Automatic Synthesis of Transiently Correct Network Updates via Petri Games

Martin Didriksen, Peter G. Jensen, Jonathan F. Jønler, Andrei-Ioan Katona, Sangey D.L. Lama, Frederik B. Lottrup, Shahab Shajarat, and Jiří Srba

Department of Computer Science, Aalborg University, Denmark

Abstract. As software-defined networking (SDN) is growing increasingly common within the networking industry, the lack of accessible and reliable automated methods for updating network configurations becomes more apparent. Any computer network is a complex distributed system and changes to its configuration may result in policy violations during the transient phase when the individual routers update their forwarding tables. We present an approach for automatic synthesis of update sequences that ensures correct network functionality throughout the entire update phase. Our approach is based on a novel translation of the update synthesis problem into a Petri game and it is implemented on top of the open-source model checker TAPAAL. On a large benchmark of synthetic and real-world network topologies, we document the efficiency of our approach and compare its performance with state-of-the-art tool NetSynth. Our experiments show that for several networks with up to thousands of nodes, we are able to outperform NetSynth's update schedule generation.

1 Introduction

Modern computer networks are met with increasing demands on scalability, security, reliability and performance. This stipulates the need of frequently updating the network configurations in order to adapt to changes in flow demands, link failures and other disturbances. The complexity of current networks shows the limitation of the traditional manual network maintenance as the risks of introducing faulty behaviour and security leaks become too high. The software defined networking (SDN) paradigm [2] is a recent methodology that aims to combat the increased complexity of network operation by centralizing the network control and hence allowing for fully automatic updates of network configurations. This enables the option of dynamically updating networks with increased frequency in order to optimize their performance, but it also requires a reliable way to govern and schedule updates, so that disruption of service due to forwarding loops or blackholes and security leaks when e.g. a critical firewall is bypassed, can be avoided.

Even though the initial and final configurations are correct and satisfy a number of desirable properties like reachability and waypointing (before a packet leaves the network a certain router, e.g. a firewall, must be visited), there is no

guarantee that any transient configuration, where individual routers are updated one by one, preserves the required policies. The update synthesis problem [10] asks, in which order to update the routers in the network so that at any moment there is never any policy violation.

As our first contribution, we propose to translate the update synthesis problem into a two-player Petri game between the controller and the environment. The objective of the controller is to reach the updated network routing from the initial one by sequentially scheduling the updates of the individual switches. The environment can at any time interrupt the construction of the update sequence and initialize a check on whether the current partially constructed update sequence satisfies the given security policies. If the policies are satisfied, the controller is the winner, otherwise the environment wins. A winning strategy for the controller then defines a transiently correct update sequence.

As a second contribution, we implement the Petri game translation on top of the open-source model checker TAPAAL [6,12] and update its game engine with efficient state-space exploration strategies for solving the game synthesis problem. Our fully automated tool chain accepts the descriptions of network topologies and the initial and final routings as a JSON file, together with policy properties that include loop-freedom, reachability and waypointing. The tool then outputs that either there does not exist any transiently correct update sequence or it synthesizes such a sequence.

Finally, we conduct a number of experiments on both synthetic and real-world benchmarks and compare the performance of our approach with state-of-the-art tool NetSynth [21] that relies on counterexample-guided search and incremental model checking techniques, and allows for the use of different model checkers including NuSMV [5] as its backend engine. The results confirm that on two, out of the three scalable synthetic networks, we obtain several orders of magnitude speed up. In one case where the synthetic network shares as many possible links in both the initial and final routing, our method is performing slower. Experiments on the real-world Internet topologies, where the routings are constructed using the common method based on shortest paths (see e.g. the Equal-Cost-MultiPath (ECMP) [11] or the Open Shortest Path First (OSPF) [22] routing protocols), demonstrate that both tools are able to solve smaller instances of the update synthesis problem below once second, however, for the larger instances our method wins by a clear margin. As an additional contribution to the research community, we also provide a publicly available reproducibility package [7] including both the code as well as all experimental data.

Related Work. The work on updates in SDN is heavily influenced by the work by Reitblatt et al. [24] that defines the per-packet and per-flow consistency. In per-packet consistency, each packet traverses the network within at most one stable configuration, whereas per-flow guarantees that all packets in a flow traverse the network in the same configuration. The per-packet consistency, which is also the main focus of our work, inspired further research in this area [3,17,20]. In particular, Mahajan and Wattenhofer [20] suggest an approach that eliminates the use of packet header rewriting and the expensive two-phase update. They devise

a solution that preserves loop-freedom with weak consistency by examining the dependencies of switches in a network and conclude that half of the updates with around 100 switches only depended on zero or a single critical switch update and in 90% of the cases, updates are only dependent on at most three switches, in contrast to Reitblatt et al. [24] who rely on updating all switches. Their work has since then been refined and extended to support more properties [18,1], including waypointing [19]. However, it is known that the update synthesis problem with waypointing and loop-freedom becomes NP-complete [18] (for a detailed complexity overview see e.g. [10]). More recently, Nate Foster et. al. [21] introduce a specialized incremental model checker NetSynth that automatically synthesises correct update sequences from LTL specifications. In [21] the authors argue that their tool is outperforming other existing approaches on a variety of network topologies for ensuring reachability and waypointing policies. NetSynth essentially performs a (heuristic) search through all possible update sequences and relies on the assumption that the routings are loop-free. Our approach can verify also the presence of loops and it uses the general concept of two-player games instead of the explicit enumeration of all possible update sequences. Another approach that allows to present updates in concurrent steps is given in [25].

A recent work by Christensen et. al. [4] introduces the tool Latte that models the problem as a timed-arc colored Petri net and its main focus is on reducing the delays between the updates of the individual routers. While it extends the analysis with timing aspects, their work relies on obtaining a correct update sequence from third-party tools (NetSynth in their case) and as such it does not solve the synthesis problem and focuses purely on the timing optimization aspects and the discovery of possible concurrent updates. Another line of work by Finkbeiner et. al. [9,8] focuses on verifying concurrent network updates against Flow-LTL specifications, using Petri nets extended with transits as the underlying modelling formalism and circuit model checking as the backend engine. The experiments show that their tool can verify in minutes networks up to a hundred of routers, however, similarly as Latte [4], they can only verify but not synthesise update sequences. To the best of our knowledge, our approach is the first one that employs the game semantics of Petri nets and allows hence for fully automate update synthesis as well as the reuse of generic game model checkers like TAPAAL. As such, the generated update sequences can be then further optimized, e.g. for the timing and concurrent aspects, by the above mentioned approaches.

2 Update Synthesis

We shall now formalize the notion of a network and routing in a network, define some essential routing properties and formulate the update synthesis problem.

Definition 1 (Network). *A* network *is a directed graph $G = (V, E)$ where V is a finite set of nodes (switches), and $E \subseteq V \times V$ is a set of edges (links) such that $(v, v) \notin E$ for all $v \in V$.*

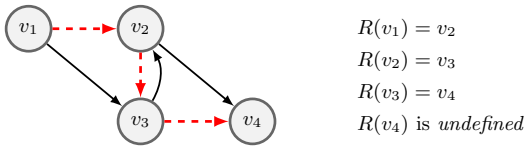


Fig. 1: A network with routing R depicted by the red dashed arrows (the black arrows represent existing links that are not used in R)

A network defines the set of links that connect the switches. For a given packet type (given by its header and usually determined by its destination), each switch contains a forwarding table defining the next hop. A switch contains this information for all different packet types and we project on a certain type in order to define its routing in a network. For the rest of this section, let $G = (V, E)$ be a fixed network.

Definition 2 (Routing). *A* routing *in G is a partial function $R : V \hookrightarrow V$ such that $(u, R(u)) \in E$ for all $u \in V$ where $R(u)$ is defined.*

Consider the network in Figure 1 with a routing R, indicated by the dashed edges. The routing naturally defines a path, which is a (unique) sequence of next hops. A path can be either infinite or finite. Any infinite path, or finite path that ends in a node with undefined next hop, is called a maximal path.

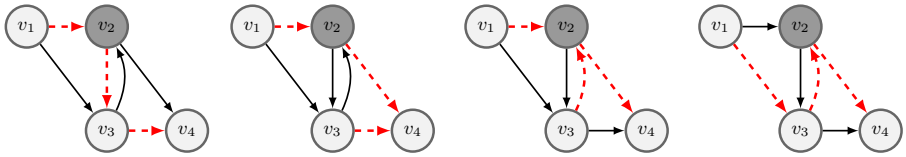
Definition 3 (Path). *A* path *π under a routing R is a sequence of nodes $v_1v_2...v_n... \in V^* \cup V^\omega$ such that $R(v_i) = v_{i+1}$ for all i. A* maximal path *is either an infinite path or a finite path that ends in a node v where $R(v)$ is undefined.*

Our example in Figure 1 contains the maximal path $v_1v_2v_3v_4$. This path demonstrates reachability between v_1 and v_4 and the routing does not contain any infinite path. Moreover, the path starting in v_1 contains the switch v_2 before v_4 is reached. A switch with this property is called a waypoint. We shall now formally define these basic properties of a routing function.

Definition 4 (Routing Properties). *Let u, v and w be three different nodes in a network. A routing R satisfies*

- *the* reachability *property reach(u, v) if there is a path $\pi = v_1v_2...v_n$ under R such that $v_1 = u$ and $v_n = v$,*
- *the* waypointing *property wp(u, v, w) if for every path $\pi = v_1v_2...v_n$ under R where $v_1 = u$ and $v_n = v$ there exists an i, $1 < i < n$, such that $v_i = w$, and*
- *the* loop-freedom *property loopfree(u) if the maximal path under R starting in u is finite.*

We shall note that (i) the waypointing property $wp(u, v, w)$ is trivially satisfied whenever there is no path under R from u to v, e.g. in our running example the property $wp(v_3, v_1, v_4)$ holds and (ii) any infinite path under a given routing must form a loop after some finite initial prefix (as the number of nodes is finite).



(a) Routing R (b) $R_1 = R^{(v_2,v_4)}$ (c) $R_2 = R_1^{(v_3,v_2)}$ (d) $R_3 = R_2^{(v_1,v_3)}$

Fig. 2: A correct update sequence for $reach(v_1, v_4)$ and $wp(v_1, v_4, v_2)$

2.1 Network Updates

In order to be able to change the given routing (and therefore to influence the routing path) we introduce the notion of an update. An update can either change the forwarding function of a given node to an alternative next hop, or it can undefine the routing function (remove the entry from the forwarding table).

Definition 5 (Update). *Given a routing R, an* update *is an element $e \in E \cup (V \times \{undefined\})$. For a given update $e = (u, u')$, the* updated routing R^e *is given by*

$$R^{(u,u')}(v) = \begin{cases} R(v) & \text{if } v \neq u \quad (1a) \\ u' & \text{if } v = u. \quad (1b) \end{cases}$$

In order to update one routing into another, a number of updates must be performed in a sequence (a so-called update sequence), executing the updates from left to right.

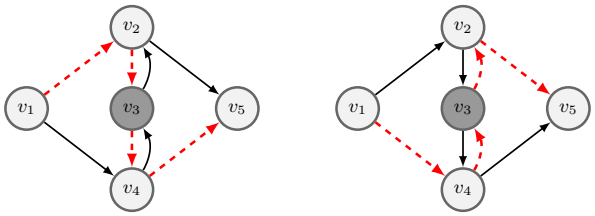
Definition 6 (Update Sequence). *Given a routing R and an* update sequence $\omega = e_1 e_2 ... e_n \in (E \cup (V \times \{undefined\}))^*$, *we inductively define the final routing R^ω by (i) $R^\epsilon = R$ and (ii) $R^{e\omega} = (R^e)^\omega$ for any $e \in E$ and any $\omega \in (E \cup V \times \{undefined\})^*$.*

Finally, we must guarantee that an update sequence transiently satisfies a given set of properties $\mathcal{P}$, containing e.g. $reach(u, v)$, $wp(u, v, w)$ and $loopfree(u)$.

Definition 7 (Correct Update Sequence). *We say that ω is a* correct update sequence *for R with respect to a set of properties $\mathcal{P}$ if for every prefix ω' of ω the routing $R^{\omega'}$ satisfies every property from $\mathcal{P}$.*

Figure 2 shows the steps of a correct update sequence $\omega = (v_2, v_4) \circ (v_3, v_2) \circ (v_1, v_3)$ on our running example, for $\mathcal{P} = \{reach(v_1, v_4), wp(v_1, v_4, v_2)\}$. At any moment during the update sequence, the node v_4 is reachable from v_1 and at the same time the node v_2 (in grey) is always present on the path from v_1 to v_4.

The update synthesis problem is, for a given initial routing R and a final routing R', to find an update sequence that transforms R into R' while preserving a given set of path properties $\mathcal{P}$. Moreover, we allow at most one update of every node in the network. As we are modelling one fixed flow in the network, we assume that the path properties always start with the same fixed node $u \in V$.



(a) Initial routing (b) Final routing

Fig. 3: A network with no solution preserving $reach(v_1, v_5)$ and $wp(v_1, v_5, v_3)$

Definition 8 (Update Synthesis Problem). *The* update synthesis problem *is a tuple* $\mathcal{U} = (G, R, R', \mathcal{P}^u)$ *where* G *is a network,* R *is an initial routing,* R' *is a final routing, and* $\mathcal{P}^u \subseteq \{wp(u, v, w), reach(u, v), loopfree(u) \mid v, w \in V\}$ *for some fixed* $u \in V$ *is the set of path properties. The question is whether there exists a correct update sequence* $\omega \in (E \cup (V \times \{undefined\}))^*$ *with respect to* $\mathcal{P}^u$ *such that* $R^\omega = R'$ *and where every* $v \in V$ *appears in* ω *at most once as a source node. We then say that* ω *is a* solution *to the update synthesis problem.*

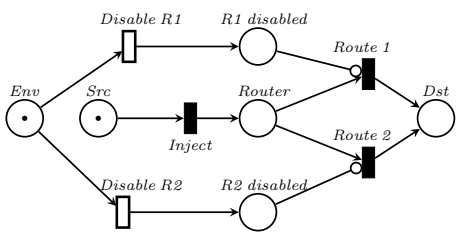
The update synthesis problem in NP-complete [18]. Figure 2 shows a solution to the update synthesis problem transforming the routing R into R_3 while preserving $reach(v_1, v_4)$ and $wp(v_1, v_4, v_2)$. We notice that this is in fact the only solution. If we in the routing R first update the router v_1, we break the waypointing property and if we instead decide to update first v_3, we create a loop and invalidate the reachability property. Similarly, in the routing R_1 the only choice is to first update v_3, as updating v_1 in R_1 will avoid the waypoint. Lastly, in Figure 3 we give an example of an update synthesis problem that does not have a solution for preserving the properties $reach(v_1, v_5)$ and $wp(v_1, v_5, v_3)$. If v_3 is updated before v_2, a loop is created and $reach(v_1, v_5)$ is violated and if v_2 is updated before v_3, the property $wp(v_1, v_5, v_3)$ is violated.

3 Petri Games

A Petri net (see e.g. [23]) is a mathematical model of distributed systems that allows us to model concurrent and nondeterministic behaviour. A Petri game extends P/T nets by introducing the controllable and environmental transitions. This kind of games were studied in [13] also including the timing aspects.

Let $\mathbb{N}^0$ be the set of natural numbers including 0 and let $\mathbb{N}^\infty$ be the set of natural numbers extended with the symbol ∞ that is larger than any other natural number.

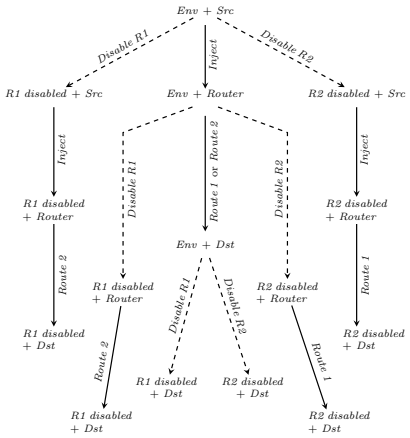
Definition 9 (Petri Game). *A* Petri game *is a 4-tuple* $N = (P, T, W, I)$ *where* P *is a finite set of* places*,* T *is a finite set of* transitions *partitioned into the controllable* T_{ctrl} *and environmental* T_{env} *ones s.t.* $T = T_{ctrl} \uplus T_{env}$ *and*



(a) A Petri game

M	$\sigma(M)$
Env + Src	*Inject*
Env + Router	*Route 1*
R1 Disabled + Src	*Inject*
R1 Disabled + Router	*Route 2*
R2 Disabled + Src	*Inject*
R2 Disabled + Router	*Route 1*

(b) A control strategy



(c) All runs under strategy σ, invariantly satisfying $\neg deadlock \vee Dst = 1$

Fig. 4: A Petri game example

$P \cap T = \emptyset$, the function $W : (P \times T) \cup (T \times P) \to \mathbb{N}^0$ assigns weights to normal arcs, and the function $I : P \times T \to \mathbb{N}^\infty$ assigns weights to inhibitor arcs.

Let $N = (P, T, W, I)$ be a fixed Petri game. Places in a Petri game are depicted as circles, controllable transitions by filled rectangles and environmental by empty rectangles and whenever $W(p,t) > 0$ or $W(t,p) > 0$ we draw an arc between p and t, resp. t and p, labeled by the corresponding weight (no label stands for the weight 1, and if $I(p,t) < \infty$ we draw an inhibitor arc depicted circle-head between p and t and annotated by the weight. An example of Petri game is given in Figure 4a and it models a scenario where a packet must travel through a network from *Src* to *Dst* through one of two possible routes *Route 1* or *Route 2*. However, the environment can disable one of the two routes, and depending on which it disables we must choose the remaining route.

A *marking* M on N is a total function $M : P \to \mathbb{N}^0$ that marks each place with 0 or more tokens. Let $\mathcal{M}(N)$ bet the set of all markings on N. We can represent a marking as a formal sum $k_0 p_0 + k_1 p_1 + \cdots + k_n p_n$ where k_i denotes the number of tokens present in p_i. A marking is graphically denoted by dots in places. A transition $t \in T$ is *enabled* in a marking M if $W(p,t) \leq M(p)$ and $I(p,t) > M(p)$ for all $p \in P$. If a transition $t \in T$ is enabled in a marking M, it can *fire* resulting in the marking M', written $M \xrightarrow{t} M'$, where $M'(p) = M(p) + W(p,t) - W(t,p)$ for all $p \in P$.

The controller's choice of which controllable transition to select in each marking, is given by the concept of a (memoryless) strategy.

Definition 10 (Strategy). *Let $N = (P, T, W, I)$ be a Petri game. A* strategy *$\sigma : \mathcal{M}(N) \hookrightarrow T_{ctrl}$ is a partial function such that if $\sigma(M) = t$ then $M \xrightarrow{t} M'$ for some M' and $\sigma(M)$ is undefined iff there is no $t \in T_{ctrl}$ enabled in M.*

An example of a strategy for the game net from Figure 4a is given in Figure 4b. A strategy determines the set of runs such that from every marking either any environmental transition or the transition proposed by the strategy can be executed. A set of all runs in our running example, organized into a tree where run prefixes are shared, is given in Figure 4c. Controllable transitions are depicted with solid lines and environmental with dashed lines.

Definition 11 (Run). *Let $N = (P, T, W, I)$ be a Petri game with an initial marking M_0 and a strategy σ. The set of finite and infinite* runs *under σ is given by $runs^\sigma(M_0) = \{M_0 M_1 M_2 ... \mid$ for all i holds $M_i \xrightarrow{t} M_{i+1}$ s.t. $t \in T_{env}$ or $M_i \xrightarrow{\sigma(M_i)} M_{i+1}$ if $\sigma(M_i)$ is defined$\}$.*

The goal of the controller in the game is to invariantly preserve a given safety objective, expressed as a *marking predicate* φ given by a Boolean combination of expressions of the form $e \bowtie e$ or *deadlock* where

$$e ::= p \mid n \mid e + e \mid e - e \mid e * e$$

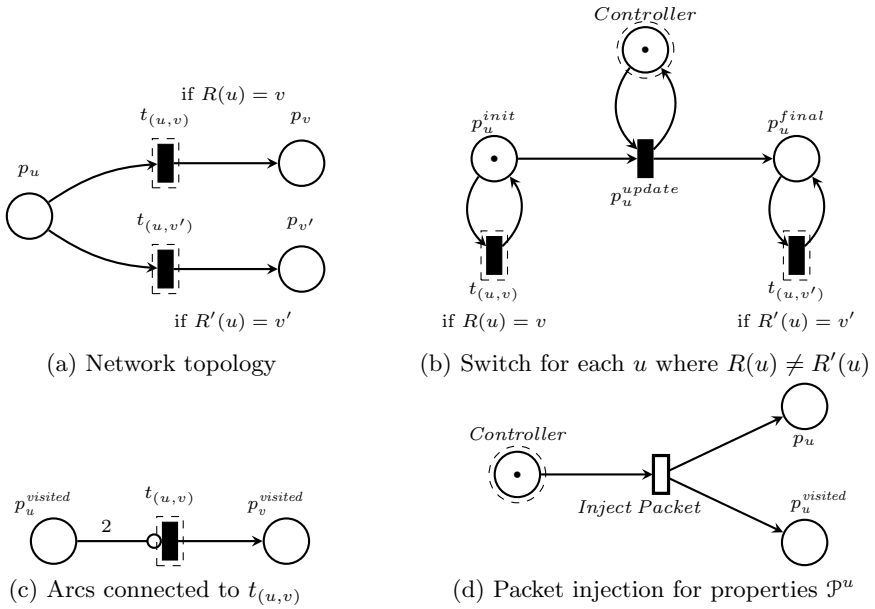
such that $p \in P$, $\bowtie \in \{\leq, <, =, \neq, \geq, >\}$ and $n \in \mathbb{N}_0$. We write $M \models \mathit{deadlock}$ if there are no enabled transitions in M and the semantics of the Boolean connectives as well as of the expressions $e \bowtie e$ is given in a natural way, assuming that p stands for $|M(p)|$. If a marking M satisfies a predicate φ, we write $M \models \varphi$.

Definition 12 (Winning Control Strategy). *Let $N = (P, T, W, I)$ be a Petri game with the initial marking M_0 and let φ be a marking predicate. We write $M_0 \models \mathit{control} : AG\, \varphi$ if there exists a strategy σ such that for every run $M_0 M_1 M_2 ... \in runs^\sigma(M_0)$ we have $M_i \models \varphi$ for all relevant i. If such a strategy exists, we say that σ is a* winning strategy*.*

In Figure 4a we have $\mathit{Env} + \mathit{Src} \models \mathit{control} : AG\,(\neg \mathit{deadlock} \vee \mathit{Dst} = 1)$ as the strategy defined in Figure 4b is a winning strategy for the safety objective $\neg \mathit{deadlock} \vee \mathit{Dst} = 1$. This can be verified by exploring the possible runs under the strategy given in Figure 4c and noticing that all deadlocked markings have a token in the place *Dst*. In general, for any bounded Petri game the existence of a winning control strategy is decidable (see e.g. [15]).

4 From Update Synthesis Problem to Petri Games

We shall now present a reduction from the update synthesis problem into a Petri game. The reduction idea is that the controller is allowed to step-wise generate any possible sequence of updates and a given control strategy fixes a concrete update sequence. The environment can at any moment decide to stop



(a) Network topology

(b) Switch for each u where $R(u) \neq R'(u)$

(c) Arcs connected to $t_{(u,v)}$

(d) Packet injection for properties $\mathcal{P}^u$

Fig. 5: Translation components

the generation process and inject a packet into the network in order to verify whether the current prefix of the update sequence satisfies the given properties. If this is not the case, the environment wins, otherwise the controller has a winning strategy that corresponds to a correct update sequence.

The reduction, for a given update synthesis problem $(G, R, R', \mathcal{P}^u)$, is split into the creation of the following components:

1. *Network topology component* based on G.
2. *Switch components* based on R and R'.
3. *Visited places* to track how many times a node is visited.
4. *Packet injection component* to start the verification phase.
5. *Satefy objective* based on $\mathcal{P}^u$.

For clarity reasons, we decompose our Petri net construction into separate components. If the same place or transition name appears in multiple components, we refer to them as shared and indicate this by surrounding them with a dashed line. We assume that the shared places and transitions are merged.

4.1 Translation to Petri Games

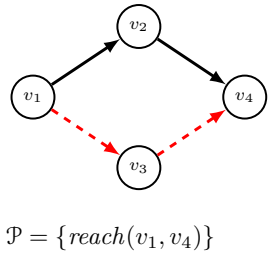
Let $\mathcal{U} = (G, R, R', \mathcal{P}^u)$ be an update synthesis problem such that $G = (V, E)$ is the underlying network. We create a Petri game $N(\mathcal{U}) = (P, T, I, W)$ where $T = T_{ctrl} \uplus T_{env}$ by defining the different components mentioned above.

1. *Network topology components.* For all $u \in V$ where $R(u)$ or $R'(u)$ is defined create the place p_u and
   - If $R(u) = v$ we create a shared transition $t_{(u,v)} \in T_{ctrl}$ and a place p_v.
   - If $R'(u) = v'$ we create a shared transition $t_{(u,v')} \in T_{ctrl}$ and a place $p_{v'}$.

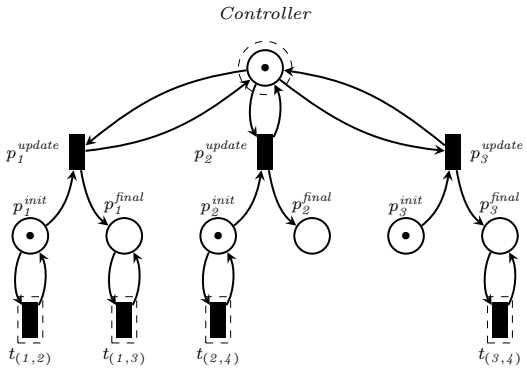
   The created places and transitions are then connected by normal arcs as illustrated in Figure 5a.
2. *Switch components.* For every $u \in V$ where $R(u) \neq R'(u)$, we create a switch component with a controllable transition $p_u^{update} \in T_{ctrl}$ connected to a globally shared place *Controller* and two places p_u^{init} (initially marked) and p_u^{final} as in Figure 5b. Moreover,
   - if $R(u) = v$, we add the arcs p_u^{init} to $t_{(u,v)}$ and $t_{(u,v)}$ to p_u^{init}, and
   - if $R'(u) = v'$, we add the arcs p_u^{final} to $t_{(u,v')}$ and $t_{(u,v')}$ to p_u^{final}.
3. *Visited places.* For each place p_u that was already created, we create a dual place $p_u^{visited}$ as illustrated in Figure 5c where
   - for all $v \in V$ such that there exists a $u \in V$ where $R(u) = v$ or $R'(u) = v$ we add an arc from $t_{(u,v)}$ to $p_v^{visited}$, and
   - for all $u \in V$ such that $R(u) = v$ or $R'(u) = v$ we add an inhibitor arc with weight of 2 from $p_u^{visited}$ to $t_{u,v}$.
4. *Packet injection component.* Here we add (the only) environmental transition *Inject Packet* $\in T_{env}$ that connects, as depicted in Figure 5d, the place *Controller* (initially marked with a token) to the places p_u and $p_u^{visited}$ where u is the initial node fixed in the set of properties $\mathcal{P}^u$.
5. *Verification queries.* For the given set of properties $\mathcal{P}^u = \{P_1, P_2, \ldots, P_k\}$ we construct the safety objective *control* : $AG\ \varphi_{P_1} \wedge \varphi_{P_2} \ldots \wedge \varphi_{P_k}$ where:
   - $\varphi_{loopfree(u)} \equiv \bigwedge_{u \in V} p_u^{visited} < 2$
   - $\varphi_{reach(u,v)} \equiv \neg deadlock \vee p_v^{visited} \geq 1$
   - $\varphi_{wp(u,v,w)} \equiv p_w^{visited} \geq 1 \vee p_v^{visited} = 0$

The translation is illustrated by a small example in Figure 6. For the network in Figure 6a, we want to update the initial routing via v_1, v_2 and v_4 to the final routing v_1, v_3 and v_4, while preserving the reachability between v_1 and v_4. The translation of the switch components is given in Figure 6b and the translation of the renaming components in Figure 6c, where the place *Controller* as well as the transitions $t_{(1,2)}$, $t_{(1,3)}$, $t_{(2,4)}$ and $t_{(3,4)}$ are shared between the two figures. We note that since $R(v_4) = R'(v_4) = undefined$, we do not create a switch component for v_4.

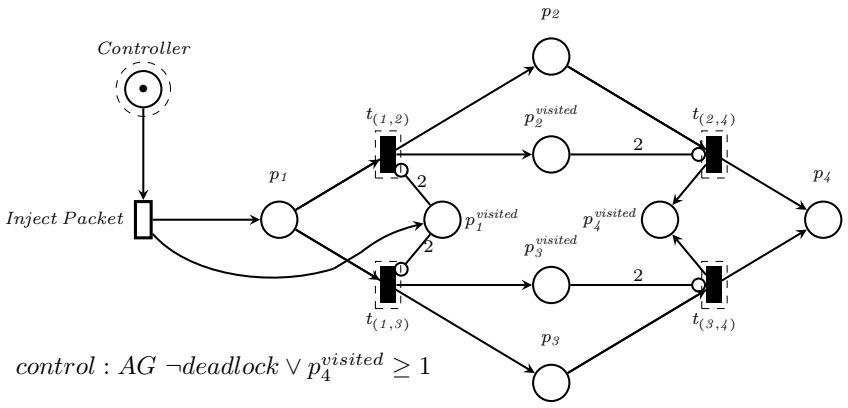
Finally, we notice the once the environmental *Inject Packet* transition fires, a token is removed from the place *Controller* and it is no longer possible to change the current transient routing. The construction guarantees that token injected to p_1 now follows exactly the path corresponding to the current routing and every time a place receives a token, the corresponding visited place also gets a token. Should there be a loop, the first marking where one of the visited places obtains a second token deadlocks due to the introduction of the inhibitor arcs. The constructed Petri game is so guaranteed to be bounded. The property we wish to preserve is $reach(v_1, v_4)$ and it translates to the safety objective $\neg deadlock \vee p_4^{visited} \geq 1$. This guarantees that during the execution of the routing path we do not deadlock (which can be caused either by a blackhole or a loop) before the target place p_4 is reached.



(a) Initial routing in solid lines, final in dashed lines



(b) Switch components



(c) Topology, visited places and packet injection components

Fig. 6: Translation example

4.2 Translation Correctness

Let $\mathcal{U} = (G, R, R', \mathcal{P}^u)$ be an update synthesis problem and let $N(\mathcal{U})$ be the constructed Petri game with the initial marking M_0 and the safety objective φ. The main correctness theorem is stated as follows.

Theorem 1. *The problem $\mathcal{U}$ has a solution iff $M_0 \models control : AG\varphi$ in $N(\mathcal{U})$.*

We shall first observe that all runs in the constructed Petri game are finite.

Lemma 1. *The net $N(\mathcal{U})$ contains no infinite run from the initial marking M_0.*

Proof. Assume by contradiction that there is an infinite run from M_0. Clearly, the firing of each p_u^{update} can happen at most once for each u, so necessarily

the transition *Inject Packet* must fire during such a sequence, initializing the execution in the topology component. There must be now a transition $t_{(u,v)}$ that fires at least twice in the infinite run. Each time $t_{(u,v)}$ fires, a new token is deposited into $p_v^{visited}$ and hence the place contains two tokens after the second time $t_{(u,v)}$ is fired. However, all outgoing transitions from the place p_v (containing a token after $t_{(u,v)}$ is fired) are disabled due to the inhibitor arcs of weight 2 from $p_v^{visited}$. Hence the net deadlocks and this contradicts the existence of the infinite run. □

Now we prove Theorem 1 by establishing each direction of the claim.

Lemma 2. *If ω is a correct update sequence for $\mathcal{U}$ then in $N(\mathcal{U})$ there is a winning control strategy σ for the formula control : $AG\ \varphi$.*

Proof. Let $\omega = e_1 e_2 ... e_n$ be a correct update sequence such that $e_i = (u_i, u_i')$ where $u_i \in V$ and $u_i' \in V \cup \{undefined\}$. We shall now define a winning strategy σ for the controller, given the initial marking M_0, as follows: $\sigma(M_0) = p_{u_1}^{update}$ and we let $M_0 \xrightarrow{p_{u_1}^{update}} M_1$; we continue to define $\sigma(M_{i-1}) = p_{u_i}^{update}$ for all i, $1 < i \leq n$, where we let $M_{i-1} \xrightarrow{p_{u_i}^{update}} M_i$. For all other markings M, we let $\sigma(M) = M'$ for an arbitrary M' such that $M \xrightarrow[t]{M'}$ and $t \in T_{ctrl}$, if such marking M' exists; otherwise is $\sigma(M)$ undefined. In other words, the controller fires the controllable $p_{u_i}^{update}$ transitions in the order specified in the update sequence ω. We also notice that once the *Inject Packet* transition is fired by the environment, there is in any reachable marking at most one enabled controllable transition, exactly simulating the path under the currently generated sequence of updates. As stated in Lemma 1, this path is finite as the net deadlocks as soon as the same node is visited the second time.

Now we argue that the strategy σ is a winning control strategy by showing that every marking M reachable under the strategy σ satisfies φ. As φ is a conjunction of marking predicates of three different types (depending on the properties in $\mathcal{P}^u$), we discuss the three cases.

- $\varphi_{loopfree(u)} \equiv \bigwedge_{u \in V} p_u^{visited} < 2$. Before the environment fires *Inject Packet*, no token can be placed in any $p_u^{visited}$ and the invariant is satisfied. If *Inject Packet* is fired, because ω is a correct update sequence it follows by Definition 7 that the currently generated prefix of ω yields a loop-free routing and therefore any reachable marking satisfies $p_u^{visited} < 2$.
- $\varphi_{reach(u,v)} \equiv \neg deadlock \vee p_v^{visited} \geq 1$. Before firing *Inject Packet*, the net cannot deadlock as *Inject Packet* is still enabled and hence the property holds. Once *Inject Packet* is fired, due to our assumption that the currently generated prefix of ω is correct and the corresponding routing hence eventually reaches the node v, and because the Petri net faithfully mimics the routing path, we know that the Petri net execution eventually marks the place $p_v^{visited}$ and cannot deadlock before this: there cannot be any blackhole as this contradicts the reachability of v and there cannot be any node

visited twice either before reaching v, as the path is deterministic and it will imply the existence of a loop and contradict the reachability of v. Hence $\neg \mathit{deadlock} \vee p_v^{visited} \geq 1$ invariantly holds.
- $\varphi_{wp(u,v,w)} \equiv p_w^{visited} \geq 1 \vee p_v^{visited} = 0$. Before firing *Inject Packet* the invariant clearly holds as none of the places $p_u^{visited}$ can be marked for all nodes u, including v. After *Inject Packet* is fired, as we assume that any prefix of ω corresponds to a correct routing, meaning that the node v cannot be marked before the node w. Hence the invariant holds also in this case. □

Lemma 3. *If σ is a winning control strategy in $N(\mathcal{U})$ for the formula control : $AG\ \varphi$ then there exists a correct update sequence ω solving $\mathcal{U}$.*

Proof. Assume that σ is a winning strategy for the formula *control* : $AG\ \varphi$. Given the initial marking M_0, the strategy σ generates the sequence of markings $M_0, M_1, \ldots, M_n$ such that $\sigma(M_{i-1}) = p_{u_i}^{update}$ and $M_{i-1} \xrightarrow{p_{u_i}^{update}} M_i$ for all i, $1 \leq i \leq n$. This sequence naturally defines the update sequence $\omega = e_1 e_2 ... e_n$ where $e_i = (u_i, R'(u_i))$ for all i, $1 \leq i \leq n$.

We know that once *Inject Packet* fires at a marking M_i, there exists a unique path in the Petri net, following exactly the routing defined by update sequence $e_1 e_2 \ldots e_i$. As σ is a winning control strategy, we know that φ holds in every marking on such a path. We shall argue by case analysis that the routing after applying the update sequence $e_1 e_2 \ldots e_i$ satisfies every path property $P \in \mathcal{P}^u$.

- $P \equiv \mathit{loopfree}(u)$. Then the path in the Petri net satisfies the marking property $\varphi_{loopfree(u)} \equiv \bigwedge_{u \in V} p_u^{visited} < 2$ which by the net construction implies that the path must be finite.
- $P \equiv \mathit{reach}(u, v)$. Then the path in the Petri net satisfies $\varphi_{reach(u,v)} \equiv \neg \mathit{deadlock} \vee p_v^{visited} \geq 1$. This implies that the net cannot deadlock before marking the place v, which gives that v must be necessarily reached as there is no infinite run due to Lemma 1. The property P hence holds.
- $P \equiv \mathit{wp}(u, v, w)$. Then every marking in the path satisfies $\varphi_{wp(u,v,w)} \equiv p_w^{visited} \geq 1 \vee p_v^{visited} = 0$, exactly formulating the requirement that p_v cannot be marked before p_w gets marked, again implying the property P. □

4.3 Optimization for Reachability and Waypointing

It is a natural requirement that every transient routing in an update sequence should preserve at least the reachability between the source and the target node. We may also assume that once the target node is reached, any further routing from the target becomes undefined as the packet is considered delivered. In this case, the preservation of reachability also implies loop freedom. Finally, we may also allow to use multiple waypointing properties for different waypoints, as long as they are between the source and the destination that are connected in every transient routing. Formally, we define a set of *reachability and waypointing properties* for a given source $u \in V$ and target $v \in V$ as

$$\mathcal{P}^{(u,v)} = \{\mathit{reach}(u, v)\} \cup \{\mathit{wp}(u, v, w) \mid w \in W\}$$

where $W \subseteq V$ is a given set of waypoints. For this restricted set of path properties, we can notice that the construction of the Petri game solving the update synthesis problem can be optimized as follows.

Let $\mathcal{U} = (G, R, R', \mathcal{P}^{(u,v)})$ be an update synthesis problem with the set of reachability and waypointing properties $\mathcal{P}^{(u,v)}$ such that $G = (V, E)$ is the underlying network. We partition all relevant switches $u \in V$ where $R(u) \neq R'(u)$ into three categories:

- $V^{init} = \{u \in V \mid R(u) \neq R'(u), R(u) \text{ is undefined }\}$,
- $V^{final} = \{u \in V \mid R(u) \neq R'(u), R'(u) \text{ is undefined }\}$, and
- $V^{both} = \{u \in V \mid R(u) \neq R'(u), \text{ both } R(u) \text{ and } R'(u) \text{ are defined }\}$.

We shall now observe that if there is a correct update sequence transforming the routing R into R' while preserving $\mathcal{P}^{(u,v)}$ then there is also one where all switches from V^{init} are placed (in arbitrary order) at the beginning of the update sequence and all switches from V^{final} can be (again in arbitrary order) updated at the very end of the update sequence.

Lemma 4. *Let ω be a correct update sequence for the update synthesis problem $\mathcal{U} = (G, R, R', \mathcal{P}^{(u,v)})$ with reachability and waypointing property set $\mathcal{P}^{(u,v)}$. Let ω' be a subsequence of ω containing only updates of the form (u, u') where $u \in V^{both}$. Let $\omega^{init} = (v_1, R'(v_1)) \circ \ldots \circ (v_k, R'(v_k))$ be a sequence of updates of all nodes from the set $V^{init} = \{v_1, \ldots, v_k\}$. Let $\omega^{final} = (u_1, \textit{undefined}) \circ \ldots \circ (u_\ell, \textit{undefined})$ be a sequence of updates of all nodes from the set $V^{final} = \{u_1, \ldots, u_\ell\}$. Then $\omega^{init} \circ \omega' \circ \omega^{final}$ is also a correct update sequence.*

Proof. Let ω be a correct update sequence such that $\omega = \omega_1 \circ (x, x') \circ \omega_2$ where either (i) $x \in V^{init}$ meaning that $R(x)$ is undefined, or (ii) $x \in V^{final}$ meaning that $R'(x)$ is undefined. We want to show that in case (i) $(x, x') \circ \omega_1 \circ \omega_2$ and in case (ii) $\omega_1 \circ \omega_2 \circ (x, x')$ is also a correct update sequence. These two facts imply the statement in the lemma.

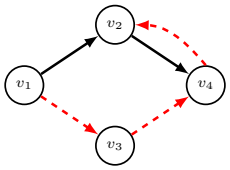
In case (i), we know that for any prefix of ω_1, the corresponding routing path always connects u and v a hence it cannot pass thought the node x for which the next hop is undefined. Hence moving (x, x') to the beginning of the update sequence does not change the routing path and after the sequence of updates $(x, x') \circ \omega_1$ and $\omega_1 \circ (x, x')$ we arrive to the identical switch configuration.

In case (ii), we know that the reachability between u and v is preserved all the time during the update sequence ω. As the update $(x, \textit{undefined})$ creates a blackhole, after the sequence ω_1 is applied, the switch x cannot be part of the prefix of the routing path from u until v is reached. This implies that moving the update to the end does not influence the given set of path properties. □

We can apply this lemma to improve the efficiency of our translation by creating a reduced Petri game $N'(\mathcal{U})$ where from the original net $N(\mathcal{U})$ we remove the whole switch component for every u where $u \in V^{init} \cup V^{final}$. This means that for the switches in V^{init}, the final routing will be enabled already from the start of the net execution and for the switches from V^{final} we never undefine the routing function. We can so conclude with the following theorem.

Theorem 2. *Let* $\mathcal{U} = (G, R, R', \mathcal{P}^{(u,v)})$ *be an update synthesis problem with reachability and waypointing properties* $\mathcal{P}^{(u,v)}$*. Let* $N'(\mathcal{U})$ *be the reduced Petri game with the initial marking* M_0 *defined above and* φ *the safety objective constructed from* $\mathcal{P}^{(u,v)}$*. Then* $\mathcal{U}$ *has a solution iff* $M_0 \models \mathit{control} : AG\ \varphi$ *in* $N'(\mathcal{U})$*.*

In Figure 7 we show that Theorem 2 does not hold e.g. for the property $\mathit{loopfree}(v_1)$ alone. As usual, the initial routing is in black solid lines and the final routing in dashed red lines. Clearly, $(v_1, v_3) \circ (v_3, v_4) \circ (v_4, v_2) \circ (v_2, \mathit{undefined})$ is a correct update sequence transforming the initial routing to the final one. However, even though the final routing for the node v_4 is undefined, it is not possible to move this update to the beginning of the update sequence as updating first the node v_4 creates a forwarding loop and hence breaks the $\mathit{loopfree}(v_1)$ property.



$\mathcal{P} = \{\mathit{reach}(v_1, v_4)\}$

Fig. 7: Counter example

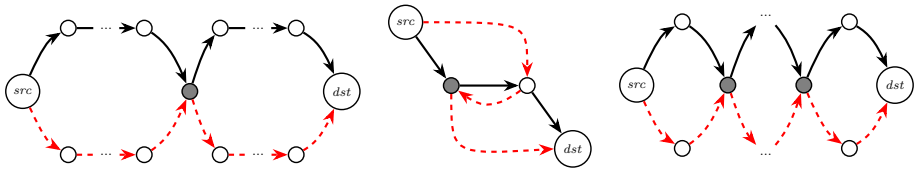
5 Implementation and Experiments

We implemented a prototype tool in Python that translates the update synthesis problem into Petri game. Our tool accepts a JSON file with the description of the network topology, initial and final routing as well as the list of required security policies: reachability, loop-freedom and (multiple) waypointing. The tool provides three types of output: (i) update synthesis problem definition in NetSynth [21] input format, (ii) XML file with Petri game and a safety query to be opened in the GUI of TAPAAL model checker [6], and (iii) XML Petri game model file and query file to be used with the command-line engine `verifypn` (part of the TAPAAL framework).

The game engine `verifypn` is based on the algorithms presented in [13,15], utilizing PTries for efficient state-storage [14], and we designed a new state-space exploration strategy in order to speedup the game synthesis algorithm. The engine `verifypn` is further extended to output the synthetized game strategy from which the update sequence can be derived. In order to experiment with network topologies in the standard `gml` format as used e.g. in the Topology Zoo dataset [16], our tool also facilitates the generation of update synthesis problems (in the TAPAAL and NetSynths formats) directly from the `gml` input files. We evaluate the performance of our tool (using the engine `verifypn` from the TAPAAL model checker) on both synthetic and real-world ISP topologies.

5.1 Synthetic Network Topologies

The synthetic topologies presented in Figure 8 define scalable network update problems that model two extreme situations where the initial and final routing paths are either *disjoint* or fully *dependent*, as well as a third, more realistic,



(a) Disjoint (b) Dependent (c) Shared

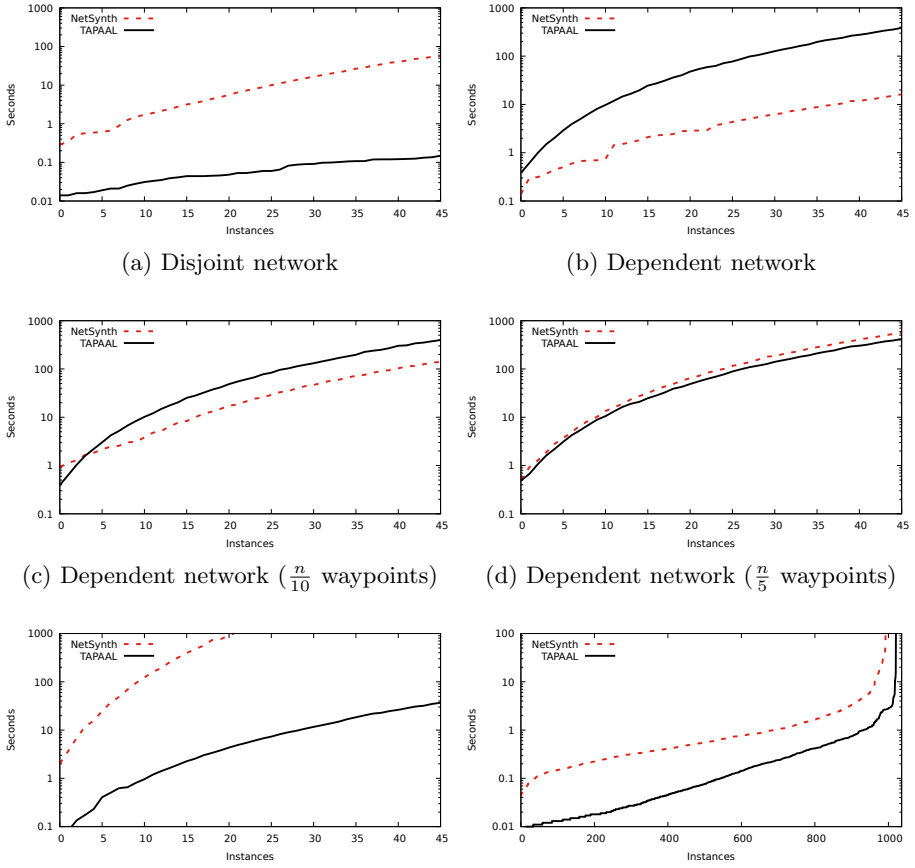
Fig. 8: Synthetic network topologies; initial routing in black solid lines and final routing in red dashed lines

scenario where the two routing paths *share* a number of waypoints, while still using independent routers in between.

The disjoint network template from Figure 8a follows the structure used in NetSynth benchmarks from [21]. The size of the update problem is scaled by increasing the lengths of the disjoint paths. The dependent network type in Figure 8b aims to minimize the number of correct update sequences as many of the possible update sequences either create a loop or avoid the waypoint drawn in gray. The problem is scaled by sequentially concatenating (repeating) the same structure number of times. Finally, the shared topology from Figure 8c combines a number of shared nodes that are connected by short disjoint paths of length two. The scaling is achieved by repeating the depicted pattern several times. The verified properties in all synthetic networks are $reach(src, dst)$ in conjunction with $wp(src, dst, w)$ where w is a single selected waypoint that is shared on the initial and final routing path (for disjoint topology there is exactly one such node drawn in gray). For the case of dependent topologies, we also study the variant with multiple waypoint properties.

5.2 Topology Zoo Benchmark

The topology Zoo database [16] contains 261 network topologies (with up to 700 nodes) from real internet service providers. In order to achieve additional scaling and larger instances, we combine existing topologies by further nesting/concatenating them. To create realistic initial and final routing paths, we emulate the standard protocols like OSPF [22] and ECMP [11] that are based on routing along the shortest paths in the weighted network (the weights are typically manually assigned by network operators). For each topology, we compute the diameter of the graph in order to identify the source *src* and destination *dst* nodes that maximize the smallest number of hops between them (for large network configurations, only a random subset of nodes are chosen in order to limit the computational effort for generating the test-cases). We randomly assign the weights up to 5 to every edge and set the initial routing path as one of the shortest paths between *src* and *dst*. We then increase the weights on the initial path that enforces a change in the set of shortest paths between *src* and *dst* in order to determine an alternative final routing path. The verified properties are



(a) Disjoint network

(b) Dependent network

(c) Dependent network ($\frac{n}{10}$ waypoints)

(d) Dependent network ($\frac{n}{5}$ waypoints)

(e) Shared network

(f) Topology Zoo

Fig. 9: Cactus plots where x-axis shows the increasing problem instances and y-axis depicts (on logarithmic scale) the synthesis time; n is the number of nodes

$reach(src, dst)$ in conjunction with $wp(src, dst, w)$ where w is a randomly chosen waypoint that is shared by both the initial and final path (routing paths with no common waypoint are discarded).

5.3 Results

We compare the performance of our tool translating the update synthesis problem into a Petri game and using the TAPAAL engine as the backend (this tool chain is referred to as TAPAAL in the plots) against the state-of-the-art update synthesis tool NetSynth [21]. The results are presented in the form of *cactus plots* where the synthesis times used by each tool are (independently of each

other) sorted in nondecreasing sequence and plotted on the x-axis, while the y-axis shows the concrete runtime for each instance. These graphs do not provide instance-to-instance runtime comparison but instead give an overall picture of the tools performances. All of our experiments are conducted using the Linux 5.8.0 kernel running on AMD EPYC 7551 processors with hyperthreading disabled and limited to 7 GB of memory (the memory limit was though never exceeded). The tool source code as well as the experimental setup that allows us to rerun all experiments is available in [7].

Figure 9a shows the cactus plot for the disjoint network experiment. While NetSynth uses 57 seconds to solve the largest instance with 1000 nodes, our tool with TAPAAL backend uses only 0.15 seconds (contributed mainly to the frequent applicability of Lemma 4). For the case of dependent networks in Figure 9b with a single waypoint, the NetSynth incremental algorithm with build-in loop detection check outperforms TAPAAL, as many update sequence candidates create forwarding loops that TAPAAL is less efficient to detect. As a result, we need almost 386 seconds to solve the largest instance while NetSynth can synthetise the update sequence in about 16 seconds. However, once we require that every 10th node must be a waypoint, the relative performance of TAPAAL improves as seen in Figure 9c and once every 5th node is set as a waypoint, we already outperform NetSynth as shown in Figure 9d. This demonstrates that our approach scales better with increasing complexity of the required path properties. Moreover, our performance on the more realistic shared network in Figure 9e is several orders of magnitude faster than NetSynth and NetSynth only solves 21 instances within the 1000 second timeout, while we need less than 37 seconds to solve even the largest instance.

Finally, Figure 9f shows the performance on the dataset of existing networks emulating a realistic network operators behaviour. Here the routing paths are computed via a shortest path algorithm. We observe that our approach is in the middle case 8.9 times faster than NetSynth. We manage to solve the majority (933 instances) of problems in less than 1 second, while NetSynth solved only 689 instances within 1 second. We also remark that NetSynth is unable to solve 42 problems within the time limit of 100 seconds while TAPAAL manages to solve all but the 12 hardest instances (not surprising as already deciding the existence of a correct update sequence is an NP-complete problem [18]). In particular, we notice that NetSynth is noticeably slower at providing answers for problems where no update sequence exists.

6 Conclusion

We presented a fully automatic approach for synthetising correct-by-construction update sequences in programmable networks. The obtained update sequences are guaranteed to satisfy a number of network policies like preservation of reachability, loop-freedom and additional waypointing requirements. Our approach is based on reducing the problem to Petri games and employing the generic model checker TAPAAL for solving the game synthesis problem. The experi-

ments demonstrate that our method is significantly outperforming the state-of-the-art tool NetSynth, except for the case of fully dependent networks where both the initial and final routings share every single node in the network. However, this scenario is unlikely to appear during the operation of real networks as the packets are commonly routed along the shortest paths in the network. Indeed, the experiments on over one thousand of real network topologies, using the shortest paths routing algorithms like OSPF and ECMP, document that we are consistently (in median 8.9 times) faster in synthetising the correct update sequences compared to NetSynth. In particular, we are able to solve majority of the realistic update synthesis problems in less than 1 second, which is important in nowadays dependable networks that must be promptly updated with increasing frequencies in order to react e.g. to sudden changes in the traffic.

The Petri net model and the generated update sequence is well suited for a further integration with the tool Latte [4], a plug-in to TAPAAL that extends the model with timing features and enables further reduction of the duration of network updates. In the future work, we plan to directly integrate the two tools into a single tool chain in order to reduce the overhead from parsing/exchanging of different file formats. We will also consider adding more network update policies like service chaining (where a given sequence of switches must be visited in a prescribed order); we expect that this will require only smaller modifications to our reduction.

Acknowledgements. We thank to Anders Mariegaard for his help with setting up NetSynth. This work received a support from the DFF project QASNET.

References

1. S.A. Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht. Congestion-free rerouting of flows on DAGs. In *ICALP'18)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 143:1–143:13. Dagstuhl, 2018.
2. K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui. Software-defined networking (SDN): A survey. *Security and Comm. Networks*, 9(18):5803–5833, 2016.
3. S. Brandt, K. Förster, and R. Wattenhofer. On consistent migration of flows in SDNs. In *INFOCOM'16*, pages 1–9. IEEE, 2016.
4. N. Christesen, M. Glavind, S. Schmid, and J. Srba. Latte: Improving the latency of transiently consistent network update schedules. In *IFIP PERFORMANCE'20*, volume 48, no. 3 of *Performance Evaluation Review*, pages 14–26. ACM, 2020.
5. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV'02*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.
6. A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *TACAS'12*, volume 7214 of *LNCS*, page 492–497. Springer, 2012.
7. M. Didriksen, P.G. Jensen, J.F. Jønler, A.-I. Katona, S.D.L. Lama, F.B. Lottrup, S. Shajarat, and J. Srba. Artefact for: Automatic Synthesis of Transiently Correct Network Updates via Petri Games, February 2021. https://doi.org/10.5281/zenodo.4497000.

8. B. Finkbeiner, M. Gieseking, J. Hecking-Harbusch, and E.-R. Olderog. Model checking data flows in concurrent network updates. In *ATVA'19*, volume 11781 of *LNCS*, pages 515–533. Springer, 2019.
9. B. Finkbeiner, M. Gieseking, J. Hecking-Harbusch, and E.-R. Olderog. AdamMC: A model checker for Petri nets with transits against flow-LTL. In *CAV'20*, volume 12225 of *LNCS*, pages 64–76. Springer, 2020.
10. K. Foerster, S. Schmid, and S. Vissicchio. Survey of consistent software-defined network updates. *IEEE Communications Surveys Tutorials*, 21(2):1435–1461, 2019.
11. Ch. Hopps et al. Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, November, 2000.
12. J.F. Jensen, T. Nielsen, L.K. Oestergaard, and J. Srba. TAPAAL and reachability analysis of P/T nets. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 9930:307–318, 2016.
13. P.G. Jensen, K.G. Larsen, and J. Srba. Real-time strategy synthesis for timed-arc Petri net games via discretization. In *SPIN'16*, volume 9641 of *LNCS*, pages 129–146. Springer, 2016.
14. P.G. Jensen, K.G. Larsen, and J. Srba. Ptrie: Data structure for compressing and storing sets via prefix sharing. In *ICTAC'17*, volume 10580 of *LNCS*, pages 248–265. Springer, 2017.
15. P.G. Jensen, K.G. Larsen, and J. Srba. Discrete and continuous strategies for timed-arc Petri net games. *STTT*, 20(5):529–546, 2018.
16. S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology Zoo. *IEEE Journal on Selected Areas in Comm.*, 29(9):1765 –1775, 2011.
17. H.H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. Zupdate: Updating data center networks with zero loss. *SIGCOMM Comput. Commun. Rev.*, 43(4):411–422, August 2013.
18. A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. Transiently secure network updates. In *ACM SIGMETRICS*, pages 273–284. ACM, 2016.
19. A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *PODC'15*, page 13–22. ACM, 2015.
20. R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. HotNets-XII, New York, NY, USA, 2013. ACM.
21. J. McClurg, H. Hojjat, P. Černy, and N. Foster. Efficient synthesis of network updates. *ACM Sigplan Notices*, 50(6):196–207, 2015.
22. J. Moy. RFC2328: OSPF version 2, 1998. https://tools.ietf.org/html/rfc2328.
23. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
24. M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM'12*, pages 323–334. ACM, 2012.
25. S. Vissicchio and L. Cittadini. FLIP the (flow) table: Fast lightweight policy-preserving SDN updates. In *INFOCOM'16*, pages 1–9. IEEE, 2016.