NetStack: A Game Approach to Synthesizing Consistent Network Updates

Stefan Schmid
TU Berlin & University of Vienna

Bernhard Clemens Schrenk
University of Vienna

Álvaro Torralba
Aalborg University

***Abstract*—Availability and policy-compliance of many communication networks must be guaranteed at any time, even during updates. Accordingly, over the last years, the problem of how to update networks in a manner which *transiently* preserves desirable properties, has received much attention, especially in the context of Software-Defined Networks (SDNs). While important insights have been obtained for many different problem variants, in general, the design of efficient network update algorithms remains challenging, and usually new algorithms have to be developed on a case-by-case basis.**

This paper is motivated by the vision of fully automated communication networks in which consistent update schedules are synthesized automatically. In particular, we propose a game approach to the network update synthesis problem, and present *NetStack*, a tool based on Stackelberg games which transiently ensures fundamental properties such as reachability, loop-freedom, and waypointing. Our approach features a high flexibility. For example, with a simple extension, *NetStack* can also support concurrent updates, where in each round multiple routers are updated simultaneously. Our empirical evaluation shows that *NetStack* scales to realistic network sizes, and can compute optimal concurrent schedules.

I. Introduction

Communication networks are the backbone of our digital society. The constant availability of communication services is critical for private and business users, and due to dual-use technology, even for emergency response and crisis communication. In order to meet their stringent dependability requirements, communication networks need to provide reachability and be policy-compliant *even during updates*. Indeed, with the increasing scale and the current trend to operate networks in a more adaptive or even "self-driving" manner, such updates are likely to become more frequent.

However, ensuring even basic correctness properties, such as reachability, loop-freedom, or waypointing, during network updates is challenging: Even if the old and and the new network configurations are policy-compliant and valid, these properties may be violated *transiently*. Given the relevance of the problem, over the last years, hundreds of approaches to schedule network updates consistently have been proposed in the literature, especially in the context of Software-Defined Networks (SDNs). Most of the solutions to this NP-hard [1] problem however are manually-designed algorithms or operational approaches [2].

In this paper, we are interested in automated tools to *synthesize* network update schedules which provably ensure different properties transiently. Our perspective is motivated by the recent success of automated tools to verify and synthesize a *given* configuration of networks [3]–[5], as well as the emergence of first automated tools to *update* configurations. For example, to update networks consistently, the NetSynth tool uses counterexample-guided search and incremental model checking [6], the recent Petri nets tool models the router network as a distributed system [7], [8], and Snowcap synthesizes configuration updates considering hard and soft goals [9].

Our main ***contribution*** is *NetStack*[1], a flexible synthesis tool for the network update problem, based on a Stackelberg planning game. Our tool guarantees reachability, loop freedom, and waypoint enforcement throughout the whole update process, and also supports concurrent updates, finding the optimal solution with the minimum number of update rounds. We report on an empirical evaluation using a state-of-the-art Stackelberg planning tool, and find that compared to existing tools, *NetStack* can solve most of the real-world network examples within comparable time and memory limits. We are not aware of any other such game approach to solving the network update problem.

To support follow-up work and to ensure reproducibility, we make all sources, tools, scripts and experimental data publicly available as a reproducibility package together with this paper [10].

II. Model and Preliminaries

In this section we present the network update problem more formally. We also introduce the preliminaries which are necessary to be able to model the problem with Stackelberg planning games in the following sections, and to synthesize consistent network update schedules.

In general, the network update problem asks for a schedule (a sequence of rounds), consisting of configuration update operations, which transition a communication network from an old to a new configuration while preserving consistency guarantees at any time. Each round consists of an update sent from an SDN controller to the routers, waiting for the acknowledgements that confirm the successful update, and

We thank Thomas A. Henzinger from IST Austria for suggesting to look into synthesis games. Research supported by the Vienna Science and Technology Fund (WWTF), ICT19-045 (WHATIF), 2020-2024.

ISBN 978-3-903176-48-5© 2022 IFIP

[1]The term 'stack' in *NetStack* stands for Stackelberg.

then initiating the next round. The update operations are in general modifications of rules in the routing table. The consistency guarantees are similar to those checked during configuration validation and will be formally defined later.

If every round contains only one update, we say that the schedule is sequential. It is possible that a round can contain multiple updates, i.e., changes of multiple rules at multiple nodes are triggered. We will call these concurrent updates. For concurrent updates, operations within a round are not executed simultaneously by the routers. This causes the relative order of the updates within a round to be undefined [2].

A. Network, Paths, and Routing

In this paper we follow the definitions as given by Didriksen et al. [7]:

Definition (Network)**.** *A network is a directed graph with routers represented as nodes and links as edges.*

Each router contains a routing table (the configuration) with rules defining which link is taken for a specific packet type. The packet type is defined by its header attributes, mainly by its destination. We can treat each packet type as an independent (separate) problem.

Definition (Routing)**.** *The outgoing link for our considered packet type as implied by the rules in one router is defined as the routing of the packet in that router.*

Definition (Routing Configuration)**.** *The set of routings for all routers is the routing configuration.*

A packet following the forwarding rules (i.e., the routing) from router to router results in a sequence of visited routers.

Definition (Path)**.** *The sequence of routers a packet travels along, according to the routing rules (the route implied by the different rules), is called a path.*

B. Consistency Guarantees

We aim to preserve consistency guarantees covering connectivity and policy aspects as they are typically considered in the literature [2].

Definition (Reachability)**.** *The guarantee* reachable(u,v) *is satisfied if there is a path from node u to node v.*

Definition (Loop-Freedom)**.** *The guarantee* loopfree(u) *is satisfied if the path starting at node u does not end in a loop (does not hit the same node more than once).*

Loop-Freedom is given if the path ends in a node without a defined routing. For a combined Loop-Freedom and Reachability guarantee, with the same starting node u, the final node v must therefore be the one without a rule for further routing. These two properties together give the connectivity guarantee.

Definition (Waypointing)**.** *The guarantee* waypoint(u,v,w) *is satisfied if every path from node u to node v contains the waypoint node w.*

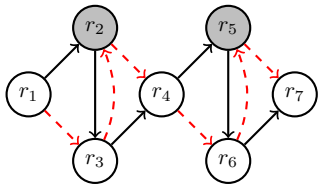


Fig. 1. A network with 7 nodes. The initial routing is shown with black arrows and the final routing is shown with red dashed arrows. Reachability $r_1 \rightarrow r_7$ must be guaranteed and the waypoints at r_2 and r_5 must always be in the path.

Waypointing is a policy guarantee and is often used to guarantee that all packets get routed via special nodes like firewall nodes.

C. Network Updates and Scheduling

To replace an existing network configuration with a new one, the routing rules must be updated. So we define the following for the sequential update case:

Definition (Routing Update)**.** *The change of the routing of our considered packet type within one router from an existing to a new routing, is called routing update.*

As we consider only one packet type and at most one update per node, the existing routing is the routing of the initial configuration, and the new routing is the routing according to the final configuration. This also means that nodes with a different initial and final routing will be updated exactly once and nodes with the same initial and final routing will not be updated at all. For a router r, with $init(r)$ we denote the router pointed in the initial configuration and with $end(r)$ the one pointed in the final configuration.

To transition the whole network from the initial to the final configuration, a sequence of multiple routing updates is necessary. As we use sequential updates, each step in this sequence will represent one round of network updates.

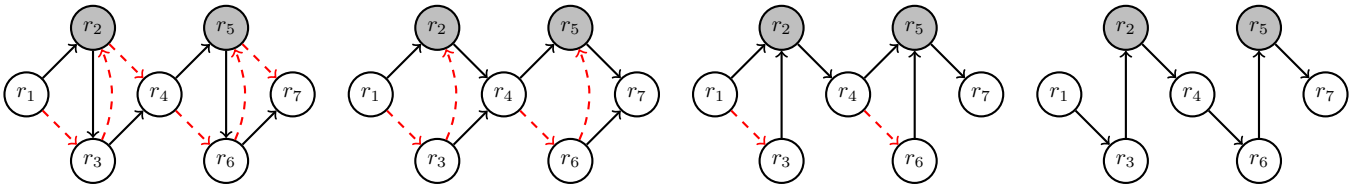
Definition (Network Update Schedule)**.** *A network update schedule is a sequence of routing updates which transition the network from the initial routing to the final routing configuration.*

After each round, a different routing configuration is active. The consistency guarantees can be checked after each round.

Definition (Consistent Network Update Schedule)**.** *A network update schedule is consistent in respect to a given set of consistency guarantees if and only if after each round of the update schedule all given consistency guarantees are fulfilled.*

The goal is now to find a consistent network update schedule which transitions our network configuration from the initial to the final routing configuration. We can therefore define:

Definition (Network Update Problem)**.** *The network update problem for a given set of initial and final routings and a given set of consistency guarantees, is to find a consistent network update schedule which transitions the network from the initial*



Initial configuration **Batch 1:** Updating r_2 and r_5. **Batch 2:** Updating r_3 and r_6. **Batch 3:** Updating r_1 and r_4.

Fig. 2. A consistent concurrent network update schedule for the network update problem from Fig. 1. Current routings shown with black arrows, target routing shown using red dashed arrows.

to the final routings, fulfilling the consistency guarantees in each round.

Fig. 1 shows a network with 7 nodes. Packets originating at r_1 must reach r_7 and always travel over r_2 and r_5. Six of those routers need to be updated and therefore a sequential update schedule needs six rounds, e.g., $r_2, r_3, r_1, r_5, r_6, r_4$. By contrast, the simple $r_1, r_2, r_3, r_4, r_5, r_6$ is not consistent as, after updating r_1, packets could go to r_6 without passing via r_2.

D. Concurrent Updates

In addition to be consistent, network update schedules should also be short, using a minimal number of rounds. As a schedule for sequential updates contains per definition only one node per round, to be able to reduce the number of update rounds, the network updates should be combined into batches.

Definition (Update Batch)**.** *We define an update batch as a set of routing updates which can be scheduled concurrently but their execution order is undefined.*

As the execution order of the routing updates within a batch is not defined, all permutations of the updates can occur. Therefore the consistency guarantees must be checked for every permutation.

Definition (Consistent Update Batch)**.** *An update batch is consistent with respect to a given set of consistency guarantees if and only if every permutation of a subset of the update batch generates a routing configuration which fulfills the consistency guarantees. We define such an update batch as a consistent update batch.*

Those update batches can be combined into a new network update schedule.

Definition (Consistent Concurrent Network Update Schedule)**.** *We call a combination of consistent update batches, which transition the network from the initial to the final routing configuration, as a* consistent concurrent network update schedule.

The number of batches can be any number from 1 (assuming that there is at least 1 routing update) up to the number of necessary routing updates (containing 1 routing update per batch).

Definition (Batch Count)**.** *We define the number of batches within a concurrent network update schedule as the batch count.*

The goal is to minimize the batch count to reduce the amount of update rounds necessary to reconfigure the network.

Definition (Optimal Consistent Concurrent Network Update Schedule)**.** *We define every* consistent concurrent network update schedule *for a network update problem for which no other* consistent concurrent network update schedule *with a smaller batch count exists as an* optimal consistent concurrent network update schedule.

The network shows two similar parts (separated by r_4). Those parts of the network can be updated concurrently reducing the rounds to 3 update batches, each updating 2 routers, as shown in Fig. 2. As there is no other consistent update schedule with less batches this is an optimal consistent concurrent network update schedule.

E. Stackelberg Planning and Games

In this paper, we propose a Stackelberg approach to flexibly and automatically solve the network update problem. More specifically, we are interested in Stackelberg planning, a framework introduced by Speicher et al. [11] where there are two players, a leader and a follower, that perform a sequence of actions. The leader acts first, executing the entire action sequence. Then, after observing the resulting outcome, the follower decides on its own sequence of actions. The objective of the follower is to minimize the cost of its own plan, which is just the sum of the costs of the actions performed, or infinity, ensuring that the resulting state satisfies a given goal condition, while minimizing the cost of its own action sequence.

The follower aims to perform the sequence of actions of minimum cost such that the resulting state satisfies a given goal condition. The leader executes a sequence of actions before the follower, aiming to find a sequence of actions of minimum cost, such as the resulting state maximizes the followers plan cost. An equilibrium is a pair of a leader and a follower plan, such that the leader cannot decrease the cost of his own plan without decreasing the cost for the follower as well. Multiple equilibria may be found, producing a Pareto frontier.

Formally, a Stackelberg planning task is a tuple $(\mathcal{F}, \mathcal{A}_L, \mathcal{A}_F, \mathcal{I}, \mathcal{G}_F)$ [11]: $\mathcal{F}$ is a set of Boolean propositions

that may be true or false. A state $s \subseteq \mathcal{F}$ is a subset of facts, corresponding to the facts that are true in s. $\mathcal{I}$ is the initial state. The goal condition $\mathcal{G}_F \subseteq \mathcal{F}$ is a set of facts that the follower must make true, so that the follower's plan must end in a state s_G such that $\mathcal{G}_F \subseteq s_G$.

$\mathcal{A}_L$ and $\mathcal{A}_F$ are the set of actions available to the leader and follower agents, respectively. Each action $a \in \mathcal{A}_L \cup \mathcal{A}_F$ has a precondition $pre(a) \subseteq \mathcal{F}$, a negative precondition $pre^-(a) \subseteq \mathcal{F}$, add effects $add(a) \subseteq \mathcal{F}$, delete effects $del(a) \subseteq \mathcal{F}$, and a non-negative cost $cost(a) \in \mathbb{Z}^+$. An action a is applicable in a state s if its preconditions are true, i.e., $pre(a) \subseteq s$ and $pre^-(a) \cap s = \emptyset$. In that case, the result of applying a in s is $s[a] = (s \setminus del(a)) \cup add(a)$, i.e., the state is changed by making all facts in add (a) true and all facts in del (a) false. All other facts remain unchanged. A sequence of actions $\pi = a_1, \ldots, a_n$ is applicable in s_0 if a_i is applicable in s_{i-1} and $s_i = a_i(s_{i-1})$ for all $i \in [n]$. The resulting state is $\pi(s_0) = s_n$. The cost of a sequence of actions is just the summed up costs of the individual actions $cost(\pi) = \sum_{i \in [n]} cost(a_i)$.

A leader plan π_L is any sequence of leader actions in $\mathcal{A}_L$ that is applicable in $\mathcal{I}$. A follower plan is any sequence of actions in $\mathcal{A}_F$ that is applicable in $\pi_L(\mathcal{I})$ such that the resulting state satisfies the goal: $\pi_F(\pi_L(\mathcal{I})) \subseteq \mathcal{G}_F$.

Given a leader plan, π_L, the corresponding follower cost is the minimum number of actions in any valid follower plan or ∞ if no such plan exists. The objective of the leader is to minimize the cost of their own plan, while maximizing the follower cost. The result is a Pareto frontier containing all non-dominated plans. The best leader and follower plans, building up the Pareto frontier, can be found by solving the Stackelberg task with an off-the-shelf solver/planning tool. Here, we will be interested in what is the minimum cost under which the leader can force an infinite cost for the follower. In that case, we say that the task is solvable by the leader.

III. Sequential Updates

We first present *NetStack*'s synthesis approach for the sequential update problem. In particular, we show how the network update problem can be reduced to a Stackelberg planning task, by splitting it into two phases: Finding a network update schedule followed by validating its consistency. These two phases can be perfectly distributed between leader and follower. The leader's plan consists of a sequence of router updates, assigning each switch to a separate batch. After this, the follower's goal is to prove that the leader plan is not a valid consistent network update schedule. Therefore, the follower plan consists of a trace that breaks one or more guarantees. The leader plan is therefore a consistent network update schedule if and only if the follower cannot find a plan to reach its goal.

Given a network update problem, we define the Stackelberg planning task as follows. We denote the set of routers by R. We define a set of time steps $T = \{t_0, \ldots, t_{|R|}\}$, one for each router, plus a last step. To model the network guarantees, we define a set of packets P, such that we have one packet $p \in P$ for every reachability guarantee $(p.u, p.v)$. We denote $p.W$ to be the set of routers such that there is a guaranteed waypoint $(p.u, p.v, p.w)$ for $p.w \in p.W$.

Algorithm 1: Sequential Network Update

```
Data: Network update problem: routers R, packets P
Result: Consistent network update schedule
while router permutations left do
    leaderPlan ←
     LeaderChoosesShortestUpdateSequence();
    if not all routers assigned in leaderPlan then
        continue;
    if ConsistentSchedule (leaderPlan) then
        return leaderPlan;
return No consistent schedule exists;

Function ConsistentSchedule(leaderPlan):
    foreach timestep ∈ t_0, ..., t_|R| do
        // Follower Constructs Routing Table
        foreach router in leaderPlan do
            if router in leaderPlan after timestep then
                routingTable[router] ← init(router);
            else
                routingTable[router] ← end(router);
        // Follower sends packets
        foreach p ∈ P do
            packetPos ← p.u;
            packetPath ← {p.u};
            while packetPos ≠ p.v do
                if packetPos is dead-end(p.v) then
                    return false;
                // move packet
                packetPos ←
                 routingTable[packetPos];
                // check for loop
                if packetPos in packetPath then
                    return false ;
                packetPath.insert(packetPos);
            if not every waypoint in packetPath then
                return false ;
    return true;
```

Algorithm 1 shows a pseudocode that encodes the same checks as are encoded in the Stackelberg planning task. Both are logically equivalent in the sense that they will return the same result. We remark, however, that algorithms solving the Stackelberg planning task will use optimizations to avoid enumerating all possible options.

Then, the Stackelberg task has the following facts:

- $update_{r,t}$: whether the leader has chosen to update router $r \in R$ at time step $t \in T$.
- $updated_r$: whether the leader has chosen to update router $r \in R$ at any time step.
- $step\text{-}update_t$: whether time step t is the first one that has not been assigned by the leader.
- $step\text{-}attack_t$: whether the follower decides to attack the update schedule right before time step t.
- $chosen$: whether the follower has already chosen at which time step to attack.
- $link_{r,r'}$: whether packets can be moved from r to r'
- $at_{p,r}$: whether packet p is at router r.

TABLE I
ACTIONS IN THE STACKELBERG TASK.

Action name	Parameter Constraints	Precondition	Add	Delete
SCHEDULE-UPDATE$^L(r,t)$		$\neg updated_r$, $step\text{-}update_t$	$updated_r$, $update_{r,t}$, $step\text{-}update_{t+1}$	$step\text{-}update_t$
CHOOSE-STEP$^F(t)$		$\neg chosen$	$step\text{-}attack_t$, $chosen$	
ADD-LINK$^F(r,r',t,t')$	$(t' < t \wedge r' = end(r)) \vee$ $(t' \geq t \wedge r' = init(r))$	$step\text{-}attack_t$, $update_{r,t'}$	$link_{r,r'}$	
MOVE$^F(p,r,r')$	$r \neq p.v$	$link_{r,r'}$, $at_{p,r}$	$at_{p,r'}$, $visited_{p,r'}$	$at_{p,r}$
REACHED-LOOP$^F(p,r,r')$	$r \neq p.v$	$link_{r,r'}$, $at_{p,r}$, $visited_{p,r'}$	$schedule\text{-}invalid$	
REACHED-DEAD-END$^F(p,r,r')$	$r \neq p.v, r' \in dead_{p.v}$	$link_{r,r'}$, $at_{p,r}$,	$schedule\text{-}invalid$	
WAYPOINT-MISSED$^F(p,r,r',w)$	$r' = p.v, w \in p.W$	$link_{r,r'}$, $at_{p,r}$, $\neg visited_{p,w}$	$schedule\text{-}invalid$	
NOT-SCHEDULED$^F(r)$		$\neg updated_r$	$schedule\text{-}invalid$	

- $visited_{p,r}$: whether packet p has passed through r.
- $schedule\text{-}invalid$: whether the follower has proven the schedule suggested by the leader to be invalid.

In the initial state, all facts are false except $step\text{-}update_0$, $at_{p,p.u}$, and $visited_{p,p.u}$ for the starting position of each packet. The follower goal is $\mathcal{G}_F = \{schedule\text{-}invalid\}$. Table I details the actions that are available for the leader and the follower, parametrized by time steps $t, t' \in T$, routers $r, r', w \in R$, and packets $p \in P$. The task has an action for each combination of the parameters, subject to the parameter constraints.

A. Leader Plan

All leader actions are of the form SCHEDULE-UPDATE$^L(r,t)$ and correspond to scheduling the update of r at time step t. Applying the action sets the corresponding $update_{r,t}$ fact to true, and increases the time step. This ensures that the leader can update exactly one router at each time step. The precondition $\neg updated_r$ ensures that the same router cannot be assigned to two time-steps so each leader action sequence corresponds to an update schedule, i.e., a permutation of (a subset of) the routers.

B. Follower Plan

The follower plan starts after the leader has decided the update schedule by setting the value of all $update_{r,t}$ variables. The follower aims to attack the network update sequence by finding a situation where a consistency guarantee is violated. To do so, there are follower actions that will allow the follower to reach the goal fact $schedule\text{-}invalid$ if and only if the schedule chosen by the leader is not valid. We identify four conditions under which the schedule is invalid:

1) Some router has not been scheduled.
2) A packet can perform a loop, reaching the same node twice.
3) A packet can reach a dead-end from which the target cannot be reached. We denote by $dead_r$ the set of routers from which r cannot be reached.
4) A packet can reach its target without passing through some waypoint.

The first case, is easily handled by actions NOT-SCHEDULED$^F(r)$ that immediately add the follower's goal and whose precondition evaluates to true if there is at least one router not assigned.

For the other cases, the follower needs to move a packet across the network in a way that is consistent with the update schedule chosen by the leader. To do that, the first action, CHOOSE-STEP$^F(t)$, chooses a single update round under which the packets will be sent. Then, the follower can apply ADD-LINK^F actions to generate the routing table for the network at the chosen update round. The parameter constraints model the fact that all routers updated in a previous round will be assigned to its final routing, whereas the remaining routers will use their initial routing.

Once links have been established, the follower also has actions MOVE$^F(p,r,r')$ to move packets through the network. According to the initial state, the packet starts at node u defined by the guarantee which needs to be checked. Moving the packet ends after it reached the target node v. This is specified by the parameter constraint, which forbids moving a package if it is already at the target router.

The last three actions correspond to the winning conditions for the follower. REACHED-LOOP^F checks if the packet can be moved to a router that has already been visited (as tracked by the $visited_{p,r}$ facts). REACHED-DEAD-END^F checks if it is possible to reach a node from which the target is unreachable. Finally, WAYPOINT-MISSED^F checks if it is possible to reach the target without visiting any of the waypoints. If any of those actions is applicable, the follower can achieve the goal, proving the schedule chosen by the leader to be invalid.

Proposition 1. *Given a Network Update Problem, the Pareto front of the corresponding Stackelberg planning task contains an entry whose follower cost is ∞ if and only if a consistent network update is possible.*

Proof Sketch. Assume the contrary.

Case 1: If the Pareto front does not have an entry with follower cost ∞, then there is no consistent schedule. Assume that there is a consistent network update. Then, the leader plan π_L corresponding to such update ends in some state $\pi_L(\mathcal{I})$ for which there exists a follower plan reaching $schedule\text{-}invalid$. Such a plan provides a trace of MOVE^F actions that violates a reachability and/or waypoint guarantee, contradicting that the network update is consistent.

Case 2: If the Pareto front has an entry with follower cost ∞, then there is a network update corresponding to the leader plan π_L. If such an update is inconsistent, then some guarantee could be violated. If this is a (u, v)-reachability guarantee, then there is a point in the update sequence t_i where a packet being sent would not reach the target. But then, there is a follower plan with finite cost that starts with CHOOSE-STEP$^F(t_i)$, where the packet will end up in a dead-end and or loop, contradicting that the follower cost is ∞. Similarly, if a (u, v, w)-waypoint guarantee is violated, then there is a plan where the packet is moved to v without going through w, so that the action WAYPOINT-MISSED^F becomes applicable. This contradicts that some guarantee is violated and therefore π_L corresponds to a consistent schedule. □

Corollary 1. NetStack *is sound and complete.*

C. Solving Stackelberg Tasks

The most straightforward way for solving Stackelberg tasks is to enumerate all possible sequences of leader and follower actions. However, a main advantage of encoding the problem as Stackelberg planning task is that one can make use of existing solvers that can avoid the exploration of large portions of the state space. We use the Stackelberg planning tool, Symbolic Leader Search [12] and provided as a free open source tool [13]. It is an extension of the Fast Downward planning system [11], [14].

The tasks described in previous sections are encoded in the Planning Domain Definition Language (PDDL) [15], [16], which divides the definition into a domain and a problem file. The domain file is written manually, and resembles the task definition from Table I, where all actions are parametrized by objects of different types. For each concrete network update problem, there is a distinct problem file, which specifies the objects for all routers and timesteps in a particular network including the initial state and all details of the model (e.g., $init(r)$ and $end(r)$ for every router). *NetStack* automatically generates PDDL problem files from network update problem descriptions in JSON format. Our tool uses the same data format as used by Didriksen et al. in their experiments with the TAPAAL tool [7]. This allows us to run experiments with the same input data and get comparable results.

Symbolic Leader Search enumerates the space of possible sequences of leader actions using symbolic search [17]–[19] with Binary Decision Diagrams (BDDs) [20]. As every router can be assigned to any time slot, the representation of all possible update schedules of a given length as BDDs is often compact.

Then, the solver will pick a leader schedule in the set and perform a secondary symbolic search with follower actions in order to find an inconsistency. If no inconsistency is found, then the solvers stops successfully returning the consistent schedule. If an inconsistency is found, regression is used to find the set of all schedules that contain the same inconsistency and all of them are removed from the set of leader schedules. Furthermore all plan suffixes are also efficiently stored as BDDs, so that subsequent follower searches can terminate early if such a plan suffix can be re-used.

We remark that this is only a candidate algorithm for solving Stackelberg planning problems. Our compilation can be used with alternative algorithms, e.g., which perform branch-and-bound with partial-order reduction [11], [21], [22].

We apply a minor modification to the solver, ignoring the cost of follower plans and only keeping information to whether the follower cost is ∞ or not. This is beneficial as it avoids the computation of the entire Pareto front, and allows stopping the follower searches as soon as a plan is found (i.e., as soon as an inconsistency is found in the schedule chosen by the leader) without looking for the lowest-cost follower plan.

D. Preprocessing

Using preprocessing of the network update problems, we can implement further optimizations before compiling them into Stackelberg planning tasks. These optimizations aim to reduce the number of necessary rounds by minimizing the number of routers. We identify two classes of routers that can be removed. We find that many of the real-world examples of the Topology Zoo [23], [24] have such non-changing and dead-end routers defined in the following, and hence these optimizations can greatly improve the performance of our tool.

1) Non-Changing Routers: As we consider cases where every router gets updated at most once, routers with the same initial and final configuration will not be updated at all. Therefore, there is no need to include those routers in the update schedule. However, it is still necessary to consider their routing when sending packets through the network. Therefore the generator tool will remove all non-changing routers, and update the remaining routing tables by changing the targets of the remaining routers whenever they pointed to one of the removed routers. Seeing the network as a graph this is equivalent to replacing a node (the non-changing router) including its incoming and outgoing edge with a single edge. Also, if any waypoint w is removed, then because w is also a non-changing router, always routing to q, q will become the new waypoint.

2) Dead-End Routers: Another special case are routers which have either at their initial or their final configuration a route to a dead-end (dropping the packets).[2] First we consider the special case of routers which only have a target node in their final configuration and drop all packets in the initial configuration. As Didriksen et al. showed, those routers will not influence the network behavior with respect to the reachability and waypoint policy if they received their update already before all other routers [7]. This can easily be seen, as any packet reaching its destination cannot have a dead-end router in the path. The packet would have been dropped before it reached the destination. Therefore these routers can not see any of the packets and so their configuration can not influence the network behaviour. So we can update those routers already

[2]We can ignore routers here which always have a dead-end configuration as they were removed by the previous optimization step already.

before we update all other routers. This means that we do not need to consider them in the Stackelberg model at all. We remove them in the same way as we removed the non-changing routers, only considering their final configuration.

Due to symmetry we can do the same for routers which drop all their packets in the final configuration. We can update them in rounds at the end of our schedule and also remove them from the Stackelberg model.

IV. Concurrent Updates

We now show that *NetStack* can be generalized quite flexibly to more involved settings. In particular, we consider concurrent updates, which can not only reduce the number of rounds to be considered by the Stackelberg model, but also the number of rounds of the final network update schedule. Concurrent updates can therefore significantly improve the time needed to roll out the update to the network.

The extension needs two main modifications on our Stackelberg model. On the one hand, the leader must generate a schedule of update batches instead of single updates. On the other hand, to ensure that all batches are consistent update batches, the follower must be extended to try to find a subset of any permutation of updates within a batch which breaks a guarantee. Both extensions are rather easy to add to our model due to the flexibility we have by using Stackelberg planning games. We split the leader action, SCHEDULE-UPDATE^L, which assigns a router to an update round and then increases the round number, into two actions:

- SCHEDULE-UPDATE-CONCURRENT^L(r, t): Assigns a router to the current update round, without increasing the round number.
- SCHEDULE-NEXT-STEP^L(t): Increases the round number.

The leader will therefore generate plans which can contain multiple routers in one round. We assign all SCHEDULE-UPDATE-CONCURRENT^{L}actions a cost of 0, and all other actions have a cost of 1, so that the cost of a leader plan directly corresponds to its number of batches.

To allow the follower to evaluate all subsets of all permutations of updates within a round, we have to modify the parameter restrictions of ADD-LINK^F by $(t' \leq t \wedge r' = end(r)) \vee (t' \geq t \wedge r' = init(r))$, so that for all the routers scheduled at the chosen round we allow for adding the link to either $init(r)$ or $end(r)$. This allows the solver to choose between both configurations for routers that are being updated in the chosen round. This non-determinism forces the solver to check all variants.

These two extensions allow our Stackelberg model to generate consistent concurrent network update schedules. As lower-cost leader plans are preferred, the Stackelberg Pareto front will contain a consistent schedule with the least amount of batches if and only if one exists.

A special consideration is required regarding the dead-end router optimization. Dead-end routers can be added as additional first and last rounds. This can result in suboptimal solutions with up to 2 rounds more then necessary. In some situations other routers can be combined with the dead-end routers making these extra rounds avoidable. To be able to consider these cases in the model, without adding the dead-end routers to the model, we use four different routing sets for every router. The model will use one set dependent on the round the router is assigned to. In the first round the initially dead-end routers are still set to drop packets, therefore all routers which are targets, will also be set to drop packets. This is a valid assumption as the worst case which can occur is that in the first round all dead-end routers will not get updated, but all other routers of the first batch will be. The same assumption can be done for the last round, which also needs a separate routing set. All other rounds use the same routing set as we use for the sequential update model. The fourth special routing set is necessary for the case that the solver tries to find a solution with one round only.

All these optimizations are implemented in the *NetStack* tool, and in the Stackelberg models, which we use for the experiments. We provide separate models implemented in two domain files for the sequential and the concurrent update mode. The problem files generated by *NetStack* are compatible with both the sequential and the concurrent model as the initial state in the problem files contains facts for both.

TABLE II
Solved Instances of our Tool within 1 hour

Tool	Solved Instances	Timed-Out(>1 hr)
Stackelberg sequential	859	57
Stackelberg concurrent	878	38

V. Empirical Evaluation

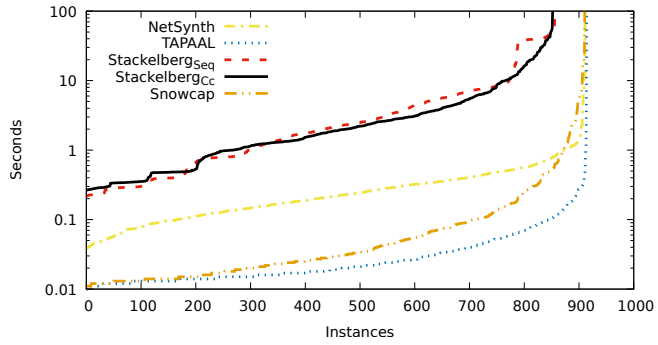
We conducted an empirical evaluation to study the performance of our approach and quality of *NetStack*'s solutions, also compared to existing tools: NetSynth [6], TAPAAL [7], and Snowcap [9].

A. Methodology

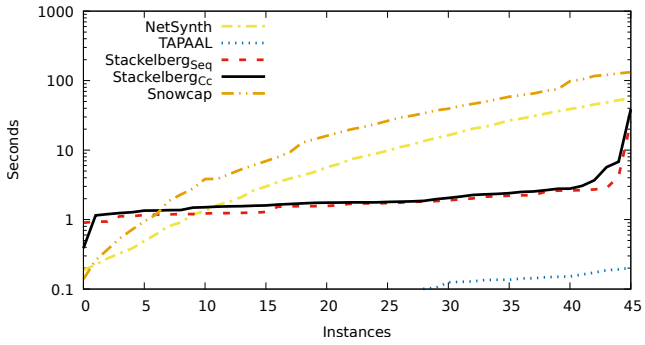
As base data for our experiments we use the network update problems generated by Didriksen et al. [24] out of the topology Zoo [23] database, which is a set of network topologies from real internet providers. This data set contains realistic networks and network updates together with reachability and waypoint guarantees. It also contains synthetically generated networks.

We validated that *NetStack* provides correct answers. On the one hand, we run our tool on a set of simple problems with between four to seven nodes for which solutions are known. On the other hand, we compared the results against the output of the TAPAAL tool [7] regarding solvability. Both tests are included in our reproducibility package.

We run our experiments on a cluster of AMD EPYC 7551 CPUs with disabled hyperthreading. We limit the execution time of one problem solver to 1 hour and the memory to 7 GB. These settings are the same as those which were used by the tools we compare with [7]. For the sequential update problems, we nevertheless rerun the experiments of the tools we compare with to have comparable performance values. Everything necessary to reproduce the experiments is available in the reproducibility package.



(a) Topology Zoo capped at 100 seconds.



(b) Synthetic Disjoint Networks.

Fig. 3. Number of instances solved as a function of runtime.

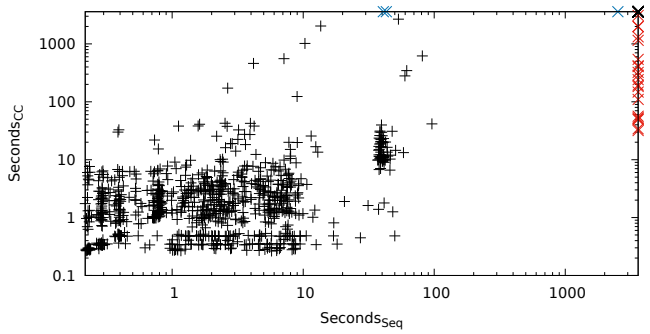


Fig. 4. Topology Zoo - Sequential vs. Concurrent Runtime. The crosses at the border are showing instances which could only be solved by sequential (blue) or concurrent (red) mode.

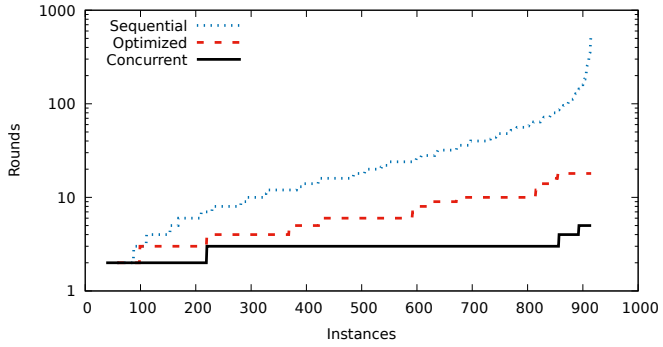


Fig. 5. Topology Zoo - Rounds needed for Sequential, Sequential with Dead-End Router Optimization and Concurrent Updates.

B. Results

Fig. 3a compares our tool to the state of the art on generating sequential network update schedules. The results show that our tool is not fully competitive when restricted to generating sequential schedules. Fig. 3b shows the comparison on synthetic networks, which contain a high number of dead-end routers. In this case, our tool is able to solve all instances outperforming NetSynth for most instances. This shows the effect of the dead-end router optimization. The results of TAPAAL, which uses the same optimization, confirms this having even better performance.

One of the main advantages of *NetStack*, however, is that it has great flexibility for modelling additional constraints. As explained in Section IV, a minor modification to the PDDL models is sufficient for obtaining consistent concurrent schedules that minimize the number of rounds. This is desirable because it will reduce the time required for the network to converge to the new update.

The results in Fig. 3a already show that, even though in principle, there are many more possible concurrent update schedules than sequential ones, the Stackelberg planning tool is not heavily affected by this. In fact, the reduced number of rounds may be beneficial. Table II shows that, within the full runtime of our experiment of 1 hour, the concurrent mode can solve more instances (95.9%) of the real-world examples in the Topology Zoo data then the sequential mode (93.8%).

We ran a more detailed comparison of the runtime of computing the optimal sequential and concurrent network update on the same network. Fig. 4 shows that there is no clear winner, and that sometimes our tool can find the concurrent schedule in less runtime than the sequential one.

Fig. 5 shows that the dead-end router optimization can reduce the number of rounds substantially, by performing all the updates of the ignored routers at the same time. The concurrent mode reduces the number of rounds even more, and most updates can be performed in two rounds.

VI. Additional Related Work

The question of how to update networks consistently lies at the heart of more automated and adaptive network operations, and has already been studied intensively in the recent literature. Besides accounting for temporal changes of the traffic demand [25], network updates are also needed to adapt to policy changes, to support planned maintenance or service relocation, or to react to failures. For a comprehensive overview of the field we refer to the survey by Foerster et al. [2]. Our model in this paper is generally known as the *per-node* update model in the literature, which has the advantage that it does not require packet tagging [26]. Originally introduced by Mahajan and Wattenhofer [27], in this model, batches of updates are scheduled over time. The model has

already been subject to extensive research, e.g., [1], [28]–[33], and due to space constraints, we refer to the survey [2] for an overview. While most existing literature on this model focuses on feasibility [33], sometimes also the duration of the update, measured in the number of rounds, is considered [34]. Ludwig et al. [28] showed that 2-round loop-free updates can be computed in polynomial time, but finding 3-round schedules is already NP-hard. Most existing literature on the per-node update model revolves around logical properties like loop-freedom [28], blackhole-freedom [35], waypointing and service chaining [1], etc.

VII. Conclusion

We developed a tool for synthesizing consistent network update schedules using a Stackelberg game approach. We showed that the tool can synthesize schedules for both sequential and concurrent update batches. Even if our tool cannot outperform current tools generating schedules for sequential updates, we showed the flexibility of our approach by extending our Stackelberg model to support concurrent updates. Indeed, we consider the possibility to simply add new rules/actions a substantial advantage of our approach. Our simulations on real-world networks showed that the concurrent extension does not have a negative performance impact on our Stackelberg based tool, but generates much shorter sequences. This can be a major advantage for quickly rolling out those network updates. Therefore we believe that this approach has the potential for solving similar problems, which will be interesting to investigate in future work. We also see a potential for further optimizing our Stackelberg model for the used solver tool, which we did not explore yet. This could improve the performance also for the cases where we could not compete with other tools yet.

References

[1] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently secure network updates," *ACM SIGMETRICS*, vol. 44, no. 1, 2016.

[2] K. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent software-defined network updates," *IEEE Communications Surveys Tutorials*, vol. 21, no. 2, 2019.

[3] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. NSDI '21*. USENIX, 2012.

[4] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. SIGCOMM '17*. ACM, 2017.

[5] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "Aalwines: A fast and quantitative what-if analysis tool for mpls networks," in *Proc. CoNEXT '20*. ACM, 2020.

[6] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *Proc. PLDI '15*. ACM, 2015.

[7] M. Didriksen, P. G. Jensen, J. F. Jønler, A.-I. Katona, S. D. L. Lama, F. B. Lottrup, S. Shajarat, and J. Srba, "Automatic synthesis of transiently correct network updates via petri games," in *Application and Theory of Petri Nets and Concurrency*. Springer International Publishing, 2021.

[8] M. Glavind, N. Christensen, J. Srba, and S. Schmid, "Latte: Improving the latency of transiently consistent network update schedules," in *Proc. 38th International Symposium on Computer Performance, Modeling, Measurements and Evaluation (PERFORMANCE)*, 2020.

[9] T. Schneider, R. Birkner, and L. Vanbever, "Snowcap: Synthesizing network-wide configuration updates," in *Proc. ACM SIGCOMM*, 2021.

[10] S. Schmid, B. C. Schrenk, and Á. Torralba, "Artifact for "NetStack: A Game Approach to Synthesizing Consistent Network Updates"," Apr. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6506194

[11] P. Speicher, M. Steinmetz, M. Backes, J. Hoffmann, and R. Künnemann, "Stackelberg planning: Towards effective leader-follower state space search," *Proc. AAAI '18*, vol. 32, no. 1, Apr. 2018.

[12] Á. Torralba, P. Speicher, R. Künnemann, M. Steinmetz, and J. Hoffmann, "Faster stackelberg planning via symbolic search and information sharing," *Proc. AAAI '21*, vol. 35, no. 13, 2021.

[13] ——, "stackelberg-planner-sls," https://gitlab.com/atorralba_planners/stackelberg-planner-sls, 2021.

[14] M. Helmert, "The fast downward planning system," *The Journal of artificial intelligence research*, vol. 26, 2006.

[15] D. McDermott, M. Ghallab, A. E. Howe, C. A. Knoblock, A. Ram, M. M. Veloso, D. S. Weld, and D. E. Wilkins, "Pddl-the planning domain definition language," Yale Center for Computational Vision and Control, Tech. Rep., 1998.

[16] M. Fox and D. Long, "Pddl2.1: An extension to pddl for expressing temporal planning domains," *The Journal of artificial intelligence research*, vol. 20, 2003.

[17] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] P. Kissmann and S. Edelkamp, "Improving cost-optimal domain-independent symbolic planning," in *Proc. AAAI*. AAAI Press, 2011.

[19] Á. Torralba, V. Alcázar, P. Kissmann, and S. Edelkamp, "Efficient symbolic search for cost-optimal planning," *Artificial Intelligence*, vol. 242, 2017.

[20] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, 1986.

[21] A. Valmari, "Stubborn sets for reduced state space generation," in *Proc. of the 10th Int. Conf. on Applications and Theory of Petri Nets (APN 1989)*, vol. 483. Springer-Verlag, 1989.

[22] M. Wehrle and M. Helmert, "About partial order reduction in planning and computer aided verification," in *Proc. ICAPS*. AAAI Press, 2012.

[23] S. Knight, H. X. Nguyen, N. Falkner, R. A. Bowden, and M. Roughan, "The internet topology zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, 2011.

[24] M. Didriksen, P. G. Jensen, J. F. Jønler, A.-I. Katona, S. D. Lama, F. B. Lottrup, S. Shajarat, and J. Srba, "Artefact for: Automatic Synthesis of Transiently Correct Network Updates via Petri Games," Feb. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4501982

[25] C. Avin, M. Ghobadi, C. Griner, and S. Schmid, "On the complexity of traffic traces and implications," in *Proc. ACM SIGMETRICS*, 2020.

[26] S. Vissicchio and L. Cittadini, "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*, 2016, pp. 1–9. [Online]. Available: https://doi.org/10.1109/INFOCOM.2016.7524419

[27] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. HotNets '13*. ACM, 2013.

[28] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *Proc. PODC '15*. ACM, 2015.

[29] K.-T. Foerster, T. Luedi, J. Seidel, and R. Wattenhofer, "Local checkability, no strings attached:(a) cyclicity, reachability, loop free updates in SDNs," *Theoretical Computer Science*, vol. 709, 2018.

[30] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proc. HotNets '14*. ACM, 2014.

[31] S. Dudycz, A. Ludwig, and S. Schmid, "Can't touch this: Consistent network updates for multiple policies," in *Proc. DSN '16*. IEEE, 2016.

[32] J. Zerwas, P. Kalmbach, C. Fuerst, A. Ludwig, A. Blenk, W. Kellerer, and S. Schmid, "Ahab: Data-driven virtual cluster hunting," in *Proc. IFIP Networking*, 2018.

[33] S. Akhoondian Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht, "Congestion-free rerouting of flows on dags," in *45th International Colloquium on Automata, Languages, and Programming (ICALP)*, vol. 107. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[34] S. Amiri, S. Dudycz, M. Parham, S. Schmid, and S. Wiederrech, "On polynomial-time congestion-free software-defined network updates," in *Proc. IFIP Networking*, 2019.

[35] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *Proc. IFIP Networking*. IEEE, 2016.