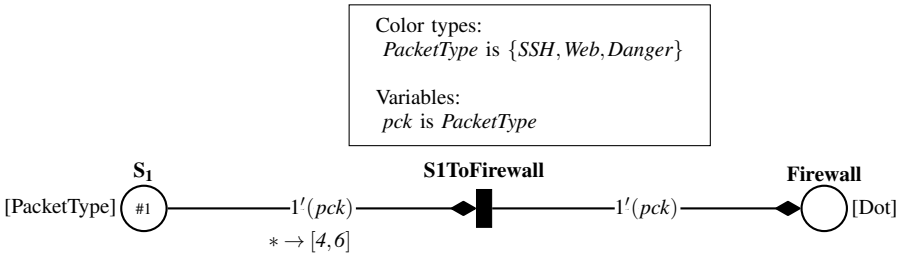


Fig. 4: Enabledness example of Figure 3

To give an example of a marked TACPN we have isolated the part where the $\mathbf{S_1}$ sends packets to **Firewall** of Figure 3 as shown on Figure 4, 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 9: **Timed Transition System**
The semantics of a marked TACPN (N, M_0) is defined as a timed transition system $[\![N]\!]_{sem} = (\mathcal{M}(N), M_0, \rightarrow)$, where $\rightarrow : \mathcal{M}(N) \times T \cup \mathbb{R}^{\geq 0} \times \mathcal{M}(N)$ 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 $(M_1, t, M_2) \in \rightarrow$, 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 $(M_1, d, M_2) \in \rightarrow$, denoted by $M_1 \xrightarrow{d} M_2$.

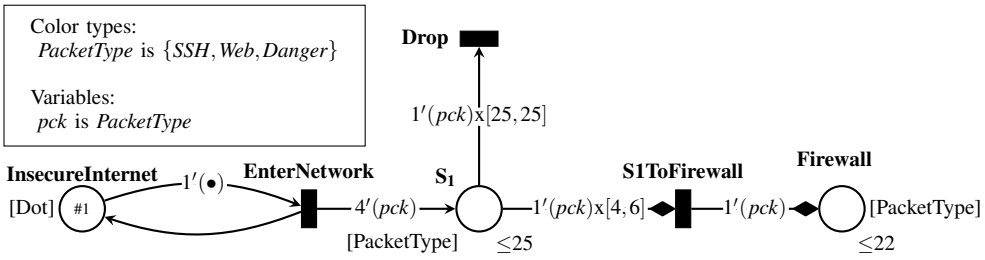


Fig. 5: Transition firing example of Figure 3

To provide an example of transition firing, we have isolated the start of the network form Figure 3. This is shown on Figure 5.

A fragment of timed transition system of the net in Figure 5 is presented in Figure 6. **EnterNetwork** is abbreviated **EN** and **InsecureInternet** is **II**.

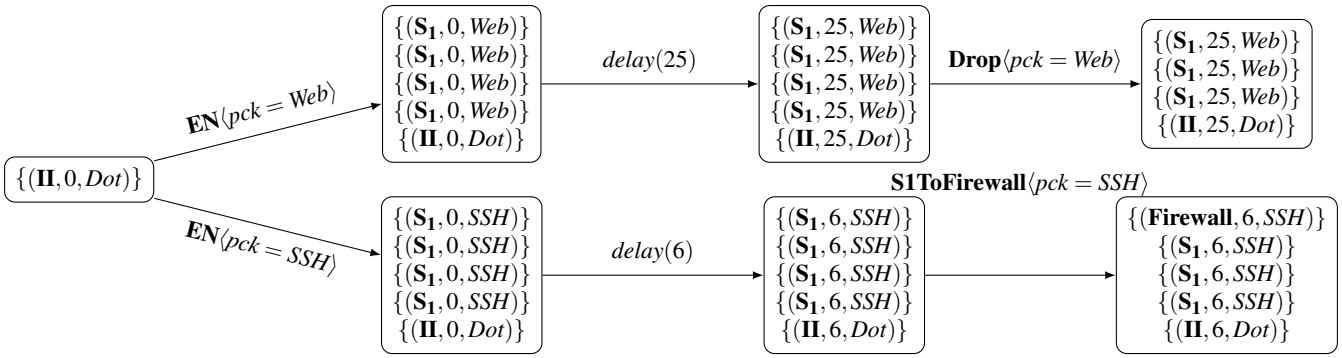


Fig. 6: Semantics of the timed-arc colored Petri net on Figure 5. EnterNetwork is abbreviated *EN* and InsecureInternet is *II*

Transition **EnterNetwork** can be fired with three bindings, either $\langle pck = SSH \rangle$, $\langle pck = Web \rangle$, or $\langle pck = Danger \rangle$ creating the markings $\{(\mathbf{S_1}, 0, SSH)\}$, $\{(\mathbf{S_1}, 0, Web)\}$, and $\{(\mathbf{S_1}, 0, Danger)\}$. In this example we will only use two bindings, one with the binding *SSH* and the second with binding *Web*.

Neither of the transitions **Drop** or **SiToFirewall** are enabled yet, because of the time guards on the arcs. Because of the construction of the net, **EnterNetwork** can keep firing, creating more and more tokens in $\mathbf{S_1}$, but this is not interesting in order to explain for the semantics.

The invariant of $\mathbf{S_1}$ allows us to delay up to 25 time units, but the enabledness of any transition connected to $\mathbf{S_1}$ will not change before 4 time unites. To show different scenarios, we delay with 6 time units where *SSH* is the binding, and 25 where *Web* is the binding.

When performing the delay of 6 time units transition **S1ToFirewall** becomes enabled, and can be fired changing the marking for one token to $\{(\mathbf{Firewall}, 6, SSH)\}$ while the marking of the dot will change to $\{(\mathbf{II}, 6, Dot)\}$ by the delay. Since the transition is connected with transport arcs, the time of the tokens does not change.

In the other scenario we delay with 25 time units, enabling the transition **Drop** that when fired removes one token from the net, changing the marking to containing three tokens with binding $\{(\mathbf{S_1}, 25, Web)\}$ and one with binding $\{(\mathbf{II}, 25, Dot)\}$.

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.

IV. Unfolding of a Timed-Arc Colored Petri Net

In section III, we defined the TACPN, a formalism that extends TAPNs with colors. However, 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 bisumulation, and lastly a proof of this theorem.

A. Timed-Arc Petri Net

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

Definition 10: **(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: $TAPN = (P, T, T_{urg}, IA, OA, INA, TA, W, W_I, TG, I)$.

B. Unfolding Timed-Arc Colored Petri Net to Timed-Arc Petri Net

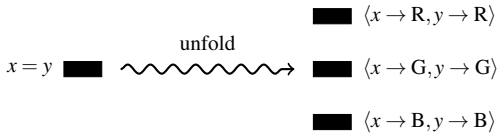
In order to unfold a TACPN into TAPN that perserves the behavior of the original net, we must be able to express the features of colors, without actually having them. This subsection will explain how this is achieved.

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.

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.



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.

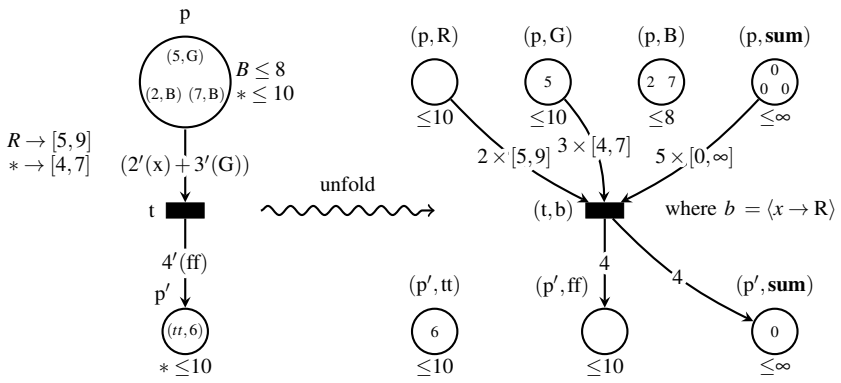


Fig. 7: Example of unfolding of a timed-arc colored Petri net

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.

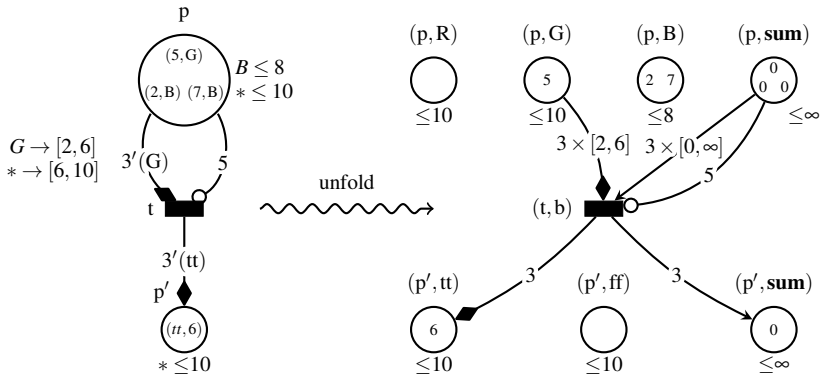


Fig. 8: Timed-arc colored Petri net unfolding example with transport and inhibitor arc

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

Definition 11: **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) $T'_{urg} = \{(t,b) \mid t \in T_{urg} \wedge (t,b) \in T'\}$
4) $W'((p,c),(t,b)) = W((p,t))\langle b \rangle(c)$
5) $W'((p,\mathbf{sum}),(t,b)) = \begin{cases} |W((p,t))\langle b \rangle| & \text{if } (p,t) \in IA \\ n \text{ where } W_T((p,t,p')) = (n,\tau,\tau') & \text{if } (p,t,p') \in TA \end{cases}$
6) $W'((t,b),(p,c)) = W((t,p))\langle b \rangle(c)$
7) $W'((t,b),(p,\mathbf{sum})) = \begin{cases} |W((t,p))\langle b \rangle| & \text{if } (t,p) \in OA \\ n \text{ where } W_T((p',t,p)) = (n,\tau,\tau') & \text{if } (p',t,p) \in TA \end{cases}$
8) $W'((p,c),(t,b),(p',c')) = n$ 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 11 we use Figure 7 and 8 to explain. P' on Figure 7 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 7 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 7 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.

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

Definition 12: **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 $\mathit{unfold} : \mathcal{M}(N) \to \mathcal{M}(N')$ defined as:

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

The initial marking M_0 of the unfolded net is defined as $\mathit{unfold}(M_0)$ 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 $(p,\ \textbf{sum})$ we as mentioned creates a token for each token at the place in the original net.

C. 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 [17].

Definition 13: **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 1 that if two markings only differ in the ages of tokens in place **sum** then they are up to strong timed bisimular.

Lemma 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 $c \neq \textbf{sum}$ and $|M(p, \textbf{sum})| = |M'(p, \textbf{sum})|$ then $M \sim M'$.

Theorem 1: Let M be a marking in a TACPN N. The corresponding unfolded marking $\mathit{unfold}(M)$ is strongly timed bisimilar with M, i.e. $\mathit{unfold}(M) \sim M$.

D. Equivalence proof

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

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

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 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.

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 8. 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 12 case 1

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

  $|M_1(p)| =$ by Definition 12 case 1

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

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

  $unfold(In)(p,\,c)(x) =$ by Definition 12 case 2

  $0 =$ by Definition 12 case 2

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

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

  $unfold(In)(p,\,c)(x) =$ by Definition 12 case 3

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

  $M_1(p)(x,\,c) =$ by Definition 12 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 $c = \mathbf{sum} \wedge x = 0$

  $unfold(In)(p,\, c)(x) =$ by Definition 12 case 1

  $|In(p)| =$ by Definition 11 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 12 case 2

  0

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

  $unfold(In)(p,\, c)(x) =$ by Definition 12 case 3

  $In(p)(x,\, c) =$ by Definition 11 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 12 case 1

  $|Out(p)| =$ by Definition 11 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 12 case 2

  0

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

  $unfold(Out)(p,\, c)(x) =$ by Definition 12 case 3

  $Out(p)(x,\, c) =$ by Definition 11 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 12 case 2

$unfold(In(p, c))(0) =$ by Definition 12 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 11 case 5

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

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

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

  $\sum_{x \in \mathbb{R}^{\geq 0}} unfold(In)(p, c)(x) =$ by Definition 12 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 11 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 $c = \textbf{sum} \wedge x = 0$

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

  $\sum_{x \in \mathbb{R}^{\geq 0}} unfold(Out)(p, c)(x) =$ by Definition 12 case 2

  $unfold(Out(p, c))(0) =$ by Definition 12 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 11 case 7

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

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

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

  $\sum_{x \in \mathbb{R}^{\geq 0}} unfold(Out)(p, c)(x) =$ by Definition 12 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 11 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 11 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 12 case 2

  $unfold(In(p,\, c))(0) =$ by Definition 12 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 12 case 2

  $unfold(Out(p',\, c'))(0) =$ by Definition 12 case 1

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

  n

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

  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*

$$\sum_{x \in \mathbb{R}^{\geq 0}} unfold(In)(p, c)(x) = \quad \text{by Definition 12 case 3}$$

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

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

$$n$$

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

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

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

$$\sum_{x \in \mathbb{R}^{\geq 0}} unfold(Out)(p', c')(x) = \quad \text{by Definition 12 case 3}$$

$$\sum_{x \in \mathbb{R}^{\geq 0}} Out(p')(c') = \quad \text{by definition of function } nt$$

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

$$n$$

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 $c = \textbf{sum}$
  By Definition 11 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 $c \neq \textbf{sum}$
  Assume $x \in TG((p, t)(c))$.

  By Definition 11 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 $(p, \textbf{sum})$ 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 $\forall (p, c) \in {}^\bullet(t, b).\ ((p, \textbf{sum}), (t, b)) \in INA' \Rightarrow |unfold(M(p, \textbf{sum}))| < W'_I((p, \textbf{sum}), (t, b))$.

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

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

By Definition 11 case 9 then $W'_I((p, \mathbf{sum}), (t, b)) = W_I((p, t))\langle b \rangle(c)$.

By Definition 12 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 12 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'.

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 8. 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 12 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 12 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.

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 the Definition 11 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 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 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 $M' \sim unfold(M_2)$.

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.

■

V. Implementation

In this section we discuss the implementation which is based upon the theory presented previously. The implementation is a further development of the tool TAPAAL [13] in order to support TACPN. TAPAAL makes it possible to create Petri nets in a user-friendly way. A official beta release will be available at *http://www.tapaal.net/download/* while the development branch with the exam version of TACPN can be found at *https://code.launchpad.net/ cpn-gui/tapaal/TACPN*. Figure 9 shows the four primary models supported by TAPAAL. When creating a new net you have to choose the model type, but it is still interchangeable as with the unfolding which converts a TACPN into a TAPN/PN. TACPN are the new additions made to TAPAAL and in this section we will highlight some of the implementation.

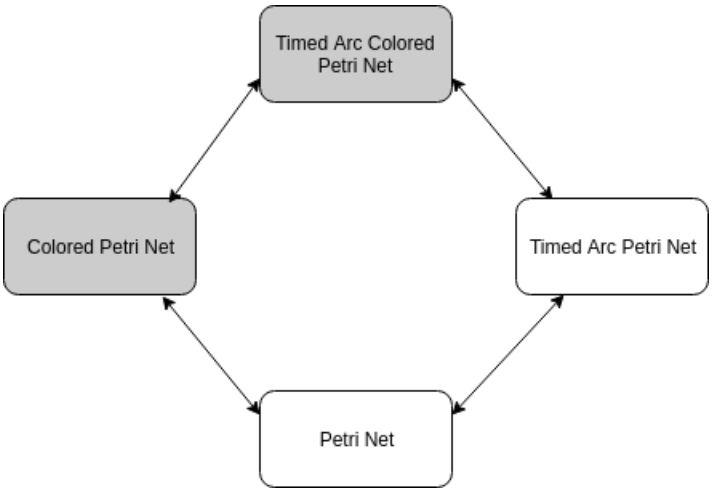


Fig. 9: Architecture of supported Petri Nets in TAPAAL

A. New features in TAPAAL

Since TAPAAL has never support the TACPN model before a lot of new feature have been developed. Work on this started on our last semester where most of the CPN only features were implemented. This semester those features have been extended to support time with some additional new ones. An overview

of the implementation in TAPAAL can be seen at list V-A followed by a detailed examination of the most important items.

- Import and export of TACPN in TAPN format
- Export TACPN or CPN models in TikZ format
- GUI for creating color type and variables
- GUI for editing places with colors and time
- GUI for editing transitions with colors, time and guard expressions
- GUI for editing arcs with colors, time and arc expressions
- GUI for editing color expressions
- Extending transport and inhibitor arcs with colors
- Extending shared places and the components with TACPN features
- Typechecking analysis

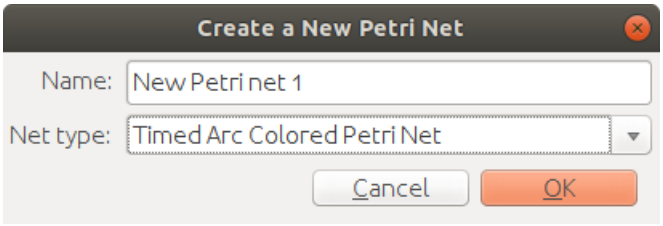


Fig. 10: When creating a new net the user is asked to choose whether it should be TACPN or TAPN

Figure 10 shows the first window the user meet, whenever they want to create a new net. Only if the Timed Arc Colored Petri Net option is chosen, will they have access to the features of TACPN. This has primarily two functions. The first is to reduce information clutter from the user. If they only want to create a TAPN, it would be confusing having to look at the GUI elements for TACPN. The second reason is that the information for which net type is being used, helps throughout the system with how to process the model/data. If the user later regrets their choice, they can always change, e.g. going from TACPN to TAPN by unfolding the net.

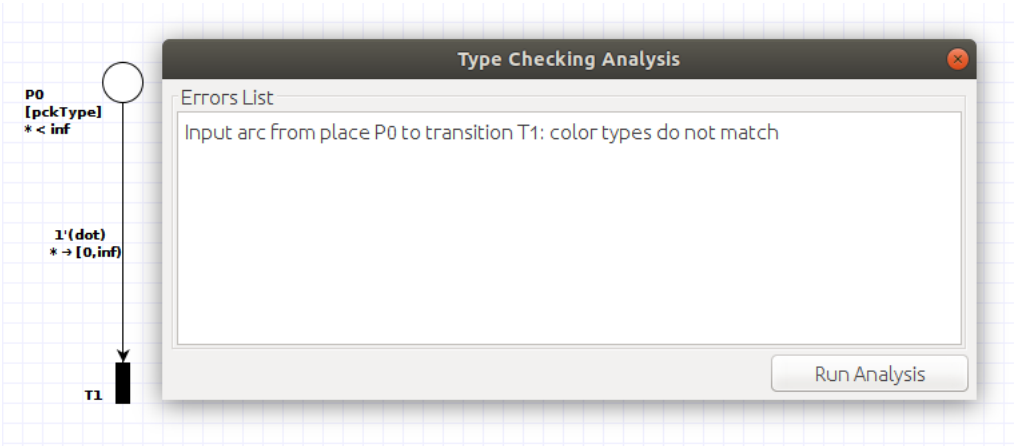


Fig. 11: Example of a small net with a type error shown in the type checking analysis panel

Figure 11 shows the typechecking analysis panel next to a small net with a type error. In this case we have a place with the color type *pckType* with an input arc taking a *dot*. In TACPN this is not allowed as the color types have to match. This mistake is caught by the *Type Checking analysis* as can be seen

in the text list. This is important as the tool almost never stops the user from creating something that is not verifiable. This is because the user could be in the process of editing his net, e.g. in this case changing the color type of either the arc expression or the place. At the same time the user should be able to save and load his progress even though the net is not correct. However, this can confuse the user as to what they are allowed to do and introduces the problem of them trying to verify a model that is semantically wrong. In order to combat this, a type checking tool has been implemented. The user can choose to run the type checking manually, but it will still be done automatically before it tries to verify a query.

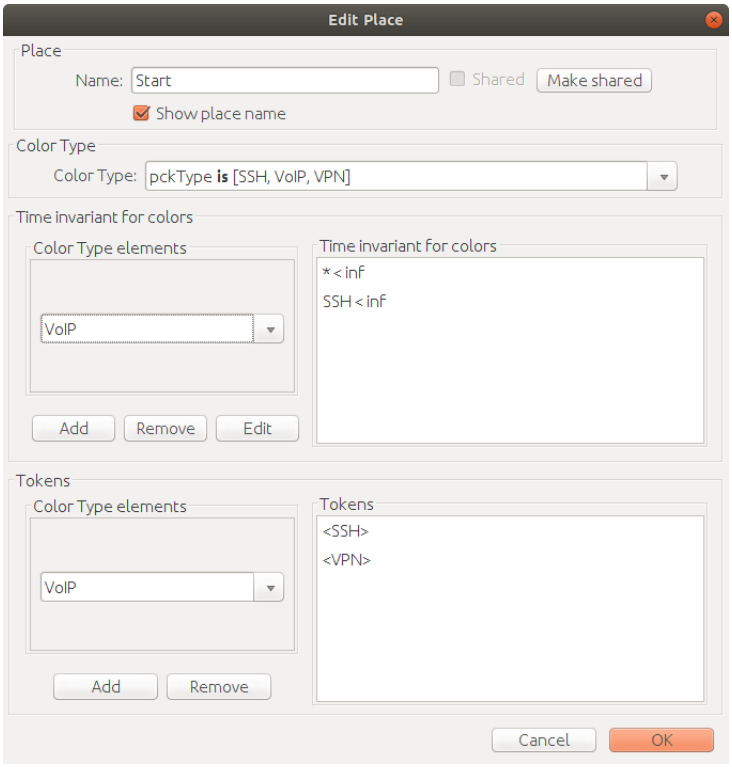


Fig. 12: Panel for editing colored places

Figure 12 shows the panel for editing colored places. Everything from below the first panel *Place* are new additions only found in TACPN. the first panel *Color Type* chooses the color type of the place. As default this is the *dot* color type. In this case we are using a custom cyclic enumeration color type called *pckType* with three colors *SSH, VoIP* and *VPN*. In the next panel *Time invariant for colors* specific time invariants for each color in the chosen color type can be set.There will always be a color invariant for * in the list as default. If a color has not been chosen and added with the combobox on the left, they will use the default time invariant *. When either double clicking on a color invariant in the list or using the edit button, a panel for editing the selected invariant will open.
A similar system applies to the last panel *Tokens*. Here the users can choose the color from the combobox and add it to the list of active tokens in the given place. The user can add as many as they want of all the colors in the color type. In the *Color Type elements* sub panel there is room for more than one combobox. This is because if the user had chosen a product type, they would have to choose a color for each color type and the panel would then create a combobox for each color type.

Figure 13 shows the panel for editing colored arcs. The top part works in the same way as it did with invariants in the edit place panel. However, instead of invariants we use intervals for all arcs. If the user had opened an output arc this panel would not be shown, as output arcs does not have any time. The bottom part of the panel shows the *Arc Expressions* panel. In this panel the user can create

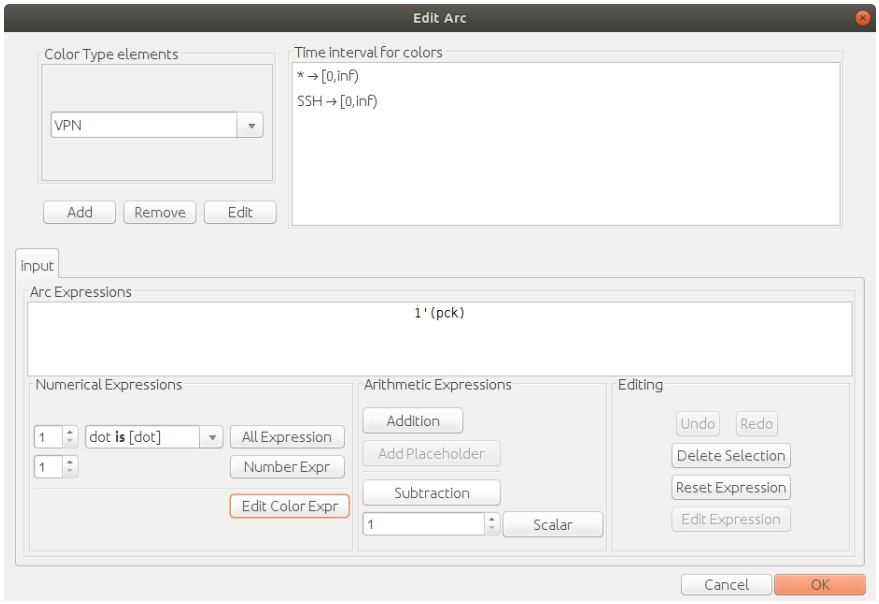


Fig. 13: Panel for editing input and output arcs

an arc expression by using the buttons representing the different elements an arc expression can consist of. In this case we have a number expression with the value 1, specifying that we want 1 of the color expression inside the parentheses. In this case the color expression is a variable expression created with the color expression panel, which can be seen in Figure 15. The variable expression has the variable *pck*, which has the color type *pckType* presented earlier.
The textfield showing the arc expressions can be interacted with, such that the user can select specific elements in the expression. If the user were to press the surround parts of *pck* they would select the number expression. In this case they would be able to change the value or replace it with a new expression such as the all expression. If they were to select the *pck* part only the *Edit Color Expr* button would be available, as a color expression has been selected. This makes it easy for the user to edit their expressions, and helps them understand which actions are legal depending on their selection.

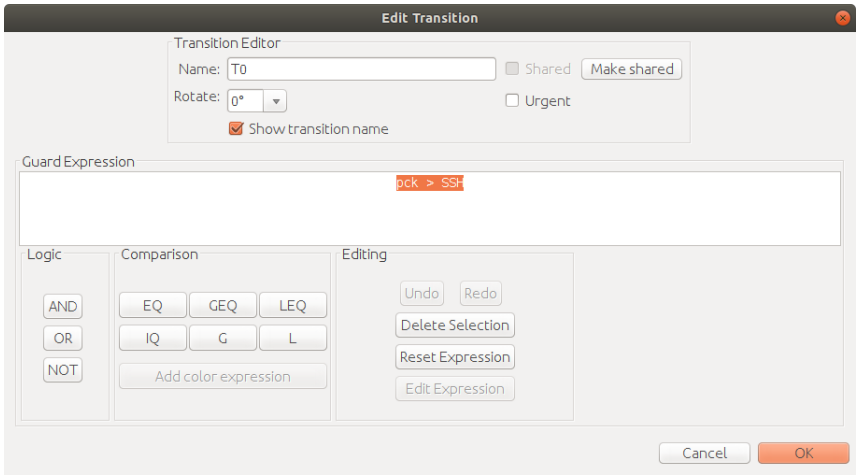


Fig. 14: Panel for editing transitions

Figure 14 shows the panel for editing a colored transition. The new addition is the *Guard Expression* panel which works in the same way as panel for editing arc expressions. However, here it is guard expressions as they are used for transitions. In the example we are using a greater than guard expression,

which in this case means the color binding of the variable *pck* must be greater than SSH. This means *pck* can either have the value *VPN* or *VoIP* as they are both after *SSH* in the enumeration order *SSH, VPN, VoIP*. In the expression field the whole expression is orange, as the greater than expression has been selected, showing it can either be replaced with another comparison expression or chained with one of the logic options.

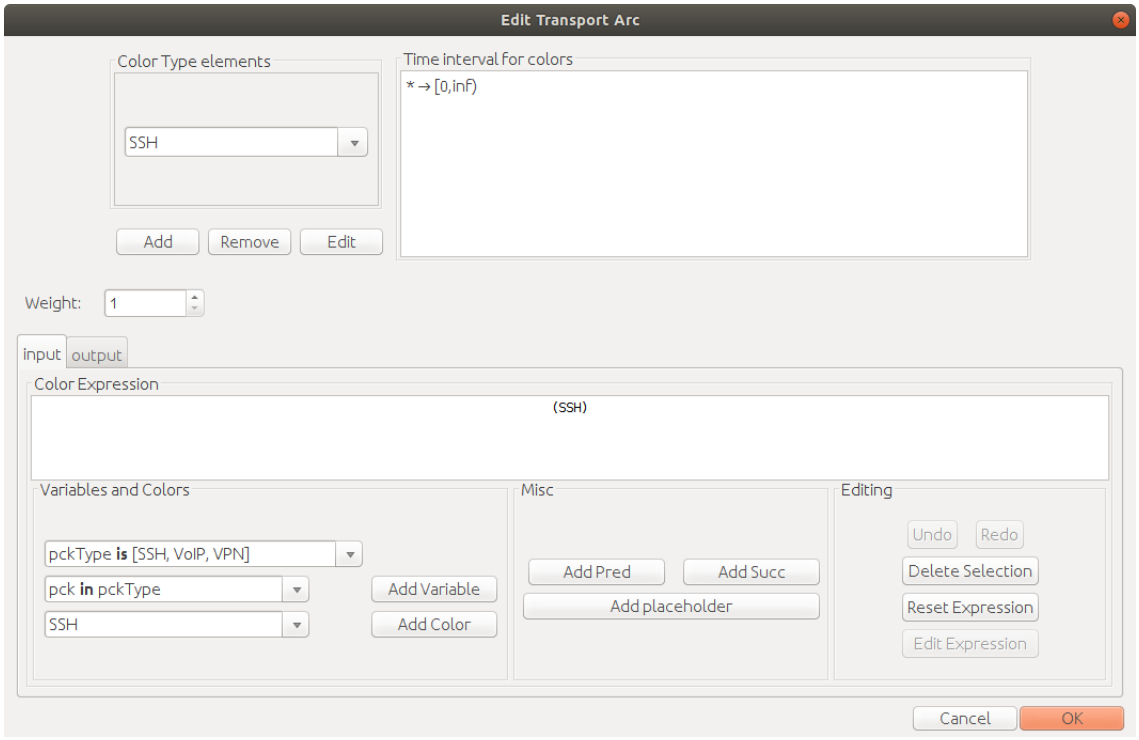


Fig. 15: Panel for editing a transport arc

Figure 15 shows the panel editing transport arcs. This is mostly the same as the panel presented for input arcs. The big difference is, instead of having the arc expression panel at the bottom, transport arc shows the color expression panel. This is because a transport arc can only have one source and one destination. We can therefore only use number expressions, and this is represented with the weight value above the *Color Expression* panel. This weight is the same for both the input and output expression, which ensures we follow the rules for transport arcs. Both the input and output expression can also be edited in the same panel for transport arcs. This is accessible with the tabs *input* and *output*.

B. Unfolding and Verification of TACPN

In TAPAAL all verification is done by separate engines. Each support one or multiple versions of Petri nets, and each fill their own niche depending on how you want the verification to be done. In our case we have to unfold from TACPN before verification is possible with either *verifydtapn* or *verifytapn* as neither supports colors. *Verifydtapn* is a verification engine that uses discrete time and support EF, AG, EG and AF queries. It also supports weighted arcs and closed intervals. The branch with *unfoldTACPN* can be found at *https://code.launchpad.net/ verifypn-cpn/verifypn/unfoldTACPN*. Meanwhile *verifytapn* uses continous time and support open intervals. The process in order to verify a TACPN model can be seen at Figure 16. The first step is exporting the model and queries to the engine *unfoldTACPN*, which is a modification of the engine *verifypn* [18]. *Verifypn* can unfold and verify CPN models. The unfolding has been extended, such that is supports timed features, transport arcs and inhibitor arcs. The

unfolded model and query is then exported and TAPAAL now calls either *verifydTAPN* or *verifyTAPN* in order to get a verification result back.

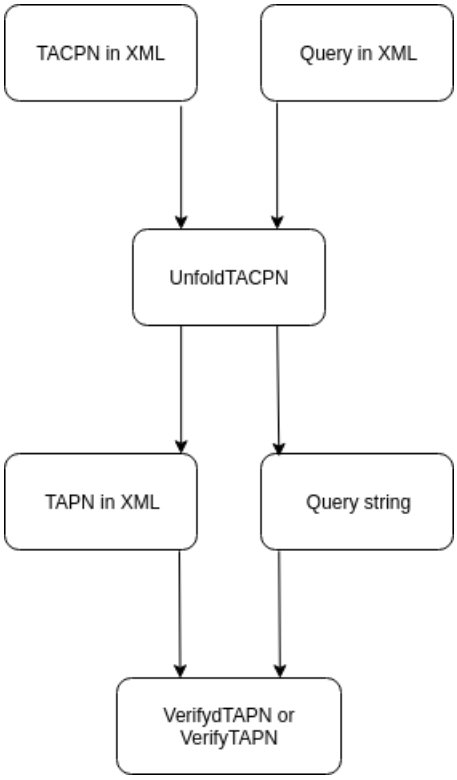


Fig. 16: Flow diagram displaying the verification process from TACPN to TAPN

To give an example of how unfolding works in TAPAAL we have modelled Figure 3 as can be seen in Figure 17.

At Figure 18 the same model is shown after it has been unfolded. Since *PacketType* got three colors we, in places as **S1** that got *PacketType* as its colors type, we create four places. Given that the order of the colors are in the original color type is $\{SSH, Web, Danger\}$ **S1Sub0** representing *SSH*. For each of the places we also generate a **Sum** which connects to every transition, in order to update the sum whenever a transition is fired. Since we have an inhibitor arc from **Firewall** to **S1ToFirewall** we create inhibitor arcs from **FirewallSum** to all transitions generated from **S1ToFirewall**.
The example here shows that the unfolding adheres to all the rules we have setup.

C. Testing

Before we can use the model we have created, we have to make sure that the results we get from our unfolding and verifications are correct. The unfolding is verified by manually checking the result of different nets. Here the focus has been on testing the unfolding of colored time constraints, inhibitor and transport arcs. This is because these features were not present in original engine *verifypn* which has been modified into *unfoldTACPN*. The nets used to test unfolding and verification can be found at *https://drive.google.com/drive/folders/1qGwmfqrrLM_3eWntMMvBZiJWdkL9s1u?usp=sharing*.
Ensuring verification is done in a similar matter. A large amount of nets used for verifying verification results for *verifydtapn* and *verifytapn* has been supplied. A sample of these nets have been recreated as TACPN's. The nets were then unfolded and verified with the same engines, where all gave the expected result.

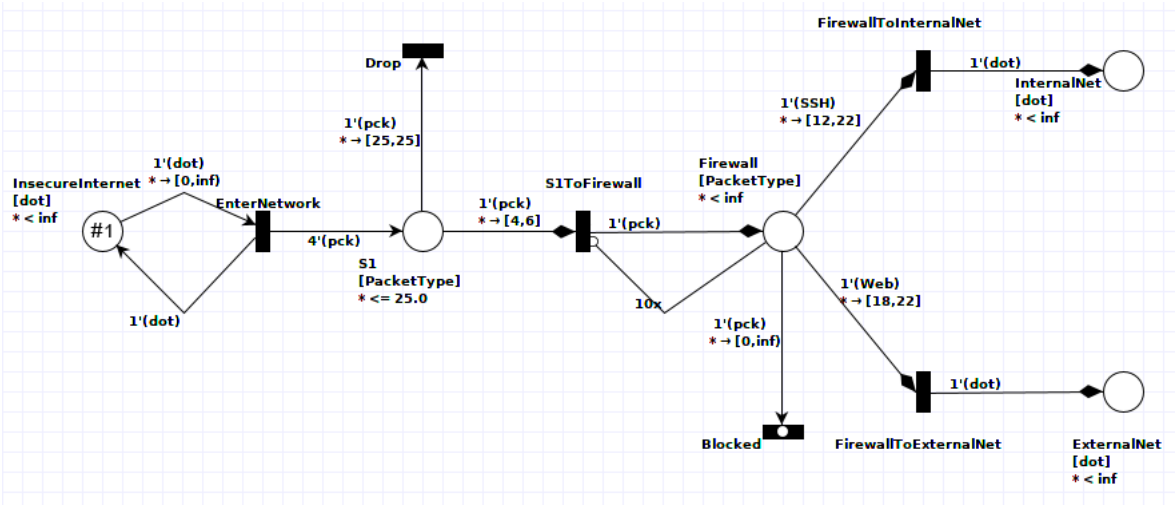


Fig. 17: Figure 3 drawn in TAPAAL

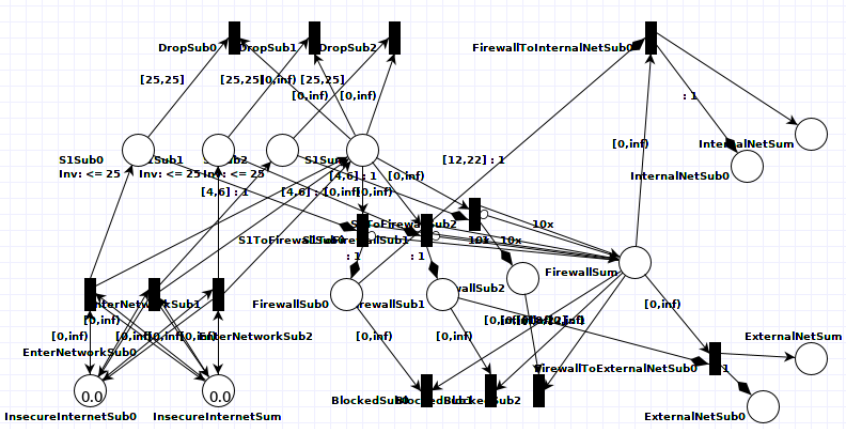


Fig. 18: Figure 17 unfolded

VI. Translation of Network Configurations and Updates into TACPN

In this section we present small introduction to networking, and in particular *Software-Defined Networks* (SDN), and some of the current problems in networks. We also present the safety property *Waypoint Enforcement* (WPE) [19] which we aim to verify while updating to a new network configuration.

A. Introduction to Updating Networks and Waypoint Enforcement

A modern network is a complicated structure so a simple task as sending a single packet throughout the network, can require that hundreds of servers to respond. Errors such as unexpected protocol interactions, or updating of configurations are a possibility since the packets sent over the network are being modified by all the different devices, and it is therefore difficult to reason about its behaviour. It can therefore be challenging to ensure correctness and guarantee that security policies are always kept under updates. There are numerous examples of network engineers deploying a new configuration to their network, which had severe implications. The whole country of Japan lost their internet connection for several hours in 2017, because Google made a bad update to a routing table [20]. This shows how vulnerable the internet is in its current state when updates are performed, which is not an uncommon occurrence.

To combat many of these issues a new approach called SDN is being developed.With the introduction of the SDN paradigm it is now possible through a centralized and programmable controller, to install,

update, and verify the network [21]. By introducing SDN, there is now a more dynamic network management, but this have also introduced some new problems. A problem is that the connection between the controller and the data layer is a distributed system, and therefore the sequence of when updates is applied is not guaranteed. When an update is occurring, the centralized controller has to instruct each switch to add, remove or change some forwarding rules. This becomes an issue if you want to update the network without taking it down, because we have to be sure that any packets in the network is not lost and that they will arrive at the desired destination.

There are several algorithms which solve this problem by creating a safe update sequence. In a safe update sequence, it will either update one or multiple switches at the same time in stages, ensuring the packet always arrives at the destination. An example of such an algorithm is WayUp, which also supports WPE [19]. This can be important, if your packet has to visit a specific switch, e.g a firewall, before reaching its destination. In order to illustrate an update consider Figure 19 which is from [19]. Here the old policy π_1 is the solid line while the new policy π_2 is the dashed line. The switch *s2* is a waypoint that all packets in the network have to traverse through. In this case three of the routers have to be updated, and WayUp solves this by making two sequences *(s3, s2),(s1)*. In this case WayUp tells us that *s3* and *s2* can be updated in parallel.

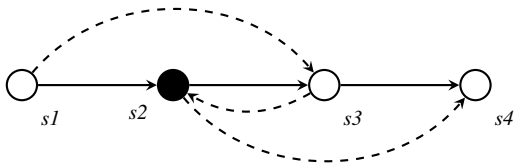


Fig. 19: Updating of a network from the solid to the dashed line that may break WPE

However, the second update can only begin when an acknowledgement has been received from all switches in the previous step. This can take a long time, and is therefore a big bottleneck, when it comes to deploying the whole update sequence. If update steps could be started before the acknowledgement is received, without breaking any policies for the packets in the network, the update could be optimized. In the following sections we will present a possible solution to optimizing these update sequences with Timed-arc Colored Petri Nets.

B. Latency for different packet sizes

In order to test the update sequences we have to account that processing times for different packets may be different, e.g., depending on the packet size. The reason behind this, is that a switch can have varying latency when processing packets with different sizes. To see the different latency's for the different packet sizes we are using Figure 3 from [7]. In order to have the most variation we are considering latency data from the hardware switch Pica8 P3297 [22]. For each packet size we have a variation of 0.5 μs upper and lower bound on the latency, since the latency's from Figure 3 in [7] are an average of their measured data. Different packet sizes are presented in Table I

Packet type	**Size** [Byte]	**Latency** [$\mu s \pm 0.5$]	**Interval**
VoIP [23]	218	2.2	[1, 3]
SSH [24]	312	2.1	[1, 3]
VPN [25]	1300	3.2	[2, 4]

TABLE I: Different network packet types and their size in bytes. Latency is deciphered from [7] Figure 3(a)

C. Router time calculation

The hardware that is used in network also has an impact on the time, since the switch scheduler, how fast it calculates the next-hop are different for different switches. Pereini et. al. created a scheduler for SDN in [3]. In their Figure 5, they show the cumulative distribution function of flow installation for 1000 flows on a FatTree topology with 20 switches. In Table II we decipher the completion time of different scheduling algorithms, when installing a batch of new flows into the FatTree topology.

Algorithm	**Minimum** $[ms]$	**Maximum** $[ms]$	**Interval**$[\mu s]$
Optimal	0	2.5	[0, 250000]
Batch-ready	0.2	2.5	[20000, 250000]
No scheduler	0.5	2.5	[50000, 250000]

TABLE II: Time for scheduler algorithms to install a batch of new flow into FatTree topology. Timing is deciphered from [3] Figure 5

The interval for the router update, is in μs to have the same time unit as packet type. Since it is the most interesting case that the router cannot start the update in time 0, we choose the interval from algorithm **No scheduler**.

D. Generating configurations and update sequence

In order to get network topologies, initial and final configurations of the network as well as update sequence between the configuration we use the tool Netsynth [5]. Netsynth takes a network topology, creates one or more source and destination pairs in the given topology, and creates a initial and final configuration. Netsynth takes a variety of parameters, where one of them is waypoints. This is done by specifying an arbitrary amount waypoints that the flow of the network have to traverse, which Netsynth will insure that both the initial and final configuration will traverse. The network topologies that are used in this thesis are from the *Zoo Topology* [14]. Netsynth provides two outputs. One output is a SDN program which is represented as an update sequence of commands to the controller, with a switch ID and rule index or with a # to represent a wait. An example of the update sequence a can be see on Figure 20.

```
(16:0), #, (18:0), #, (17:0), #,
(13:0), #, (15:0), #, (0:-1), #,
(1:-1), #, (2:-1), #, (3:-1), #,
(4:-1), #, (5:-1), #, (6:-1), #,
(7:-1), #, (8:-1), #, (9:-1), #,
(10:-1), #, (11:-1), #, (12:-1), #,
(14:0), #, (14:1), #, (19:-1), #, (20:-1)
```

Fig. 20: An update sequence outputted by Netsynth. Negative rule index represents an atomic update of all rules on a specific switch. # represents a wait between updates

The second output from Netsynth is a ltl file containing all switches from the topology, each port on the switch, the initial and finial configuration on each switch, the links between the ports of the switches, and a Linear Temporal Logic (LTL) which describes the traces in the model. An example of a switch with its ports and initial and final configuration can be seen on Figure 21. The first set of rules are for the initial configuration. What ports that should be updated, are based on the source and destination switch. How each port is updated is depending if it is the initial or final configuration. An example is line 3 of Figure 21. Here port 5 is getting a link to port 18 in the initial configuration, while on line 9 it is getting a link to port 17 in the final.

To start start experiments we need to calculate a maximum time for an update sequence we know is safe. In order to do this we use Equation 1. This equation represents the worst case scenario for

```
switch 18 (15, 11, 17, 5, 18, 32, 20, 58, 22, 64, 24, 66) {
  rule 11 => 18 [((src=20) & (dst=13))]1.
  rule 5  => 18 [((src=18) & (dst=13))]1.
  rule 11 =>    [((src=19) & (dst=14))]1.
  rule 5  =>    [((src=17) & (dst=14))]1.
}
final {
  rule 11 =>    [((src=20) & (dst=13))]1.
  rule 5  => 17 [((src=18) & (dst=13))]1.
  rule 11 => 18 [((src=19) & (dst=14))]1.
  rule 5  => 18 [((src=17) & (dst=14))]1.
}
```

Fig. 21: Switch 18 outputted by Netsynth with all its different ports, the update of the ports with different links, based on the initial or final configuration, and the source and destination switch

updating a single switch. We will therefore use this equation to assume the time it should take, before the next step in an update sequence can be executed.

$$Maximum\,time = Sumpath \times slowest\ packet\ hop + slowest\ switch\ update \tag{1}$$

Where *Sumpath* is the sum of the routers that will get updated in $initial + final$ configuration, *slowest packet hop*, is the packet type with the longest processing and transmit time, and *slowest switch update* is the switch with the longest update time.

E. Modeling networks into TACPN

In order to make a network into a TACPN model, we use the data from Table I and II, the initial and final configuration from Figure 21 and the update sequence from Figure 20. The input is then processed by our translation algorithm that creates six different types of components and a query for verification. A conceptual overview is displayed on Figure 22.

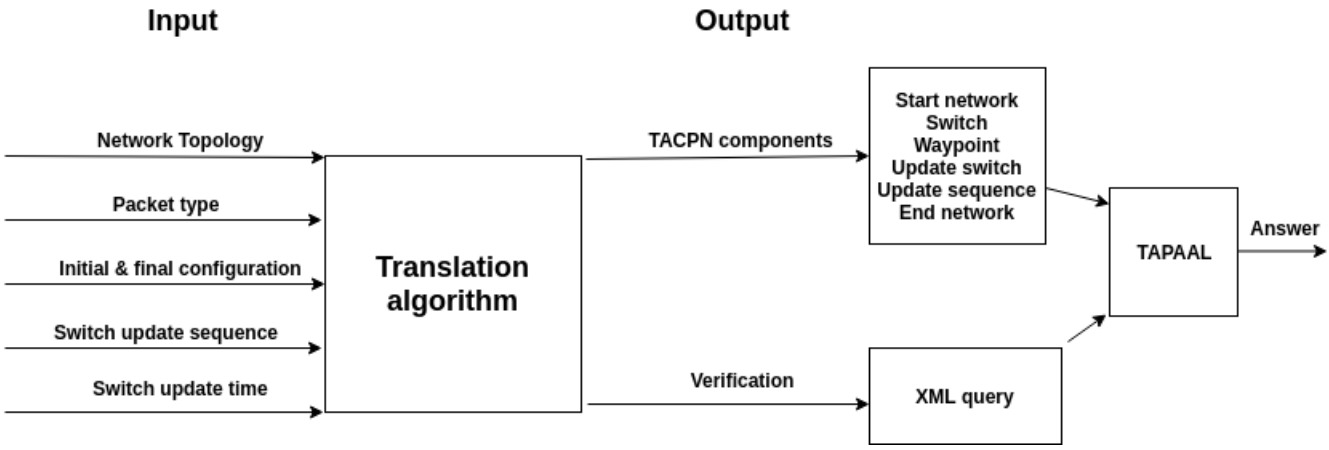


Fig. 22: Input to and output from the translation algorithm

To provide an understanding of how we create the components, we show a overview of the translation algorithm below.

Translation algorithm:

1) Create time constraints based upon packet input
2) For each packet type, create color in color type: *PckType*

3) Create start and end component with start and end switch
4) For all switches in initial or final configuration, create switch component
  a) If node is waypoint in either initial or final configuration, create as waypoint component
5) For each switch & waypoint in update sequence create update component, and create update sequence component
  a) The order in update sequence is created from switch update sequence input
6) Create *maximumTime* based upon Equation 1 and populate update sequence component
7) Create XML query for verification

By using the translation algorithm we can translate an arbitrary large network into the seven different TACPN components. By using colors, we are able to model each packet type, as its own color. As mentioned in Subsection VI-A there are many time aspects though out the network. By using intervals on arcs, we are modelling the different latency's presented in Table I. By using invariant of places, we are modelling that a packet will eventually get processed by the switch. As mentioned in Table II the router update time is different based upon what hardware and scheduler that is used. When starting the experiments, we use Equation 1 to calculate a safe update time, and uses this update time, for all updates.

To display the different components we generate when making a network in TACPN, we have modelled Figure 19 as a TACPN in TAPAAL.

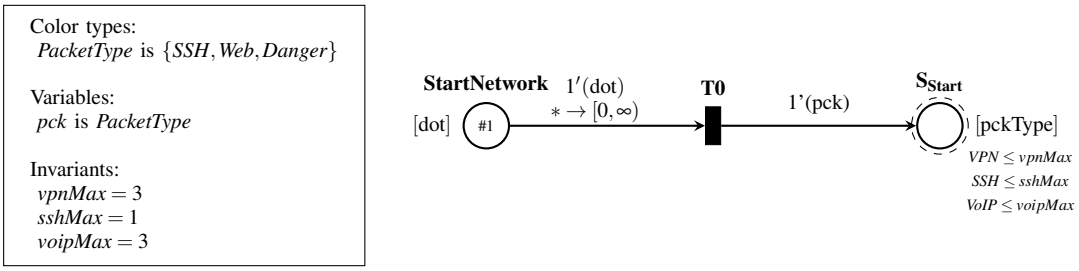


Fig. 23: Component displaying the start of the network

Figure 23 is an initial component which is the start of network. First a packet is send to the network represented as a token in place **StartNetwork**. The transition is then fired and creating a token of type *pck* in the shared place $\mathbf{S_{start}}$. $\mathbf{S_{start}}$ is a **Shared Place** meaning that this place, is the same place as a **S** on Figure 25 or 27 but it is a place on each component. On Figure 23, in the box to the left are the color type, variable, and time information displayed. The color type and variable is the same in all the components, and are therefore only presented once.

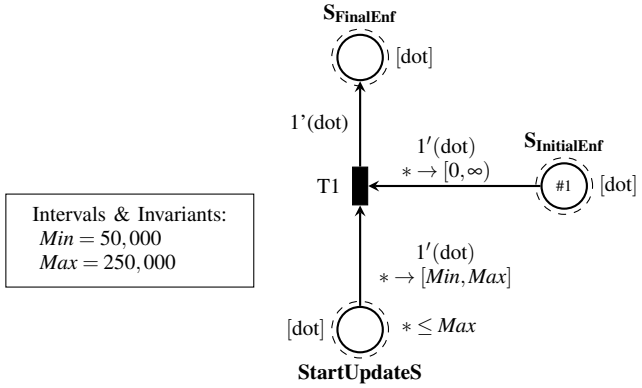


Fig. 24: Component displaying the update sequence of a switch S

Figure 24 is a component representing the updating sequence of a switch or waypoint. The update for a specific switch/waypoint is initialised by the invariant on place **StartUpdateS** and the interval on the input arc to **T1**. When **T1** is fired, it removes the token from $\mathbf{S_{InitialEnf}}$ and creates a token in $\mathbf{S_{FinalEnf}}$ such that packets now will use the final configuration.

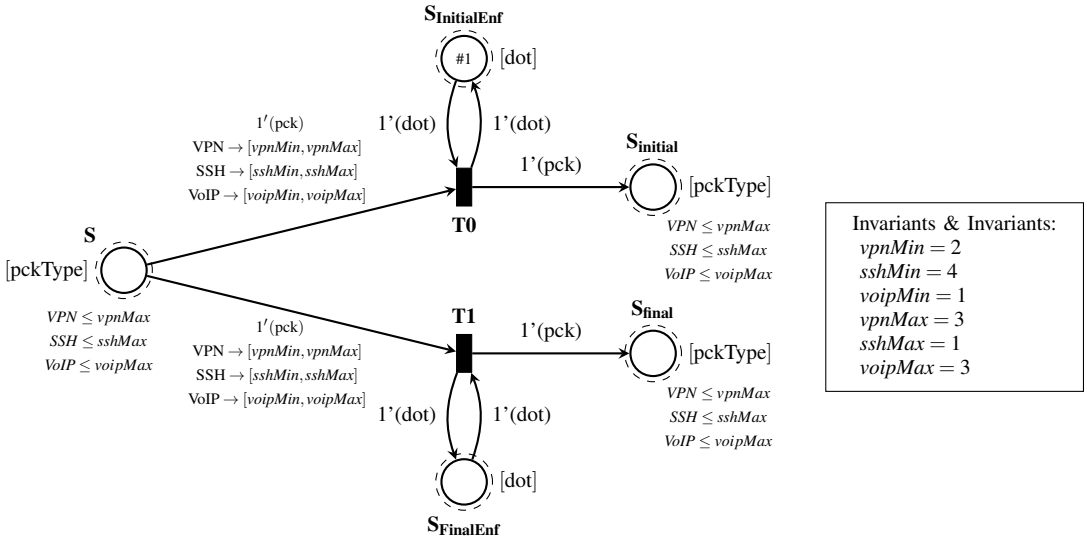


Fig. 25: Component displaying a switch S

Figure 25 is a component of a single switch. A packet of will be send to the switch by the shared place **S** and can either use the initial or the final configuration. At Figure 25 the initial configuration is active, since there is a token in the shared place $\mathbf{S_I nitialEnf}$, transition **T0** is enabled and can be fired, creating a token in $\mathbf{S_{initial}}$.

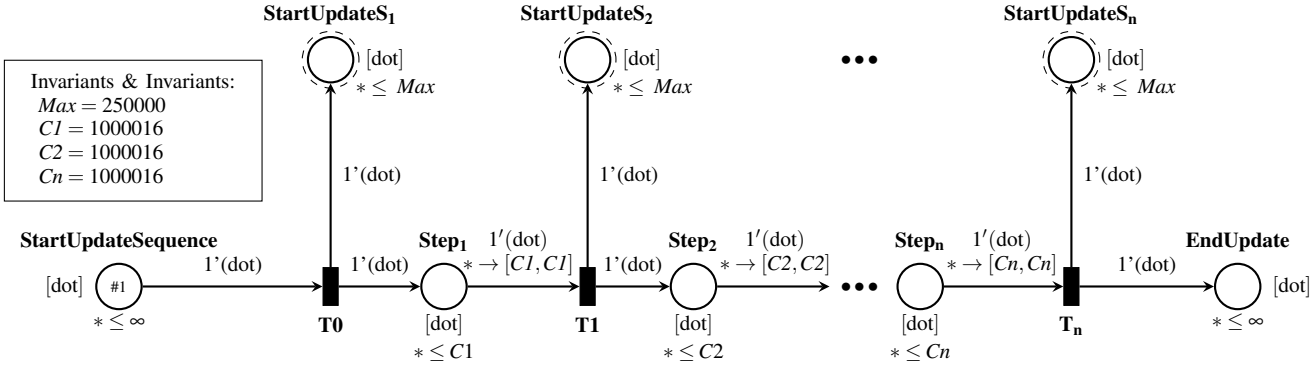


Fig. 26: Component displaying the update sequence for all switches S_1, S_2 ,$\cdots$, S_n. The size of n is equal to how many switches and waypoints that will get updated

Figure 26 is a update sequence of the network. The update can start at any time, hence the $\leq \infty$ invariant on place **StartUpdateSequence** and no time guard on the arc. The length of the update sequence is generated from how many switches and waypoints that will be updated. On Figure 26 the length is three. The order of the update is created from the update switch sequence input to the translation algorithm. *C1* and *C2* are both 1000016 since we know this a safe time for the update sequence, since they are calculated from Equation 1. This is the constants we will try to reduce, since they represent the time it takes to update a switch, and when we can start a new update without breaking network policies such as WPE.

Figure 27 is showing a waypoint component. By running the example we found a loop with the shared place **WaypointVisited** which is used in verification, to make sure that we traverse the waypoint. In order

to bypass this loop we created an extra transition **T4**, and made an inhibitor arc from **WaypointVisited** to **T0**. First time a waypoint is visited, only **T0** is enabled but it can only be fired once. When **T0** have been fired it creates a token in **WaypointVisited**, enforcing that only **T4** is enabled and therefore we know that the waypoint have been visited.

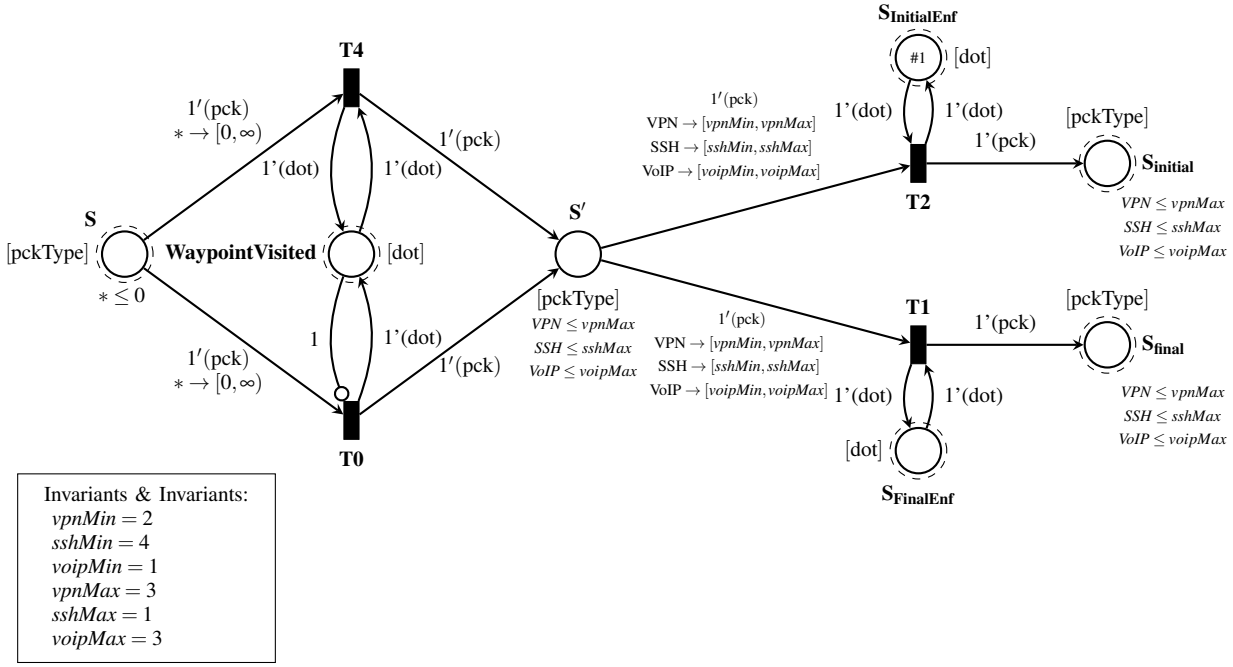


Fig. 27: Component displaying a waypoint S

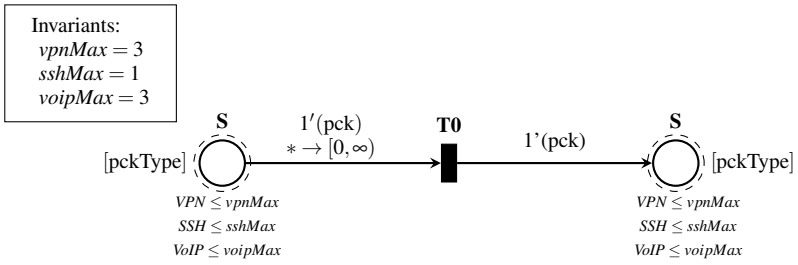


Fig. 28: Component displaying a switch S with no updates

A switch can have the same switch as the next-hop in both the initial and final configuration and therefore not have any updates between the configurations. This component is shown on Figure 28.

As mentioned we use both place **WaypointVisited** and **EndNetwork** for validation, by using a CTL query such as:

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

specifying that for all paths all reachable marking must satisfy that either **EndNetwork** is 0 or **WaypointVisited** is greater or equal 1.

Figure 29 is a component for the end of the network. On this component we use the **EndNetwork** place in the verification, to make sure that a packet has left the network.

To give a full view of all the components in the same net, Figure 30 is the composed TACPN of Figure 19.

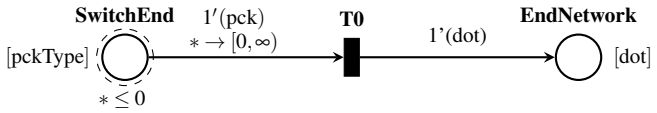


Fig. 29: Component displaying the end of the network

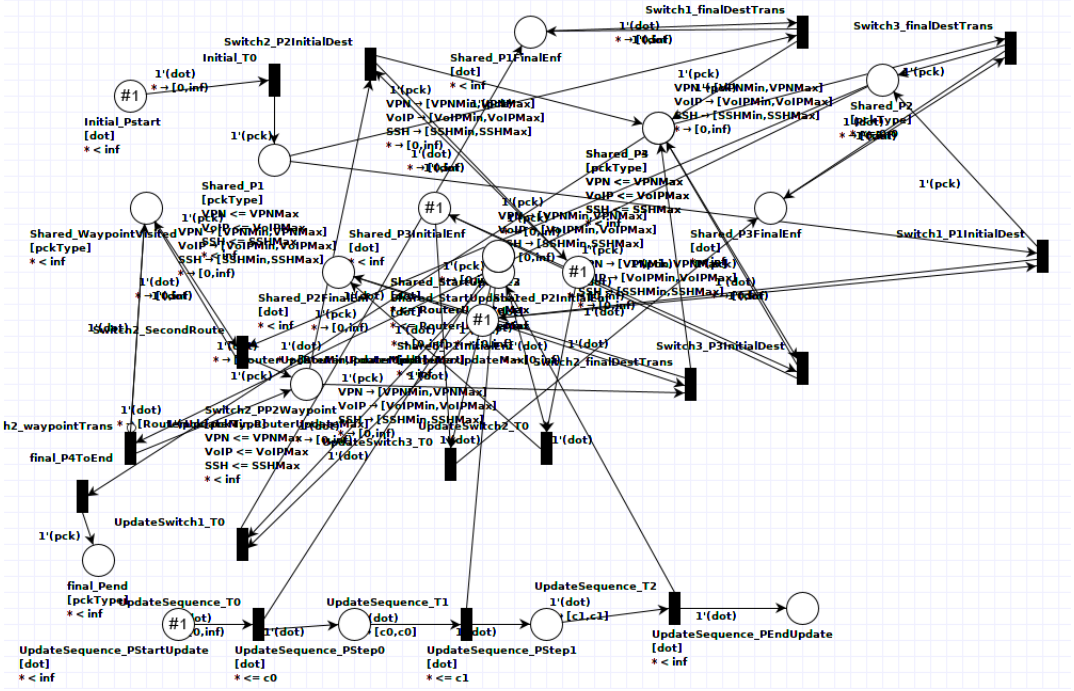


Fig. 30: Composed version of Figure 19

VII. Case Study: Updating Configurations in Network

In this section we will experiment if by using TACPN to model networks, we can optimize the updating between two configurations on switches. We will begin with testing on our running example where Netsynth has generated the solution. Next we will do the same with topologies taken from the Topology Zoo. All experiments have been done on a 64-bit Ubuntu 18.04 machine with 16 GB RAM, Intel Core i7-7700HQ CPU @ 2.80GHz x 8.

A. Minimizing all update sequence constants

As it was presented in the Equation 1 we know how to calculate the worst case update time for each of the switches. However, this is a theoretical max, and the real number may be much lower. In order to find the minimum of this constant for each update we use the bisection algorithm presented in Algorithm 1. The input is all constants used in the update sequence model set to the maximum from Equation 1, and the output is the same constants minimized, but where verification of the model still produces *true* based on the query presented in Subsection VI-E. For each constant the first thing we do is set it to *0* and try to verify with the new constant in the model. If *verify()* still returns true, then we know that the previous and current switch can start their update in parallel without affecting any packets. However, if it returns false we will have to find the minimum value for the constant that still produces true. This is done in the while loop from line 8-15. Here we use the *lastTrue* and *lastFalse* values, in the first case original value of *c* and *0*, to get us near the minimum value as seen on line 9. Based on the result received from a new call to *verify()* we will either update the value of *lastTrue* or *lastFalse*. We continue this until we find the bisection.

Algorithm 1 Find bisection for all possible C constants

Input: $C_1 ... C_n$
Output: $Minimumvalueforc_1 ... C_n$
1: **for** $(c \; in \; C)$ **do**
2: $lastFalse \leftarrow 0$
3: $lastTrue \leftarrow value(c)$
4: $setValue(c) \leftarrow 0$
5: **if** $verify(AG(\textbf{EndNetwork} = 0 \vee \textbf{WaypointVisited})$ **then**
6: **continue**
7: **else**
8: **while** $(lastTrue - lastFalse \neq 1)$ **do**
9: $setValue(c) \leftarrow ((lastTrue - lastFalse)/2) + lastFalse$
10: **if** $verify(AG(\textbf{EndNetwork} = 0 \vee \textbf{WaypointVisited})$ **then**
11: $lastTrue \leftarrow value(c)$
12: **else**
13: $lastFalse \leftarrow value(c)$
14: **end if**
15: **end while**
16: **end if**
17: **end for**
18: **return** $C_1 ... C_n$

B. Running example optimization results

Before we begin running the optimization on real topologies from the Topology Zoo [14] with NetSynth, we will use our optimization method on our running example, which illustrates some of the issues we have designed our solution for, such as when the paths intersect.

Route length]	**Unfold** $[s]$	**Verify**$[s]$	**Total bisection**$[s]$	**Cycles**	**Default time** $[s]$	**Optimized time** $[s]$	**Decrease** $[\%]$
6	0.66	0.044	5.86	22	2.25	0.45	77.50%

TABLE III: Results when using running example

In Table III **Route length** is the sum of switches in the initial and final path. **Unfold** and **Verify** are both timing information on how long it takes to unfold the model and verify it. **Total bisection** is the total time Algorithm 1 used to reduce all constants, where **Cycles** is the amount of iterations needed for all constants. **Default time** is the default update time based on the worst case calculations, **Optimized time** is the optimized update time after using Algorithm 1, and **Decrease** is the percentage improvement.
There are multiple interesting points to draw from this. First of all we can see that the **total bisection** time for a **route length** of *6* only takes *5.86 s*. This can indicate that in most cases it will be trivial to process the update optimization. A second thing is the **decrease** in update time, which is significant with *77.50%*. This is a promising result, given it is the same for real network topologies. Another thing to note is that $(\textbf{Unfold} + \textbf{Verify}) * \textbf{Cycles} \neq \textbf{Total bisection}$. The reason for this is the **Unfold** and **Verify** time displayed is peak time used in any of the *22* cycles and the time it takes can change.

C. Optimizing Netsynth update sequences

Using our optimization method on the running example showed promising results. However, it is an artificial constructed topology, configurations and update sequence. In order to show how the method handles real topologies we will be applying the tool Netsynth to create both initial and final configuration and an update sequence to solve the transition [5]. Netsynth will use topolgogies from the Topology Zoo. We will use the route length to order the complexity of the model. The results from the tests are presented in Table IV.

As it can be seen we have a high reduced update time, with better percentage reduction than the running example, since it is almost always reduced to *0.25s*. The reason all of these cases end at the same **Optimized time** is because algorithm 1 is reducing all the constants in the update sequence to

Name	Route length	Unfold [s]	Verify[s]	Total bisection[s]	Cycles	Default time [s]	Optimized time [s]	Decrease [%]
TLex	4	0.25	0.02	0.73	3	1.75	0.25	85.71%
HiberIreland	5	0.81	0.05	2.11	4	3.25	0.25	92.31%
Harnet	6	0.20	0.07	1.15	5	4.25	0.25	94.12%
UniC	7	0.30	0.16	1.90	6	6.50	0.25	96.15%
Oxford	8	0.36	0.37	3.13	7	7.75	0.25	96.77%
Xeex	10	0.30	2.14	7.07	9	12.25	0.25	97.96%
Sunet	11	0.32	5.32	13.78	10	15.95	0.25	98.43%
SwitchL3	12	1.17	13.07	31.40	11	17.75	0.25	98.59%
Psinet	14	0.50	78.46	156.04	13	24.25	0.25	98.97%
Uunet	15	0.40	194.45	379.23	14	29.50	0.25	99.15%
Renater2010	16	1.53	462.94	895.40	15	31.75	0.25	99.21%
Columbus	20	1.49	188.12	TIMEDOUT	13	49.75	16.7	66.43%
Missouri	25	2.20	329.87	TIMEDOUT	13	80.75	38.75	52.01%
Syringa	35	2.69	213.45	TIMEDOUT	12	157.00	104.75	33.28%
VtlWavenet2011	35	2.23	357.99	TIMEDOUT	13	157.01	100	36.31%

TABLE IV: Using Algorithm 1 to calculate constants to reduce update time between initial and final configuration from Netsynth. The verifier will timeout after 10 minutes

0. This means all updates can start concurrently and the worst case for update time will be the same as the worst case interval for updating a switch. This is not in it self a problem, as it shows that Netsynth always inherently produces safe configurations and update sequences for them. None of the configurations produced by Netsynth ever have nodes intersecting, as we do with the running example. For Netsynth this makes sense, as if it can be avoided to have crossing routes that is always preferred. It is only when the routes intersect that unexpected things can happen to the packet. However, for our case we want the intersecting routes as our solution is good at optimizing them.
When we look at the peak **Verify** time we can see it increases a lot with the **Route length**. This is a product of all the routers being able to update concurrently. For updating the network this is good to know, but it makes it more complex for the *verifytapn* tool to verify the query, as it has to explore all possibilities and they increase by $n!$ with n being the number of switches to update. This results in the bigger nets not completing, as they exceed 10 minute timeout. If we did not have the timeout we would not even get a partial result, as it would never finish. This way we are still able to get some reduction as it can be seen in Table IV. We can also see **Unfold** is increasing with **Route length** but not enough for it to be a bottle neck on the **Total bisection** time.

Since we know calculation time increase because of having to verify multiple switches updating concurrently, there could be an improvement in changing the order the update sequence constants are minimized in. The order the constants are reduced in does not matter, as we have to try and reduce them all before the algorithm is done. We can exploit this by creating gaps in the sequence, such that *verifytapn* does not have to verify the update of all switches at once, until the last constant has been set to *0*, removing the gap.
We achieve this by finding the minimum of every second constant in a loop until the time out has been reached, or the algorithm is done. The results from running with our new constant scheduler, but still on the same topologies, initial and finial configurations, and update sequence as in Table IV is presented in Table V.

In the tests with the optimized scheduler the time out is set to *3* minutes instead of *10*, however even with less time there is a significant improvement for the nets that time out. *Missouri* has gone from a *51.87%* **Decrease** to *90.99%*. At the same time we see that the **Verify** has fallen dramatically for the time out nets. It is still high for nets such as *Uunet* since they are still finishing, meaning the optimized scheduler is not helping on the last iteration. However, with the optimized scheduler able to optimize the update more and faster. This difference can also be seen in Figure 31 which shows the optimized scheduler increases the size of update sequences we are able to optimize and how much they can be optimized.
We reduce the calculation time for total bisection with almost half when the route length is at *18*, but

Name	Route length	Unfold [s]	Verify[s]	Total bisection[s]	Cycles	Default time [s]	Optimized time [s]	Decrease [%]
TLex	4	0.14	0.02	0.42	3	1.75	0.25	85.71%
HiberIreland	5	0.22	0.04	0.87	4	3.25	0.25	92.31%
Harnet	6	0.30	0.07	1.45	5	4.25	0.25	94.12%
UniC	7	0.22	0.16	1.53	6	6.50	0.25	96.15%
Oxford	8	0.22	0.38	2.12	7	7.75	0.25	96.77%
Xeex	10	0.27	2.18	6.03	9	12.25	0.25	97.96%
Sunet	11	0.31	5.34	10.16	10	15.95	0.25	98.43%
SwitchL3	12	0.32	13.09	18.80	11	17.75	0.25	98.59%
Psinet	14	0.37	78.84	89.28	13	24.25	0.25	98.97%
Uunet	15	0.40	194.59	206.84	14	29.50	0.25	99.15%
Renater2010	16	0.39	460.05	474.70	15	31.75	0.25	99.21%
Columbus	20	2.22	6.42	(TIMEDOUT) 198.40	17	49.75	5.75	88.44%
Missouri	25	1.70	9.50	(TIMEDOUT) 210.41	22	80.75	7.25	91.02%
Syringa	35	2.50	27.88	(TIMEDOUT) 201.23	29	157.00	24	84.71%
VtlWavenet2011	35	2.11	23.17	(TIMEDOUT) 181.92	29	157.01	24	84.71%

TABLE V: Using optimized calculation for constants to reduce update time between initial and final configuration from Netsynth

both curves are factorial graphs. However it should be able to implement an algorithm that calculates constants in batches, then skips one and then calculate another batch. By deciphering from Figure 31 the batch sizes can be up to *14*, with both the sequential and optimized constant calculation. With this implementation, it should be possible not to find the optimal, but to over-approximate a result for even the longest paths in the largest networks.

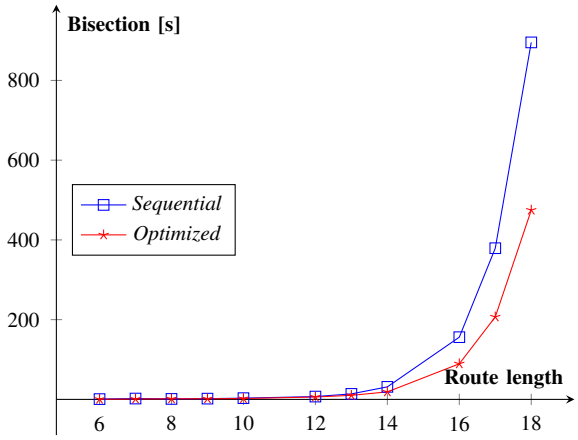


Fig. 31: Displaying the route length and the time to calculate the total bisection

D. Optimization on updates with intersections

As discussed in Subsection VII-C and VII-B testing the optimization algorithm on more configurations where the two routes intersect, would be interesting as in these cases it will not always be possible to set all update constants to *0*. However, it has not been possible to create such configurations automatically with the tool Netsynth. In order to still try and test more of these cases, we instead chose interesting topologies and created our own routes manually. However, this presents a new problem, as we do not have a safe update sequence for the two configurations we create.

To handle this we create update sequences randomly, by shuffling the update sequence. If the model cannot verify, it shows its a bad update sequence, and we can continue to the next one. This has the added benefit of showing that our model can verify whether a update sequence is safe, showing it could likely be used to verifying the result of update sequences produced by algorithms dedicated to this.
On Table VI the manually created routes and the optimization of their update time can be seen. In

the first four nets we used the topologies from the Topology Zoo, while the last two are completely artificially constructed. The first is a small net with many intersections and possibility for loops. The second is an extension of the running example also used in Subsection VII-B. Here we have taken the original model, copied it and added to the model again. This means the packet has to travel through a net with the same pattern twice. The reason for this is that if the original model had the requirements to create a gap, then the extended model could have two gaps. However, this is not the case. As Table VI shows all **Optimized time** are either *0.45s* or *0.25s*. This shows a pattern where if a constant is not reduced to *0* it always reduces to *0.2s* as the last *0.25s* is the update time for the last switch based on worst case for switch update interval. The reason for this is that the time constraints for all our models are static. In all the tests we have performed we use the same packet types and the same type of switch. If these time constraints were to change, e.g. if the net had different kinds of switches, the bisection value would also change.

An interesting point is, the bisection algorithm could be improved if a previous model with the same time constraints has already been calculated, or if a second gap is found. In this case the algorithm could jump straight to the last known bisection value and see if it still is true. This would reduce the amount of cycles necessary a lot.

Unfortunately we were unable to find a problem with two gaps. If this is because such a model does not exist or because we have not found it yet, is an open question. However, the fact that the constants are not always reduced to *0* when routes intersect, shows that our method can be applied to some update sequences and get usable optimizations from it.

Name	Route length	Unfold [s]	Verify[s]	Total bisection[s]	Cycles	Default time [s]	Optimized time [s]	Decrease [%]
Harnet	10	0.27	0.07	0.96	3	5.25	0.25	95.24%
Bics	16	0.31	0.17	8.62	23	4.75	0.45	90.53%
Globalcenter	12	0.29	0.32	11.39	25	8.25	0.45	94.54%
Ans	25	0.38	0.45	15.37	25	9.25	0.45	95.14%
PossibleLoop	8	0.21	0.04	4.48	22	2.75	0.45	83.63%
WPEExtended	14	0.61	0.81	19.99	26	10.25	0.45	95.61%

TABLE VI: Creating our own paths with random update sequences and default scheduler.

VIII. Conclusion & Future Work

In this thesis we present an algorithm to reduce the worst case update time in a network from one configuration to another. We show that for almost any route lengths in our experiments we can optimize the worst case update time by at least 80 %. When configurations do not have any intersections this optimization is trivial, as packet will only ever follow one of the routes. In intersecting routes a more interesting application is seen, as the routes can now intertwine and performing updates at the wrong time can break security policies of the network. Our algorithm minimizes this update without breaking any policies.
To accomplish this minimization we have presented a new mathematical modeling language called Timed-arc Colored Petri net with formal semantics. We present an unfolding algorithm and a proof of correctness. We have implemented TACPN in the tool TAPAAL, and an unfolding implementation.

A. Future work

There are several things which could be interesting to continue working with regarding this project. One thing is to optimize the current algorithm even more than with the optimized scheduler. Introducing intelligent behavior as discussed in Subsection VII-D could reduce the amount of cycles. Further improvements could be found by removing the step of building the model and unfolding it in each cycle. If the *unfoldTACPN* engine hold onto the constants and keep them in the model it outputs, we

would only have to do this step once, as we could now change the constants in the TAPN model instead of the TACPN model.
We would also like to create a more optimal algorithm to find initial and final configurations that will intersect, instead of creating them manually. Automatically generating a large amount of configurations with intersections, would make it easier to explore the possibility of having more than one gap.
In the future it would also be interesting to change the update sequence component, such that as soon as an update is completed, the next step will be fired. For now we are only exploring the reducing of maximal delay, while with this change, it will be possible to explore the minimal delay between updates. To this point, changing the hardware processing time and packet delay may also provide other results.

IX. Bibliographic Remarks

In this thesis Definitions 1-11 and 13 are from [26] with a modification to point 14 and 15 in Definition 7 and an explanation to the changes. Section II and III except Subsection II-B and III-A, and Figure 4, 5, and 6 with explanation, are also from [26].

In Section IV the introduction, Subsection IV-A and IV-B except Figure 7 and 8 which is extended with the color specific syntax, and introduction to Subsection IV-C are from [26].

In Section V the introduction and Figure 14 is slightly modified, and Figure 9 are from [26].

ACKNOWLEDGMENT

We would like to thanks Stefan Schmid for providing supervision, and helping us with contacting people throughout the making of this thesis.
We send a thanks to Nate Foster and Jedidiah McClurg for providing us with the implementation of Netsynth.
We also would like to thank Peter Gjøl Jensen for helping to us with our implementation and code review of TACPN in both TAPAAL and the unfolder in C++.
We send a thanks to Kenneth Yrke Jørgensen for providing help with parts of the Java implementation.
A thanks to Andreas Blenk for providing articles and information about how switches have different latency based on the size of the packets.
We would like to thank Thomas Frandsen, who was a part of the thesis in the start, and has worked on many of the first definitions.
Lastly we would like to send a special thanks to Jiri Srba for guidance and supervision in the creation of this thesis.

References

[1] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 49–54. ACM, 2013.
[2] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 20. ACM, 2013.
[3] 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.
[4] 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.
[5] 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.
[6] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang. A survey on network verification and testing with formal methods: Approaches and challenges. *IEEE Communications Surveys Tutorials*, pages 1–1, 2018.
[7] 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.
[8] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universitt Hamburg, 1962.
[9] Kurt Jensen. Coloured petri nets and the invariant-method. *Theoretical computer science*, 14(3):317–336, 1981.
[10] Chander Ramchandani. *Analysis of asynchronous concurrent systems by timed petri nets.* PhD thesis, Massachusetts Institute of Technology, 1973.

[11] Tommaso Bolognesi. From timed petri nets to timed lotos. In *Proc. IFIP WG6. 1, 10th Int. Workshop on Protocol Specification, Testing and Verification*, pages 377–406, 1990.

[12] Lasse Jacobsen, Morten Jacobsen, Mikael H Møller, and Jiří Srba. Verification of timed-arc petri nets. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 46–72. Springer, 2011.

[13] Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H Møller, and Jiří Srba. Tapaal 2.0: Integrated development environment for timed-arc petri nets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 492–497. Springer, 2012.

[14] 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.

[15] Andreas H Klostergaard. Efficient unfolding and approximation of colored petri nets with inhibitor arcs, 2018.

[16] F.M. Boenneland, P.G. Jensen, K.G. Larsen, M. Muniz, and J. Srba. Start pruning when time gets urgent: Partial order reduction for timed systems. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV'18)*, volume 10981 of *LNCS*, pages 527–546. Springer-Verlag, 2018.

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

[18] Frederik Bønneland, Jakob Dyhr, Peter G Jensen, Mads Johannsen, and Jiří Srba. Simplification of ctl formulae for efficient model checking of petri nets. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 143–163. Springer, 2018.

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

[20] Google accidentally broke japan's internet, 2017. - Web article. https://www.popularmechanics.com/technology/news/a27971/google-accidentally-broke-japans-internet/ (2017).

[21] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise, 2007.

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

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

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

[25] 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/.

[26] Thomas Frandsen Mark Glavind, Niels Christensen. Applying timed-arc colored petri net for network update synthesis. 9. semester article, 01 2019.