Model Checking Data Flows in Concurrent Network Updates*

Bernd Finkbeiner[1], Manuel Gieseking[2], Jesko Hecking-Harbusch[1], and Ernst-Rüdiger Olderog[2]

[1] Saarland University, Saarbrücken, Germany
[2] University of Oldenburg, Oldenburg, Germany

Abstract. We present a model checking approach for the verification of data flow correctness in networks during concurrent updates of the network configuration. This verification problem is of great importance for software-defined networking (SDN), where errors can lead to packet loss, black holes, and security violations. Our approach is based on a specification of temporal properties of individual data flows, such as the requirement that the flow is free of cycles. We check whether these properties are simultaneously satisfied for all active data flows while the network configuration is updated. To represent the behavior of the concurrent network controllers and the resulting evolutions of the configurations, we introduce an extension of Petri nets with a transit relation, which characterizes the data flow caused by each transition of the Petri net. For safe Petri nets with transits, we reduce the verification of temporal flow properties to a circuit model checking problem that can be solved with effective verification techniques like IC3, interpolation, and bounded model checking. We report on encouraging experiments with a prototype implementation based on the hardware model checker ABC.

1 Introduction

Software-defined networking (SDN) [33,7] is a networking technology that separates the packet forwarding process, called the *data plane*, from the routing process, called the *control plane*. Updates to the routing configuration can be initiated by a central controller and are then implemented in a distributed manner in the network. The separation of data plane and control plane makes the management of a software-defined network dramatically more efficient than a traditional network. The model checking of network configurations and concurrent updates between them is a serious challenge. The distributed update process can cause issues like forwarding loops, black holes, and incoherent routing which, from the perspective of the end-user, result in performance degradation, broken connections, and security violations. Correctness of concurrent network updates

* This work was supported by the German Research Foundation (DFG) Grant Petri Games (392735815) and the Collaborative Research Center Foundations of Perspicuous Software Systems (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (683300).

has previously been addressed with restrictions like *consistent updates* [39]: every packet is guaranteed during its entire journey to either encounter the initial routing configuration or the final routing configuration, but never a mixture in the sense that some switches still apply the old routing configuration and others already apply the new routing configuration. Under these restrictions, updates to network configurations can be synthesized [12,32]. Ensuring consistent updates is slow and expensive: switches must store multiple routing tables and messages must be tagged with version numbers.

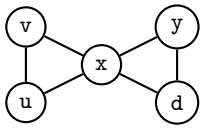
In this paper, we propose the *verification* of network configurations and concurrent updates between them. We specify desired properties of the data flows in the network, such as the absence of loops, and then automatically check, for a given initial routing configuration and a concurrent update, whether the specified properties are simultaneously satisfied for all active data flows while the routing configuration is updated. This allows us to check a specific concurrent update and to thus only impose a sequential order where this is strictly needed to avoid an erroneous configuration during the update process.

Our approach is based on temporal logic and model checking. The control plane of the network can naturally be specified as a Petri net. Petri nets are convenient to differentiate between sequential and parallel update steps. The data plane, however, is more difficult to specify. The standard flow relation of a Petri net does not describe which ingoing token of a transition transits to which outgoing token. In theory, such a connection could be made with *colored* Petri nets [22], by using a uniquely colored token for each data flow in the network. Since there is no bound on the number of packets, this would require infinitely many tokens and colors to track the infinitely many data flows. To avoid this problem, we develop an extension of Petri nets called *Petri nets with transits*, which augment standard Petri nets with a *transit relation*. This relation specifies the precise data flow between ingoing and outgoing tokens of a transition. In Petri nets with transits, a single token can carry an unbounded number of data flows.

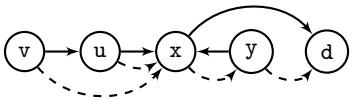
We introduce a linear-time temporal logic called *Flow-LTL* to specify the correct data flows in Petri nets with transits. The logic expresses requirements on several separate timelines: global conditions, such as fairness, are expressed in terms of the global timeline of the system run. Requirements on individual data flows, such as that the data flow does not enter a loop, on the other hand, are expressed in terms of the timeline of that specific data flow. The next operator, for example, refers to the next step taken by the particular data flow, independently of the behavior of other, simultaneously active, data flows.

Concurrent updates of software-defined networks can be modeled as safe Petri nets with transits. We show that the model checking problem of the infinite state space of Petri nets with transits against a Flow-LTL formula can be reduced to the LTL model checking problem for Petri nets with a finite number of tokens; and that this model checking problem can in turn be reduced to checking a hardware circuit against an LTL formula. This ultimately results in a standard verification problem, for which highly efficient tools such as ABC [2] exist.

Proofs and more detailed constructions can be found in the full paper [15].



(a) Example of a network topology with five switches and six connections.



(b) Solid arrows show the forwarding rules before the update, dashed arrows the intended forwarding rules after the update.

```
ingress = {v}        ingress = {v}
v.fwd(u)                   v.fwd(x)
u.fwd(x)                   u.fwd(x)
x.fwd(d)                   x.fwd(y)
y.fwd(x)                   y.fwd(d)
egress = {d}         egress = {d}
```

(c) Network programs for the routing configurations before and after the update.

Fig. 1: Example (due to [19]) of an update to a software-defined network.

2 Motivating Example

We motivate our approach with a typical network update problem taken from the literature [19]. Consider the simple network topology shown in Fig. 1a. From the global point of view, our goal is to update the network from the routing configuration shown with solid lines in Fig. 1b to the routing configuration shown with dashed lines. Such routing configurations are typically given as static NetCore [20,35] programs like the ones shown in Fig. 1c. The `ingress` and `egress` sections define where packets enter and leave the network, respectively. Expressions of the form `v.fwd(u)` define that switch `v` forwards packets to switch `u`.

It is not straightforward to see how the update from Fig. 1b can be implemented in a distributed manner. If switch `x` is updated to forward to switch `y` *before* `y` is updated to forward to switch `d`, then any data flow that reaches `x` is sent into a loop between `x` and `y`. A correct update process must thus ensure sequentiality between switch updates `upd(y.fwd(d))` and `upd(x.fwd(y))`, in this order. The only other switch with changing routing is switch `v`. This update can occur in any order. A correct concurrent update would thus work as follows:

$$\texttt{(upd(y.fwd(d)) >> upd(x.fwd(y))) || upd(v.fwd(x))},$$

where `>>` and `||` denote sequential and parallel composition, respectively.

Figure 2 shows a Petri net model for the network topology and the concurrent update from the initial to the final routing configuration from Fig. 1. The right-hand side models the control plane, where, beginning in *update_start*, the update of `v` and, concurrently, the sequential update to `y` and then to `x` is initiated. Each marking of the net represents a control state of the network. Changes to the control state are thus modeled by the standard flow relation. Leaving out the control plane allows us to verify configurations of network topologies.

On the left-hand side, we model the data plane by extending the Petri net with a transit relation. This new type of Petri nets will be defined formally in the next section. We only depict the update to the data flow in and from switch `v`. Places $sw_{\texttt{u}}$, $sw_{\texttt{v}}$, and $sw_{\texttt{x}}$ represent the switches `u`, `v`, and `x`, respectively. The data plane is modeled by the transit relation which indicates the extension of the data flows during each transition at the switches.

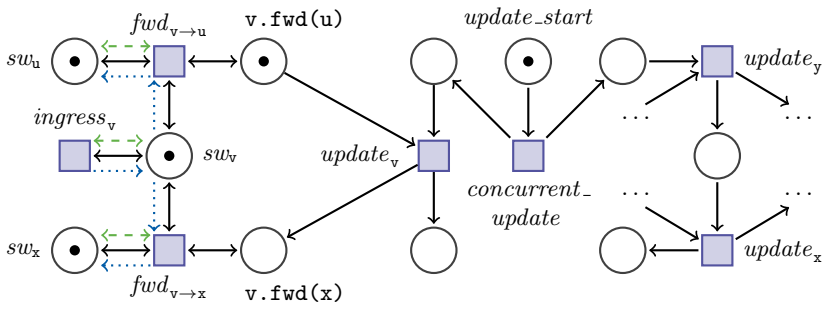


Fig. 2: Example Petri net with transits encoding the data plane on the left and the control plane on the right. The standard flow relation, describing the flow of tokens, is depicted by solid black arrows, the transit relation by colored arrows. Colors that occur on both ingoing and outgoing arrows of a transition define that the transition extends the data flow. If an outgoing arrow has a color that does not appear on an ingoing arrow, a new data flow is initiated.

The standard flow relation is depicted by solid black arrows and the transit relation by colored arrows. If an outgoing arrow has a color that does *not* appear on an ingoing arrow, then a new data flow is initiated. In our example, data flows are initiated by transition $ingress_{\mathtt{v}}$ and the (dotted) blue arrow. Colors that occur on both in- and outgoing arrows extend the data flow. In transition $fwd_{\mathtt{v}\to\mathtt{u}}$, the (dotted) blue arrows indicate the extension of the data flow from $sw_{\mathtt{v}}$ to $sw_{\mathtt{u}}$. The (dashed) green arrows between $fwd_{\mathtt{v}\to\mathtt{u}}$ and $sw_{\mathtt{u}}$ indicate that, in addition to the incoming data flow from $sw_{\mathtt{v}}$, there may be data flows that have previously reached $sw_{\mathtt{u}}$ and have not yet departed from $sw_{\mathtt{u}}$. These flows stay in $sw_{\mathtt{u}}$.

Notice that $ingress_{\mathtt{v}}$, $fwd_{\mathtt{v}\to\mathtt{u}}$, and $fwd_{\mathtt{v}\to\mathtt{x}}$ do not actually move tokens because of the double-headed arrows. None of these transitions change the control state, they only model the data flow. As the switches `u`, `v`, and `x` remain continuously active, their tokens in sw_u, sw_v, and sw_x are never moved. By contrast, $update_{\mathtt{v}}$ moves the token from `v.fwd(u)` to `v.fwd(x)`, thus disabling the data flow from $sw_{\mathtt{v}}$ to $sw_{\mathtt{u}}$ and enabling the data flow from $sw_{\mathtt{v}}$ to $sw_{\mathtt{x}}$. We specify the correctness of our update process with formulas of the temporal logic *Flow-LTL*. The formula $\mathbb{A}\Diamond\mathtt{d}$ expresses *connectivity* requiring that all data flows ($\mathbb{A}$) eventually ($\Diamond$) arrive at the egress switch `d`. Flow-LTL and the specification of data flow properties are discussed in more detail in Sec. 4 and Sec. 5. The general construction of the motivating example is formalized in the full paper [15].

3 Petri Nets with Transits

We give the formal definition of *Petri nets with transits*. We assume some basic knowledge about standard Petri nets [37]. A safe *Petri net with transits* (PNwT) is a structure $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In, \Upsilon)$, where the set of *places* $\mathcal{P}$, the set of *transitions* $\mathcal{T}$, the (*control*) *flow relation* $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$, and the *initial*

marking $In \subseteq \mathscr{P}$ are as in *safe* Petri nets. In safe Petri nets, each reachable marking contains at most one token per place. We add the *transit relation* Υ of tokens for transitions to obtain Petri nets with transits. For each transition $t \in \mathscr{T}$, we postulate that $\Upsilon(t)$ is a relation of type $\Upsilon(t) \subseteq (pre^{\mathscr{N}}(t) \cup \{\triangleright\}) \times post^{\mathscr{N}}(t)$, written in infix notation, where the symbol $\triangleright$ denotes a *start*. $p\ \Upsilon(t)\ q$ defines that the token in place p *transits* via transition t to place q and $\triangleright\ \Upsilon(t)\ q$ defines that the token in place q marks the start of a new data flow via transition t. The graphic representation of $\Upsilon(t)$ in Petri nets with transits uses a *color coding* as can be seen in Fig. 2. Black arrows represent the usual *control flow*. Other matching colors per transition are used to represent the transits of tokens. Transits allow us to specify which data flows are moved forward, split, and merged, which data flows are removed, and which data flows are newly created.

Data flows can be of infinite length and can be created at any point in time. Hence, the number of data flows existing in a place during an execution depends on the causal past of the place. Therefore, we recall informally the notions of unfoldings and runs [13,14] and apply them to Petri nets with transits. In the unfolding of a Petri net $\mathscr{N}$, every transition stands for the unique occurrence (instance) of a transition of $\mathscr{N}$ during an execution. To this end, every loop in $\mathscr{N}$ is unrolled and every join of transitions in a place is expanded by duplicating the place. Forward branching, however, is preserved. Formally, an *unfolding* is a branching process $\beta^U = (\mathscr{N}^U, \lambda^U)$ consisting of an occurrence net $\mathscr{N}^U$ and a homomorphism λ^U that labels the places and transitions in $\mathscr{N}^U$ with the corresponding elements of $\mathscr{N}$. The unfolding exhibits concurrency, causality, and nondeterminism (forward branching) of the unique occurrences of the transitions in $\mathscr{N}$ during all possible executions. A *run* of $\mathscr{N}$ is a subprocess $\beta = (\mathscr{N}^R, \rho)$ of β^U, where $\forall p \in \mathscr{P}^R : |post^{\mathscr{N}^R}(p)| \leq 1$ holds, i.e., all nondeterminism has been resolved but concurrency is preserved. Thus, a run formalizes one concurrent execution of $\mathscr{N}$. We introduce the *unfolding of Petri nets with transits* by lifting the transit relation to the unfolding $\beta^U = (\mathscr{N}^U, \lambda^U)$. We define the relation Υ^U as follows: For any $t \in \mathscr{T}^U$, the transit relation $\Upsilon^U(t) \subseteq (pre^{\mathscr{N}^U}(t) \cup \{\triangleright\}) \times post^{\mathscr{N}^U}(t)$ is defined for all $p, q \in \mathscr{P}^U$ by $p\ \Upsilon^U(t)\ q \Leftrightarrow \lambda^U(p)\ \Upsilon(\lambda^U(t))\ \lambda^U(q)$.

We use the transit relation in the unfolding to introduce (data) flow chains. A (*data*) *flow chain* in β^U is a maximal sequence $\xi = p_0, t_0, p_1, t_1, p_2, \ldots$ of places in $\mathscr{P}^U$ with connecting transitions in $\mathscr{T}^U$ such that

1. $\exists t \in \mathscr{T}^U : \triangleright\ \Upsilon^U(t)\ p_0$,
2. if ξ is infinite then for all $i \geq 0$ the transit relation $p_i\ \Upsilon^U(t_i)\ p_{i+1}$ holds,
3. if ξ is finite, say $p_0, t_0, \ldots, t_{n-1}, p_n$ for some $n \geq 0$, then for all i with $0 \leq i < n$ the transit relation $p_i\ \Upsilon^U(t_i)\ p_{i+1}$ holds, and there is no place $q \in \mathscr{P}^U$ and no transition $t \in \mathscr{T}^U$ with $p_n\ \Upsilon^U(t)\ q$.

4 Flow-LTL for Petri Nets with Transits

We recall LTL applied to Petri nets and define our extension *Flow-LTL* to specify the behavior of flow chains in Petri nets with transits. We fix a Petri net with transits $\mathscr{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In, \Upsilon)$ throughout the section.

4.1 Linear Temporal Logic for Petri nets

We define $AP = \mathscr{P} \cup \mathscr{T}$ as the set of *atomic propositions*. The set $\mathtt{LTL}$ of *linear temporal logic* (LTL) formulas over AP has the following syntax $\psi ::= \mathit{true} \mid a \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \bigcirc\psi \mid \psi_1 \,\mathsf{U}\, \psi_2$, where $a \in AP$. Here, $\bigcirc$ is the *next* operator and U is the *until* operator. We use the abbreviated temporal operators $\Diamond$ (*eventually*) and $\Box$ (*always*) as usual. A *trace* is a mapping $\sigma : \mathbb{N} \longrightarrow 2^{AP}$. The trace $\sigma^i : \mathbb{N} \longrightarrow 2^{AP}$, defined by $\sigma^i(j) = \sigma(i+j)$ for all $j \in \mathbb{N}$, is the ith *suffix* of σ.

We define the traces of a Petri net based on its runs. Consider a run $\beta = (\mathcal{N}^R, \rho)$ of $\mathcal{N}$ and a finite or infinite firing sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle M_2 \cdots$ of $\mathcal{N}^R$ with $M_0 = In^R$. This sequence *covers* β if $(\forall p \in \mathscr{P}^R \; \exists i \in \mathbb{N} : p \in M_i) \wedge (\forall t \in \mathscr{T}^R \; \exists i \in \mathbb{N} : t = t_i)$, i.e., all places and transitions in $\mathcal{N}^R$ appear in ζ. Note that several firing sequences may cover β. To each firing sequence ζ covering β, we associate an infinite trace $\sigma(\zeta) : \mathbb{N} \longrightarrow 2^{AP}$. If ζ is finite, say $\zeta = M_0[t_0\rangle \cdots [t_{n-1}\rangle M_n$ for some $n \geq 0$, we define 1. $\sigma(\zeta)(i) = \rho(M_i) \cup \{\rho(t_i)\}$ for $0 \leq i < n$ and 2. $\sigma(\zeta)(j) = \rho(M_n)$ for $j \geq n$. Thus, we record for $0 \leq i < n$ (case 1) all places of the original net $\mathcal{N}$ that label the places in the marking M_i in $\mathcal{N}^R$ and the transition of $\mathcal{N}$ that labels the transition t_i in $\mathcal{N}^R$ outgoing from M_i. At the end (case 2), we *stutter* by repeating the set of places recorded in $\sigma(\zeta)(n)$ from n onwards, but repeat no transition. If ζ is infinite we apply case 1 for all $i \geq 0$ as no stuttering is needed to generate an infinite trace $\sigma(\zeta)$.

We define the *semantics* of $\mathtt{LTL}$ on Petri nets by $\mathcal{N} \models_{\mathtt{LTL}} \psi$ iff for all runs β of $\mathcal{N}$: $\beta \models_{\mathtt{LTL}} \psi$, which means that for all firing sequences ζ covering β : $\sigma(\zeta) \models_{\mathtt{LTL}} \psi$, where the latter refers to the usual binary satisfaction relation $\models_{\mathtt{LTL}}$ between traces σ and formulas $\psi \in \mathtt{LTL}$ defined by: $\sigma \models_{\mathtt{LTL}} \mathit{true}$, $\sigma \models_{\mathtt{LTL}} a$ iff $a \in \sigma(0)$, $\sigma \models_{\mathtt{LTL}} \neg\psi$ iff not $\sigma \models_{\mathtt{LTL}} \psi$, $\sigma \models_{\mathtt{LTL}} \psi_1 \wedge \psi_2$ iff $\sigma \models_{\mathtt{LTL}} \psi_1$ and $\sigma \models_{\mathtt{LTL}} \psi_2$, $\sigma \models_{\mathtt{LTL}} \bigcirc\psi$ iff $\sigma^1 \models_{\mathtt{LTL}} \psi$, $\sigma \models_{\mathtt{LTL}} \psi_1 \,\mathsf{U}\, \psi_2$ iff there exists a $j \geq 0$ with $\sigma^j \models_{\mathtt{LTL}} \psi_2$ and for all i with $0 \leq i < j$ the following holds: $\sigma^i \models_{\mathtt{LTL}} \psi_1$.

4.2 Definition of Flow-LTL for Petri Nets with Transits

For Petri nets with transits, we wish to express requirements on several separate timelines. Based on the global timeline of the system run, global conditions like fairness and maximality can be expressed. Requirements on individual data flows, e.g., that the data flow does not enter a loop, are expressed in terms of the timeline of that specific data flow. *Flow-LTL* comprises of *run formulas* φ specifying the usual LTL behavior on markings and *data flow formulas* φ_F specifying properties of flow chains inside runs:

$$\varphi ::= \psi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \psi \rightarrow \varphi \mid \varphi_F \quad \text{and} \quad \varphi_F ::= \mathbb{A}\, \psi$$

where formulas $\psi \in \mathtt{LTL}$ may appear both inside φ and φ_F.

To each flow chain ξ in a run β, we associate an infinite *flow trace* $\sigma(\xi) : \mathbb{N} \longrightarrow 2^{AP}$. If ξ is finite, say $\xi = p_0, t_0 \ldots, t_{n-1}, p_n$ for some $n \geq 0$, we define (1) $\sigma(\xi)(i) = \{\rho(p_i), \rho(t_i)\}$ for $0 \leq i < n$ and (2) $\sigma(\xi)(j) = \{\rho(p_n)\}$ for $j \geq n$.

Thus, we record for $0 \leq i < n$ (case 1) the place and the transition of the original net $\mathcal{N}$ that label the place p_i in $\mathcal{N}^R$ and the transition t_i in $\mathcal{N}^R$

outgoing from p_i. At the end (case 2), we *stutter* by repeating the place recorded in $\sigma(\xi)(n)$ infinitely often. No transition is repeated in this case.

If ξ is infinite we apply case 1 for all $i \geq 0$. Here, no stuttering is needed to generate an infinite flow trace $\sigma(\xi)$ and each element of the trace consists of a place and a transition.

A Petri net with transits $\mathcal{N}$ satisfies φ, abbr. $\mathcal{N} \models \varphi$, if the following holds:

$$
\begin{array}{ll}
\mathcal{N} \models \varphi & \text{iff for all runs } \beta \text{ of } \mathcal{N}: \ \beta \models \varphi \\
\beta \models \varphi & \text{iff for all firing sequences } \zeta \text{ covering } \beta: \ \beta, \sigma(\zeta) \models \varphi \\
\beta, \sigma(\zeta) \models \psi & \text{iff } \sigma(\zeta) \models_{\mathtt{LTL}} \psi \\
\beta, \sigma(\zeta) \models \varphi_1 \wedge \varphi_2 & \text{iff } \beta, \sigma(\zeta) \models \varphi_1 \text{ and } \beta, \sigma(\zeta) \models \varphi_2 \\
\beta, \sigma(\zeta) \models \varphi_1 \vee \varphi_2 & \text{iff } \beta, \sigma(\zeta) \models \varphi_1 \text{ or } \beta, \sigma(\zeta) \models \varphi_2 \\
\beta, \sigma(\zeta) \models \psi \to \varphi & \text{iff } \beta, \sigma(\zeta) \models \psi \text{ implies } \beta, \sigma(\zeta) \models \varphi \\
\beta, \sigma(\zeta) \models \mathbb{A}\ \psi & \text{iff for all flow chains } \xi \text{ of } \beta: \ \sigma(\xi) \models_{\mathtt{LTL}} \psi
\end{array}
$$

5 Example Specifications

We illustrate Flow-LTL with examples from the literature on software-defined networking. Specifications on data flows like loop and drop freedom are encoded as data flow formulas. Fairness assumptions for switches are given as run formulas.

5.1 Data Flow Formulas

We show how properties from the literature can be encoded as data flow formulas. For a network topology, let Sw be the set of all switches, $Ingr \subseteq Sw$ the ingress switches, and $Egr \subseteq Sw$ the egress switches with $Ingr \cap Egr = \emptyset$. The connections between switches are given by $Con \subseteq Sw \times Sw$.

Loop freedom. Loop freedom [29] requires that a data flow visits every switch at most once. In Sec. 2, we outlined that arbitrarily ordered updates can lead to loops in the network. The following data flow formula expresses that each data flow is required to not visit a non-egress switch anymore after it has been forwarded and therefore left that switch (realized via the U-operator):

$$\mathbb{A}\Box\Big(\bigwedge_{\mathtt{s} \in Sw \setminus Egr} \mathtt{s} \to (\mathtt{s}\ \mathsf{U}\ \Box\neg\mathtt{s})\Big)$$

Drop freedom. Drop freedom [38] requires that no data packets are dropped. Packets are dropped by a switch if no forwarding is configured. We specify that all data flows not yet at the egress switches are extended by transitions from a set Fwd encoding the connections Con between switches (details of the encoding can be found in the full paper [15]). We obtain the following data flow formula:

$$\mathbb{A}\Box\Big(\bigwedge_{\mathtt{e} \in Egr} \neg\mathtt{e} \to \bigvee_{f \in Fwd} f\Big)$$

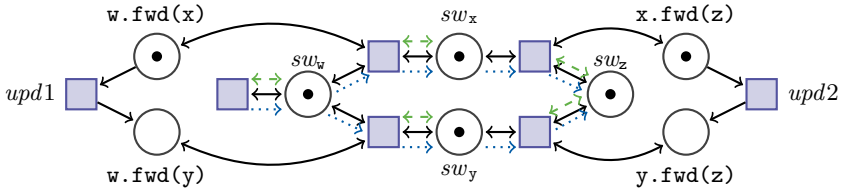


Fig. 3: Concurrent network update that does not preserve drop freedom.

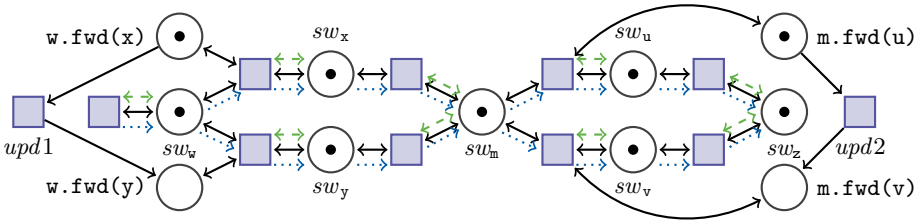


Fig. 4: Concurrent network update that does not preserve packet coherence.

Example 1. Figure 3 shows an example update that violates drop freedom. Packets are forwarded from switch `w` to switch `z` either via switch `x` or via switch `y`. If the forwarding of `x` is deactivated by firing transition $upd2$ *before* the forwarding of switch `w` is updated by firing $upd1$, then all packets still forwarded from `w` to `x` are dropped as no outgoing transitions from `x` will be enabled.

Packet coherence. Packet coherence [1] requires that every data flow follows one of two paths: either the path according to the routing before the update or the path according to the routing after the update. The paths $Path_1$ and $Path_2$ are defined as the sets of switches of the forwarding route before and after the update. This results in the following data flow formula:

$$\mathbb{A}(\square(\bigvee_{\mathtt{s}\in Path_1} \mathtt{s}) \vee \square(\bigvee_{\mathtt{s}\in Path_2} \mathtt{s}))$$

Example 2. In Fig. 4, the encoding of an update to a double-diamond network topology [8] is depicted as a simple example for a packet incoherent update. Before firing the update transitions $upd1$ and $upd2$, packets are forwarded via switches `x`, `m`, and `u`, after the complete update, via switches `y`, `m`, and `v`. If `m` is updated by firing transition $upd2$ while packets have been forwarded to `x` then these packets are forwarded along the incoherent path `x`, `m`, and `v`.

We note that loop and drop freedom are incomparable requirements. Together, they imply that all packets reach one egress switch. Connectivity, in turn, implies drop freedom but not loop freedom, because an update can allow some loops.

5.2 Run Formulas

Data flow formulas require behavior on the maximal flow of packets and switches are assumed to forward packets in a fair manner. Both types of assumptions are expressed in Flow-LTL as run formulas. We typically consider implications between run formulas and data flow formulas.

Maximality. A run β is *interleaving-maximal* if, whenever some transition is enabled, some transition will be taken: $\beta \models \square(\bigvee_{t\in\mathcal{T}} \mathit{pre}(t) \rightarrow \bigvee_{t\in\mathcal{T}} t)$.
A run β is *concurrency-maximal* if, when a transition t is from a moment on always enabled, infinitely often a transition t' (including t itself) sharing a precondition with t is taken: $\beta \models \bigwedge_{t\in\mathcal{T}}(\diamondsuit\square\ \mathit{pre}(t) \rightarrow \square\diamondsuit \bigvee_{p\ \in\ \mathit{pre}(t), t'\ \in\ \mathit{post}(p)}\ t')$.

Fairness. A run β is *weakly fair* w.r.t. a transition t if, whenever t is always enabled after some point, t is taken infinitely often: $\beta \models \diamondsuit\square\ \mathit{pre}(t) \rightarrow \square\diamondsuit t$.
A run β is *strongly fair* w.r.t. t if, whenever t is enabled infinitely often, t is taken infinitely often: $\beta \models \square\diamondsuit\ \mathit{pre}(t) \rightarrow \square\diamondsuit t$.

6 Model Checking Flow-LTL on Petri Nets with Transits

We solve the model checking problem of a Flow-LTL formula φ on a Petri net with transits $\mathcal{N}$ in three steps:

1. $\mathcal{N}$ is encoded as a Petri net $\mathcal{N}^{>}$ without transits obtained by composing suitably modified copies of $\mathcal{N}$ such that each flow subformula in φ can be checked for correctness using the corresponding copy.
2. φ is transformed to an LTL-formula $\varphi^{>}$ which skips the uninvolved composition copies when evaluating run and flow parts, respectively.
3. $\mathcal{N}^{>}$ and $\varphi^{>}$ are encoded in a circuit and fair reachability is checked with a hardware model checker to answer if $\mathcal{N} \models \varphi$ holds.

Given a Petri net with transits $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \mathit{In}, \Upsilon)$ and a Flow-LTL formula φ with subformulas $\varphi_{F_i} = \mathbb{A}\,\psi_i$, where $i = 1, \ldots, n$ for some $n \in \mathbb{N}$, we produce a Petri net $\mathcal{N}^{>} = (\mathscr{P}^{>}, \mathscr{T}^{>}, \mathscr{F}^{>}, \mathscr{F}_I^{>}, \mathit{In}^{>})$ with inhibitor arcs (denoted by $\mathscr{F}_I^{>}$) and an LTL formula $\varphi^{>}$. An *inhibitor arc* is a directed arc from a place p to a transition t, which only enables t if p contains no token. Graphically, those arcs are depicted as arrows equipped with a circle on their arrow tail.

6.1 From Petri Nets with Transits to P/T Petri nets

We informally introduce the construction of $\mathcal{N}^{>}$ and Fig. 5 visualizes the process by an example. Details for this and the following constructions, as well as all proofs corresponding to Sec. 6 can be found in the full paper [15].

The *original part* of $\mathcal{N}^{>}$ (denoted by $\mathcal{N}_O^{>}$) is the original net $\mathcal{N}$ without transit relation and is used to check the run part of the formula. To $\mathcal{N}_O^{>}$, a *subnet* for each subformula $\mathbb{A}\,\psi_i$ of φ is composed (denoted by $\mathcal{N}_i^{>}$, with places $\mathscr{P}_i^{>}$ and transitions $\mathscr{T}_i^{>}$), which serves for checking the corresponding data flow part of the formula φ. The subnet introduces the possibility to decide for the

tracking of up to one specific flow chain by introducing a copy $[p]_i$ of each place $p \in \mathcal{P}$ and transitions simulating the transits. The place $[\iota]_i$ serves for starting the tracking. Each run of a subnet simulates one possible flow chain of $\mathcal{N}$, i.e., every firing sequence covering any run of $\mathcal{N}$ yields a flow chain.

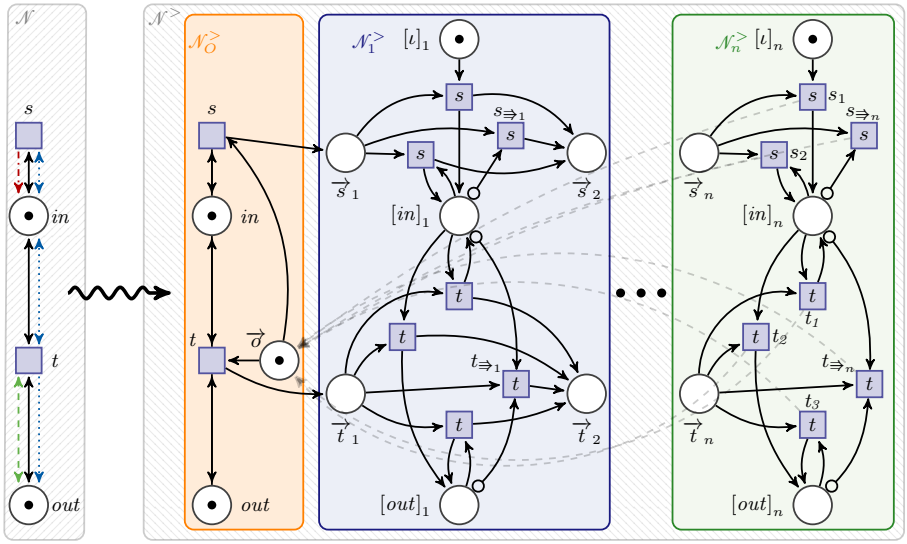


Fig. 5: An overview of the constructed P/T Petri net $\mathcal{N}^>$ (on the right) for an example Petri net with transits $\mathcal{N}$ (on the left) and n flow subformulas $\mathbb{A}\,\psi_i$.

An *activation token* iterates sequentially through these components via places $\overrightarrow{t}$ for $t \in \mathcal{T}$. In each step, the active component has to fire exactly one transition and pass the active token to the next component. The sequence starts by $\mathcal{N}_O^>$ firing a transition t and proceeds through every subnet simulating the data flows according to the transits of t. This implies that the subnets have to either move their data flow via a t-labelled transition t' ($\lambda(t') = t$) or use the skipping transition $t_{\Rightarrow_i}$ if their chain is not involved in the firing of t or a newly created chain should not be considered in this run.

Lemma 1 (Size of the Constructed Net). *The constructed Petri net $\mathcal{N}^>$ has $\boldsymbol{O}(|\mathcal{N}| \cdot n + |\mathcal{N}|)$ places and $\boldsymbol{O}(|\mathcal{N}|^3 \cdot n + |\mathcal{N}|)$ transitions.*

6.2 From Flow-LTL Formulas to LTL Formulas

The two different kinds of timelines of φ are encoded in the LTL formula $\varphi^>$. On the one hand, the data flow formulas $\mathbb{A}\,\psi_i$ in φ are now checked on the corresponding subnets $\mathcal{N}_i^>$ and, on the other hand, the run formula part of φ is checked on the original part of the net $\mathcal{N}_O^>$. In both cases, we need to ignore

places and transitions from other parts of the composition. This is achieved by replacing each next operator $\bigcirc\,\phi$ and atomic proposition $t \in \mathcal{T}$ inside φ with an until operator. Transitions which are not representing the considered timeline are called *unrelated*, others *related*. Via the until operator, all unrelated transitions can fire until a related transition is fired. This is formalized in Tab. 1 using the sets $O = \mathcal{T}^{>} \setminus \mathcal{T}$ and $O_i = (\mathcal{T}^{>} \setminus \mathcal{T}_i^{>}) \cup \{t_{\Rrightarrow_i} \in \mathcal{T}_i^{>} \mid t \in \mathcal{T}\}$, for the unrelated transitions of the original part and of the subnets, respectively. The related transitions of the original part are given by $\mathcal{T}$ and for the subnets by $M_i(t) = \{t' \in \mathcal{T}_i^{>} \setminus \{t_{\Rrightarrow_i}\} \mid \lambda(t') = t\}$ and $M_i = \mathcal{T}_i^{>} \setminus \{t_{\Rrightarrow_i} \in \mathcal{T}_i^{>} \mid t \in \mathcal{T}\}$.

Table 1: Row 1 considers the substitutions in the run part of φ, row 2 the substitutions in each subformula φ_{F_i}. Column 1 considers simultaneously substitutions, column 2 substitutions from the inner- to the outermost occurrence.

$t \in \mathcal{T}$	$\bigcirc\,\phi$
$(\bigvee_{t' \in O} t')\,\mathsf{U}\,t$	$((\bigvee_{t \in O} t)\,\mathsf{U}\,((\bigvee_{t' \in \mathcal{T}} t') \wedge \bigcirc\,\phi)) \vee (\Box\,(\neg(\bigvee_{t' \in \mathcal{T}} t')) \wedge \phi)$
$(\bigvee_{t_o \in O_i} t_o)\,\mathsf{U}\,(\bigvee_{t_m \in M_i(t)} t_m)$	$((\bigvee_{t \in O_i} t)\,\mathsf{U}\,((\bigvee_{t \in M_i} t) \wedge \bigcirc\,\phi)) \vee (\Box\,(\neg(\bigvee_{t \in M_i} t)) \wedge \phi)$

Additionally, every atomic proposition $p \in \mathcal{P}$ in the scope of a flow operator is simultaneously substituted with its corresponding place $[p]_i$ of the subnet. Every flow subformula $\mathbb{A}\,\psi_i$ is substituted with $\Box\,[\iota]_i \vee ([\iota]_i\,\mathsf{U}\,(\neg\,[\iota]_i \wedge \psi_i'))$, where $\Box\,[\iota]_i$ represents that no flow chain is tracked and ψ_i' is the result of the substitutions of atomic propositions and next operators described before. With $[\iota]_i\,\mathsf{U}\,(\neg\,[\iota]_i \wedge \psi_i')$ we ensure to only check the flow subformula at the time the chain is created. Finally, restricting runs to not end in any of the subnets yields the final formula $\varphi^{>} = (\Box \diamondsuit \overrightarrow{o}) \rightarrow \varphi^{\mathbb{A}}$ with $\overrightarrow{o}$ being the activation place of the original part of the net and $\varphi^{\mathbb{A}}$ the result of the substitution of all flow subformulas.

Lemma 2 (Size of the Constructed Formula). *The size of the constructed LTL formula $\varphi^{>}$ is in $\mathcal{O}(|\mathcal{N}|^3 \cdot n \cdot |\varphi| + |\varphi|)$.*

Lemma 3 (Correctness of the Transformation). *For a Petri net with transits $\mathcal{N}$ and a Flow-LTL formula φ, there exists a safe P/T Petri net $\mathcal{N}^{>}$ with inhibitor arcs and an LTL formula $\varphi^{>}$ such that $\mathcal{N} \models \varphi$ iff $\mathcal{N}^{>} \models_{\text{LTL}} \varphi^{>}$.*

6.3 Petri Net Model Checking with Circuits

We translate the model checking of an LTL formula ψ with places and transitions as atomic propositions on a safe P/T Petri net with inhibitor arcs $\mathcal{N}$ to a model checking problem on a circuit. We define the circuit $\mathcal{C}_{\mathcal{N}}$ simulating $\mathcal{N}$ and an adapted formula ψ', which can be checked by modern model checkers [9,18,2].

A *circuit* $\mathcal{C} = (\mathcal{I}, \mathcal{O}, \mathcal{L}, \mathcal{F})$ consists of boolean variables $\mathcal{I}$, $\mathcal{O}$, $\mathcal{L}$ for input, output, latches, and a boolean formula $\mathcal{F}$ over $\mathcal{I} \times \mathcal{L} \times \mathcal{O} \times \mathcal{L}$, which is deterministic in $\mathcal{I} \times \mathcal{L}$. The formula $\mathcal{F}$ can be seen as transition relation from a

valuation of the input variables and the current state of the latches to the valuation of the output variables and the next state of the latches. A circuit $\mathcal{C}$ can be interpreted as a Kripke structure such that the satisfiability of a formula ψ' (denoted by $\mathcal{C} \models \psi'$) can be defined by the satisfiability in the Kripke structure.

The desired circuit $\mathcal{C}_{\mathcal{N}}$ has a latch for each place $p \in \mathcal{P}$ to store the current *marking*, a latch `i` for *initializing* this marking with *In* in the first step, and a latch `e` for handling *invalid* inputs. The inputs $\mathcal{I}$ consider the firing of a transition $t \in \mathcal{T}$. The latch `i` is true in every but the first step. The latch `e` is true whenever invalid values are applied on the inputs, i.e., the firing of not enabled, or more than one transition. The marking latches are updated according to the firing of the valid transition. If currently no valid input is applied, the marking is kept from the previous step. There is an output for each place (the current marking), for each transition (the transition leading to the next marking), and for the current value of the invalid latch. We create ψ' by skipping the initial step and allowing invalid inputs only at the end of a trace: $\psi' = \bigcirc(\square(\mathtt{e} \rightarrow \square\,\mathtt{e}) \rightarrow \psi)$. This allows for finite firing sequences. The concrete formula $\mathcal{F}$, the Kripke structure, and the corresponding proofs can be found in the full paper [15]. The circuit $\mathcal{C}_{\mathcal{N}}$ can be encoded as an and-inverter graph in the Aiger format [4].

Lemma 4 (Correctness of the Circuit). *For a safe P/T Petri net with inhibitor arcs $\mathcal{N}$ and an LTL formula ψ, there exists a circuit $\mathcal{C}_{\mathcal{N}}$ with $|\mathcal{P}| + 2$ latches and $\mathcal{O}(|\mathcal{N}|^2)$ gates, and ψ' of size $\mathcal{O}(|\psi|)$ such that $\mathcal{N} \models_{\mathtt{LTL}} \psi$ iff $\mathcal{C}_{\mathcal{N}} \models \psi'$.*

Theorem 1. *A safe Petri net with transits $\mathcal{N}$ can be checked against a Flow-LTL formula φ in single-exponential time in the size of $\mathcal{N}$ and φ.*

Checking a safe Petri net with transits against Flow-LTL has a PSPACE-hard lower bound because checking a safe Petri net against LTL is a special case of this problem and reachability of safe Petri nets is PSPACE-complete.

7 Implementation Details and Experimental Results

We implemented our model checking approach in a prototype tool based on the tool Adam [16]. Our tool takes as input a Flow-LTL specification and a Petri net with transits, and carries out the transformation described in Sec. 6 to obtain an LTL formula and an Aiger circuit. We then use MCHyper [18] to combine the circuit and the LTL formula into another Aiger circuit. MCHyper is a verification tool for HyperLTL [10], which subsumes LTL. The actual model checking is carried out by the hardware model checker ABC [2]. ABC provides a toolbox of state-of-the-art verification and falsification techniques like IC3 [5]/PDR [11], interpolation (INT) [34], and bounded model checking [3] (BMC, BMC2, BMC3). We prepared an artifact to replicate our experimental results [21].

Our experimental results cover two benchmark families (SF/RP) and a case study (RU) from software-defined networking on real-world network topologies: **Switch Failure** (SF) (*Parameter:* n switches): From a sequence of n switches with the ingress at the beginning and the egress at the end, a failing switch is

chosen at random and removed. Then, data flows are bypassed from the predecessor to the successor of the failing switch. Every data flow reaches the egress node no matter of the update (connectivity).
Redundant Pipeline (RP) (*Parameters:* n_1 switches in pipeline one / n_2 switches in pipeline two / v version): The *base version (B)* contains two disjoint sequences of switches from the ingress to the egress, possibly with differing length. For this and the next two versions, it is required that each data flow reaches the egress node (connectivity) and is only forwarded via the first or the second pipeline (packet coherence). *Update version (U):* Two updates are added that can concurrently remove the first node of any pipeline and return the data flows to the ingress. If both updates happen, data flows do not reach the egress. Returning the data flows violates packet coherence. *Mutex version (M):* A mutex is added to the update version such that at most one pipeline can be broken. Updates can happen sequentially such that data flows are in a cycle through the ingress. *Correct version (C):* The requirements are weakened such that each data flow only has to reach the egress when updates do not occur infinitely often.
Routing Update (RU) is a case study based on realistic software-defined networks. We picked 31 real-world network topologies from [26]. For each network, we choose at random an ingress switch, an egress switch, and a loop- and drop-free initial configuration between the two. For a different, random final configuration, we build a sequential update in reverse from egress to ingress. The update overlaps with the initial configuration at some point during the update or is activated from the ingress in the last step. It is checked if all packets reach the egress (T) and if all packets reach another specific switch as an egress (F).

Table 2 presents our experimental results and indicates for each benchmark the model checking approach with the best performance. In the benchmarks where the specification is satisfied (✓), IC3 is the clear winner, in benchmarks where the specification is violated (✗), the best approach is bounded model checking with dynamic unrolling (BMC2/3). The results are encouraging: hardware model checking is effective for circuits constructed by our transformation with up to 400 latches and 27619 gates; falsification is possible for larger circuits with up to 1288 latches and 269943 gates. As a result, we were able to automatically verify with our prototype implementation updates for networks with topologies of up to 10 switches ($\#S$) and to falsify updates for topologies with up to 38 switches within the time bound of 30 minutes.

We investigated the cost of specifications drop and loop freedom compared with connectivity and packet coherence. Table 3 exemplarily shows the results for network topology *Napnet* from RU. Connectivity, packet coherence, and loop freedom have comparable runtimes due to similar formula and circuit sizes. Drop freedom is defined over transitions and, hence, expensive for our transformation.

8 Related Work

There is a large body of work on software-defined networks, see [28] for a good introduction. Specific solutions that were proposed for the network update prob-

Table 2: Experimental results from the benchmark families Switch Failure (SF) and Redundant Pipeline (RP), and the case study Routing Update (RU). The results are the average over five runs on an Intel i7-2700K CPU with 3.50 GHz, 32 GB RAM, and a timeout of 30 minutes.

			PNwT			Translated PN			Circuit		Result		
Ben.	Par.	$\#S$	$\lvert\mathscr{P}\rvert$	$\lvert\mathscr{T}\rvert$	$\lvert\varphi\rvert$	$\lvert\mathscr{P}^{>}\rvert$	$\lvert\mathscr{T}^{>}\rvert$	$\lvert\psi'\rvert$	Lat.	Gat.	Sec.	Algo.	$\models$
SF	3	4	4	5	35	17	22	60	90	2796	2.7	IC3	✓
	...					...					...		
	9	10	10	11	95	35	46	138	186	8700	1359.9	IC3	✓
	10	11	11	12	105	38	50	151	202	9964	TO	-	?
RP	1/1/B	4	4	5	43	17	22	68	100	2989	4.0	IC3	✓
	...					...					...		
	4/4/B	10	10	11	103	35	46	146	196	8893	646.4	IC3	✓
	4/5/B	11	11	12	113	38	50	159	212	10157	TO	-	?
	1/1/U	6	6	9	63	25	36	100	136	5535	1.6	BMC2	✗
	...					...					...		
	5/4/U	13	13	16	133	46	64	191	248	14523	945.1	BMC3	✗
	5/5/U	14	14	17	143	49	68	204	264	16127	TO	-	?
	1/1/M	6	9	11	63	30	42	106	146	6908	8.1	BMC3	✗
	...					...					...		
	4/3/M	11	14	16	113	45	62	171	226	13573	1449.6	BMC2	✗
	4/4/M	12	15	17	123	48	66	184	242	15146	TO	-	?
	1/1/C	6	9	11	70	30	42	113	151	7023	63.1	IC3	✓
	...					...					...		
	3/3/C	10	13	15	110	42	58	165	215	12195	1218.0	IC3	✓
	3/4/C	11	14	16	120	45	62	178	231	13688	TO	-	?
RU	Arpanet196912T	4	14	10	117	31	39	154	188	7483	22.7	IC3	✓
	Arpanet196912F	4	14	10	117	31	39	154	188	7483	2.0	BMC3	✗
	NapnetT	6	23	17	199	48	64	254	292	15875	95.1	IC3	✓
	NapnetF	6	23	17	199	48	64	254	292	15875	4.7	BMC3	✗
	...					...					...		
	NetrailT	7	30	23	271	62	88	344	380	26101	145.3	IC3	✓
	NetrailF	7	30	23	271	62	88	344	380	26101	58.3	BMC3	✗
	Arpanet19706T	9	33	24	281	67	89	354	400	27619	507.8	IC3	✓
	Arpanet19706F	9	33	24	281	67	89	354	400	27619	49.7	BMC3	✗
	NsfcnetT	10	31	22	261	65	87	334	376	26181	304.8	IC3	✓
	NsfcnetF	10	31	22	261	65	87	334	376	26181	8.4	BMC3	✗
	...					...					...		
	TwarenF	20	65	45	531	130	170	664	736	87493	461.5	BMC3	✗
	MarnetF	20	77	57	679	156	224	854	908	138103	746.1	BMC3	✗
	JanetlenseF	20	91	71	847	184	280	1064	1104	203595	514.2	BMC2	✗
	HarnetF	21	71	50	593	143	193	744	812	108415	919.0	BMC3	✗
	Belnet2009F	21	71	50	597	145	199	754	816	113397	1163.3	BMC2	✗
	...					...					...		
	UranF	24	56	38	449	106	133	552	618	57950	143.2	BMC3	✗
	KentmanFeb2008F	26	82	56	669	167	223	844	920	142291	111.2	BMC3	✗
	Garr200212F	27	86	59	703	174	232	884	964	153509	324.2	BMC3	✗
	IinetF	31	104	73	871	210	288	1094	1176	227153	1244.5	BMC3	✗
	KentmanJan2011F	38	117	79	943	236	312	1184	1288	269943	112.6	BMC3	✗

Table 3: For the network topology Napnet and a concurrent update between two randomly generated topologies, our four standard requirements are checked.

		PN w. Transits			Translated PN			Circuit		Result		
Ben.	Req.	$\lvert\mathscr{P}\rvert$	$\lvert\mathscr{T}\rvert$	$\lvert\varphi\rvert$	$\lvert\mathscr{P}^{>}\rvert$	$\lvert\mathscr{T}^{>}\rvert$	$\lvert\psi'\rvert$	Latches	Gates	Sec.	Algo.	$\models$
Napnet	connectivity	23	17	199	48	64	254	292	15875	95.1	IC3	✓
	p. coherence	23	17	208	48	64	267	298	16041	31.9	IC3	✓
	loop-free	23	17	237	48	64	296	305	16289	52.6	INT	✓
	drop-free	23	17	257	48	64	2288	325	30449	165.9	IC3	✓

lem include *consistent updates* [39,8] (cf. the introduction), *dynamic scheduling* [23], and *incremental updates* [25]. Model checking, including both explicit and SMT-based approaches, has previously been used to verify software-defined networks [6,31,30,43,1,36]. Closest to our work are models of networks as Kripke structures to use model checking for synthesis of correct network updates [12,32]. While they pursue *synthesis*, rather than verification of network updates, the approach is still based on a model checking algorithm that is called in each step of the construction of a sequence of updates. The model checking subroutine of the synthesizer assumes that each packet sees at most one switch that was updated after the packet entered the network. This restriction is implemented with explicit waits, which can afterwards often be removed by heuristics. Our model checking routine does not require this assumption. As it therefore allows for more general updates, it would be very interesting to add the new model checking algorithm into the synthesis procedure. Flow correctness also plays a role in other application areas like *access control* in physical spaces. Flow properties that are of interest in this setting, such as "from every room in the building there is a path to exit the building", have been formalized in a temporal logic [42].

There is a significant number of model checking tools (e.g., [40,41,24]) for Petri nets and an annual model checking contest [27]. In this contest, however, only LTL formulas with places as atomic propositions are checked. To the best of our knowledge, other model checking tools for Petri nets do not provide places and transitions as atomic propositions. Our encoding needs to reason about places and transitions to pose fairness conditions on the firing of transitions.

9 Conclusion

We have presented a model checking approach for the verification of data flow correctness in networks during concurrent updates of the network configuration. Key ingredients of the approach are Petri nets with transits, which superimpose the transit relation of data flows onto the flow relation of Petri nets, and Flow-LTL, which combines the specification of local data flows with the specification of global control. The model checking problem for Petri nets with transits and Flow-LTL specifications reduces to a circuit model checking problem. Our prototype tool implementation can verify and falsify realistic concurrent updates of software-defined networks with specifications like packet coherence.

In future work, we plan to extend this work to the synthesis of concurrent updates. Existing synthesis techniques use model checking as a subroutine to verify the correctness of the individual update steps [12,32]. We plan to study Flow-LTL specifications in the setting of Petri games [17], which describe the existence of controllers for asynchronous distributed processes. This would allow us to synthesize concurrent network updates without a central controller.

References

1. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: Vericon: towards verifying controller programs in software-defined networks. In: Proc. of PLDI (2014)
2. Berkeley Logic Synthesis and Verification Group: ABC: A system for sequential synthesis and verification, version 1.01 81030
3. Biere, A., Clarke, E.M., Raimi, R., Zhu, Y.: Verifiying safety properties of a power PC microprocessor using symbolic model checking without BDDs. In: Proc. of CAV (1999)
4. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Johannes Kepler University, Linz (2011)
5. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. of VMCAI (2011)
6. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A NICE way to test openflow applications. In: Proc. of NSDI (2012)
7. Casado, M., Foster, N., Guha, A.: Abstractions for software-defined networks. Commun. ACM **57**(10) (2014)
8. Cerný, P., Foster, N., Jagnik, N., McClurg, J.: Optimal consistent network updates in polynomial time. In: Proc. of DISC (2016)
9. Claessen, K., Eén, N., Sterin, B.: A circuit approach to LTL model checking. In: Proc. of FMCAD (2013)
10. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Proc. of POST (2014)
11. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proc. of FMCAD (2011)
12. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.T.: Network-wide configuration synthesis. In: Proc. of CAV (2017)
13. Engelfriet, J.: Branching processes of Petri nets. Acta Inf. **28**(6) (1991)
14. Esparza, J., Heljanko, K.: Unfoldings – A Partial-Order Approach to Model Checking. Springer (2008)
15. Finkbeiner, B., Gieseking, M., Hecking-Harbusch, J., Olderog, E.: Model checking data flows in concurrent network updates (full version). arXiv preprint arXiv:1907.11061 (2019)
16. Finkbeiner, B., Gieseking, M., Olderog, E.: Adam: Causality-based synthesis of distributed systems. In: Proc. of CAV (2015)
17. Finkbeiner, B., Olderog, E.: Petri games: Synthesis of distributed systems with causal memory. Inf. Comput. **253** (2017)
18. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Proc. of CAV (2015)
19. Förster, K., Mahajan, R., Wattenhofer, R.: Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In: Proc. of IFIP Networking (2016)

20. Foster, N., Harrison, R., Freedman, M.J., Monsanto, C., Rexford, J., Story, A., Walker, D.: Frenetic: a network programming language. In: Proc. of ICFP (2011)
21. Gieseking, M., Hecking-Harbusch, J.: AdamMC – A Model Checker for Petri Nets With Transits and Flow-LTL (2019). https://doi.org/10.6084/m9.figshare.8313344
22. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1. Springer (1992)
23. Jin, X., Liu, H.H., Gandhi, R., Kandula, S., Mahajan, R., Zhang, M., Rexford, J., Wattenhofer, R.: Dynamic scheduling of network updates. In: Proc. SIGCOMM (2014)
24. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-performance language-independent model checking. In: Proc. of TACAS (2015)
25. Katta, N.P., Rexford, J., Walker, D.: Incremental consistent updates. In: Proc. of HotSDN. ACM (2013)
26. Knight, S., Nguyen, H.X., Falkner, N., Bowden, R.A., Roughan, M.: The internet topology zoo. IEEE Journal on Selected Areas in Communications **29**(9) (2011)
27. Kordon, F., et al.: Complete Results for the 2019 Edition of the Model Checking Contest (2019)
28. Kreutz, D., Ramos, F.M.V., Veríssimo, P.J.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: A comprehensive survey. Proc. of the IEEE **103**(1) (2015)
29. Liu, H.H., Wu, X., Zhang, M., Yuan, L., Wattenhofer, R., Maltz, D.A.: zUpdate: updating data center networks with zero loss. In: Proc. of SIGCOMM (2013)
30. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P.B., King, S.T.: Debugging the data plane with anteater. SIGCOMM Comput. Commun. Rev. **41**(4) (2011)
31. Majumdar, R., Tetali, S.D., Wang, Z.: Kuai: A model checker for software-defined networks. In: Proc. of FMCAD (2014)
32. McClurg, J., Hojjat, H., Cerný, P.: Synchronization synthesis for network programs. In: Proc. of CAV (2017)
33. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G.M., Peterson, L.L., Rexford, J., Shenker, S., Turner, J.S.: Openflow: enabling innovation in campus networks. Computer Communication Review **38**(2) (2008)
34. McMillan, K.L.: Craig interpolation and reachability analysis. In: Proc. of SAS (2003)
35. Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D.: Composing software defined networks. In: Proc. of NSDI (2013)
36. Padon, O., Immerman, N., Karbyshev, A., Lahav, O., Sagiv, M., Shoham, S.: Decentralizing SDN policies. In: Proc. of POPL (2015)
37. Reisig, W.: Petri Nets: An Introduction. Springer (1985)
38. Reitblatt, M., Canini, M., Guha, A., Foster, N.: Fattire: declarative fault tolerance for software-defined networks. In: Proc. of HotSDN (2013)
39. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D.: Abstractions for network update. In: Proc. of SIGCOMM (2012)
40. Schmidt, K.: Lola: A low level analyser. In: Proc. of ICATPN (2000)
41. Thierry-Mieg, Y.: Symbolic model-checking using ITS-tools. In: Proc. of TACAS (2015)
42. Tsankov, P., Dashti, M.T., Basin, D.: Access control synthesis for physical spaces. In: Proc. of CSF (2016)
43. Wang, A., Moarref, S., Loo, B.T., Topcu, U., Scedrov, A.: Automated synthesis of reactive controllers for software-defined networks. In: Proc. of ICNP (2013)