Aalborg Universitet

Kaki

Efficient Concurrent Update Synthesis for SDN

Johansen, Nicklas Slorup; Kær, Lasse Brink; Madsen, Andreas Leicht; Nielsen, Kristian Ødum; Srba, Jiri; Tollund, Rasmus Grønkjær

Published in:
Formal Aspects of Computing

DOI (link to publication from Publisher):
10.1145/3605952

Creative Commons License
CC BY 4.0

Publication date:
2023

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Johansen, N. S., Kær, L. B., Madsen, A. L., Nielsen, K. Ø., Srba, J., & Tollund, R. G. (2023). Kaki: Efficient Concurrent Update Synthesis for SDN. *Formal Aspects of Computing*, *35*(3), 1-22. Article 20. https://doi.org/10.1145/3605952

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: December 05, 2025

Kaki: Efficient Concurrent Update Synthesis for SDN

NICKLAS S. JOHANSEN, LASSE B. KÆR, ANDREAS L. MADSEN, KRISTIAN Ø. NIELSEN, JIŘÍ SRBA, and RASMUS G. TOLLUND, Department of Computer Science, Aalborg University, Denmark

Modern computer networks based on the software-defined networking (SDN) paradigm are becoming increasingly complex and often require frequent configuration changes in order to react to traffic fluctuations. It is essential that forwarding policies are preserved not only before and after the configuration update but also at any moment during the inherently distributed execution of such an update. We present Kaki, a Petri game based tool for automatic synthesis of switch batches which can be updated in parallel without violating a given (regular) forwarding policy like waypointing or service chaining. Kaki guarantees to find the minimum number of concurrent batches and supports both splittable and nonsplittable flow forwarding. In order to achieve optimal performance, we introduce two novel optimisation techniques based on static analysis: decomposition into independent subproblems and identification of switches that can be collectively updated in the same batch. These techniques considerably improve the performance of our tool Kaki, relying on TAPAAL's verification engine for Petri games as its backend. Experiments on a large benchmark of real networks from the Internet Topology Zoo database demonstrate that Kaki outperforms the state-of-the-art tools Netstack and FLIP. Kaki computes concurrent update synthesis significantly faster than Netstack and compared to FLIP, it provides shorter (and provably optimal) concurrent update sequences at similar runtimes.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Additional Key Words and Phrases: Computer networks, software defined networking, concurrent update synthesis, security policies

ACM Reference format:
Nicklas S. Johansen, Lasse B. Kær, Andreas L. Madsen, Kristian Ø. Nielsen, Jiří Srba, and Rasmus G. Tollund. 2023. Kaki: Efficient Concurrent Update Synthesis for SDN. *Form. Asp. Comput.* 35, 3, Article 20 (September 2023), 22 pages.
https://doi.org/10.1145/3605952

1 INTRODUCTION

Software defined networking (SDN) [7] delegates the control of a network's routing to the control plane, allowing for programmable control of the network and creating a higher degree of flexibility and efficiency. If a group of switches fail, a new routing of the network flows must be established in

This work was supported by DFF project QASNET.
Authors' address: N. S. Johansen, L. B. Kar, A. L. Madsen, K. O. Nielsen, J. Srba, and R. G. Tollund, Department of Computer Science, Aalborg University, Selma Lagerlofs Vej 300, Aalborg, Denmark, 9220; emails: {nslorup, lasse.b.kaer, andreasmadsen327, kristianodum}@gmail.com, srba@cs.aau.dk, rasmusgtollund@gmail.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0934-5043/2023/09-ART20 $15.00
https://doi.org/10.1145/3605952

order to avoid sending packets to the failed switches, resulting ultimately in packet drops. While updating the routing in an SDN network, the network must preserve a number of policies like waypointing that requires that a given firewall (waypoint) must be visited before a packet in the network is delivered to its destination. The update synthesis problem [7] is to find an update sequence (ordering of switch updates) that preserves a given policy.

In order to reduce the time of the update process, it is of interest to update switches in parallel. However, due to the asynchronous nature of networks, attempting to update all switches concurrently may lead to transient (i.e., during the update) policy violations before the update is completed. This raises the problem of finding a concurrent update strategy (sequence of batches of switches that can be updated concurrently) while preserving a given forwarding policy during the update. We study this *concurrent update synthesis problem* and provide an efficient translation of the problem of finding an optimal (shortest) concurrent update sequence into Petri net games. Our translation, implemented in the tool Kaki, guarantees that we preserve a given forwarding policy, expressed as a regular language over the switches describing the sequences of all acceptable hops.

Popular routing schemes like Equal-Cost-MultiPath (ECMP) [8] allow for switches to have multiple next hops that split a flow along several paths to its destination in order to account for traffic engineering like load balancing, using e.g., hash-based schemes [1]. In our translation approach, we support concurrent update synthesis that takes into account such multiple forwarding (splittable flows) modelled using nondeterminism.

To solve the concurrent update synthesis problem, our framework, called Kaki, translates a given network and its forwarding policy into a Petri game and synthesises a winning strategy for the controller using TAPAAL's Petri game engine [9, 10]. Kaki guarantees to find a concurrent update sequence that is minimal in the number of batches. We provide two novel optimisation techniques based on static analysis of the network that reduce the complexity of solving a concurrent update synthesis problem, which is known to be NP-hard even if restricted only to the basic loop-freedom and waypointing properties [16]. The first optimisation, topological decomposition, effectively splits the network with its initial and final routing into two subproblems that can be solved independently and even in parallel. The second optimisation identifies collective update classes (sets of switches) that can always be updated in the same batch.

Finally, we conduct a thorough comparison of our tool against the state-of-the-art update synthesis tools Netstack [23] and FLIP [26], and another Petri game tool [4] (though only allowing for sequential updates). We benchmark on the set of 8,759 problem instances of realistic network topologies with various policies required by network operators. Kaki manages to solve a similar number of problems as FLIP, however, in 9% of cases it synthesises a solution with a smaller number of batches than FLIP. The tool Netstack synthesises also provably optimal concurrent update solutions, however, at almost an order of magnitude slower running time. When Kaki is specialised to produce only singleton batches and policies containing only reachability and single waypointing, it performs similarly as the Petri game approach from [4] that is also using TAPAAL verification engine as its backend but solves a simpler problem. This demonstrates that our more elaborate translation that supports concurrent updates does not create any considerable performance overhead when applied to the simpler setting.

Related Work

The update synthesis problem recently attracted lots of attention (see e.g., the recent overview [7]). State-of-the-art solutions/tools include NetSynth [19], FLIP [26], Snowcap [24], AllSynth [14], Netstack [23], and a Petri game based approach [4].

The tools NetSynth [19] and AllSynth [14] use the generic LTL logic for policy specification but support the synthesis of only sequential updates. NetSynth is using incremental model checking approach and the authors in [4] argue that their tool outperforms NetSynth. AllSynth is based on the BDD technology in order to compactly represent all sequential solutions, however, it does not support concurrent updates either.

The update synthesis tool FLIP [26] supports general policies, and moreover it allows to synthesise concurrent update sequences. Similarly to Kaki, it handles every flow independently but Kaki provides more advanced structural decomposition (that can be possibly applied also as a preprocessing step for FLIP). FLIP provides a faster synthesis compared to NetSynth (see [26]) but the tool's performance is negatively affected by more complicated forwarding policies. FLIP synthesises policy-preserving update sequences by constructing constraints that enforce precedence of switch updates, implying a partial order of updates and hence allowing FLIP to update switches concurrently. FLIP, contrary to our tool Kaki, does not guarantee to find the minimal number of batches and it sometimes reverts to an undesirable two-phase commit approach [22] via packet tagging. This is suboptimal as it doubles the required (expensive) ternary content-addressable memory (TCAM) [15].

The tool Netstack [23] is a very recent addition to the family of update synthesis tools that support concurrent updates. Netstack reduces the concurrent update synthesis problem to Stackelberg games [25] but the update policies are restricted to basic reachability and waypointing. Our approach instead reduces the concurrent update problem to Petri games and moreover it allows for the specification of generic (regular) network policies. The performance of our tool Kaki is almost an order of magnitude faster than Netstack. To the best of our knowledge, FLIP and Netstack are the only tools supporting concurrent updates and we provide an extensive performance comparison of FLIP and Netstack against Kaki on a large benchmark of concurrent update problems.

The update synthesis problem via Petri games was recently studied in [4]. Our work generalises this work in several dimensions. The translation in [4] considers only sequential updates and reduces the problem to a simplistic type of game with only two rounds and only one environmental transition. Our translation uses the full potential of Petri games with multiple rounds where the controller and environment switch turns—this allows us to encode the concurrent update synthesis problem. Like many others [17, 18], the work in [4] fails to provide general forwarding policies and defines only a small set of predefined policies. Our tool, Kaki, solves the limitation by providing a regular language for the specification of forwarding policies and it is also the first tool that considers splittable flows with multiple (nondeterministic) forwarding.

A recent work introduces Snowcap [24], a generic update synthesis tool allowing for both soft and hard specifications. A hard specification specifies a forwarding policy, whereas the soft specification is a secondary objective that should be minimised. Snowcap uses LTL logic for the hard specification but it supports only sequential updates and, as documented in [23] on the same benchmark as used in this paper, it is significantly slower than the approach from [4] that we compare against in our experiments.

Other recent works relying on the Petri net formalism include timing analysis for network updates [2] and verification of concurrent network updates against Flow-LTL specifications [6], however, both approaches focus solely on the analysis/verification part for a given update sequence and do not discuss how to synthesise such sequences.

This paper is an extended version of the conference paper [12] with full proofs of all theorems and lemmas, additional examples in Figures 3 and 4 and their descriptions, and extended experiments that compare Kaki performance with a recently released tool Netstack [23] (Figure 9) and a comparison plot of deterministic and nondeterministic forwarding (Figure 10).

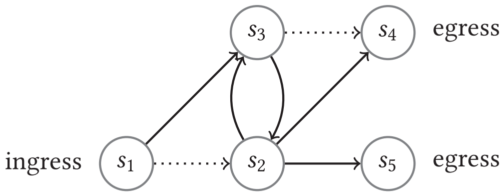


Fig. 1. Network and a routing function (dotted lines are links present in the network but not used in the routing) for the flow $\mathcal{F} = (\{s_1\}, \{s_4, s_5\})$ where $R(s_1) = \{s_3\}$, $R(s_2) = \{s_3, s_4, s_5\}$, $R(s_3) = \{s_2\}$ and $R(s_4) = R(s_5) = \emptyset$.

2 CONCURRENT UPDATE SYNTHESIS

We shall now formally define a network, routing of a flow in a network, flow policy as well as the concurrent update synthesis problem.

A *network* is a directed graph $G = (V, E)$ where V is a finite set of *switches* (nodes) and $E \subseteq V \times V$ is a set of *links* (edges) such that $(s, s) \notin E$ for all $s \in V$. A *flow* in a network is a pair $\mathcal{F} = (S_I, S_F)$ of one or more initial (*ingress*) switches and one or more final (*egress*) switches where $\emptyset \neq S_I, S_F \subseteq V$. A flow aims to forward packets such that a packet arriving to any of the ingress switches eventually reaches one of the egress switches. Packet forwarding is defined by network routing, specifying which links are used for forwarding of packets. Given a network $G = (V, E)$ and a flow $\mathcal{F} = (S_I, S_F)$, a *routing* is a function $R : V \to 2^V$ such that $s' \in R(s)$ implies that $(s, s') \in E$ for all $s \in V$, and $R(s_f) = \emptyset$ for all $s_f \in S_F$. We write $s \to s'$ if $s' \in R(s)$, as an alternative notation to denote the edges in the network that are used for packet forwarding in the given flow.

Figure 1 shows a network example together with its routing. Note that we allow nondeterministic forwarding as there may be defined multiple next-hops—this enables splitting of the traffic through several paths for load balancing purposes.

We now define a trace in a network as a maximal sequence of switches that can be observed when forwarding a packet under a given routing function. A *trace* t for a routing R and a flow $\mathcal{F} = (S_I, S_F)$ is a finite or infinite sequence of switches starting in some ingress switch $s_0 \in S_I$ where for the infinite case we have $t = s_0 s_1 \dots s_i \dots$ where $s_i \in R(s_{i-1})$ for $i \geq 1$, and for the finite case $t = s_0 s_1 \dots s_i \dots s_n$ where $s_i \in R(s_{i-1})$ for $1 \leq i \leq n$ and $R(s_n) = \emptyset$ for the final switch in the sequence s_n. For a given routing R and a flow $\mathcal{F}$, we denote by $T(R, \mathcal{F})$ the set of all traces.

In our example from Figure 1, the set $T(R, (\{s_1\}, \{s_4, s_5\}))$ contains e.g., the traces $s_1 s_3 s_2 s_4$, $s_1 s_3 s_2 s_3 s_2 s_4$ as well as the infinite trace $s_1 (s_3 s_2)^\omega$ that exhibits (undesirable) looping behaviour as the packets are never delivered to any of the two egress switches.

2.1 Routing Policy

A routing policy specifies all allowed traces on which packets (in a given flow) can travel. Given a network $G = (V, E)$, a *policy* P is a regular expression over V describing a language $L(P) \subseteq V^*$. Given a routing R for a flow $\mathcal{F} = (S_I, S_F)$, a policy P is *satisfied* by R if $T(R, \mathcal{F}) \subseteq L(P)$. Hence, all possible traces allowed by the routing must be in the language $L(P)$. As $L(P)$ contains only finite traces, if the set $T(R, \mathcal{F})$ contains an infinite trace then it never satisfies the policy P.

Our policy language can define a number of standard routing policies for a flow $\mathcal{F} = (S_I, S_F)$ in a network $G = (V, E)$.

- *Reachability* is expressed by the policy $(V \setminus S_F)^* S_F$. It ensures loop and black hole freedom as it requires that an egress switch must always be reached.

- *Waypoint enforcement* requires that packets must visit a given waypoint switch $s_w \in V$ before they are delivered to an egress switch (where, by our assumption, the trace ends) and it is given by the policy $V^* s_w V^*$.
- *Alternative waypointing* specifies two waypoints s and s' such that at least one of them must be visited and it is given by the union of the waypoint enforcement regular languages for s and s', or alternatively by $V^*(s + s')V^*$.
- *Service chaining* requires that a given sequence of switches $s_1, s_2, \dots, s_n$ must be visited in the given order and it is described by the policy $(V \setminus \{s_1, \dots, s_n\})^* s_1 (V \setminus \{s_2, \dots, s_n\})^* s_2 \dots (V \setminus \{s_n\})^* s_n V^*$.
- *Conditional enforcement* is given by a pair of switches $s, s' \in V$ such that if s is visited then s' must also be visited and it is given by the policy $(V \setminus \{s\})^* + V^* s' V^*$.

Regular languages are closed under union and intersection, hence the standard policies can be combined using Boolean operations. As reachability is an essential property that we always want to satisfy, we shall assume that the reachability property is always assumed in any other routing policy.

In our translation, we represent a policy by an equivalent nondeterministic finite automaton (NFA) $A = (Q, V, \delta, q_0, F)$ where Q is a finite set of states, V is the alphabet equal to set of switches, $\delta : Q \times V \rightarrow 2^Q$ is the transition function, q_0 is the initial state and F is the set of final states. We extend the δ function to sequences of switches by $\delta(q, s_0 s_1 \dots s_n) = \bigcup_{q' \in \delta(q, s_0)} \delta(q', s_1 \dots s_n)$ in order to obtain all possible states after executing $s_0 s_1 \dots s_n$. We define the language of A by $L(A) = \{w \in V^* \mid \delta(q_0, w) \cap F \neq \emptyset\}$. An NFA where $|\delta(q, s)| = 1$ for all $q \in Q$ and $s \in V$ is called a deterministic finite automaton (DFA). It is a standard result that NFA, DFA, and regular expressions have the same expressive power (w.r.t. the generated languages).

2.2 Concurrent Update Synthesis Problem

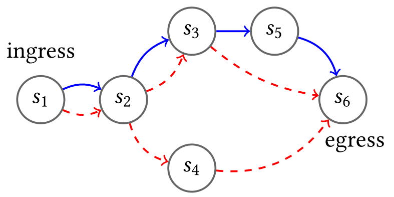
Let R_i and R_f be the *initial* and *final* routing, respectively. We aim to update the switches in the network so that the packet forwarding is changed from the initial to the final routing. The goal of the concurrent update synthesis problem is to construct a sequence of nonempty sets of switches, called *batches*. We want to guarantee that when we update the switches from their initial to the final routing in every batch concurrently (while waiting so that all updates in the batch are finished before we update the next batch), a given routing policy is transiently preserved. Our aim is to synthesise an update sequence that is optimal, i.e., minimises the number of batches.

During the update, only switches that actually change their forwarding function need to be updated. Given a network $G = (V, E)$, an initial routing R_i and a final routing R_f, the set of *update switches* is defined by $U = \{s \in V \mid R_i(s) \neq R_f(s)\}$. An *update* of a switch $s \in U$ changes its routing from $R_i(s)$ to $R_f(s)$.

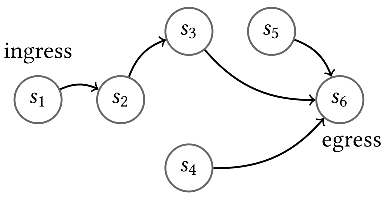
Definition 1. Let $G = (V, E)$ be a network, let R and R_f be the current and final routing, respectively, and let U the set of update switches. An *update* of a switch $s \in U$ results in the updated routing R^s given by

$$R^s(s') = \begin{cases} R(s') & \text{if } s \neq s' \\ R_f(s) & \text{if } s = s'. \end{cases}$$

A *concurrent update sequence* $\omega = X_1 \dots X_n \in (2^U \setminus \emptyset)^*$ is a sequence of nonempty batches of switches such that each update switch appears in exactly one batch of ω. As a network is a highly distributed system with asynchronous communication, the switch updates can be executed in any permutation of the batch, even if all switches in the batch are commanded to start the update at the same time. An *execution* $\pi = p_1 p_2 \dots p_n \in U^*$ respecting a concurrent update sequence



(a) Initial routing (solid lines) and a final routing (dashed lines).



(b) Intermediate routing after updating s_3 and s_4 in the first batch.

Fig. 2. Network with an optimal concurrent update sequence $\{s_3, s_4\}\{s_2, s_5\}$.

$\omega = X_1 \ldots X_n$ is the concatenation of a permutation of each batch in ω such that $p_i \in perm(X_i)$ for all i, $1 \leq i \leq n$, where $perm(X_i)$ denotes the set of all permutations of the switches in X_i.

Given a routing R and an execution $\pi = s_1 s_2 \ldots s_n$ where $s_i \in U$ for all i, $1 \leq i \leq n$, we inductively define the *updated routing* R^π by (*i*) $R^\epsilon = R$ and (*ii*) $R^{s\pi} = (R^s)^\pi$ where $s \in U$ and ϵ is the empty execution. An *intermediate routing* is any routing $R^{\pi'}$ where π' is a prefix of π. We notice that for any given routing R and any two executions π, π' that respect a concurrent update sequence $\omega = X_1 \ldots X_m$, we have $R^\pi = R^{\pi'}$, whereas the sets of intermediate routings can be different.

Given an initial routing R_i and a final routing R_f for a flow (S_I, S_F), a concurrent update sequence ω where $R_i^\omega = R_f$ *satisfies a policy* P if R' satisfies P for all intermediate routings R' generated by any execution respecting ω.

Definition 2. The *concurrent update synthesis problem* (CUSP) is a 5-tuple $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$ where $G = (V, E)$ is a network, $\mathcal{F} = (S_I, S_F)$ is a flow, R_i is an initial routing, R_f is a final routing, and P is a routing policy that includes reachability i.e., $L(P) \subseteq L((V \setminus S_F)^* S_F)$. A *solution* to a CUSP is a concurrent update sequence ω such that $R_i^\omega = R_f$ where ω satisfies the policy P and the sequence is *optimal*, meaning that the number of batches, $|\omega|$, is minimal.

Consider an example in Figure 2(a) where the initial routing is depicted in solid lines and the final one in dashed ones. We want to preserve the reachability policy between the ingress and egress switch. The set of update switches is $\{s_2, s_3, s_4, s_5\}$. Clearly, all update switches cannot be placed into one batch because the execution starting with the update of s_2 creates a possible black hole at the switch s_4. Hence, we need at least two batches and indeed the concurrent update sequence $\omega = \{s_3, s_4\}\{s_2, s_5\}$ satisfies the reachability policy. Any execution of the first batch preserves the reachability of the switch s_6 and brings us to the intermediate routing depicted in Figure 2(b). Any execution order of the second batch also preserves the reachability policy, implying that ω is an optimal concurrent update sequence.

3 OPTIMISATION TECHNIQUES

Before we present the translation of CUSP problem to Petri games, we introduce two preprocessing techniques that allow us to reduce the size of the problem.

3.1 Topological Decomposition

The intuition of topological decomposition is to reduce the complexity of solving CUSP $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$ where $G = (V, E)$ by decomposing it into two smaller subproblems. In the rest of this section, we use the aggregated routing $R_c(s) = R_i(s) \cup R_f(s)$ for all $s \in V$ (also denoted by the relation $\rightarrow$) in order to consider only the relevant part of the network.

We can decompose our problem at a switch $s_D \in V$ if s_D splits the network into two independent networks and there is at most one possible NFA state that can be reached by following any path

ALGORITHM 1: Potential NFA state set

```
   input  : A CUSP $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$ and NFA $A = (Q, V, \delta, q_0, F)$.
   output : Function $\mathcal{Q} : V \to 2^Q$ of potential NFA states at a given switch.
1  $\mathcal{Q}_f(s) := \emptyset$ and $\mathcal{Q}_b(s) := \emptyset$ for all $s \in V$
2  $\mathcal{Q}_f(s_i) := \delta(q_0, s_i)$ for all $s_i \in S_I$
3  $\mathcal{Q}_b(s_f) := F$ for all $s_f \in S_F$
   // $s \to s'$ can be relaxed if it changes $\mathcal{Q}_f(s')$ or $\mathcal{Q}_b(s)$
4  while there exists $s \to s' \in R_c$ that can be relaxed do
5      $\mathcal{Q}_f(s') := \mathcal{Q}_f(s') \cup \bigcup_{q \in \mathcal{Q}_f(s)} \delta(q, s')$
6      $\mathcal{Q}_b(s) := \mathcal{Q}_b(s) \cup \{q \in Q \mid \delta(q, s') \cap \mathcal{Q}_b(s') \neq \emptyset\}$
7  return $\mathcal{Q}(s) := \mathcal{Q}_f(s) \cap \mathcal{Q}_b(s)$ for all $s \in V$
```

from any of the ingress switches to s_D, and the path has a continuation to some of the egress switches while reaching an accepting NFA state. By $\mathcal{Q}(s)$ we denote the set of all such possible NFA states for a switch s. Algorithm 1 computes the set $\mathcal{Q}(s)$ by iteratively relaxing edges, i.e., by forward propagating the potential NFA states and storing them in the function $\mathcal{Q}_f$ and in a backward manner it also computes NFA states that can reach a final state and stores them in $\mathcal{Q}_b$. An edge $s \to s'$ can be relaxed if it changes the value of $\mathcal{Q}_f(s')$ or $\mathcal{Q}_b(s)$ and the algorithm halts when no more edges can be relaxed.

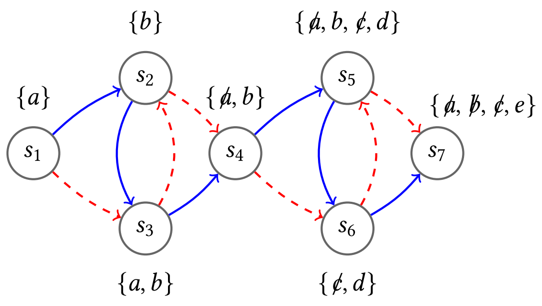
Figure 3 shows a network and a policy NFA. Here, the switches are annotated with the potential NFA states from the forward propagation, and those crossed out are the ones that are pruned by the backward propagation. For instance, at switch s_4 it is possible to be in either NFA state a or b, however, only the state b can reach a final state, since from state a the switch s_2 must be visited, which is impossible.

Lemma 1. *Let $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$ be a CUSP where $\mathcal{F} = (S_I, S_F)$ is a flow and let (Q, V, δ, q_0, F) be an NFA describing its routing policy P. Algorithm 1 terminates and the resulting function $\mathcal{Q}$ has the property that $q \in \mathcal{Q}(s_i)$ iff there exists a trace $s_0 \dots s_i \dots s_n \in T(R_c, \mathcal{F})$ such that $s_0 \in S_I$, $s_n \in S_F$, $q \in \delta(q_0, s_0 \dots s_i)$ and $\delta(q, s_{i+1} \dots s_n) \cap F \neq \emptyset$.*

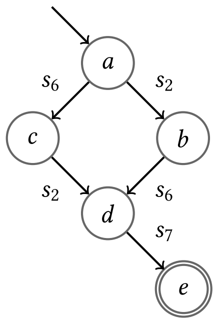
Proof. The algorithm terminates because in each iteration of the while loop, an NFA state is added either to Q_f or Q_b . Since there are only finitely many states, it must terminate.

We now prove that at line 7 the set $\mathcal{Q}_f(s)$ contains the NFA states can be reached from an initial switch to s, and afterwards, we prove that $\mathcal{Q}_b(s)$ contains the NFA states that can reach a final state from s. We prove by induction on the number of hops from an initial switch, with the induction hypothesis $H_f(n)$ = "*$q \in \mathcal{Q}_f(s)$ iff from the initial state, q can be reached by a path of length at most n from an initial switch to s*".
Base case (0 hops): This is trivially true, because the only switches reachable with no hops is the initial switches, and $\mathcal{Q}_f$ is initialised to the NFA states reached from q_0.
Induction step: Assume $H_f(n)$, we now show $H_f(n+1)$. ($\Rightarrow$) After the while loop has terminated, there are no more edges that can be relaxed forwards. Therefore, for switches s' where $s' \to s$, if an NFA state q can be reached in s' with n hops, then relaxing $s' \to s$ will ensure that $\delta(q, s) \subseteq \mathcal{Q}_f(s)$. ($\Leftarrow$) A state is only added when a relaxation adds NFA states that can be reached from the initial state (follows from the induction hypothesis), therefore no superfluous states are in $\mathcal{Q}_f(s)$.

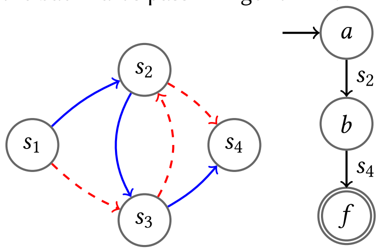
We now prove by induction on the number of hops to a final switch that $H_b(n)$ = "*$q \in \mathcal{Q}_b(s)$ iff from q a final state can be reached by a path of at most n switches from s to a final switch*".
Base case (0 hops): This is trivially true, because the only switches that can reach a final switch with no hops are final switches, and $\mathcal{Q}_f$ is initialised to the final NFA states.



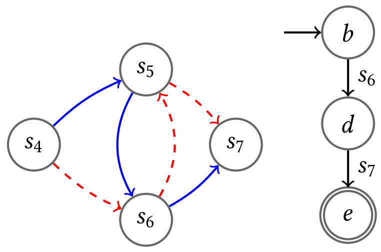
(a) Network. Blue solid arrow is initial routing and red dashed arrow is final routing. Each state is marked with the potential NFA states, where crossed out states are those removed doing the backwards pass in Algorithm 1.



(b) Policy NFA for reachability of s_7 and way-pointing on both s_2 and s_6. Self-loops are not drawn (each state has self-loops for all switches that are not an outgoing edge from the state).



(c) First subproblem, dealing only with the switches s_1, s_2, s_3, s_4 and the part of the policy NFA until state b but with the added condition that it must reach s_4.



(d) Second subproblem, dealing only with the switches s_4, s_5, s_6, s_7 and the part of the policy NFA from state b and onward.

Fig. 3. Example of network depicted in (a) and a simplified NFA for the policy seen on (b). The decomposition is shown in (c) and (d).

Induction step: Assume $H_b(n)$, we now show $H_b(n+1)$. ($\Rightarrow$) After the while loop, for switches s' where $s \to s'$, if an NFA state q can reach a final state from s' with n hops, then relaxing $s \to s'$ will ensure that $\{q' \in Q \mid q \in \delta(q', s')\} \subseteq Q_b(s)$. ($\Leftarrow$) A state is only added when a relaxation adds NFA states that can reach a final state, so from the induction hypothesis no superfluous states are added to $Q_b(s)$.

Finally, the intersection of Q_f and Q_b will contain only those states that can be reached from the initial switch and that can reach a final state. This proves both directions. $\square$

Definition 3. Let $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$ be a CUSP where $G = (V, E)$, $\mathcal{F} = (S_I, S_F)$ and where P is expressed by an equivalent NFA $A = (Q, V, \delta, q_0, F)$. A switch $s_D \in V$ is a *topological decomposition point* if $|Q(s_D)| = 1$ and for all $s \in V \setminus \{s_D\}$ either *(i)* $s \to^* s_D$ and $s_D \not\to^* s$ or *(ii)* $s \not\to^* s_D$ and $s_D \to^* s$.

We can notice that in the network from Figure 3(a) the switch s_4 is a topological decomposition point as it satisfies all conditions of Definition 3.

Let s_D be a decomposition point. We construct two CUSP subproblems $\mathcal{U}'$ and $\mathcal{U}''$, the first one containing the switches $V' = \{s \in V \mid s \to^* s_D\}$ and the latter one the switches $V'' = \{s \in V \mid s_D \to^* s\}$. Let $G[\overline{V}]$ be the induced subgraph of G restricted to the set of switches $\overline{V} \subseteq V$.

The first subproblem is given by $\mathcal{U}' = (G[V'], \mathcal{F}', R_i', R_f', P')$ where (i) $\mathcal{F}' = (S_I, \{s_D\})$, (ii) $R_i'(s) = R_i(s)$ and $R_f'(s) = R_f(s)$ for all $s \in V' \setminus \{s_D\}$ and $R_i'(s_D) = R_f'(s_D) = \emptyset$, and (iii) $L(P') = L(A') \cap L((V' \setminus \{s_D\})^* s_D)$ where $A' = (Q, V, \delta, q_0, F')$ with $F' = \mathcal{Q}(s_D)$. In other words, the network and routing are projected to only include the switches from V' and the policy ensures that we must reach s_D as well as the potential NFA state of s_D.

The second subproblem is given by $\mathcal{U}'' = (G[V''], \mathcal{F}'', R_i'', R_f'', P'')$ where (i) $\mathcal{F}'' = (\{s_D\}, S_F)$, (ii) $R_i''(s) = R_i(s)$ and $R_f''(s) = R_f(s)$ for all $s \in V''$, and (iii) $L(P'') = L(A'')$ where $A'' = (Q, V, \delta, q_0', F)$ and $\{q_0'\} = \mathcal{Q}(s_D)$. The policy of the second subproblem ensures that starting from the potential NFA state q_0' for the switch s_D, a final state of the original policy can be reached.

Figure 3 shows an example of topological decomposition. By analysing the network 3(a), we find that s_4 is a topological decomposition point because $\mathcal{Q}(s_4)$ only contains one viable NFA state, namely b. We then construct in Figures 3(c) and 3(d) two subproblems concerned with the switches s_1, s_2, s_3, s_4 and s_4, s_5, s_6, s_7, respectively. The concurrent update sequences solving the two subproblems are $\{s_2\}\{s_3\}\{s_1\}$ and $\{s_4\}\{s_5\}\{s_6\}$. Merging the solutions for the two subproblems yields the concurrent update sequence $\{s_2, s_4\}\{s_3, s_5\}\{s_1, s_6\}$ for the original problem. We shall now argue that such merging always produces an (optimal) concurrent update sequence.

First, we prove that from the optimal solutions of the subproblems, we can synthesise an optimal solution for the original problem.

Theorem 4. *Let $\omega' = X_1'X_2' \ldots X_j'$ and $\omega'' = X_1''X_2'' \ldots X_k''$ be optimal solutions for $\mathcal{U}'$ and $\mathcal{U}''$, respectively. Then $\omega = (X_1' \cup X_1'')(X_2' \cup X_2'') \ldots (X_m' \cup X_m'')$ where $m = \max\{j, k\}$ and where by conventions $X_i' = \emptyset$ for $i > j$ and $X_i'' = \emptyset$ for $i > k$, is an optimal solution to $\mathcal{U}$.*

Proof. We first prove that ω is a solution. Trivially, $R_i^\omega = R_f$ because $V' \cup V'' = V$, so all switches are updated. We show that for any prefix $\pi = s_i s_{i+1} \ldots s_n$ of any execution of ω the routing R_i^π satisfies the given policy P, and therefore that $t \in L(P)$ for all traces $t = s_0 s_1 \ldots s_n \in T(R_i^\pi, \mathcal{F})$. Let π' be the subsequence of π consisting of updates for switches from $\mathcal{U}'$, and π'' be those from $\mathcal{U}''$. We then examine the behaviour of the subproblems after the partial update. From the definition of $\mathcal{U}'$ we know that an injected packet must reach the decomposition point s_D. From the definition of $\mathcal{U}''$ we know that an injected package in s_D must reach a final switch. Therefore, the trace must be of the form $t = s_0 s_1 \ldots s_D \ldots s_n$ where $s_0 \in S_I$ and $s_n \in S_F$. By the assumption that ω' is correct, the trace $t' = s_0 s_1 \ldots s_D$ must end in the final state q_f of the NFA for $\mathcal{U}'$. By the assumption that ω'' is correct, the trace $t'' = s_D \ldots s_f$ starting from the state q_f must end in a final state of $\mathcal{U}$. Therefore, t must also satisfy P.

We now prove by contradiction that ω is optimal. Assume that there exists an $\overline{\omega} = X_1 \ldots X_k$ solution s.t. $|\overline{\omega}| < |\omega|$. We then pick the subproblem with the longest optimal solution, w.l.o.g. let it be ω'. Notice that $|\overline{\omega}| < |\omega'|$. We can then construct a new (and shorter) solution for this subproblem by extracting the update switches from the subproblem from $\overline{\omega}$, i.e., $\overline{\omega}' = (X_1 \cap V') \ldots (X_k \cap V')$. This contradicts ω' being an optimal solution. □

Second, we realise that a solution to $\mathcal{U}$ implies the existence of solutions to both $\mathcal{U}'$ and $\mathcal{U}''$.

Theorem 5. *If $\omega = X_1 \ldots X_n$ is a solution to $\mathcal{U}$ then $\omega' = (X_1 \cap V') \ldots (X_n \cap V')$ and $\omega'' = (X_1 \cap V'') \ldots (X_n \cap V'')$, where empty batches are omitted, are solutions to $\mathcal{U}'$ and $\mathcal{U}''$, respectively.*

Proof. The argument is similar to Theorem 4. Since the routings of the two subproblems do not affect the part of the policy they each are concerned with, delineated by the single potential NFA state of the decomposition point, the subproblems' updates are independent. Therefore, solutions to $\mathcal{U}'$ and $\mathcal{U}''$ can directly be extracted from ω. □

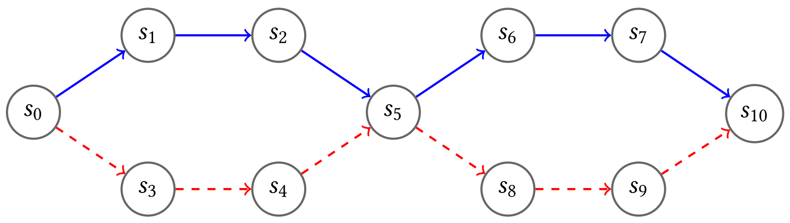


Fig. 4. Network with initial and final routing. $\aleph_i = \{s_3, s_4, s_8, s_9\}$ and $\aleph_f = \{s_1, s_2, s_6, s_7\}$.

Hence, if the original problem has a solution and can be decomposed into two subproblems, then these subproblems also have solutions and from the optimal solutions of the subproblems, we can construct an optimal solution for the original problem. Importantly, since the subproblems are themselves also CUSPs, they may be subject to further decompositions.

3.2 Collective Update Classes

We now present the notion of a *collective update class*, or simply *collective updates*, which is a set of switches that can be always updated in the same batch in an optimal concurrent update sequence. The switches in a collective update class can then be viewed only as a single switch, thus reducing the complexity of the synthesis by reducing the number of update switches.

The first class of collective updates is inspired by [4] where the authors realize that in case of sequential updates, update switches that are undefined in the initial routing can be always updated in the beginning of the update sequence and similarly update switches that should become undefined in the final routing can always be moved to the end of the update sequence. Consider e.g., Figure 4 where we can w.l.o.g. assume that the routers s_3, s_4, s_8, and s_9 can be all updated (initialised) in the first batch and the update (removal of forwarding rules) of the routers s_1, s_2, s_6, and s_7 can be scheduled in the last batch. This observation is generalised (for concurrent update sequences) in the following theorem.

Theorem 6. *Let $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$ be a CUSP. Let $\aleph_i = \{s \in V \mid R_i(s) = \emptyset \ \wedge R_f(s) \neq \emptyset\}$ and $\aleph_f = \{s \in V \mid R_f(s) = \emptyset \ \wedge R_i(s) \neq \emptyset\}$. If $\mathcal{U}$ is solvable then it has an optimal solution of the form $X_1 \dots X_n$ where $\aleph_i \subseteq X_1$ and $\aleph_f \subseteq X_n$.*

Proof. Let $\omega = X_1 \dots X_n$ be an optimal concurrent update sequence. Recall that P must contain reachability. The switches in $\aleph_i$ and $\aleph_f$ can only be updated when they are not reachable, because otherwise they create a black hole. Additionally, updating an unreachable switch does not violate the policy as it does not affect the traces of the current routing. The switches in $\aleph_i$ have no initial next-hop, and therefore they are not in the initial routing; otherwise, it violates reachability. Therefore, $\aleph_i$ is not reachable in the first batch and can therefore be in the first batch. There are no other switches in the first batch whose update can make any switch in $\aleph_i$ reachable, because if a switch s makes some switch in $\aleph_i$ reachable, then the intermediate routing after updating s creates a black hole, and therefore ω is not a solution. Similarly, $\aleph_f$ cannot be reachable in the last batch, and those switches can therefore be updated in the last batch. □

In Figure 5 we show another class of collective updates with a chain-like structure where the initial and final routings forward packets in opposite directions. We claim that the switches $\aleph_c = \{s_3, s_4, s_5\}$ can be always updated in the same batch. As long as the intermediate routing is passing through the switches, updating any switch in $\aleph_c$ introduces a looping behaviour, and hence they cannot be updated at this moment. Once the switches in $\aleph_c$ are not a part of the intermediate

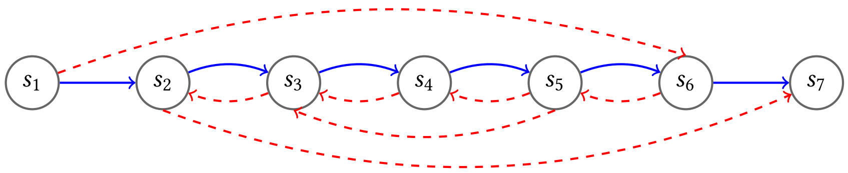


Fig. 5. Chain structure with initial (solid) and final (dashed) routings.

routing, we can update all of them in the same batch without causing any forwarding issues. The notion of chain-reducible collective updates is formalised as follows.

Definition 7. Let $C \subseteq V$ be a strongly connected component w.r.t. $\rightarrow$ such that $|C| \geq 4$. The triple $(s_e, s_{e'}, C)$, where $s_e, s_{e'} \in C$, is *chain-reducible* if it satisfies:

(i) if $s \in C \setminus \{s_e, s_{e'}\}$ and $s' \rightarrow s$ then $s' \in C$,
(ii) if $s \in C \setminus \{s_e, s_{e'}\}$ and $s \rightarrow s'$ then $s' \in C$, and
(iii) for every $s \in C \setminus \{s_e, s_{e'}\}$ if there exists a switch $s' \in R_f(s)$ then $s' \rightarrow^* s$ using only the initial routing or $R_i(s') = \emptyset$.

The restriction $|C| \geq 4$ is included so that reduction in size can be achieved. Cases (i) and (ii) ensure that the switches in $C \setminus \{s_e, s_{e'}\}$ do not influence or are influenced by any of the switches not in C and can be part of a collective update. Case (iii) guarantees that updating a reachable switch $s \in C \setminus \{s_e, s_{e'}\}$ induces either a loop or a black hole.

Theorem 8. *Let $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$ be a CUSP and let $(s_e, s_{e'}, C)$ be chain-reducible and let $\aleph_c = C \setminus \{s_e, s_{e'}\}$. If $\mathcal{U}$ has an optimal solution $\omega = X_1 \dots X_n$ then there exists another optimal solution $\omega' = X_1 \setminus \aleph_c \dots X_k \cup \aleph_c \dots X_n \setminus \aleph_c$ for some k, $1 \leq k \leq n$.*

Proof. Let X_k be the first batch of the optimal concurrent update sequence $\omega = X_1 \dots X_k \dots X_n$ that contains a switch $s \in \aleph_c$, where s is routed to in both the initial and final routing. We construct another concurrent update sequence $\omega' = X_1 \setminus \aleph_c \dots X_k \cup \aleph_c \dots X_n \setminus \aleph_c$ and prove that it is an optimal solution to $\mathcal{U}$.

Let $s_k \in \aleph_c \cap X_k$ be one of the switches first updated in $\aleph_c$. Notice that P always contains reachability and by *(iii)* that updating any switch $s \in \aleph_c$ introduces a loop or black hole if $\aleph_c$ is reachable, therefore, $s \in \aleph_c$ can only be updated when $\aleph_c$ is unreachable. The collective update class $\aleph_c$ can only again become reachable when it is completely updated as it transiently contains loops. By *(i)* and *(ii)* only s_e and $s_{e'}$ have incoming or outgoing routings of C, therefore, all other switches $s \in \aleph_c$ have no influence on any intermediate routing of ω. Therefore, all switches of $\aleph_c$ can be updated in X_k since their updates cannot change the traces of any intermediate routing, i.e., $T(R^{\pi_i}, \mathcal{F}) = T(R^{\pi'_i}, \mathcal{F})$, for all prefixes π_i of π, where π respects ω and for all prefixes π'_i of π', where π' respects ω'. □

4 TRANSLATION TO PETRI GAMES

We shall first present the formalism of Petri games and then reduce the concurrent update synthesis problem to this model.

4.1 Petri Games

A Petri net is a mathematical model for distributed systems focusing on concurrency and asynchronicity (see [20]). A Petri game [4, 10] is a 2-player game extension of Petri nets, splitting the transitions into controllable and environmental ones. We shall reduce the concurrent update synthesis problem to finding a winning strategy for the controller in a Petri game with a reachability objective.

A *Petri net* is a 4-tuple (P, T, W, M) where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \emptyset$, $W : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}^0$ is a weight function and $M : P \rightarrow \mathbb{N}^0$ is an initial marking that assigns a number of tokens to each place. We depict places as circles, transitions as rectangles and draw an arc (directed edge) between a transition t and place p if $W(t, p) > 0$, or place p and transition t if $W(p, t) > 0$. When an arc has no explicit weight annotation, we assume that it has the weight 1.

The semantics of a Petri net is given by a labeled transition system where states are Petri net markings and we write $M \xrightarrow{t} M'$ if $M(p) \geq W(p, t)$ for all $p \in P$ (the transition t is enabled in M) and $M'(p) = M(p) - W(p, t) + W(t, p)$.

Marking properties are given by a formula φ which is a Boolean combination of the atomic predicates of the form $p \bowtie n$ where $p \in P$, $\bowtie \in \{<, \leq, >, \geq, =, \neq\}$ and $n \in \mathbb{N}^0$. We write $M \models p \bowtie n$ iff $M(p) \bowtie n$ and extend this naturally to the Boolean combinators. We use the classical CTL operator AF and write $M \models AF\ \varphi$ if *(i)* $M \models \varphi$ or *(ii)* $M' \models AF\ \varphi$ for all M' such that $M \xrightarrow{t} M'$ for some $t \in T$, meaning that on any maximal firing sequence from M, the marking property φ must eventually hold.

A *Petri game* [4, 10] is a two-player game extension of Petri nets where transitions are partitioned $T = T_{ctrl} \uplus T_{env}$ into two distinct sets of *controller* and *environment* transitions, respectively. During a play in the game, the environment has a priority over the controller in the decisions: the environment can always choose to fire its own fireable transition, or ask the controller to fire one of the controllable transitions. The goal of the controller is to find a strategy in order to satisfy a given $AF\ \varphi$ property whereas the environment tries to prevent this. Formally, a (controller) *strategy* is a partial function $\sigma : \mathcal{M}_N \rightharpoonup T$, where $\mathcal{M}_N$ is the set of all markings, that maps a marking to a fireable controllable transition (or it is undefined if no such transition exists). We write $M \xrightarrow{t}_\sigma M'$ if $M \xrightarrow{t} M'$ and $t \in T_{env} \cup \{\sigma(M)\}$. A Petri game satisfies the reachability objective $AF\ \varphi$ if there exists a controller strategy σ such that the labelled transition system under the transition relation $\rightarrow_\sigma$ satisfies $AF\ \varphi$.

4.2 Translation Intuition

We now present the intuition for our translation from CUSP to Petri games. For a given CUSP instance, we compositionally construct a Petri game where the controller's goal is to select a valid concurrent update sequence and the environment aims to show that the controller's update sequence is invalid. The game has two phases: generation phase and verification phase.

The generation phase has two modes where the controller and environment switch turns in each mode. The controller proposes the next update batch (in a mode where only controller's transitions are enabled) and when finished, it gives the turn to the environment that sequentialises the batch by creating an arbitrary permutation of the update switches in the batch (in this mode only environmental transitions are enabled). At any moment during the batch sequentialisation, the environment may decide to enter the second phase that is concerned with validation of the current intermediate routing.

The verification phase begins when the environment injects a packet (token) to the network and wishes to examine the currently generated intermediate routing. In this phase, a next hop of the packet is simulated in the network according to the current switch configuration; in case of nondeterministic forwarding it is the environment that chooses the next switch. A hop in the network is followed by an update of the current state of a DFA that represents the routing policy.

These two steps alternate, until (i) an egress switch is reached, (ii) the token ends in a black hole (deadlock), or (iii) the packet forwarding forms a loop, wherefrom the execution is deadlocked by only allowing to visit each switch once. The controller wins the game only in situation (i), providing that the currently reached state in the DFA is an accepting state.

The controller now has a winning strategy if and only if the CUSP problem has a solution. By restricting the number of available batches and using the bisection method (binary search), we can further identify an optimal concurrent update sequence.

4.3 Translation of Network Topology and Routings

Let $(G, \mathcal{F}, R_i, R_f, P)$ be a concurrent update synthesis problem where $G = (V, E)$ is a network and $\mathcal{F} = (S_I, S_F)$ is the considered flow. We construct a Petri game $N(\mathcal{U}) = (P, T, W, M)$. This subsection describes the translation of the network and routings, and the next subsection deals with the policy translation.

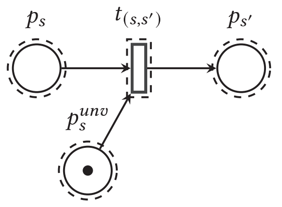
Figure 6 shows the Petri game building blocks for translating the network and the routings. Environmental transitions are denoted by empty rectangles and controller transitions are depicted as black/filled. The captions of each subfigure quantify for which switches such components are created. The final net is then constructed as a composition of all such components and if a transition/place is surrounded by a dashed line then it has only a single copy in the final net—such a place/transition is shared across all components that use this transition/place.

Network Topology Component (Figure 6(a)). This component represents the network and its current routing. For each $s \in V$, we create the shared places p_s and a shared unvisited place p_s^{unv} with one token. The unvisited place tracks whether the switch has been visited and prevents looping. We use uncontrollable transitions so that the environment can decide how to traverse the network in case of nondeterminism. The switch component ensures that these transitions are only fireable in accordance with the current intermediate routing.

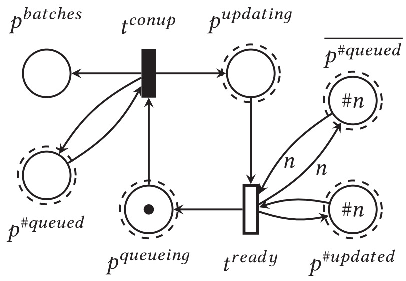
Update Mode Component (Figures 6(b) and 6(d)). These components handle the bookkeeping of turns between the controller and the environment. A token present in the place $p^{queueing}$ enables the controller to queue updates into a current batch. Once the token is moved to the place $p^{updating}$, it enables the environment to schedule (in an arbitrary order) the updates from the batch. The dual places $p^{\#queued}$ and $\overline{p^{\#queued}}$ count how many switches have been queued in this batch and how many switches have not been queued, respectively. The place $p^{\#updated}$ is decremented for each update implemented by the environment. Hence, the environment is forced to inject a token to the network, latest once all update switches are updated. Additionally, the number of produced batches is represented by the number of tokens in the place $p^{batches}$.

Switch Component (Figure 6(c)). This component handles the queueing (by controller) and activation (by environment) of updates. For every $s \in V$ where $R_i(s) \neq R_f(s)$ we create a switch component. Let U be the set of all such update switches. Initially, we put one token in p_s^{init} (the switch forwards according to its initial routing) and $p_s^{limiter}$ (making sure that each switch can be queued only once). Once a switch is queued (by the controller transition t_s^{queue}) and updated (by the environment transition t_s^{update}), the token from p_s^{init} is moved into p_s^{final} and the switch is now forwarding according to the final routing function.

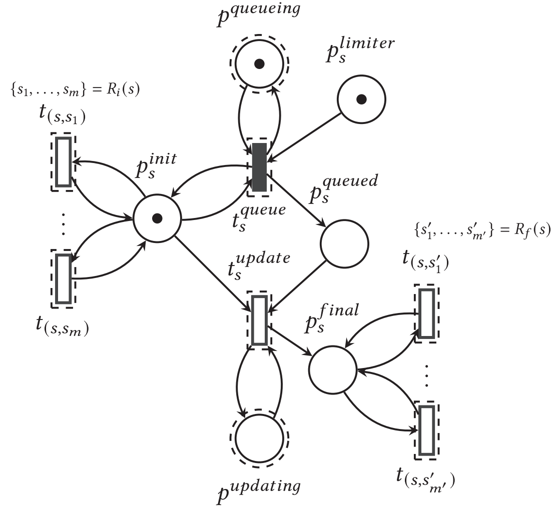
Packet Injection Component (Figure 6(e)). The environment can at any moment during the sequentialisation mode use the transition t_s^{inject} to inject a packet into any of the ingress routers and enter the second verification phase.



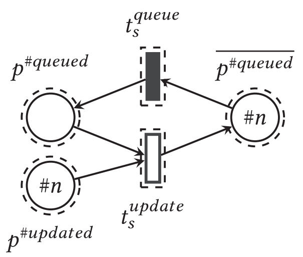
(a) Topology component for each switch s and $s' \in R_i(s) \cup R_f(s)$.



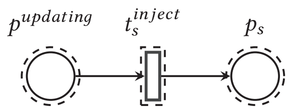
(b) Update mode component where $n = |U|$.



(c) Switch component for each $s \in U$. Transitions $t(s,s_1)\ldots t(s,s_m)$ are for the initial routing; $t(s,s'_1)\ldots t(s,s'_{m'})$ for the final one.



(d) Counter component where $n = |U|$ added for each $s \in U$.



(e) Packet injection component for every $s \in S_I$ in flow (S_I, S_F).

Fig. 6. Construction of Petri game components; U is the set of update switches.

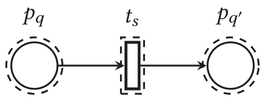
4.4 Policy Translation

Given a CUSP $(G, \mathcal{F}, R_i, R_f, P)$, we now want to encode the policy P into the Petri game representation. We assume that P is given by a DFA $A(P)$ such that $L(P) = L(A(P))$. We translate $A(P)$ into a Petri game so that DFA states/transitions are mapped into corresponding Petri net places/transitions which are connected to the earlier defined Petri game for the topology and routing.

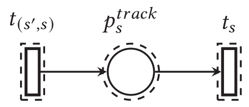
Figure 7 presents the components for the policy translation.

(1) *DFA transition component (Figure 7(a)).* This component creates places/transitions for each DFA state/transition. Note that if a Petri game transition is of the form t_s then it corresponds to a DFA-transition, contrary to transitions of the form $t_{(s,s')}$ that represent network topology links.

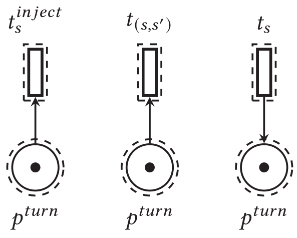
(2) *Policy tracking component (Figure 7(b)).* For all $s \in V$, we create the place p_s^{track} in order to track the current position of a packet in the network.



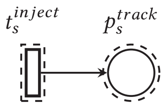
(a) Component for each DFA transition $q \xrightarrow{s} q'$; if $q = q_0$ then p_q gets a token.



(b) Tracking component for each already added transition $t_{(s',s)}$ and each switch $s \in V$; creates a new transition t_s.



(c) Turn component for all created transitions t_s^{inject} and $t_{(s,s')}$ and t_s.



(d) Injection component for each $s \in S_I$ in the flow (S_I, S_F).

Fig. 7. Policy checking components.

(3) *Turn component (Figure 7(c)).* The intuition here is that whenever the environment fires the topology transition $t_{(s,s')}$ then the DFA-component must match it by firing a DFA-transition $t_{s'}$. The token in the place p^{turn} means that it is the environment turn to challenge with a next hop in the network topology.

(4) *DFA injection component (Figure 7(d)).* For all inject transitions t_s^{inject} to the switch s, we add an arc to its tracking place p_s^{track}. This initiates the second phase of verification of the routing policy.

4.5 Reachability Objective and Translation Correctness

We finish by defining the reachability objective $C(k)$ for each positive number k that gives an upper bound on the maximum number of allowed batches (recall that F is the set of final DFA states): $C(k) = AF\ p^{batches} \leq k \wedge \bigvee_{q \in F} p_q = 1$.

The query expresses that all runs that follow the controller's strategy must use fewer than k batches and eventually end in an accepting DFA state. Note that since reachability is assumed as a part of the policy P and that the final switch has no further forwarding, there can be no next-hop in the network after the DFA gets to its final state.

The query can be iteratively verified (e.g., using the bisection method) while changing the value of k, until we find k such that $C(k)$ is true and $C(k-1)$ is false (which implies that also $C(\ell)$ is false for every $\ell < k-1$). Then we know that the synthesised strategy is an optimal solution. If $C(k)$ is false for $k = |U|$ where U is the set of update switches then there exists no concurrent update sequence solving the CUSP. The correctness of the translation is summarised in the following theorem.

Theorem 9. *A concurrent update synthesis problem $\mathcal{U}$ has a solution with k or fewer batches if and only if there exists a winning strategy for the controller in the Petri game $N(\mathcal{U})$ for the query $C(k)$.*

Let us note that a winning strategy for the controller in the Petri game can be directly translated to a concurrent update sequence. The firing of controllable transitions of the form t_s^{queue} indicates that the switch s should be scheduled in the current batch and the batches are separated from each other by the firings of the controllable transitions t^{conup}.

4.6 Correctness of Translation

Let $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$ be a concurrent update synthesis problem, and let $N(\mathcal{U}) = (P, T, W, M)$ be the Petri game resulting from translating $\mathcal{U}$ into a Petri game using the translation process from Section 4.2. Also, let $C(k)$ be the query from Section 4.5.

We first want to prove that the state-space of the constructed Petri game is finite. This is done by proving that there exists no infinite run in $N(\mathcal{U})$.

Theorem 10. *Given the CUSP $\mathcal{U} = (G, \mathcal{F}, R_i, R_f, P)$, the Petri game $N(\mathcal{U}) = (P, T, W, M)$ never produces an infinite run.*

Proof. First, observe that the update switch component transitions t_s^{queue} and t_s^{update} can be fired at most once. The transition t_s^{queue} is restricted by the place $p_s^{limiter}$, and t_s^{update} can only be fired after t_s^{queue} has been fired. Second, t_s^{inject} can be fired exactly once because it removes the token from $p^{updating}$, and $p^{updating}$ can never regain its lost token. Third, any topology transition $t_{(s,s')}$ can be fired at most once. This is ensured by the limiter place $p_{s'}^{unv}$ as it contains one token by the initial marking, and it never regains tokens.

Notice that the transitions t^{conup} can happen at most $|U|$ times since it requires a token from $p^{\#queued}$, and such a token indicates that a switch update has been queued. Furthermore, t^{ready} can only fire after t^{conup} has fired, which can therefore also only fire a finite number of times. Lastly regarding the policy-component, the turn switch enforces that any DFA-transition t_s can only fire after a topology transition $t_{(s',s)}$ or t_s^{inject} has fired, which both only happen a finite number of times. □

We now prove the correctness of Theorem 9. The theorem states a bi-implication; therefore, its proof is divided into two separate lemmas, which are presented below. First, we prove that if ω is a solution to a CUSP $\mathcal{U}$ then there exists a winning strategy σ for $N(\mathcal{U})$ with the query $C(k)$.

Lemma 2. *If ω is a solution to a CUSP $\mathcal{U}$, where $|\omega| \leq k$, then there exists a winning strategy σ for the controller player in the Petri game $N(\mathcal{U})$ with the query $C(k)$.*

Proof. Let $\omega = X_1 \dots X_k$ be a concurrent update sequence, s.t. $X_i \subseteq U$. We now define a winning strategy σ w.r.t. $C(k)$ for the controller, starting with the initial marking M_0.

Notice that if $M(p^{queueing}) = 1$ then only the controller can fire transitions. After t^{conup} is fired, the token of $P^{queueing}$ is moved to $p^{updating}$, and the environment can update switches (or alternatively inject a packet), and at some point move the token back by firing t^{ready}. The strategy of the controller is to fire all queue transitions that correspond to the batches from ω, starting with X_1 and followed by X_2, X_3, and so on, in the next rounds. The controller queues a batch $X = \{s_1, \dots, s_n\}$ by firing the transitions $t_{s_1}^{queue} \dots t_{s_n}^{queue} t^{conup}$—this adds a token to $p^{batches}$ and gives the turn to the environment. Notice that the order in which the transitions t_s^{queue} are fired is irrelevant.

During the updating phase, i.e., when $M(p^{updating}) = 1$, the environment is able to fire transitions corresponding to the switches that were queued by the controller, trying to find their permutation breaking the given policy. Hence, if the controller fired t_s^{queue} in the queuing phase then the transition t_s^{update} will be fired by the environment during its following updating phase (where no controllable transitions are enabled so there is no need to define the controller's strategy here).

We now prove that $M_0 \models C(k)$ under the strategy σ. Recall $C(k)$ from Section 4.5, which states that for all possible runs of $N(\mathcal{U})$ the number of batches used is limited to k and after t_s^{inject} is fired, any sequence of transition firings (determined purely by the environment) results in an accepting DFA state.

The predicate $AF(p^{batches} \leq k)$ is assured because each batch adds a token to $p^{batches}$ and we have k batches and every batch is queued exactly once. We then argue that t_s^{inject} is guaranteed to fire eventually, so that we must eventually enter the verification phase. The environment can inject in the generation phase anytime it is its turn, and it is forced to do so after all switches have been updated. This enforcement is ensured by the place $p^{\#updated}$ as it loses a token after each update, and after all updates are executed, the transition t^{ready} can no longer be fired, and inject is the only option left for the environment.

We now prove that any run after firing of t_s^{inject} always results in $M(p_q) = 1$ for some $q \in F$, assuming that the controller follows the strategy σ. Once the Petri game enters the verification phase by the environment firing the transition t_s^{inject}, the place p_s^{track} gets a token. Now, the environment chooses the only available transition t_s, as all other transitions are unfireable because they lack a token in their respective track place; this removes a token from p_{q_0} and p_s^{track} and puts a token into $p_{q'}$. After this the environment fires some transition $t_{(s,s_j)}$ in the topology and a token is put into $p_{s_j}^{track}$. Again, the environment is forced to match this by firing a transition t_{s_j}; and so on. Effectively, the DFA-component matches the trace that the environment simulates in a turn-wise manner. Any path the environment can simulate this way is a trace in some intermediate routing of ω, and we know all possible intermediate routings of ω satisfy the policy P. Therefore, any simulation path chosen by the environment results in an accepting DFA-state. □

We now prove the other implication of Theorem 9.

Lemma 3. *If σ is a winning strategy for the controller in the Petri game $N(\mathcal{U})$ with the query $C(k)$ then there exists a solution ω to the CUSP $\mathcal{U}$, where $|\omega| = k$.*

Proof. Let σ be a winning strategy for the Petri Game $N(\mathcal{U})$ with the query $C(k)$. Whenever $p^{queueing} = 1$ then σ must fire one or more queue transitions and then the t^{conup} transition. Therefore, the strategy must be sequences of $t_{s_1}^{queue} \dots t_{s_j}^{queue} \dots t_{s_n}^{queue} t^{conup}$ repeated i times, where $1 \leq i \leq k$. This naturally produces a concurrent update sequence $\omega = X_1 \dots X_k$. In between the queuing of batches, the environment updates the queued switches and has the option to fire the inject transition at any moment. However, because σ is a winning strategy, no inject can violate the policy. We now prove by contradiction that the derived concurrent update sequence ω satisfies the policy P. Assume that ω does not satisfy P, then there must exist an execution of ω where its prefix $\pi = s_1 s_2 \dots s_k$ yields a routing R_i^π s.t. $t \notin L(P)$ for some $t \in T(R_i^\pi, \mathcal{F})$. However, such a trace cannot exist: in the Petri game, the environment is able to simulate the intermediate routing R_i^π by updating switches in correspondence with π. It can then inject a token and enter the verification phase. If the produced trace t is an infinite trace then the network topology will deadlock due to the p_s^{unv} places, and σ is not a winning strategy; if t is finite, then $M(p_q) \neq 1$ for all $q \in F$ because the DFA in the Petri game recognises exactly P, but this also contradicts σ being a winning strategy. Therefore, ω satisfies P.

Finally, $|\omega| \leq k$ because $\sigma \models C(k)$ which implies that $M_i(p^{batches}) \leq k$ for all markings M_i of σ. Therefore, there are queued no more than k batches. □

5 EXPERIMENTAL EVALUATION

We implemented the translation approach and optimisation techniques in our tool Kaki. The tool is coded in Kotlin and compiled to JVM. It uses the Petri game engine of TAPAAL [3, 9, 10] as its backend for solving the Petri games. The source code of Kaki is publicly available on GitHub.[1]

[1]https://github.com/Ragusaen/Kaki.

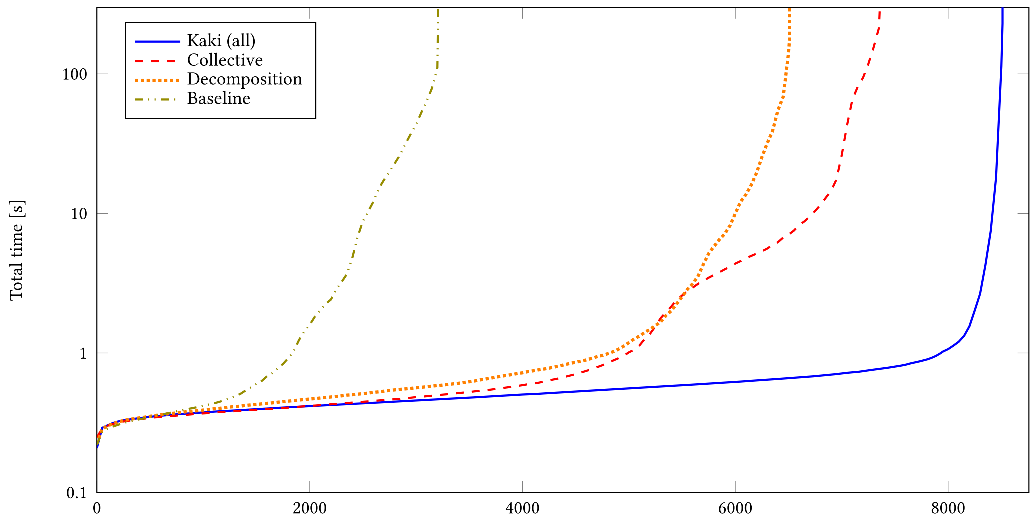


Fig. 8. Kaki optimisation techniques comparison (y-axis is logarithmic) on extended benchmark.

We shall discuss the effect of our novel optimisation techniques and compare the performance of our tool to FLIP [26] and Netstack [23] as well as the tool for sequential update synthesis from [4], referred to as SEQ. We use the benchmark [5] of update synthesis problems from [4], based on 229 real-network topologies from the Internet Topology Zoo database [13]. The benchmark includes four update synthesis problems for reachability and single waypointing for each topology, totalling 916 problem instances. As Kaki and FLIP support a richer set of policies, we further extend this benchmark with additional policies for multiple waypointing, alternative waypointing, and conditional enforcement, giving us 8,759 instances of the concurrent update synthesis problem.

All experiments (each using a single core) are conducted on a compute-cluster running Ubuntu version 18.04.5 on an AMD Opteron(tm) Processor 6376 with a 1GB memory limit and 5 minute timeout. A reproducibility package is available in [11] and it includes executable files to run Kaki, pre-generated outputs that are used to produce the figures as well as the benchmark and related scripts.

5.1 Results

To compare the Kaki optimisation techniques introduced in this paper, we include a baseline without any optimisation techniques, its extension with only topological decomposition technique and only collective update classes, and also the combination of both of them. Each method decides the existence of a solution for the concurrent update synthesis problem and in the positive case it also minimises the number of batches. Figure 8 shows a cactus plot of the results where the problem instances on the x-axis are (for each method independently) sorted by the increasing synthesis time shown on the y-axis. The experiments are run on the extended benchmark and we can observe that both of the optimisation techniques provide a significant improvement over the baseline and their combination is clearly beneficial as it solves 97% of the problems in the benchmark within the 5 minute timeout.

In Figure 9 we also show a cactus plot for Kaki, FLIP, and Netstack on the benchmark of concurrent update synthesis problems that include reachability and waypointing only (because Netstack cannot handle other network policies). As Kaki has to first generate the Petri game file and then call the external TAPAAL engine for solving the Petri game, there is an initial overhead that

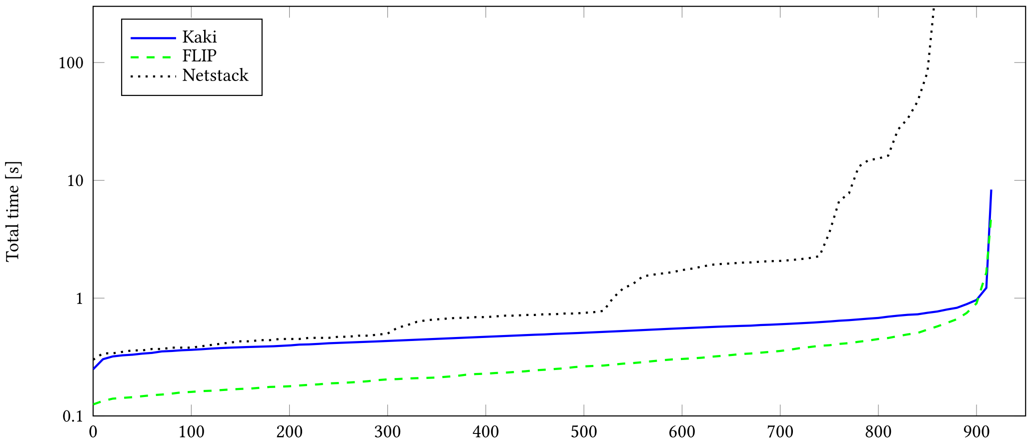


Fig. 9. Comparison with FLIP and Netstack (y-axis is logarithmic) on basic benchmark.

Table 1. Number of Solved Problems for Kaki and FLIP (Suboptimal and Tagging Refers to FLIP)

	reachability	1-wp	2-wp	4-wp	8-wp	1-alt-wp	2-alt-wp	4-alt-wp	1-cond-enf	2-cond-enf	all	percentage
Total	856	916	916	844	647	916	916	916	916	916	8759	100.0%
Only Kaki	0	0	17	37	63	0	5	8	1	2	133	1.5%
Only FLIP	0	0	0	0	0	17	20	35	40	84	196	2.2%
Suboptimal	0	11	18	14	4	283	198	104	41	114	787	9.0%
Tagging	0	0	47	55	21	4	39	100	1	1	268	3.1%

implies that the single-purpose tool FLIP is faster on the smaller and easy-to-solve instances of the problem that can be answered below 1 second. For the more difficult instances both Kaki and FLIP obtain a similar performance and solve the most difficult instance in 8.3 and 5.1 seconds, respectively. The most recent tool Netstack computes the optimal solutions similarly as Kaki, however, at significantly slower running times and it times out for the more challenging instances of the problems.

We also notice that FLIP does not always produce the minimal number of batches, which is critical for practical applications because updating a switch can cause forwarding instability for up to 0.4 seconds [21]. Hence, minimising the number of batches where switches can be updated in parallel significantly decreases the forwarding vulnerability (some networks in the benchmark have up to 700 switches). In fact, on the full benchmark of concurrent update synthesis problems, FLIP synthesises a strictly larger number of batches in 787 instances, compared to the minimum number of possible batches (that Kaki is guaranteed to find). The distribution of the solved problems for the different policies is shown in Table 1. Here we can also notice that FLIP uses the less desirable tag-and-match update strategy in 268 problem instances, even though there exists a concurrent update sequence as demonstrated by Kaki. In conclusion, Kaki has a slightly larger overhead on easy-to-solve instances but scales almost as well as FLIP, however, FLIP in more than 12% of cases does not find the optimal update sequence or reverts to the less desirable two-phase commit protocol.

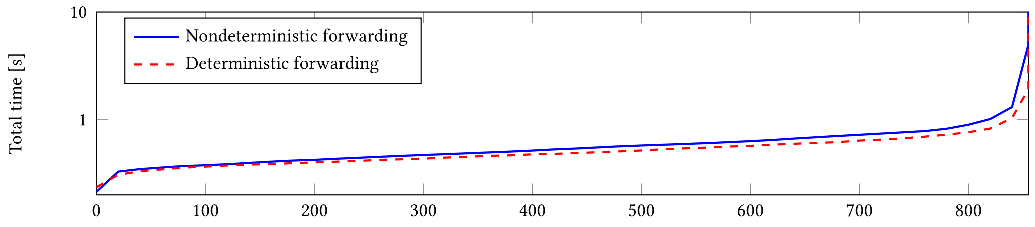


Fig. 10. Total time taken for Kaki using splittable and nonsplittable forwarding.

Comparison with SEQ from [4] is more difficult as SEQ supports only reachability and single waypointing and computes only sequential updates (single switch per batch). When we restrict the benchmark to the subset of these policies and adapt our tool to produce sequential updates, we observe that Kaki's performance is in the worst case 0.06 seconds slower than SEQ when measuring the verification time required by the TAPAAL engine. We remark that SEQ solved all problems in under 0.55 seconds, except for two instances where it timed out, while Kaki was able to solve both of them in under 0.1 second.

We further enlarged the extended benchmark with nondeterministic forwarding that models splittable flows (using the Equal-Cost-MultiPath (ECMP) protocol [8] that divides a flow along all shortest paths from an ingress to an egress switch). We observe that verifying the routing policies in this modified benchmark implies only a negligible (3.4% on the median instance) overhead in running time. The running times are summarised in Figure 10.

6 CONCLUSION

We presented Kaki, a tool for update synthesis that can deal with (i) concurrent updates, (ii) synthesises solutions with a minimum number of batches, (iii) extends the existing approaches with nondeterministic forwarding and can hence model splittable flows, and (iv) verifies arbitrary (regular) routing policies. It extends the state-of-the-art approaches with respect to generality but given its efficient TAPAAL backend engine, it is also fast and provides more optimal solutions compared to the competing tool FLIP and runs almost an order of magnitude faster than the tool Netstack.

Kaki's performance is the result of its efficient translation in combination with optimisations techniques that allow us to reduce the complexity of the problem while preserving the optimality of its solutions. Kaki uses less than 1 second to solve 90% of all concurrent update synthesis problems for real network topologies and hence provides a practical approach to concurrent update synthesis.

ACKNOWLEDGMENTS

We thank Peter G. Jensen for his help with executing the experiments and Anders Mariegaard for his assistance with setting up FLIP.

REFERENCES

[1] Zhiruo Cao, Zheng Wang, and Ellen W. Zegura. 2000. Performance of hashing-based schemes for internet load balancing. In *Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26–30, 2000.* IEEE Computer Society, 332–341. https://doi.org/10.1109/INFCOM.2000.832203

[2] N. Christesen, M. Glavind, S. Schmid, and J. Srba. 2020. Latte: Improving the latency of transiently consistent network update schedules. In *IFIP PERFORMANCE'20 (Performance Evaluation Review)*, Vol. 48, no. 3. ACM, 14–26.

[3] A. David, L. Jacobsen, M. Jacobsen, K. Y. Jørgensen, M. H. Møller, and J. Srba. 2012. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12) (LNCS)*, Vol. 7214. Springer-Verlag, 492–497.

[4] Martin Didriksen, Peter G. Jensen, Jonathan F. Jønler, Andrei-Ioan Katona, Sangey D. L. Lama, Frederik B. Lottrup, Shahab Shajarat, and Jiří Srba. 2021. Automatic synthesis of transiently correct network updates via Petri games. In *Application and Theory of Petri Nets and Concurrency*, Didier Buchs and Josep Carmona (Eds.). Springer International Publishing, Cham, 118–137.

[5] Martin Didriksen, Peter G. Jensen, Jonathan F. Jønler, Andrei-Ioan Katona, Sangey D. L. Lama, Frederik B. Lottrup, Shahab Shajarat, and Jiří Srba. 2021. Artefact for: Automatic Synthesis of Transiently Correct Network Updates via Petri Games. (Feb. 2021). https://doi.org/10.5281/zenodo.4501982

[6] B. Finkbeiner, M. Gieseking, J. Hecking-Harbusch, and E.-R. Olderog. 2020. AdamMC: A model checker for Petri nets with transits against flow-LTL. In *CAV'20 (LNCS)*, Vol. 12225. Springer, 64–76.

[7] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. 2019. Survey of consistent software-defined network updates. *IEEE Commun. Surv. Tutorials* 21, 2 (2019), 1435–1461.

[8] Christian Hopps et al. 2000. *Analysis of an Equal-cost Multi-path Algorithm.* Technical Report. RFC 2992, November.

[9] J. F. Jensen, T. Nielsen, L. K. Oestergaard, and J. Srba. 2016. TAPAAL and reachability analysis of P/T nets. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)* 9930 (2016), 307–318. https://doi.org/10.1007/978-3-662-53401-4_16

[10] P. G. Jensen, K. G. Larsen, and J. Srba. 2016. Real-time strategy synthesis for timed-arc Petri net games via discretization. In *Proceedings of the 23rd International SPIN Symposium on Model Checking of Software (SPIN'16) (LNCS)*, Vol. 9641. Springer-Verlag, 129–146. https://doi.org/10.1007/978-3-319-32582-8_9

[11] N. S. Johansen, L. B. Kær, A. L. Madsen, K. Ø. Nielsen, J. Srba, and R. G. Tollund. 2022. Artefact for Kaki: Concurrent Update Synthesis for Regular Policies via Petri Games. (Oct. 2022). https://doi.org/10.5281/zenodo.6379555 https://doi.org/10.5281/zenodo.6379555.

[12] N. S. Johansen, L. B. Kaer, A. L. Madsen, K. O. Nielsen, J. Srba, and R. G. Tollund. 2022. Kaki: Concurrent update synthesis for regular policies via Petri games. In *Proceedings of the 17th International Conference on Integrated Formal Methods (iFM'22) (LNCS)*, Vol. 13274. Springer-Verlag, 249–267.

[13] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Alistair Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE J. Sel. Areas Commun.* 29, 9 (2011), 1765–1775. https://doi.org/10.1109/JSAC.2011.111002

[14] K. G. Larsen, A. Mariegaard, S. Schmid, and J. Srba. 2022. AllSynth: Transiently correct network update synthesis accounting for operator preferences. In *Proceedings of the 16th International Symposium on Theoretical Aspects of Software Engineering (TASE'22) (LNCS)*, Vol. 13299. Springer, 344–362.

[15] Alex X. Liu, Chad R. Meiners, and Eric Torng. 2010. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.* 18, 2 (2010), 490–500. https://doi.org/10.1145/1816262.1816274

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

[17] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. 2015. Scheduling loop-free network updates: It's good to relax!. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21–23, 2015*, Chryssis Georgiou and Paul G. Spirakis (Eds.). ACM, 13–22. https://doi.org/10.1145/2767386.2767412

[18] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. 2014. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks, HotNets-XIII, Los Angeles, CA, USA, October 27–28, 2014*, Ethan Katz-Bassett, John S. Heidemann, Brighten Godfrey, and Anja Feldmann (Eds.). ACM, 15:1–15:7. https://doi.org/10.1145/2670518.2673873

[19] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. 2015. Efficient synthesis of network updates. *SIGPLAN Not.* 50, 6 (June 2015), 196–207. https://doi.org/10.1145/2813885.2737980

[20] Tadao Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580.

[21] Peter Pereíni, Maciej Kuzniar, Marco Canini, and Dejan Kostić. 2014. ESPRES: Transparent SDN update scheduling. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*. Association for Computing Machinery, New York, NY, USA, 73–78. https://doi.org/10.1145/2620728.2620747

[22] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. In *ACM SIGCOMM 2012 Conference, Helsinki, Finland*, Lars Eggert, Jörg Ott, Venkata N. Padmanabhan, and George Varghese (Eds.). ACM, 323–334.

[23] Stefan Schmid, Bernhard Clemens Schrenk, and Álvaro Torralba. 2022. NetStack: A game approach to synthesizing consistent network updates. In *IFIP Networking Conference, IFIP Networking 2022, Catania, Italy, June 13–16, 2022.* IEEE, 1–9.

[24] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. 2021. Snowcap: Synthesizing network-wide configuration updates. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23–27, 2021*, Fernando A. Kuipers and Matthew C. Caesar (Eds.). ACM, 33–49. https://doi.org/10.1145/3452296.3472915

[25] Patrick Speicher, Marcel Steinmetz, Michael Backes, Jörg Hoffmann, and Robert Künnemann. 2018. Stackelberg planning: Towards effective leader-follower state space search. *Proceedings of the AAAI Conference on Artificial Intelligence* 32, 1 (2018).

[26] Stefano Vissicchio and Luca Cittadini. 2016. FLIP the (flow) table: Fast lightweight policy-preserving SDN updates. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10–14, 2016*. IEEE, 1–9.

Received 22 October 2022; revised 4 April 2023; accepted 11 June 2023