AdamMC: A Model Checker for Petri Nets with Transits against Flow-LTL *

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. The correctness of networks is often described in terms of the individual data flow of components instead of their global behavior. In software-defined networks, it is far more convenient to specify the correct behavior of packets than the global behavior of the entire network. Petri nets with transits extend Petri nets and Flow-LTL extends LTL such that the data flows of tokens can be tracked. We present the tool AdamMC as the first model checker for Petri nets with transits against Flow-LTL. We describe how AdamMC can automatically encode concurrent updates of software-defined networks as Petri nets with transits and how common network specifications can be expressed in Flow-LTL. Underlying AdamMC is a reduction to a circuit model checking problem. We introduce a new reduction method that results in tremendous performance improvements compared to a previous prototype. Thereby, AdamMC can handle software-defined networks with up to 82 switches.

1 Introduction

In networks, it is difficult to specify correctness in terms of the global behavior of the entire system. Instead, the individual *flow* of components is far more convenient to specify correct behavior. For example, loop and drop freedom can be easily specified for the flow of each packet. Petri nets and LTL lack this local view. Petri nets with transits and Flow-LTL have been introduced to overcome this restriction [10]. A transit relation is introduced to follow the *flow* induced by tokens. *Flow-LTL* is a temporal logic to specify both the *local* flow of data and the *global* behavior of markings. The global behavior as in Petri nets and LTL is still important for maximality and fairness assumptions. In this paper, we present the tool AdamMC[3] as the first model checker for Petri nets with transits against Flow-LTL and its application to software-defined networking.

In Fig. 1, we present an example of a Petri net with transits that models the security check at an airport where passengers are checked by a security guard.

* 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).

[3] AdamMC is available online at https://uol.de/en/csd/adammc [12].

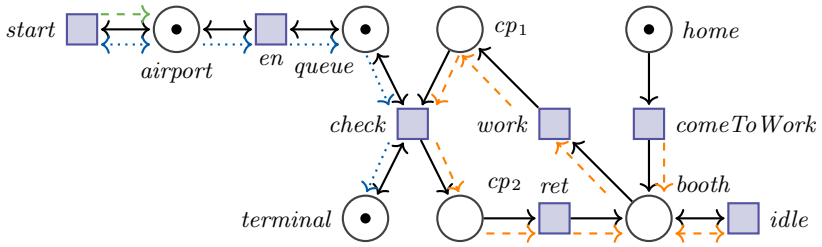

Fig. 1: Access control at an airport modeled as Petri net with transits. Colored arrows display the transit relation and define flow chains to model the passengers.

The number of passengers entering the airport is unknown in advance. Rather than introducing the complexity of an infinite number of tokens, we use a fixed number of tokens to model possibly infinitely many *flow chains*. This is done by the transit relation which is depicted with colored arrows.

The left-hand side of Fig. 1 models passengers who want to reach the terminal. There are three tokens in the places *airport*, *queue*, and *terminal*. Thus, transitions *start* and *en* are always enabled. Each firing of *start* creates a new flow chain as depicted by the green arrow. This models a new person arriving at the *airport*. Meanwhile, the double-headed blue arrow maintains all flow chains that are still in place *airport*. Passengers have to *en*ter the *queue* and wait until the security *check* is performed. Therefore, transition *en* continues every flow chain in *airport* to *queue*. Checking the passengers is carried out by transition *check* which becomes enabled if the security guard *work*s. Thus, passengers residing in *queue* have to wait until the guard *check*s them. Afterwards, they reach the *terminal*. The security guard is modeled on the right-hand side of Fig. 1. By firing *comeToWork* and thus moving the token in place *home*, her flow chain starts and she can repeatedly either *idle* or *work*, *check* passengers, and *ret*urn. Her transit relation is depicted in orange and models exactly one flow chain.

In Fig. 1, we define the checkpoints cp_1 and cp_2 and the *booth* as a security zone and require that passengers never enter the security zone and eventually reach the *terminal*. The flow formula $\varphi = \mathbb{A}(\mathit{airport} \rightarrow (\square\neg(\mathit{cp}_1 \vee \mathit{cp}_2 \vee \mathit{booth}) \wedge \diamondsuit \mathit{terminal}))$ specifies this. AdamMC verifies the example from Fig. 1 against the formula $\square\diamondsuit \mathit{check} \rightarrow \varphi$ specifying that if passengers are checked regularly then they cannot access the security zone and eventually reach the terminal.

In this paper, we present AdamMC as a full-fledged tool. First, AdamMC can handle Petri nets with transits and Flow-LTL formulas in general. Second, AdamMC has an input interface for a concurrent update and a software-defined network and encodes both of them as a Petri nets with transits. Common assumptions on fairness and requirements for network correctness are also provided as Flow-LTL formulas. This allows users of the tool to model check the correctness of concurrent updates and to prevent packet loss, routing loops, and network congestion. Third, AdamMC provides algorithms to check safe Petri nets against LTL with *both* places and transitions as atomic propositions which makes it especially easy to specify fairness and maximality assumptions.

The tool reduces the model checking problem for safe Petri nets with transits against Flow-LTL to the model checking problem for safe Petri nets against LTL. We develop the new *parallel approach* to check global and local behavior in parallel instead of sequentially. This approach yields a tremendous speed-up for a few local requirements and realistic fairness assumptions in comparison to the sequential approach of a previous prototype [10]. In general, the parallel approach has worst-case complexity inferior to the sequential approach even though the complexities of both approaches are the same when using only one flow formula.

As last step, AdamMC reduces the model checking problem of safe Petri nets against LTL to a circuit model checking problem. This is solved by ABC [2,4] with effective verification techniques like IC3 and bounded model checking. AdamMC verifies concurrent updates of software-defined networks with up to 38 switches (31 more than the prototype) and falsifies concurrent updates of software-defined networks with up to 82 switches (44 more than the prototype).

The paper is structured as follows: In Sec. 2, we recall Petri nets with transits and Flow-LTL. In Sec. 3, we outline the three application areas of AdamMC: checking safe Petri nets with transits against Flow-LTL, checking concurrent updates of software-defined networks against common assumptions and specifications, and checking safe Petri nets against LTL. In Sec. 4, we algorithmically encode concurrent updates of software-defined networks in Petri nets with transits. In Sec. 5, we introduce the parallel approach for the underlying circuit model checking problem. In Sec. 6, we present our experimental evaluation.

Further details can be found in the full paper [13].

2 Petri Nets With Transits and Flow-LTL

A safe *Petri net with transits* $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In, \Upsilon)$ [10] contains the set of *places* $\mathscr{P}$, the set of *transitions* $\mathscr{T}$, the *flow relation* $\mathscr{F} \subseteq (\mathscr{P} \times \mathscr{T}) \cup (\mathscr{T} \times \mathscr{P})$, and the *initial marking* $In \subseteq \mathscr{P}$ as in safe Petri nets [27]. In a *safe* Petri net, reachable markings contain at most one token per place. The *transit relation* Υ is for every transition $t \in \mathscr{T}$ of type $\Upsilon(t) \subseteq (pre^{\mathcal{N}}(t) \cup \{\triangleright\}) \times post^{\mathcal{N}}(t)$. With $p\ \Upsilon(t)\ q$, we define that firing transition t *transits* the flow in place p to place q. The symbol $\triangleright$ denotes a *start* and $\triangleright\ \Upsilon(t)\ q$ defines that firing transition t *starts* a new flow for the token in place q. Note that the transit relation can split, merge, and end flows. A sequence of flows leads to a *flow chain* which is a sequence of the current place and the fired outgoing transition. Thus, Petri nets with transits can describe both the global progress of tokens and the local flow of data.

Flow-LTL [10] extends Linear-time Temporal Logic (LTL) and uses places and transitions as atomic propositions. It introduces $\mathbb{A}$ as a new operator which uses LTL to specify the flow of data for *all* flow chains. For Fig. 1, the formula $\mathbb{A}(\mathit{booth} \rightarrow \Diamond\, \mathit{check})$ specifies that the guard performs at least one check. We call formulas starting with $\mathbb{A}$ *flow formulas*. Formulas around flow formulas specify the global progress of tokens in the form of markings and fired transitions to formalize maximality and fairness assumptions. These formulas are called *run formulas*. Often, Flow-LTL formulas have the form *run formula* $\rightarrow$ *flow formula*.

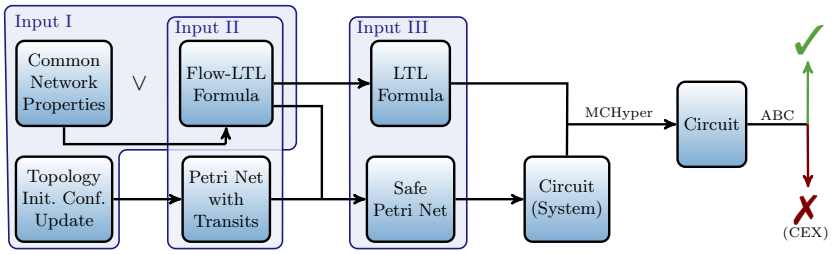

Fig. 2: Overview of the workflow of AdamMC: The application areas of the tool are given by three different input domains: software-defined network / Flow-LTL (Input I), Petri nets with transits / Flow-LTL (Input II), and Petri nets / LTL (Input III). AdamMC performs all unlabeled steps. MCHyper creates the final circuit which ABC checks to answer the initial model checking problem.

3 Application Areas

AdamMC consists of modules for three application areas: checking safe Petri nets with transits against Flow-LTL, checking concurrent updates of software-defined networks against common assumptions and specifications, and checking safe Petri nets against LTL. The general architecture and workflow of the model checking procedure is given in Fig. 2. AdamMC is based on the tool Adam [14].

Petri Nets with Transits Petri nets with transits follow the progress of tokens and the flow of data. Flow-LTL allows to specify requirements on both. For Petri nets with transits and Flow-LTL (Input II), AdamMC extends a parser for Petri nets provided by APT [30], provides a parser for Flow-LTL, and implements two reduction methods to create a safe Petri net and an LTL formula. The sequential approach is outlined in [10] and the parallel approach in Sec. 5.

Software-Defined Networks Concurrent updates of software-defined networks are the second application area of AdamMC. The tool automatically encodes an initially configured network topology and a concurrent update as a Petri net with transits. The concurrent update renews the forwarding table. We provide parsers for the *network topology*, the *initial configuration*, the *concurrent update*, and Flow-LTL (Input I). In Sec. 4, we present the creation of a Petri net with transits from the input and Flow-LTL formulas for *common network properties* like *connectivity*, *loop freedom*, *drop freedom*, and *packet coherence*.

Petri Nets AdamMC supports the model checking of safe Petri nets against LTL with both places *and* transitions as atomic propositions. It provides dedicated algorithms to check *interleaving-maximal* runs of the system. A run is interleaving-maximal if a transition is fired whenever a transition is enabled. Furthermore, AdamMC allows a concurrent view on runs and can check *concurrency-maximal* runs which demand that each subprocess of the system has to progress maximally rather than only the entire system. State-of-the-art tools like LoLA [32] and ITS-Tools [29] are restricted to interleaving-maximal runs and places as atomic propositions. For Petri net model checking (Input III), we allow Petri nets in APT and PNML format as input and provide a parser for LTL formulas.

The construction of the circuit in Aiger format [3] is defined in [11]. MCHyper [15] is used to create a circuit from a given circuit and an LTL formula. This circuit is given to ABC [2,4] which provides a toolbox of modern hardware verification algorithms like IC3 and bounded model checking to decide the initial model checking question. As output for all three modules, AdamMC transforms a possible counterexample (CEX) from ABC into a counterexample to the Petri net (with transits) and visualizes the net with Graphviz and the dot language [9]. When no counterexample exists, AdamMC verified the input successfully.

4 Verifying Updates of Software Defined Networks

We show how AdamMC can check concurrent updates of realistic examples from software-defined networking (SDN) against typical specifications [19]. SDN [25,6] separates the *data plane* for forwarding packets and the *control plane* for the routing configuration. A central controller initiates updates which can cause problems like routing loops or packet loss. AdamMC provides an input interface to automatically encode software-defined networks and concurrent updates of their configuration as Petri nets with transits. The tool checks requirements like loop and drop freedom to find erroneous updates before they are deployed.

4.1 Network Topology, Configurations, and Updates

A *network topology* T is an undirected graph $T = (Sw, Con)$ with *switches* as vertices and *connections* between switches as edges. Packets enter the network at *ingress* switches and they leave at *egress* switches. *Forwarding* rules are of the form `x.fwd(y)` with $\mathtt{x}, \mathtt{y} \in Sw$. A concurrent *update* has the following syntax:

switch update	::= `upd(x.fwd(y/z))` | `upd(x.fwd(y/-))` | `upd(x.fwd(-/z))`
sequential update	::= `(`update `>>` update `>>` ... `>>` update`)`
parallel update	::= `(`update `||` update `||` ... `||` update`)`
update	::= switch update | sequential update | parallel update

where a switch update can renew the forwarding rule of switch `x` from switch `z` to switch `y`, introduce a new forwarding rule from switch `x` to switch `y`, or remove an existing forwarding rule from switch `x` to switch `z`.

4.2 Data Plane and Control Plane as Petri Net with Transits

For a network topology $T = (Sw, Con)$, a set of *ingress* switches, a set of *egress* switches, an initial *forwarding* table, and a concurrent *update*, we show how data and control plane are encoded as Petri net with transits. Switches are modeled by tokens remaining in corresponding places `s` whereas the flow of packets is modeled by the transit relation Υ. Specific transitions $i_{\mathtt{s}}$ model ingress switches where new data flows begin. Tokens in places of the form `x.fwd(y)` configure the forwarding. Data flows are extended by firing transitions `(x,y)` corresponding to configured forwarding without moving any tokens. Thus, we model any order

of newly generated packets and their forwarding. Assuming that each existing direction of a connection between two switches is explicitly given in *Con*, we obtain Algorithm 1 which calls Algorithm 2 to obtain the control plane.

For the *update*, let SwU be the set of switch updates in it, SeU the set of sequential updates in it, and PaU the set of parallel updates in it. Depending on *update*'s type, it is also added to the respective set. The subnet for the *update* has an empty transit relation but moves tokens from and to places of the form `x.fwd(y)`. Tokens in these places correspond to the forwarding table. The order of the switch updates is defined by the nesting of sequential and parallel updates. The *update* is realized by a specific token moving through unique places of the form $u^s, u^f, s^s, s^f, p^s, p^f$ for start and finish of each switch update $u \in SwU$, each sequential update $s \in SeU$, and each parallel update $p \in PaU$. A parallel update temporarily increases the number of tokens and reduces it upon completion to one. Algorithm 2 defines the update behavior between start and finish places and connects finish and start places depending on the subexpression structure.

input : $T = (Sw, Con)$, *ingress*, *forwarding*, *update*
output: Petri net with transits $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In, \Upsilon)$ for *update* of topology T with *ingress* and *forwarding*

create empty $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In, \Upsilon)$;
for switch $\mathtt{s} \in Sw$ **do**
  add place $\mathtt{s}$ to $\mathscr{P}$;
  add place $\mathtt{s}$ to In;
  if $\mathtt{s} \in$ *ingress* **then**
    add transition $i_{\mathtt{s}}$ to $\mathscr{T}$;
    add $\mathtt{s}$ to $pre(i_{\mathtt{s}})$, $post(i_{\mathtt{s}})$;
    add creating data flow $\triangleright\ \Upsilon(i_{\mathtt{s}})\ \mathtt{s}$ to Υ;
    add maintaining data flow $\mathtt{s}\ \Upsilon(i_{\mathtt{s}})\ \mathtt{s}$ to Υ;
for connection $(\mathtt{x}, \mathtt{y}) \in Con$ **do**
  add place `x.fwd(y)` to $\mathscr{P}$;
  if `x.fwd(y)` $\in$ *forwarding* **then**
    add place `x.fwd(y)` to In;
  add transition $(\mathtt{x}, \mathtt{y})$ to $\mathscr{T}$;
  add $\mathtt{x}$, $\mathtt{y}$, `x.fwd(y)` to $pre((\mathtt{x}, \mathtt{y}))$, $post((\mathtt{x}, \mathtt{y}))$;
  add connecting data flow $\mathtt{x}\ \Upsilon((\mathtt{x}, \mathtt{y}))\ \mathtt{y}$ to Υ;
  add maintaining data flow $\mathtt{y}\ \Upsilon((\mathtt{x}, \mathtt{y}))\ \mathtt{y}$ to Υ;
$\mathcal{N}$ = call Algorithm 2 with T, *update*, $\mathcal{N}$ as input;
add place $update^s$ to In;

Algorithm 1: Data plane

input : $T = (Sw, Con)$, *update*, $\mathcal{N}$
output: $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In, \Upsilon)$
for switch update $u \in SwU$ **do**
  // $u =$ `upd(x.fwd(y/z))`
  add places u^s, u^f to $\mathscr{P}$;
  add transition u to $\mathscr{T}$;
  add u^s to $pre(u)$, u^f to $post(u)$;
  if $\mathtt{z} \neq$ `-` **then**
    add `x.fwd(z)` to $pre(u)$;
  if $\mathtt{y} \neq$ `-` **then**
    add `x.fwd(y)` to $post(u)$;
for sequential update $s \in SeU$ **do**
  // $s = [s_1, ..., s_i, ..., s_{|s|}]$
  add places s^s, s^f to $\mathscr{P}$;
  for $i \in \{0, ..., |s|\}$ **do**
    add transition s^i to $\mathscr{T}$;
    if $i == 0$ **then**
      add s^s to $pre(s^i)$;
    else
      add s^f_i to $pre(s^i)$;
    if $i = |s|$ **then**
      add s^f to $post(s^i)$;
    else
      add s^s_{i+1} to $post(s^i)$;
for parallel update $p \in PaU$ **do**
  add places p^s, p^f to $\mathscr{P}$;
  add transitions p^o, p^c to $\mathscr{T}$;
  add p^s to $pre(p^o)$, p^f to $post(p^c)$;
  for sub-update u_i of p **do**
    add u^s_i to $post(p^o)$, u^f_i to $pre(p^c)$;

Algorithm 2: Control plane

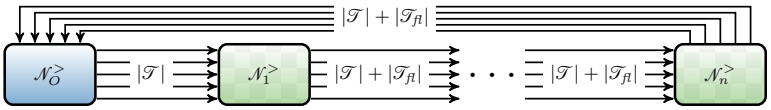

Fig. 3: Overview of the *sequential approach*: Each firing of a transition of the original net is split into first firing a transition in the subnet for the run formula and subsequently firing a transition in each subnet tracking a flow formula. The constructed LTL formula skips the additional steps with until operators.

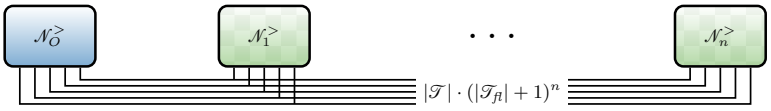

Fig. 4: Overview of the *parallel approach*: The n subnets are connected such that for every transition $t \in \mathcal{T}$ there are $(|\Upsilon(t)| + 1)^n$ transitions, i.e., there is one transition for every combination of which transit of t (or none) is tracked by which subnet. We use until operators in the constructed LTL formula to only skip steps not involving the tracking of the guessed chain in the flow formula.

4.3 Assumptions and Requirements

We use the run formula $\Diamond\Box \mathit{pre}(t) \rightarrow \Box\Diamond t$ to assume weak fairness for every transition t in our encoding $\mathcal{N}$. Transitions, which are always enabled after some point, are ensured to fire infinitely often. Thus, packets are eventually forwarded and the routing table is eventually updated. We use flow formulas to test specific requirements for all packets. Connectivity ($\mathbb{A}(\Diamond \bigvee_{s \in \mathit{egress}} s)$) ensures that all packets reach an egress switch. Packet coherence ($\mathbb{A}(\Box(\bigvee_{\mathtt{s} \in \mathit{initial}} \mathtt{s}) \vee \Box(\bigvee_{\mathtt{s} \in \mathit{final}} \mathtt{s}))$) tests that packets are either routed according to the initial or final configuration. Drop freedom ($\mathbb{A}\Box(\bigwedge_{\mathtt{e} \in \mathit{egress}} \neg \mathtt{e} \rightarrow \bigvee_{f \in \mathit{Con}} f)$) forbids dropped packets whereas loop freedom ($\mathbb{A}\Box(\bigwedge_{\mathtt{s} \in \mathit{Sw} \setminus \mathit{egress}} \mathtt{s} \rightarrow (\mathtt{s}\,\mathsf{U}\,\Box\neg\mathtt{s}))$) forbids routing loops. We combine run and flow formula into *fairness* $\rightarrow$ *requirement*.

5 Algorithms and Optimizations

Central to model checking a Petri net with transits $\mathcal{N}$ against a Flow-LTL formula φ is the reduction to a safe Petri net $\mathcal{N}^>$ and an LTL formula $\varphi^>$. The infinite state space of the Petri net with transits due to possibly infinitely many flow chains is reduced to a finite state model. The key idea is to guess and track a violating flow chain for each flow subformula $\mathbb{A}\,\psi_i$, for $i \in \{1, \ldots, n\}$, and to only once check the equivalent future of flow chains merging into a common place.

AdamMC provides two approaches for this reduction: Fig. 3 and Fig. 4 give an overview of the *sequential* approach and the *parallel* approach, respectively. Both algorithms create one subnet $\mathcal{N}_i^>$ for each flow subformula $\mathbb{A}\,\psi_i$ to track the corresponding flow chain and have one subnet $\mathcal{N}_O^>$ to check the run part of the formula. The places of $\mathcal{N}_O^>$ are copies of the places in $\mathcal{N}$ such that the current state of the system can be memorized. The subnets $\mathcal{N}_i^>$ also consist of the

Table 1: Overview of optimization parameters of AdamMC: The three reduction steps depicted in the first column can each be executed by different algorithms. The first step allows to combine the optimizations of the first and second row.

1) Petri Net with Transits $\rightsquigarrow$ Petri Net	sequential		parallel	
	inhibitor	act. token	inhibitor	act. token
2) Petri Net $\rightsquigarrow$ Circuit	explicit		logarithmic	
3) Circuit $\rightsquigarrow$ Circuit	gate optimizations			

original places of $\mathcal{N}$ but only use one token (initially residing on an additional place) to track the current state of the considered flow chain. The approaches differ in how these nets are connected to obtain $\mathcal{N}^{>}$.

Sequential Approach The places in each subnet $\mathcal{N}_i^{>}$ are connected with one transition for each transit ($\mathcal{T}_{fl} = \bigcup_{t \in \mathcal{T}} \Upsilon(t)$). An additional token iterates sequentially through the subnets to activate or deactivate the subnet. This allows each subnet to track a flow chain corresponding to firing a transition in $\mathcal{N}_O^{>}$. The formula $\varphi^{>}$ takes care of these additional steps by means of the until operator: In the run part of the formula, all steps corresponding to moves in a subnet $\mathcal{N}_i^{>}$ are skipped and, for each subformula $\mathbb{A}\,\psi_i$, all steps are skipped until the next transition of the corresponding subnet is fired which transits the tracked flow chain. This technique results in a polynomial increase of the size of the Petri net and the formula: $\mathcal{N}^{>}$ has $\mathcal{O}(|\mathcal{N}| \cdot n + |\mathcal{N}|)$ places and $\mathcal{O}(|\mathcal{N}|^3 \cdot n + |\mathcal{N}|)$ transitions and the size of $\varphi^{>}$ is in $\mathcal{O}(|\mathcal{N}|^3 \cdot n \cdot |\varphi| + |\varphi|)$. We refer to [11] for formal details.

Parallel Approach The n subnets are connected such that the current chain of each subnet is tracked simultaneously while firing an original transition $t \in \mathcal{T}$. Thus, there are $(|\Upsilon(t)| + 1)^n$ transitions. Each of these transitions stands for exactly one combination of which subnet is tracking which (or no) transit. Hence, firing one transition of the original net is directly tracked in one step for all subnets. This significantly reduces the complexity of the run part of the constructed formula, since no until operator is needed to skip sequential steps. A disjunction over all transitions corresponding to an original transition suffices to ensure correctness of the construction. Transitions and next operators in the flow parts of the formula still have to be replaced by means of the until operator to ensure that the next step of the tracked flow chain is checked at the corresponding step of the global timeline of $\varphi^{>}$. In general, the parallel approach results in an exponential blow-up of the net and the formula: $\mathcal{N}^{>}$ has $\mathcal{O}(|\mathcal{N}| \cdot n + |\mathcal{N}|)$ places and $\mathcal{O}(|\mathcal{N}|^{3n} + |\mathcal{N}|)$ transitions and the size of $\varphi^{>}$ is in $\mathcal{O}(|\mathcal{N}|^{3n} \cdot |\varphi| + |\varphi|)$. For the practical examples, however, the parallel approach allows for model checking Flow-LTL with few flow subformulas with a tremendous speed-up in comparison to the sequential approach. Formal details are in the full version of the paper [13].

Optimizations Various optimizations parameters can be applied to the model checking routine described in Sec. 3 to tweak the performance. Table 1 gives an overview of the major parameters. We found that the versions of the sequential and the parallel approach with inhibitor arcs to track flow chains are generally faster than the versions without. Furthermore, the reduction step from a Petri net into a circuit with logarithmically encoded transitions had oftentimes better

Table 2: We compare the explicit and logarithmic encoding of the sequential approach with the parallel approach. The results are the average over five runs from an Intel i7-2700K CPU with 3.50 GHz, 32 GB RAM, and a timeout (TO) of 30 minutes. The runtimes are given in seconds.

			expl. enc.		log. enc.		parallel appr.	
T / F	Network	$\#Sw$	Alg.	Time $\models$	Alg.	Time $\models$	Alg.	Time $\models$
T	Arpanet196912	4	IC3	12.08 ✓	IC3	9.89 ✓	IC3	**2.18** ✓
T	Napnet	6	IC3	146.49 ✓	IC3	96.06 ✓	IC3	**4.75** ✓
	...			...		...		...
T	Heanet	7	IC3	806.81 ✓	IC3	84.62 ✓	IC3	**30.30** ✓
T	HiberniaIreland	7	-	TO ?	-	TO ?	IC3	**26.58** ✓
T	Arpanet19706	9	-	TO ?	IC3	362.21 ✓	IC3	**11.33** ✓
T	Nordu2005	9	-	TO ?	-	TO ?	IC3	**12.67** ✓
	...			...		...		...
T	Fatman	17	-	TO ?	IC3	1543.34 ✓	IC3	**162.17** ✓
	...			...		...		...
T	Myren	37	-	TO ?	-	TO ?	IC3	**1309.23** ✓
T	KentmanJan2011	38	-	TO ?	-	TO ?	IC3	**1261.32** ✓
F	Arpanet196912	4	BMC3	2.18 ✗	BMC3	**1.85** ✗	BMC3	1.97 ✗
F	Napnet	6	BMC2	4.17 ✗	BMC2	5.22 ✗	BMC3	**1.48** ✗
	...			...		...		...
F	Fatman	17	BMC3	168.78 ✗	BMC3	169.82 ✗	BMC3	**6.72** ✗
	...			...		...		...
F	Belnet2009	21	BMC2	1146.26 ✗	BMC2	611.81 ✗	BMC3	**24.26** ✗
	...			...		...		...
F	KentmanJan2011	38	BMC3	167.92 ✗	BMC3	86.44 ✗	BMC2	**9.35** ✗
	...			...		...		...
F	Latnet	69	-	TO ?	-	TO ?	BMC2	**209.20** ✗
F	Ulaknet	82	-	TO ?	-	TO ?	BMC2	**1043.74** ✗
Sum of runtimes (in hours):				82.99		79.15		30.31
Nb of TOs (of 230 exper.):				146		138		6

performance than the same step with explicitly encoded transitions. However, several possibilities to reduce the number of gates of the created circuit worsened the performance of some benchmark families and improved the performance of others. Consequently, all parameters are selectable by the user and a script is provided to compare different settings. An overview of the selectable optimization parameters can be found in the documentation of AdamMC [12]. Our main improvement claims can be retraced by the case study in Sec. 6.

6 Evaluation

We conduct a case study based on SDN with a corresponding artifact [16]. The performance improvements of AdamMC compared to the prototype [10] are summarized in Table 2. For realistic software-defined networks [19], one ingress and one egress switch are chosen at random. Two forwarding tables between the

two switches and an update from the first to the second configuration are chosen at random. AdamMC verifies that the update maintained *connectivity* between ingress and egress switch. The results are depicted in rows starting with T. For rows starting with F, we required *connectivity* of a random switch which is not in the forwarding tables. AdamMC falsified this requirement for the update.

The prototype implementation based on an *explicit encoding* can verify updates of networks with 7 switches and falsify updates of networks with 38 switches. We optimize the explicit encoding to a *logarithmic encoding* and the number of switches for which updates can be verified increases to 17. More significantly, the *parallel approach* in combination with the logarithmic encoding leads to tremendous performance gains. The performance gains of an approach with inferior worst-case complexity are mainly due to the smaller complexity of the LTL formula created by the reduction. The encoding of SDN requires fairness assumptions for each transition. These assumptions (encoded in the run part of the formula) experience a blow-up with until operators by the sequential approach but only need a disjunction in the parallel approach. Hence, the size of networks for which AdamMC can verify updates increases to 38 switches and the size for which it can falsify updates increases to 82 switches. For rather small networks, the tool needs only a few seconds to verify and falsify updates which makes it a great option for operators when updating networks.

7 Related Work

We refer to [21] for an introduction to SDN. Solutions for correctness of updates of software-defined networks include *consistent updates* [28,7], *dynamic scheduling* [17], and *incremental updates* [18]. Both explicit and SMT-based model checking [5,23,22,31,1,26] is used to verify software-defined networks. Closest to our approach are models of networks as Kripke structures to use model checking for synthesis of correct network updates [8,24]. The model checking subroutine of the synthesizer assumes that each packet sees at most one updated switch. Our model checking routine does not make such an assumption.

There is a significant number of model checking tools (e.g., [32,29]) for Petri nets and an annual model checking contest [20]. AdamMC is restricted to safe Petri nets whereas other tools can handle bounded and colored Petri nets. At the same time, only AdamMC accepts LTL formulas with places *and* transitions as atomic propositions. This is essential to express fairness in our SDN encoding.

8 Conclusion

We presented the tool AdamMC with its three application domains: checking safe Petri nets with transits against Flow-LTL, checking concurrent updates of software-defined networks against common assumptions and specifications, and checking safe Petri nets against LTL. New algorithms allow AdamMC to model check software-defined networks of realistic size: it can verify updates of networks with up to 38 switches and can falsify updates of networks with up to 82 switches.

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: Proceedings of PLDI. pp. 282–293 (2014), http://doi.acm.org/10.1145/2594291.2594317
2. Berkeley Logic Synthesis and Verification Group: ABC: A system for sequential synthesis and verification, http://www.eecs.berkeley.edu/~alanmi/abc/, version 1.01 81030
3. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. rep. (2011)
4. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Proceedings of CAV. pp. 24–40 (2010), https://doi.org/10.1007/978-3-642-14295-6_5
5. Canini, M., Venzano, D., Peresíni, P., Kostic, D., Rexford, J.: A NICE way to test openflow applications. In: Proceedings of NSDI. pp. 127–140 (2012), https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini
6. Casado, M., Foster, N., Guha, A.: Abstractions for software-defined networks. Commun. ACM **57**(10), 86–95 (2014), http://doi.acm.org/10.1145/2661061.2661063
7. Cerný, P., Foster, N., Jagnik, N., McClurg, J.: Optimal consistent network updates in polynomial time. In: Proceedings of DISC. pp. 114–128 (2016), https://doi.org/10.1007/978-3-662-53426-7_9
8. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.T.: Network-wide configuration synthesis. In: Proceedings of CAV. pp. 261–281 (2017), https://doi.org/10.1007/978-3-319-63390-9_14
9. Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C., Woodhull, G.: Graphviz and dynagraph - static and dynamic graph drawing tools. In: Graph Drawing Software, pp. 127–148. Springer (2004), https://doi.org/10.1007/978-3-642-18638-7_6
10. Finkbeiner, B., Gieseking, M., Hecking-Harbusch, J., Olderog, E.: Model checking data flows in concurrent network updates. In: Proceedings of ATVA. pp. 515–533 (2019), https://doi.org/10.1007/978-3-030-31784-3_30
11. Finkbeiner, B., Gieseking, M., Hecking-Harbusch, J., Olderog, E.: Model checking data flows in concurrent network updates (full version). Tech. rep. (2019), http://arxiv.org/abs/1907.11061
12. Finkbeiner, B., Gieseking, M., Hecking-Harbusch, J., Olderog, E.: AdamMC – A Model Checker for Petri Nets with Transits against Flow-LTL. University of Oldenburg and Saarland University. https://uol.de/en/csd/adammc (2020)
13. Finkbeiner, B., Gieseking, M., Hecking-Harbusch, J., Olderog, E.: AdamMC: A model checker for Petri nets with transits against Flow-LTL (full version). Tech. rep. (2020), https://arxiv.org/abs/2005.07130
14. Finkbeiner, B., Gieseking, M., Olderog, E.: Adam: Causality-based synthesis of distributed systems. In: Proceedings of CAV. pp. 433–439 (2015), https://doi.org/10.1007/978-3-319-21690-4_25
15. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Proceedings of CAV. pp. 30–48 (2015), https://doi.org/10.1007/978-3-319-21690-4_3
16. Gieseking, M., Hecking-Harbusch, J.: AdamMC: A Model Checker for Petri Nets with Transits against Flow-LTL (Artifact) (2020), https://doi.org/10.6084/m9.figshare.11676171
17. Jin, X., Liu, H.H., Gandhi, R., Kandula, S., Mahajan, R., Zhang, M., Rexford, J., Wattenhofer, R.: Dynamic scheduling of network updates. In: Proceedings of SIGCOMM. pp. 539–550 (2014), https://doi.org/10.1145/2619239.2626307

18. Katta, N.P., Rexford, J., Walker, D.: Incremental consistent updates. In: Proceedings of HotSDN. pp. 49–54 (2013), https://doi.org/10.1145/2491185.2491191
19. 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), 1765–1775 (2011), https://doi.org/10.1109/JSAC.2011.111002
20. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amparore, E., Beccuti, M., Berthomieu, B., Ciardo, G., Dal Zilio, S., Liebke, T., Li, S., Meijer, J., Miner, A., Srba, J., Thierry-Mieg, Y., van de Pol, J., van Dirk, T., Wolf, K.: Complete Results for the 2019 Edition of the Model Checking Contest. http://mcc.lip6.fr/2019/results.php (April 2019)
21. Kreutz, D., Ramos, F.M.V., Veríssimo, P.J.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: A comprehensive survey. Proceedings of the IEEE **103**(1), 14–76 (2015), https://doi.org/10.1109/JPROC.2014.2371999
22. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, B., King, S.T.: Debugging the data plane with anteater. In: Proceedings ofSIGCOMM. pp. 290–301 (2011), https://doi.org/10.1145/2018436.2018470
23. Majumdar, R., Tetali, S.D., Wang, Z.: Kuai: A model checker for software-defined networks. In: Proceedings of FMCAD. pp. 163–170 (2014), https://doi.org/10.1109/FMCAD.2014.6987609
24. McClurg, J., Hojjat, H., Cerný, P.: Synchronization synthesis for network programs. In: Proceedings of CAV. pp. 301–321 (2017), https://doi.org/10.1007/978-3-319-63390-9_16
25. 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), 69–74 (2008), http://doi.acm.org/10.1145/1355734.1355746
26. Padon, O., Immerman, N., Karbyshev, A., Lahav, O., Sagiv, M., Shoham, S.: Decentralizing SDN policies. In: Proceedings of POPL. pp. 663–676 (2015), https://doi.org/10.1145/2676726.2676990
27. Reisig, W.: Petri Nets: An Introduction. Springer (1985), https://doi.org/10.1007/978-3-642-69968-9
28. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D.: Abstractions for network update. In: Proceedings of SIGCOMM. pp. 323–334 (2012), http://doi.acm.org/10.1145/2342356.2342427
29. Thierry-Mieg, Y.: Symbolic model-checking using ITS-tools. In: Proceedings of TACAS. pp. 231–237 (2015), https://doi.org/10.1007/978-3-662-46681-0_20
30. University of Oldenburg: APT – Analyse von Petri-Netzen und Transitionssystemen. https://github.com/CvO-Theory/apt (2012)
31. Wang, A., Moarref, S., Loo, B.T., Topcu, U., Scedrov, A.: Automated synthesis of reactive controllers for software-defined networks. In: Proceedings of ICNP. pp. 1–6 (2013), https://doi.org/10.1109/ICNP.2013.6733666
32. Wolf, K.: Petri net model checking with LoLA 2. In: Proceedings of PETRI NETS. pp. 351–362 (2018), https://doi.org/10.1007/978-3-319-91268-4_18