AllSynth: A BDD-Based Approach for Network Update Synthesis

Larsen, Kim Guldstrand; Mariegaard, Anders; Schmid, Stefan; Srba, Jiri

Published in:
Science of Computer Programming

DOI (link to publication from Publisher):
10.1016/j.scico.2023.102992

Publication date:
2023

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

Science of Computer Programming 230 (2023) 102992

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

AllSynth: A BDD-based approach for network update synthesis

Kim G. Larsen [a], Anders Mariegaard [a], Stefan Schmid [b], Jiří Srba [a,*]

[a] Dept. of Computer Science, Aalborg University, Denmark
[b] TU Berlin, Germany and University of Vienna, Austria

ARTICLE INFO

Article history:
Received 6 December 2022
Received in revised form 16 June 2023
Accepted 22 June 2023
Available online 28 June 2023

Keywords:
Computer networks
Software defined networking
Update synthesis
Binary decision diagrams

ABSTRACT

The increasingly stringent dependability requirements on communication networks as well as the need to render these networks more adaptive to improve performance, demand for more automated approaches to operate networks. We present AllSynth, a symbolic synthesis tool for updating communication networks in a provably correct and efficient manner. AllSynth automatically synthesizes network update schedules which transiently ensure a wide range of policy properties expressed using linear temporal logic (LTL). In particular, in contrast to existing approaches, AllSynth symbolically computes and compactly represents *all* feasible and cost-optimal solutions. At its heart, AllSynth relies on a novel parameterized use of binary decision diagrams (BDDs) which greatly improves performance. Indeed, AllSynth not only provides formal correctness guarantees and outperforms existing state-of-the-art tools in terms of generality, but also in terms of runtime as documented by experiments on a benchmark of real-world network topologies.

1. Introduction

Improving the automated operation of communication networks is considered one of the most important research problems in networking today, for two main reasons. *First*, communication networks and their configurations are highly complex, forcing operators to become "masters of complexity" [25]; many major Internet outages over the past years were caused by human errors [5,13,16]. Today's manual approach hence stands in stark contrast to the increasingly stringent dependability requirements on communication networks, which are a critical infrastructure of our digital society. *Second*, network traffic is not only constantly increasing but also features much more temporal and spatial structure [4,6,51]; this introduces a significant potential to improve operational efficiency by rendering networks more adaptive towards the actual traffic patterns they serve.

Motivated by the prospect of an increased level of automation in networks [18], over the past years, great efforts were made in laying the foundations for automated network verification, and in designing synthesis tools [3,17,28,44,47]. Furthermore, motivated by the benefits of more adaptive network operations, e.g., to improve availability and performance [29], automated tools for consistently updating network configurations have been developed [12,24,40,45,48] which overcome the limitations of existing hand-crafted algorithms [1,36,39]. However, the computation of provably consistent network update schedules remains challenging, due to the performance and expressiveness demands. The performance requirements are multidimensional: network update schedules should not only be quickly computable but also account for operator pref-

* Corresponding author.
E-mail address: srba@cs.aau.dk (J. Srba).

https://doi.org/10.1016/j.scico.2023.102992

erences, like requiring that certain switches or routers are updated first. However, existing approaches only provide one update sequence that may not satisfy additional requirements imposed by the network operator.

Our contributions. We present an automated network update synthesis tool, *AllSynth*, that computes and represents in a compact BDD (binary decision diagram) form *all* correct update sequences that respect various logical properties expressible in linear temporal logic (LTL) [43] such as reachability, waypointing and service chaining. *AllSynth* comes with formal correctness guarantees and in case it provably establishes that there does not exist any simple update schedule (where each switch is updated at most once), it can make suggestions for alternative solutions employing general update schedules (where a switch can be updated multiple times).

Despite being more general, *AllSynth* significantly outperforms state-of-the-art tools with regard to execution time on all non-trivial real-world networks from the standard Topology Zoo benchmark [30]. The update synthesis problem solved by *AllSynth* is NP-hard, even when restricted to preserving the basic loop-freedom and waypointing properties [36]. To overcome the complexity of the problem, *AllSynth* exploits a novel use of binary decision diagrams (BDDs) [34] to compactly encode not only the network topology and policy invariant, but also the set of *all* correct update sequences.

The fact that *AllSynth* computes all feasible update sequences enables future use cases for the tool, such as finding an optimal schedule with regard to a particular cost specification, providing multiple alternative solutions and filtering based on operator requirements (e.g. some switches must be updated before the rest or in a certain order). The source code of *AllSynth* and all our experimental artefacts are available at [33].

Related work. Motivated by the benefits of adaptive and software-defined (i.e., programmable) communication networks [31], as well as the increasingly stringent dependability requirements, the question of how to correctly update network configurations has received much attention over the last years. A recent survey summarizes over one hundred approaches [20].

In their seminal work, Reitblatt et al. [45] showed that strong per-packet consistency can be achieved using packet versioning during reconfigurations. Their approach, which was subsequently studied intensively in the literature [9,11,21,26,27,35,42], has the drawback that it requires packet header modifications and additional memory at the nodes: switches and routers need to store forwarding rules for each version.

A clever alternative approach, introduced by Mahajan and Wattenhofer [39], schedules batches of updates over time, where the set of updates within a batch can take effect in any order without harming consistency. This approach has also been explored extensively already [1,15,22,36–38,50], however, it can only be used to provide a subset of the consistency properties of [45]. This in turn motivated hybrid approaches such as FLIP [48]. Interestingly, similar to *AllSynth*, FLIP also supports alternative solutions in case a simple update cannot be found. However, in contrast to FLIP which relies on a heuristic algorithm, *AllSynth* only presents alternative solutions in case a simple solution *provably* does not exist. Furthermore, while FLIP resorts to a packet tagging alternative (which consumes header space and switch memory), *AllSynth* is a light-weight and fully symbolic approach aiming at updating nodes multiple times.

The need for supporting more general or even customizable consistency properties [52] as well as more automated synthesis approaches [19,24,41] has already received attention in the literature as well. However, our approach is the first one that uses the BDD-based technology for the synthesis and representation of *all* correct network updates. The competing tool NetSynth [40] for update synthesis is relying on an incremental enumeration of candidates of update sequences that are then verified by external model checkers, like NuSMV [14], and the tool terminates as soon as the first correct update sequence is found.

This article is an extended version (with full proofs, cost-optimal update synthesis and additional experiments) of TASE'22 conference paper published in [32]. The paper is organized as follows: in Section 2 we formally define the update synthesis problem, including the simple and generalized variants of the problem; in Section 3.2 we introduce our BDD-based tool for solving the problem and we construct a BDD representation of all feasible solutions to the update synthesis problem; in Section 4 we extend the methodology with the construction of a BDD representing all cost-optimal solutions; in Section 5 we evaluate the performance of our implementation against state-of-the-art approaches; finally in Section 6 we provide a conclusion and further perspectives of our work. We also include a short appendix showing how to run AllSynth.

2. A model for update synthesis

Before we formally define our problem, we shall provide an intuitive motivation for the update synthesis problem. In Fig. 1 we see a simple network with four nodes (routers). Packets from the source node s are forwarded to the destination node d along the solid edges (links) that represent the initial routing configuration. The network operator aims to change this routing to an alternative one represented by the dashed edges. The task is to schedule the order of node updates (changing the forwarding function at the updated node from the solid edge to the dashed one) so that in every intermediate routing configuration we preserve the reachability between s and d and at the same time always visit the waypoint node v_1 (representing for example a firewall).

If the node s is updated first, the new routing will follow the path s, v_2, d which preserves the reachability property but not the waypointing. On the other hand, if we first update the node v_2, we create an undesirable forwarding loop $s, v_2, v_1, v_2, v_1, \ldots$ which breaks the reachability property. Hence the only option is to update first the node v_1, after which

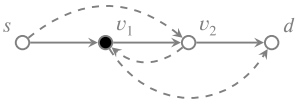


Fig. 1. Update synthesis problem aiming to preserve reachability between s and d via the waypoint v_1.

Table 1
Key notation of network model.

Notation	Description
$G = (V, E, \mathsf{src}, \mathsf{tgt})$	Network topology with nodes V and edges E
$\mathsf{src}(e), \mathsf{tgt}(e)$	Source and target node of an edge
v, v_i	Nodes
e, e_i	Edges
ρ	Routing configuration
π	A path (sequence of edges)
$\overline{\pi}$	Sequence of nodes traversed by path π
$\pi_\rho(v)$	Path induced by ρ from node v
u	An update
w	An update sequence
ρ^w	Routing after applying update sequence w
ρ_i, ρ_f	Initial and final routing configurations
φ	Policy specified in LTL logic
P	Update synthesis problem
$\mathsf{Sol}(P)$	Set of solutions to problem P

we have a correct forwarding path s, v_1, d satisfying both reachability and waypointing. After this we can update the node v_2 because this update does not change the forwarding path and lastly, we update the node s that completes the update sequence from the initial to the final routing.

We are now ready to provide the formalization of the update synthesis problem. Table 1 contains a summary of the key notation. We model the network as a multigraph, allowing us to describe multiple connections between nodes (i.e., switches or routers, which are treated as synonyms in the following); these connections can have different quantitative attributes (e.g. latency). Henceforth, we adopt graph-theory terminology and refer to such connections or links as edges.

Definition 1 *(Network topology).* A network topology is a directed multigraph $G = (V, E, \mathsf{src}, \mathsf{tgt})$ where V is the set of nodes, E is the set of edges and $\mathsf{src}, \mathsf{tgt}\colon E \to V$ are respectively the *source* and *target* functions.

In order to route traffic from a node v_0 to a node v', each node v has a forwarding rule that specifies an appropriate outgoing edge e such that $\mathsf{src}(e) = v$. This rule can be per-flow or apply to multiple flows; in the following, we do not explicitly distinguish between the two scenarios. Not all nodes need to have defined their forwarding edge (e.g. the target node v' or the nodes that are not involved in packet forwarding from v_0 to v'). We capture this formally by the notion of a routing configuration.

Definition 2 *(Routing configuration).* A *routing configuration*, or *routing* for short, in a network topology $G = (V, E, \mathsf{src}, \mathsf{tgt})$ is a partial function $\rho\colon V \rightharpoonup E$ such that $\mathsf{src}(\rho(v)) = v$ for all $v \in V$ where $\rho(v)$ is defined.

For a given network topology $G = (V, E, \mathsf{src}, \mathsf{tgt})$ with the source node $v_0 \in V$, a routing configuration ρ defines a unique sequence of edges (a path) that is finite if the routing is loop free; otherwise it is infinite. In the finite case, the *path* is given by $\pi = e_0 e_1 \cdots e_n$ such that $\rho(\mathsf{tgt}(e_{i-1})) = e_i$ for all i, $1 \le i \le n$, and $\mathsf{src}(e_0) = v_0$, and where $\rho(\mathsf{tgt}(e_n))$ is undefined. The corresponding sequence of *traversed nodes* is then $\overline{\pi} = \mathsf{src}(e_0)\mathsf{src}(e_1)\cdots\mathsf{src}(e_n)\mathsf{tgt}(e_n)$. In the infinite case, the *path* is given by $\pi = e_0 e_1 \cdots$ such that $\rho(\mathsf{tgt}(e_{i-1})) = e_i$ for all $i > 0$, and $\mathsf{src}(e_0) = v_0$. The sequence of *traversed nodes* is given by the infinite sequence $\overline{\pi} = \mathsf{src}(e_0)\mathsf{src}(e_1)\cdots$. If $\overline{\pi} = v_0 v_1 \ldots$ is a (finite or infinite) sequence of nodes then we refer to its suffix $v_i v_{i+1} \ldots$ by $\overline{\pi}_i$ and to the initial node v_0 by $\overline{\pi}[0]$. For a node $v_0 \in V$ and routing ρ, we let $\pi_\rho(v_0)$ denote the unique (finite or infinite) path induced by ρ from the source node v_0 and let $\overline{\pi}_\rho(v_0)$ be the corresponding sequence of traversed nodes.

$$\begin{aligned}
\mathsf{Reach}(d) &\equiv \mathsf{true}\, U\, d \\
\mathsf{Waypoint}(v,d) &\equiv \neg\mathsf{Reach}(d) \vee (\neg d\, U\, v \wedge \mathsf{Reach}(d)) \\
\mathsf{MultiWaypoint}(W,d) &\equiv \bigvee_{v\in W} \mathsf{Waypoint}(v,d) \\
\mathsf{Service}(v_1 v_2 \cdots v_n, d) &\equiv \begin{cases} \neg\mathsf{Reach}(d) \vee \left(\bigwedge_{i=2}^{n} \neg v_i \wedge \neg d\right) U\ (v_1 \wedge \mathsf{Service}(v_2 \cdots v_n, d)) & \text{if } n \geq 1 \\ \mathsf{true} & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 2. Encoding of standard policies where $v, d \in V$, $\emptyset \neq W \subseteq V$ and $v_1 v_2 \cdots v_n \in V^*$.

2.1. Routing policies

We shall now define a variant of LTL [43] that allows us to describe specific policies that routings must enforce (both statically and transiently).

Definition 3 *(Policy syntax).* For a network topology $G = (V, E, \mathsf{src}, \mathsf{tgt})$, a policy φ is constructed according to the following LTL-based abstract syntax, where $v \in V$:

$$\varphi ::= \mathsf{true} \mid v \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{NoLoop} \mid X\,\varphi \mid \varphi\, U\, \varphi\ .$$

In addition to the classical LTL operators, our logic includes a loop-freeness predicate. We now give the formal semantics of our logic, interpreted both on infinite and finite paths [23].

Definition 4 *(Policy semantics).* For a network topology $G = (V, E, \mathsf{src}, \mathsf{tgt})$, a policy φ is satisfied by a path $\pi \in E^* \cup E^\omega$, written $\pi \models \varphi$, iff the corresponding sequence of traversed nodes $\overline{\pi}$ satisfies $\overline{\pi} \models \varphi$, defined inductively on the structure of φ as follows:

$$\begin{aligned}
&\overline{\pi} \models \mathsf{true} && \text{always} \\
&\overline{\pi} \models v && \text{iff } \overline{\pi}[0] = v \\
&\overline{\pi} \models \neg\varphi && \text{iff } \overline{\pi} \not\models \varphi \\
&\overline{\pi} \models \varphi_1 \wedge \varphi_2 && \text{iff } \overline{\pi} \models \varphi_1 \text{ and } \overline{\pi} \models \varphi_2 \\
&\overline{\pi} \models \mathsf{NoLoop} && \text{iff } \overline{\pi} \text{ is finite} \\
&\overline{\pi} \models X\,\varphi && \text{iff } \overline{\pi}_1 \models \varphi \\
&\overline{\pi} \models \varphi_1\, U\, \varphi_2 && \text{iff } \exists j \forall i < j . \overline{\pi}_j \models \varphi_2 \quad \text{and } \overline{\pi}_i \models \varphi_1 .
\end{aligned}$$

Examples of standard routing policies are given in Fig. 2. The simplest policy, $\mathsf{Reach}(d)$, specifies that the destination node d must eventually be reached while $\mathsf{Waypoint}(v, d)$ asks that any path reaching the destination d must necessarily pass through waypoint node v. For multiple alternative waypoints, $\mathsf{MultiWaypoint}(W, d)$ specifies that any path reaching destination d must necessarily pass through *either* of the waypoints in W. Finally, $\mathsf{Service}(v_1 v_2 \cdots v_n, d)$ ensures that the sequence of waypoints in $v_1 v_2 \cdots v_n$ is visited in this fixed order.

2.2. Update synthesis

In the following we assume a fixed network topology $G = (V, E, \mathsf{src}, \mathsf{tgt})$. An *update* $u \in E \cup V$ on G under a current routing configuration ρ specifies that the source node of edge u (if $u \in E$) must now forward its traffic along u or that the routing for the node u (if $u \in V$) is set to undefined. We write ρ^u for the new routing configuration, defined for any $v \in V$ as

$$\rho^u(v) = \begin{cases} u & \text{if } u \in E \text{ and } v = \mathsf{src}(u) \\ \mathit{undefined} & \text{if } u = v \\ \rho(v) & \text{otherwise.} \end{cases}$$

We inductively extend this notation to sequences of updates by letting $\rho^\varepsilon = \rho$ and $\rho^{wu} = (\rho^w)^u$ for any $w \in (E \cup V)^*$ and $u \in E \cup V$. An update sequence may in general contain an arbitrary number of updates that change multiple times the routing of the same node, however an important set of update sequences is the class of *simple* update sequences. A simple update sequence consists of only *simple updates* that change, for a given node v, the initial routing $\rho_i(v)$ at v directly to its

(a) Network topology (b) Initial (solid) and final (dashed) routings

Fig. 3. Update synthesis problem with only a general solution.

final routing $\rho_f(v)$. This implies that it does not make sense to update the node v (using simple updates) more than once; after the first simple update of v, no further simple updates of v change anything.

Definition 5 *(Simple updates).* Let ρ_f be the final routing. An update u is *simple* if $\rho_f(\mathsf{src}(u)) = u$ whenever $u \in E$ and $\rho_f(u)$ is undefined whenever $u \in V$. A *simple update sequence* is then a sequence of simple updates, where each update appears at most once.

A basic property of simple update sequences is that any reordering results in the same final routing configuration i.e., if w is a simple update sequence and w' is any permutation of w, then $\rho^w = \rho^{w'}$ for any routing ρ.

Although any reordering of a simple update sequence yields the same final routing configuration, the intermediate routing configurations induced by each update may not uphold a given policy invariant. This is also the case for general update sequences. We therefore say that an update sequence is *correct* with respect to a policy φ and a node v, if the unique path from v induced by any intermediate routing configuration satisfies φ.

Definition 6 *(Update correctness).* An update sequence $w \in (E \cup V)^*$ on network topology G with initial routing configuration ρ is *correct* with respect to source node v_0 and a policy φ, if $\pi_{\rho^{w'}}(v_0) \models \varphi$ for any prefix w' of w.

The network update synthesis problem is thus the problem of constructing a correct update sequence that updates an initial routing to a desired final routing.

Definition 7 *((Simple) update synthesis problem).* Given a topology G, an initial routing configuration ρ_i, a final routing configuration ρ_f, source node $v_0 \in V$ and a policy φ, the *simple update synthesis problem* asks to construct a simple update sequence w that is correct with respect to v_0 and φ such that $\rho_i^w = \rho_f$. The *update synthesis problem* omits the requirement that the constructed update sequence is simple.

In the following, we let $P = (G, \rho_i, \rho_f, v_0, \varphi)$ denote a (simple) update synthesis problem and say that a constructed update sequence w that satisfies the conditions above is a *solution*. For any simple update synthesis problem P, the set of its solutions, denoted by $\mathsf{Sol}(P)$, is always finite. This is not the case for the *general* problem as there may be infinitely many (longer and longer) solutions.

While much prior work focused on simple update problems, there are examples which are only solvable with a general solution (as supported by our approach). Consider for example the network topology in Fig. 3a with initial and final routings visualised respectively as solid and dashed lines in Fig. 3b. We fix the source node s and the policy $\varphi = \mathsf{Waypoint}(v_2, d) \wedge \mathsf{Reach}(d)$ requiring that waypoint v_2 must be visited before reaching d. An update of any node v from the initial to the final routing violates φ—either by introducing a loop or it bypasses the waypoint. Hence there is no correct simple update sequence. However, the update sequence that first updates s to route to v_2, followed by the update of the nodes v_1, v_2 and v_3 and finally updating s again to route to v_3 is a correct update sequence.

2.3. Simple update sequence reordering

In case of simple update sequences, we shall now argue that for routing policies that (i) include the preservation of reachability between the source and a target, and (ii) for which it holds that once a packet is delivered, no further routing is defined from the target node, we can reorder certain updates in the sequence without invalidating the correctness of the sequence. More specifically, we shall show that if a node routing is to be changed from undefined to some concrete edge, we can safely schedule such updates (in any order) to the very beginning of the update sequence. Similarly, all nodes that change their current routing into undefined can be scheduled (again in arbitrary order) at the end of the update sequence.

Lemma 1. *Let w be a solution to a simple update synthesis problem $P = (G, \rho_i, \rho_f, v_0, \varphi)$ where $\varphi = \mathsf{Reach}(d) \wedge \varphi'$ for any policy φ' and where $\rho_i(d)$ and $\rho_f(d)$ are undefined.*

1. *If $w = w_1 u w_2$ where $u \in E$ is an update s.t. $\rho_i(\mathsf{src}(u))$ is undefined then $u w_1 w_2$ is a solution to P.*
2. *If $w = w_1 u w_2$ where $u \in V$ updates the routing in u to undefined then $w_1 w_2 u$ is a solution to P.*

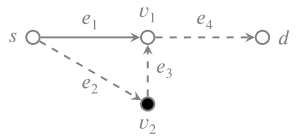


Fig. 4. Counter example for Waypoint(v_2, d); initial/final routing is in solid/dashed lines.

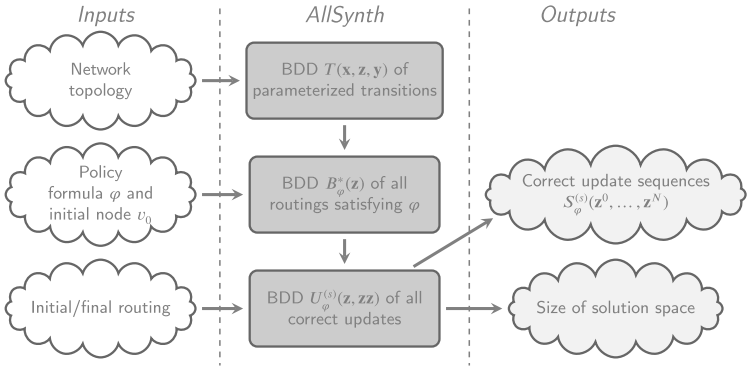


Fig. 5. *AllSynth* workflow.

Proof. As we assume that w is a correct update sequence from the initial node v_0 for the policy φ that includes the formula Reach(d), we know that for every prefix w' of w the path $\pi_{\rho_i^{w'}}(v_0)$ under the routing $\rho_i^{w'}$ necessarily ends in the node d. This implies that all nodes v on any such path (except for d) must have defined its routing function $\rho_i^{w'}(v)$.

Now consider case (1) where $w = w_1 u w_2$ such that $u \in E$ where $\rho_i(\mathsf{src}(u))$ is undefined. As the update sequence w is simple, there is only a single occurrence of the node $\mathsf{src}(u)$ in w. This implies that for any prefix w' of w_1 the path $\pi_{\rho_i^{w'}}(v_0)$ cannot contain the node $\mathsf{src}(u)$ as it will otherwise create a blackhole [39] (where a packet cannot be forwarded further) and invalidate the predicate Reach(d). Hence, moving the update u to the very beginning of the update sequence has no impact on the path from v_0, meaning that it does not change the validity of the policy φ' either.

For case (2) where $w = w_1 u w_2$ such that $u \in V$, we notice that after executing the updates in w_1 and u, the resulting routing does not define any forwarding for u, i.e. $\rho_i^{w_1 u}(u)$ is undefined. As the update sequence w is simple, the routing of u remains undefined until the end of the update sequence w, while the property Reach(d) must still hold. This implies that after executing the update sequence w_1 the path under any future routing (after executing the updates $w_1 w'$ where w' is a prefix of w_2) does not include the node u. Hence we can safely move the update u at the end of the sequence as it does not influence the validity of the policy φ'. □

Lemma 1 can be used to identify all nodes that have an undefined forwarding function in ρ_i and schedule them to the beginning of the update sequence. Symmetrically, all updates that change a node forwarding to an undefined value (in the routing ρ_f), can be placed at the end of the update sequence. This may simplify the synthesis of the update sequence by analysing only the nodes that have a defined forwarding function both in the initial and final routing.

The requirement in Lemma 1 that the policy must enforce at least the reachability of d is essential, as illustrated in Fig. 4 where $e_2 e_3 e_4$ is a correct update sequence preserving Waypoint(v_2, d). This is because until the last update, the destination d is not reachable and hence the waypointing policy trivially holds. However, even though the routing of v_1 is undefined in the initial routing, moving the update e_4 to the beginning of the update sequence creates a transient forwarding following the path $e_1 e_4$ and violates Waypoint(v_2, d).

3. BDD-based algorithm for update synthesis problem

We shall now present an overview of our tool *AllSynth*, including its inputs and outputs, followed by the theoretical foundations of the BDD-based synthesis algorithm implemented in the tool.

3.1. AllSynth tool workflow

The diagram in Fig. 5 illustrates the main components of *AllSynth*. The inputs to *AllSynth* are the network topology G, a policy of interest φ, as well as the initial routing ρ_i and final routing ρ_f from the node v_0.

From the input network topology G, a BDD representation of the edges in G is combined with the input policy φ and a source node v_0 to produce a BDD representing all routing configurations ρ where the unique path $\pi_\rho(v_0)$ satisfies φ. This

Table 2
Key notation of BDD encoding.

Notation	Description
$\mathbf{x}, \mathbf{y}$	Variables encoding sets of nodes
$\mathbf{z}, \mathbf{zz}$	Variables encoding routing configurations
$T(\mathbf{x}, \mathbf{z}, \mathbf{y})$	Transition function
$B_\varphi(\mathbf{x}, \mathbf{z})$	Satisfiability of policy φ
$B^*_\varphi(\mathbf{z})$	Routing configurations satisfying φ for a fixed source node v_0
$U_\varphi(\mathbf{z}, \mathbf{zz})$	Possible updates preserving correctness with respect to φ
$R_\varphi(\mathbf{z}, \mathbf{zz})$	Updates leading to final configuration with φ correctness preserved
$S_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N)$	All solutions of length N, using N copies of $\mathbf{z}$ variables
$U^s_\varphi / R^s_\varphi / S^s$	Variants of $U_\varphi / R_\varphi / S_\varphi$ for simple updates

BDD is then in turn combined with the initial and final routing configurations ρ_i and ρ_f, to construct a BDD representation of all correct update sequences.

3.2. The synthesis algorithm

We now describe our algorithmic solution to the update synthesis problem, based on a symbolic encoding of routing configurations using BDDs. This encoding allows for an efficient fixed-point computation of those routing configurations that satisfy a given routing policy, and subsequently to find a correct update sequence solving the synthesis problem.

Boolean decision diagrams [34] are data structures for the compact representation of a Boolean function. A BDD is a rooted directed acyclic graph (DAG), with nonleaf nodes labelled by Boolean variables, and leaf nodes labelled with 0 (false) or 1 (true). Each node that is labelled by a variable has two outgoing edges, a solid one representing the true assignment to the variable and a dotted one for the false assignment. By following the paths from the root to the leaf labelled with 1, we obtain all satisfying Boolean assignments. BDDs were introduced by Lee [34] and later Bryant [10] presented their reduced ordered version (ROBDD), where the ordering between the Boolean variables are fixed along each path from the root to a leaf, and isomorphic parts are combined. We show how to exploit ROBDDs for solving the update synthesis problem. We refer to Table 2 for a summary of key notation used in the encoding.

First, let us recall how to encode subsets of a finite set S using Boolean expressions—hence ROBDDs. The encoding is relative to a given enumeration $s_0, s_1, s_2, \ldots s_{|S|-1}$ of S and it is based on $n = \lceil \log(|S|) \rceil$ Boolean variables $\mathbf{x} = x_1, x_2, \ldots, x_n$. Now, any truth assignment μ to $\mathbf{x}$ may be seen as a binary encoding of a natural number $n(\mu) \in \mathbb{N}$ and hence an encoding of the $n(\mu)$'th element $s_{n(\mu)} \in S$. We shall use the short notation $s(\mu)$ for the element $s_{n(\mu)}$ as well as the notation $\mathbf{x}(s)$ to denote a Boolean expression over $\mathbf{x}$ encoding the singleton-set $\{s\}$. Now any Boolean expression $t(\mathbf{x})$ over $\mathbf{x}$ may be seen as encoding the subset $[\![t(\mathbf{x})]\!] = \{ s(\mu) \mid \mu \text{ satisfies } t(\mathbf{x}) \} \subseteq S$.

Example 1. Consider the network topology in Fig. 6a with the nodes $V = \{v_0, v_1, v_2, v_3\}$ enumerated by the given indices. We encode any subset of V by a Boolean expression over two Boolean variables x_1, x_2—note that the encoding of e.g. $\{v_1\}$ is $\mathbf{x}(v_1) = \neg x_1 \wedge x_2$ as the binary encoding of v_1 is 01. Conversely, the subset identified by the Boolean expression $t \equiv \neg x_1 \vee \neg x_2$ is $[\![t]\!] = \{v_0, v_1, v_2\}$ as the binary encoding of v_0, v_1, v_2 are 00, 01, 10, respectively.

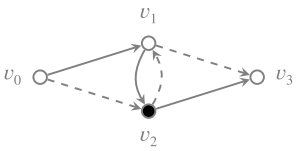
BDD encoding of routing configurations. Let $G = (V, E, \mathsf{src}, \mathsf{tgt})$ be a network topology and let $v \in V$. We denote by E_v the set of edges having v as a source-node, i.e. $E_v = \{e \in E \mid \mathsf{src}(e) = v\}$. Now, a routing configuration $\rho \colon V \rightharpoonup E$ is isomorphic to indicating for each node v whether $\rho(v)$ is defined and if so to identify an element from E_v. For the Boolean encoding of (sets of) elements from E_v we use, as described above, $\lceil \log(|E_v|) \rceil$ Boolean variables $\mathbf{z}_v$. To indicate the definedness of $\rho(v)$, we use an additional Boolean variable z^d_v. To encode the possible transitions between nodes v and v' enabled by a given routing configuration ρ, we use Boolean variables $\mathbf{x}$ for encoding the source node v and equally many Boolean variables $\mathbf{y}$ for encoding the target node v'. The following Boolean expression T encodes all possible transitions in a network:

$$T(\mathbf{x}, \mathbf{z}_{v_0}, \ldots, \mathbf{z}_{v_k}, z^d_{v_0}, \ldots, z^d_{v_k}, \mathbf{y}) = \bigvee_{v \in V} \bigvee_{e \in E_v} \left(\mathbf{x}(v) \wedge \mathbf{z}_v(e) \wedge z^d_v \wedge \mathbf{y}(\mathsf{tgt}(e)) \right)$$

where $V = \{v_0, \ldots, v_k\}$.

Example 2. Considering again the network topology from Fig. 6a, we shall use three Boolean variables z_0, z_1, z_2 for encoding routing configurations in terms of their choice of successor-node from v_0, v_1 and v_2.[1] Using the encoding of nodes from

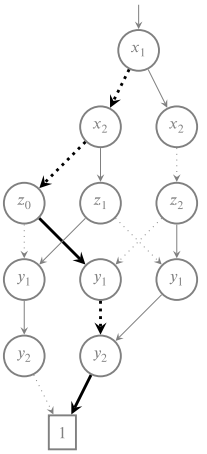
[1] In this running example, we shall for simplicity assume that routing configurations are total functions, e.g. that the variables z^d_v are true.



(a) Running example with initial (solid line) and final (dashed line) routings

$$
\begin{aligned}
&(\neg x_1 \wedge \neg x_2 \wedge z_0 \wedge \neg y_1 \wedge y_2) \vee \\
&(\neg x_1 \wedge \neg x_2 \wedge \neg z_0 \wedge y_1 \wedge \neg y_2) \vee \\
&(\neg x_1 \wedge x_2 \wedge z_1 \wedge y_1 \wedge \neg y_2) \vee \\
&(\neg x_1 \wedge x_2 \wedge \neg z_1 \wedge y_1 \wedge y_2) \vee \\
&(x_1 \wedge \neg x_2 \wedge z_2 \wedge y_1 \wedge y_2) \vee \\
&(x_1 \wedge \neg x_2 \wedge \neg z_2 \wedge \neg y_1 \wedge y_2)
\end{aligned}
$$

(b) Expression T (here $\mathbf{x}$ encodes the source node, $\mathbf{y}$ is the target node, and z_i indicates whether the node v_i follows the edge in the initial or in the final routing



(c) T as ROBDD (here dotted lines represent low edges (0) and solid lines stand for high edges (1); the highlighted path (thick black) encodes the transition (routing) from v_0 to v_1 under the initial routing

Fig. 6. Running example and encoding of the transition function.

Example 1, the possible transitions between nodes are given by the Boolean expression T in Fig. 6b. The resulting unique ROBDD in Fig. 6c with only 11 non-leaf nodes illustrates the compactness of the ROBDD data structure (the missing edges lead to 0). The highlighted path encodes the transition (routing) from v_0 to v_1 under the initial routing. Here the chosen ordering of the Boolean variables is crucial. Alternative orderings, e.g. with the $\mathbf{z}$ variables being tested first respectively last results in ROBDDs with 25 respectively 17 non-leaf nodes.

BDD encoding of routing policies. Now let $G = (V, E, \mathsf{src}, \mathsf{tgt})$ be a network topology and let φ be a routing policy expressed in the LTL logic of Definition 3. Using Boolean variables $\mathbf{x}$ for encoding nodes and Boolean variables $\mathbf{z}$ for encoding routing configurations,[2] we shall construct an ROBDD $B_\varphi(\mathbf{x}, \mathbf{z})$ such that: $(v, \rho) \in [\![B_\varphi(\mathbf{x}, \mathbf{z})]\!]$ if and only if $\pi_\rho(v) \models \varphi$ where $\pi_\rho(v)$ is the unique path starting in the node v following the routing configuration ρ.

Definition 8. Let $G = (V, E, \mathsf{src}, \mathsf{tgt})$ be a network topology and φ a routing policy. We define the ROBDD $B_\varphi(\mathbf{x}, \mathbf{z})$ inductively on φ as follows:

$$
\begin{aligned}
B_{\mathsf{true}}(\mathbf{x}, \mathbf{z}) &= 1 \\
B_v(\mathbf{x}, \mathbf{z}) &= \mathbf{x}(v) \\
B_{\neg\varphi}(\mathbf{x}, \mathbf{z}) &= \neg B_\varphi(\mathbf{x}, \mathbf{z}) \\
B_{\varphi_1 \wedge \varphi_1}(\mathbf{x}, \mathbf{z}) &= B_{\varphi_1}(\mathbf{x}, \mathbf{z}) \wedge B_{\varphi_2}(\mathbf{x}, \mathbf{z}) \\
B_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z}) &\overset{\min}{=} \forall \mathbf{y}.(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \rightarrow B_{\mathsf{NoLoop}}(\mathbf{y}, \mathbf{z})) \\
B_{X\varphi}(\mathbf{x}, \mathbf{z}) &= \exists \mathbf{y}.\big(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \wedge B_\varphi(\mathbf{y}, \mathbf{z})\big) \\
B_{\varphi_1 U \varphi_2}(\mathbf{x}, \mathbf{z}) &\overset{\min}{=} B_{\varphi_2}(\mathbf{x}, \mathbf{z}) \vee \big(B_{\varphi_1}(\mathbf{x}, \mathbf{z}) \wedge \exists \mathbf{y}.\big(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \wedge B_{\varphi_1 U \varphi_2}(\mathbf{y}, \mathbf{z})\big)\big)
\end{aligned}
$$

In Definition 8 we exploit the fact that ROBDDs are closed under Boolean operations as well as Boolean quantification. In the case of NoLoop and $\varphi_1 U \varphi_2$, the changes of Boolean variables used in the parameter lists in the right-hand sides are obtained by simple substitution of variables, an operation that may efficiently be performed on ROBDDs. Finally, note that the definitions of B_{NoLoop} and $B_{\varphi_1 U \varphi_2}$ are given as least fixed points. These fixed points, e.g. B_{NoLoop}, are obtained after a finite number of applications of the corresponding right-hand sides on increasing approximations B^n_{NoLoop}, starting with $B^0_{\mathsf{NoLoop}} = 0$, where 0 stands for false, and terminating when $B^{n+1}_{\mathsf{NoLoop}} = B^n_{\mathsf{NoLoop}}$, and similarly for $B_{\varphi_1 U \varphi_2}$. Such an n must

[2] Recall that $\mathbf{z}$ consists of variables $\mathbf{z}_{v_1}, \ldots, \mathbf{z}_{v_k}$ and $z^d_{v_1}, \ldots, z^d_{v_k}$.

necessarily exist because the applications of the right-hand sides can only monotonically increase the respective Boolean functions represented by the ROBDDs.

Lemma 2. *We have* $(v, \rho) \in [\![B_\varphi(\mathbf{x}, \mathbf{z})]\!]$ *if and only if* $\pi_\rho(v) \models \varphi$.

Proof. The proof proceeds by structural induction on the formula φ. The cases $\varphi = \mathsf{true}$ and $\varphi = v$ are trivial, while the case $\varphi = \varphi_1 \wedge \varphi_2$ follows directly by application of the induction hypothesis for the two conjuncts. We consider the remaining cases in turn.

$\varphi = \mathsf{NoLoop}$**:**

Recall that $B_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z})$ is obtained as the limit of the finite sequence of approximations $B^n_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z})$ with $B^0_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z}) = 0$ and otherwise $B^{n+1}_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z}) = \forall \mathbf{y}.(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \rightarrow B^n_{\mathsf{NoLoop}}(\mathbf{y}, \mathbf{z}))$ i.e. there exists a k for which $B^{k+1}_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z}) = B^k_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z})$.

For the left to right implication, assume $(v, \rho) \in [\![B_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z}]\!]$. The proof follows by induction on k. For the base case $k = 1$, it follows that $T(\mathbf{x}, \mathbf{z}, \mathbf{y})$ is false for all $\mathbf{y}$, implying that v does not have any outgoing transition under routing configuration ρ, hence, the path induced by ρ from v is loop-free. For the inductive step $k > 1$, we have $B^k_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z}) = \forall \mathbf{y}.(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \rightarrow B^{k-1}_{\mathsf{NoLoop}}(\mathbf{y}, \mathbf{z})$. As v has a unique successor under ρ, it must be the case that $(v', \rho) \in [\![B^{k-1}_{\mathsf{NoLoop}}(\mathbf{y}, \mathbf{z})]\!]$ for the unique successor v. Thus, we can apply the inductive hypothesis to conclude $\pi_\rho(v') \models \mathsf{NoLoop}$, i.e. the path $\pi_\rho(v')$ is finite, hence the path $\pi_\rho(v)$ must necessarily be finite and we can conclude $\pi_\rho(v) \models \mathsf{NoLoop}$ following the LTL semantics.

For the right to left implication, assume $\pi_\rho(v) \models \mathsf{NoLoop}$. Following the LTL semantics, this implies that $\pi_\rho(v)$ is finite. We proceed by induction on the length of $\pi_\rho(v)$. For the base case $|\pi_\rho(v)| = 1$, v has no successor in the graph under ρ. Then trivially $(v, \rho) \in [\![B_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z})]\!]$ as $T(\mathbf{x}, \mathbf{z}, \mathbf{y})$ is not satisfiable for any $\mathbf{y}$. For the inductive step $|\pi_\rho(v)| > 1$, v has a unique successor v' under ρ. As $\pi_\rho(v)$ may only be finite if $\pi_\rho(v')$ is finite, we can apply the induction hypothesis to conclude that $(v', \rho) \in [\![B_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z})]\!]$, hence $(v, \rho) \in [\![B_{\mathsf{NoLoop}}(\mathbf{x}, \mathbf{z})]\!]$.

$\varphi = X\,\varphi'$**:**

For the left to right implication, suppose $(v, \rho) \in [\![B_{X\varphi'}(\mathbf{x}, \mathbf{z})]\!]$. This implies that there exists a (unique) successor v' of v under routing configuration ρ, encoded by the variables $\mathbf{y}$, such that $(v', \rho) \in [\![B_{\varphi'}(\mathbf{y}, \mathbf{z})]\!]$. By applying the inductive hypothesis we immediately have that $\pi_\rho(v') \models \varphi'$, which following the LTL semantics yields $\pi_\rho(v) \models \varphi$.

For the right to left direction, suppose $\pi_\rho(v) \models X\,\varphi'$. Following the LTL semantics, we then have that for the unique successor v' of v it holds that $\pi_\rho(v') \models \varphi$. By the inductive hypothesis we then have $(v', \rho) \in [\![B_{\varphi'}(\mathbf{x}, \mathbf{x})]\!]$, hence $(v, \rho) \in [\![B_{X\varphi'}(\mathbf{x}, \mathbf{z})]\!]$.

$\varphi = \varphi_1\,U\,\varphi_2$**:**

Recall that $B_{\varphi_1 U \varphi_2}$ is obtained as the limit of the finite sequence of approximations $B^n_{\varphi_1 U \varphi_2}(\mathbf{x}, \mathbf{z})$ with $B^0_{\varphi_1 U \varphi_2}(\mathbf{x}, \mathbf{z}) = 0$ and otherwise

$$B^{n+1}_{\varphi_1 U \varphi_2}(\mathbf{x}, \mathbf{z}) = B_{\varphi_2}(\mathbf{x}, \mathbf{z}) \vee \left(B_{\varphi_1}(\mathbf{x}, \mathbf{z}) \wedge \exists \mathbf{y}.\left(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \wedge B^n_{\varphi_1 U \varphi_2}(\mathbf{y}, \mathbf{z})\right)\right)$$

i.e. there exists a k for which $B^{k+1}_{\varphi_1 U \varphi_2} = B^k_{\varphi_1 U \varphi_2}$.

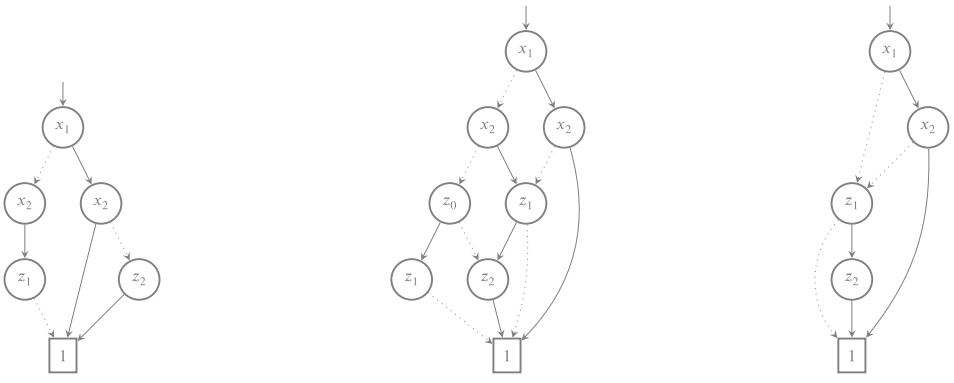
For the left to right implication, assume $(v, \rho) \in [\![B_{\varphi_1 U \varphi_2}(\mathbf{x}, \mathbf{z})]\!]$. We proceed by induction on k. For the base case $k = 1$, it follows that $(v, \rho) \in [\![B_{\varphi_2}(\mathbf{x}, \mathbf{z})]\!]$, hence $\pi_\rho(v) \models \varphi_2$ by the induction hypothesis for the outer structural induction and thus $\pi_\rho(v) \models \varphi_1\,U\,\varphi_2$ following the LTL semantics. For the inductive step $k > 1$, we have

$$B^k_{\varphi_1 U \varphi_2}(\mathbf{x}, \mathbf{z}) = B_{\varphi_2}(\mathbf{x}, \mathbf{z}) \vee \left(B_{\varphi_1}(\mathbf{x}, \mathbf{z}) \wedge \exists \mathbf{y}.\left(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \wedge B^{k-1}_{\varphi_1 U \varphi_2}(\mathbf{y}, \mathbf{z})\right)\right),$$

and directly that $(v, \rho) \in [\![B_\varphi(\mathbf{x}, \mathbf{x})]\!]$, hence $\pi_\rho(v) \models \varphi_1$ by the inductive hypothesis for the outer structural induction. Furthermore, there exists a (unique) successor v' of v such that $(v', \rho) \in [\![B^{k-1}_{\varphi_1 U \varphi_2}(\mathbf{y}, \mathbf{z})]\!]$, which, by the induction hypothesis for the inner induction on k implies that $\pi_\rho(v') \models \varphi_1\,U\,\varphi_2$, hence $\pi_\rho(v) \models \varphi_1\,U\,\varphi_2$ following the LTL semantics.

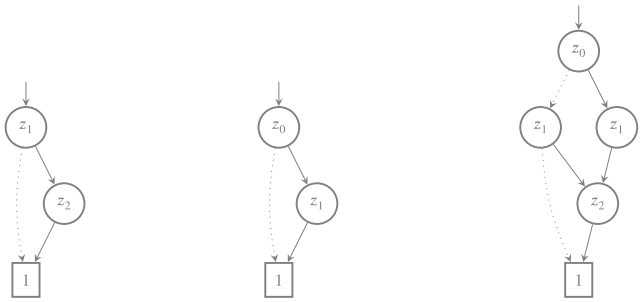
For the right to left implication, assume $\pi_\rho(v) \models \varphi_1\,U\,\varphi_2$. Suppose that $\pi_\rho(v) \models \varphi_2$. Then, by the inductive hypothesis for the outer structural induction we then have $(v, \rho) \in [\![B_{\varphi_2}(\mathbf{x}, \mathbf{z})]\!]$, hence $(v, \rho) \in [\![B_{\varphi_1 U \varphi_2}(\mathbf{x}, \mathbf{z})]\!]$. Otherwise, it must be the case that $\pi_\rho(v) \models \varphi_1$, while for the unique successor of v, v', we have $\pi_\rho(v') \models \varphi_1\,U\,\varphi_2$ by LTL semantics. Hence, by applying the inductive hypothesis for the outer induction, we have $(v, \rho) \in [\![B_{\varphi_1}(\mathbf{x}, \mathbf{z})]\!]$ and $(v', \rho) \in [\![B_{\varphi_1\,U\,\varphi_2}(\mathbf{x}, \mathbf{z})]\!]$, implying that $(v, \rho) \in [\![B_{\varphi_1 U \varphi_2}(\mathbf{x}, \mathbf{z})]\!]$. □

Example 3. Consider the network topology from Fig. 6a with the routing policy $\mathsf{Reach}(v_3)$. Given the LTL-definition of $\mathsf{Reach}(v_3)$, the ROBDD $B_{\mathsf{Reach}(v_3)}$ is given by the limit of the following inductively defined sequence: $B^{n+1}_{\mathsf{Reach}(v_3)}(\mathbf{x}, \mathbf{z}) = \mathbf{x}(v_3) \vee \exists \mathbf{y}.\left(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \wedge B^n_{\mathsf{Reach}(v_3)}(\mathbf{y}, \mathbf{z})\right)$ with $B^0_{\mathsf{Reach}(v_3)} = 0$. Fig. 7 provides some of the approximants with $B^4_{\mathsf{Reach}(v_3)}$ found to be the least fixed point.



(a) $B^2_{\mathsf{Reach}(v_3)}$ (b) $B^3_{\mathsf{Reach}(v_3)}$ (c) $B^4_{\mathsf{Reach}(v_3)}$

Fig. 7. Increasing approximants $B^n_{\mathsf{Reach}(v_3)}$.



(a) R (b) W (c) $W \wedge R$

Fig. 8. Encoding of different routing policies.

We shall denote by $B^*_\varphi(\mathbf{z})$ the ROBDD $\exists \mathbf{x}.B_\varphi(\mathbf{x},\mathbf{z}) \wedge \mathbf{x}(v_0)$, where $v_0 \in V$ is the source node. Rather than using BDDs for model-checking that individual routing configurations satisfy a given policy φ one by one, $B^*_\varphi(\mathbf{z})$ characterizes exactly in one single ROBDD the full set of routing configurations satisfying φ.

Example 4. Recall the network topology from Fig. 6a and the Boolean encoding of routing configurations and nodes from Example 2. Now consider the routing policies $W = \mathsf{Waypoint}(v_2, v_3)$ and $R = \mathsf{Reach}(v_3)$. The resulting ROBDDs for B^*_R, B^*_W and $B^*_{W \wedge R}$ are given in Fig. 8. It can be concluded that there are 6, 6 respectively 4 routing configurations satisfying the policies R, W respectively $R \wedge W$. Moreover, both ρ_i and ρ_f satisfy all three policies.

BDD encoding of update sequences. Again let $G = (V, E, \mathsf{src}, \mathsf{tgt})$ be a network topology and let φ be a routing policy, with ρ_i respectively ρ_f being initial respectively final routing configuration. We shall show how to symbolically synthesize correct (simple) update sequences using BDD encodings. The basis of the synthesis is the ROBDD $B^*_\varphi(\mathbf{z})$ encoding all routing configurations that are correct with respect to φ using Boolean variables $\mathbf{z} = \mathbf{z}_{v_0} \ldots \mathbf{z}_{v_k}, z^d_{v_0}, \ldots, z^d_{v_k}$. For simple updates it suffices to use single Boolean variables z_{v_j}, with z_{v_j} encoding $\rho_i(v_j)$ and $\neg z_{v_j}$ encoding $\rho_f(v_j)$, i.e. in case $\rho_f(v_j) \neq \rho_i(v_j)$. To encode a simple update between configurations ρ and ρ' we shall use Boolean variables $\mathbf{z}$ for encoding ρ and a corresponding (distinct) sequence of Boolean variables $\mathbf{zz}$ for encoding ρ'. The following Boolean expression U^s_φ encodes the set of possible simple updates that preserve correctness with respect to φ.

$$U^s_\varphi(\mathbf{z},\mathbf{zz}) = B^*_\varphi(\mathbf{z}) \wedge B^*_\varphi(\mathbf{zz}) \wedge \exists i.\left[z_{v_i} \wedge \neg zz_{v_i} \wedge \bigwedge_{j \neq i} z_{v_j} = zz_{v_j}\right] \tag{1}$$

Note that in this simple update the routing configuration changes for exactly one node v_i from the setting in the initial configuration ρ_i, encoded as z_{v_i}, to the setting in final configuration ρ_f, encoded as $\neg zz_{v_i}$. In the general case, the update can change the setting of any node arbitrarily, as given by the following Boolean expression U_φ.

$$U_\varphi(\mathbf{z},\mathbf{zz}) = B^*_\varphi(\mathbf{z}) \wedge B^*_\varphi(\mathbf{zz}) \wedge \exists i.\left[\mathbf{z}_{v_i} \neq \mathbf{zz}_{v_i} \wedge \bigwedge_{j \neq i} \mathbf{z}_{v_j} = \mathbf{zz}_{v_j}\right] \tag{2}$$

Lemma 3. *We have $(\rho, \rho') \in [\![U_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ (resp. $[\![U^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$) iff $\rho \neq \rho'$ and there exists an update (resp. simple update) u such that $\rho^u = \rho'$, $\pi_\rho(v_0) \models \varphi$ and $\pi_{\rho'}(v_0) \models \varphi$, where v_0 is the given source node.*

Proof. We prove the case for general updates. The simple case follows the same reasoning. First recall Equation (2). For the left to right implication, assume $(\rho, \rho') \in [\![U_\varphi(\mathbf{z}, \mathbf{zz})]\!]$. By Lemma 2, it follows from the first two conjuncts $B^*_\varphi(\mathbf{z})$ and $B^*_\varphi(\mathbf{zz})$ that the routing configurations encoded by $\mathbf{z}$ and $\mathbf{zz}$, namely ρ and ρ' have the property that the induced path (starting from v_0) satisfies φ, thus $\pi_\rho(v_0) \models \varphi$ and $\pi_{\rho'}(v_0) \models \varphi$. The last conjunct ensures that the two routing configurations ρ and ρ' differ by exactly one update u, hence $\rho' = \rho^u$.

For the right to left implication, assume the existence of an update u such that $\rho^u = \rho'$, $\pi_\rho(v_0) \models \varphi$ and $\pi_{\rho'}(v_0) \models \varphi$. By Lemma 2 we then have $(v_0, \rho) \in [\![B_\varphi(\mathbf{x}, \mathbf{z})]\!]$ and $(v_0, \rho') \in [\![B_\varphi(\mathbf{x}, \mathbf{z})]\!]$, implying $\rho \in [\![B^*_\varphi(\mathbf{z})]\!]$ and $\rho' \in [\![B^*_\varphi(\mathbf{zz})]\!]$. As $\rho^u = \rho'$ and $\rho \neq \rho'$, there exists exactly one node where the routing is changed by the update u, implying that the last conjunct is satisfied by ρ and ρ', encoded by variables $\mathbf{z}$ and $\mathbf{zz}$ respectively. We can now conclude that $(\rho, \rho') \in [\![U_\varphi(\mathbf{z}, \mathbf{zz})]\!]$. □

To enable synthesis of correct (simple) update sequences, the following recursively defined ROBDD is the key.

$$R^s_\varphi(\mathbf{z}, \mathbf{zz}) \overset{\min}{=} \mathbf{z}(\rho_f) \vee \exists \mathbf{zzz}.\big(U^s_\varphi(\mathbf{z}, \mathbf{zz}) \wedge R^s_\varphi(\mathbf{zz}, \mathbf{zzz})\big) \tag{3}$$

The expression encodes the set of simple updates that preserve correctness with respect to φ while ensuring reachability of the final routing configuration.

Lemma 4. *We have $(\rho, \rho') \in [\![R^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ iff there exists a correct simple update sequence $w = u_0 u_1 \cdots u_k$ with respect to ρ and φ such that $\rho' = \rho^{u_0}$ and $\rho^w = \rho_f$.*

Proof. We recall Equation (3) and that $R^s_\varphi(\mathbf{z}, \mathbf{zz})$ is obtained as the limit of the finite sequence of approximations $\mathcal{R}^n_\varphi(\mathbf{x}, \mathbf{z})$ with $\mathcal{R}^0_\varphi(\mathbf{x}, \mathbf{z}) = 0$ and otherwise

$$\mathcal{R}^{n+1}_\varphi(\mathbf{z}, \mathbf{zz}) = \mathbf{z}(\rho_f) \vee \exists \mathbf{zzz}.\big(U^s_\varphi(\mathbf{z}, \mathbf{zz}) \wedge \mathcal{R}^n_\varphi(\mathbf{zz}, \mathbf{zzz})\big)$$

i.e. there exists a k for which $\mathcal{R}^{k+1}_\varphi(\mathbf{x}, \mathbf{z}) = \mathcal{R}^k_\varphi(\mathbf{x}, \mathbf{z})$.

For the left to right implication, we assume $(\rho, \rho') \in [\![R^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ and proceed by induction on k. For the base case $k = 0$, we immediately have that $\rho = \rho_f$ by the first conjunct and thus there exists a trivial empty update sequence with the desired properties. For the inductive step $k > 0$, we have

$$\mathcal{R}^{k+1}_\varphi(\mathbf{z}, \mathbf{zz}) = \mathbf{z}(\rho_f) \vee \exists \mathbf{zzz}.\big(U^s_\varphi(\mathbf{z}, \mathbf{zz}) \wedge \mathcal{R}^k_\varphi(\mathbf{zz}, \mathbf{zzz})\big)\,.$$

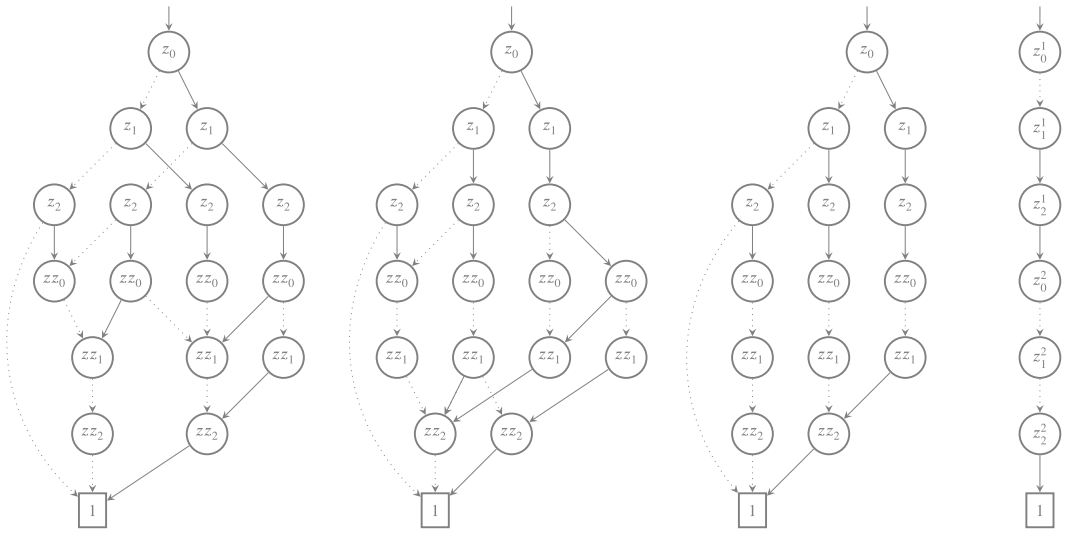
As (by assumption) $(\rho, \rho') \in [\![R^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ and therefore $(\rho, \rho') \in [\![\mathcal{R}^{k+1}\varphi(\mathbf{z}, \mathbf{zz})]\!]$, there exists a configuration ρ'' such that $(\rho', \rho'') \in [\![\mathcal{R}^k_\varphi(\mathbf{zz}, \mathbf{zzz})]\!]$ while also $(\rho, \rho') \in [\![U^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$. By application of Lemma 3, we immediately have that $(\rho, \rho') \in [\![U^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ implies the existence of a φ-preserving update u from ρ to ρ'. Furthermore, $(\rho', \rho'') \in [\![\mathcal{R}^k_\varphi(\mathbf{zz}, \mathbf{zzz})]\!]$ implies, by application of the inductive hypothesis, that there exists a correct simple update sequence $w = u_0 u_1 \cdots u_n$ with respect to ρ' and φ such that $\rho'' = \rho^{u_0}$ and $\rho^w = \rho_f$. In conclusion, there must necessarily exists an update sequence $w^* = uw$ that is simple and correct with respect to ρ and φ such that $\rho' = \rho^u$ and $\rho^{w^*} = \rho_f$.

For the right to left implication, we assume the existence of a correct simple update sequence $w = u_0 u_1 \cdots u_n$ with respect to ρ and φ such that $\rho' = \rho^{u_0}$ and $\rho^w = \rho_f$. We proceed by induction on the length of w. For the base case $|w| = 0$, it is the case that $\rho = \rho_f$ and trivially $(\rho, \rho') \in [\![R^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ by the first conjunct. For the inductive step $|w| > 0$, notice first by Lemma 3 that $(\rho, \rho') \in [\![U^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ by the property that w is correct with respect to φ. Now let $w' = u_1 u_2 \cdots u_n$ be the suffix of w starting from u_1. As w is a correct simple update sequence w.r.t. ρ and φ such that $\rho' = \rho^{u_0}$ and $\rho^w = \rho_f$, w' must be a correct simple update sequence with respect to ρ^{u_0} and φ such that $\rho^{u_0 u_1} = \rho^{u_0 u_1}$ and $\rho^{u_0^{w'}} = \rho_f$. We can now apply the inductive hypothesis as $|w'| < |w|$ to conclude that $(\rho^{u_0}, \rho^{u_0 u_1}) \in [\![R^s_\varphi(\mathbf{zz}, \mathbf{zzz})]\!]$, hence $(\rho, \rho') \in [\![R^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$. □

All correct, simple update sequences of length N may now be characterized by the following Boolean expression, where $\mathbf{z}^i$ are (distinct) Boolean variables encoding the routing configuration after i updates:

$$S^s_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N) = \mathbf{z}^0(\rho_i) \wedge \mathbf{z}^N(\rho_f) \wedge \bigwedge_{i=0}^{N-1} R^s_\varphi(\mathbf{z}^i, \mathbf{z}^{i+1})\,. \tag{4}$$

Theorem 1. *We have $(\rho_0, \rho_1, \ldots, \rho_N) \in [\![S^s_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N)]\!]$ iff there exists a simple correct update sequence $w = u_0 u_1 \cdots u_{N-1}$ with respect to φ and ρ_0 such that $\rho_{k+1} = \rho_k^{u_k}$ for all k with $0 \leq k < N$, $\rho_0 = \rho_i$ and $\rho_N = \rho_f$.*



(a) W (b) R (c) $W \wedge R$ (d) UUS

Fig. 9. Encoding of all correct simple update-steps (a-c); unique update sequence (UUS) for $W \wedge R$ (d).

Proof. We recall Equation (4). For the left to right direction, suppose $(\rho_0, \rho_1, \ldots, \rho_N) \in [\![S^s_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N)]\!]$. The first two conjuncts directly ensure that $\rho_0 = \rho_i$ and $\rho_n = \rho_f$. By repeated application of Lemma 4, the last conjunct ensures for all k with $0 \leq k < N$ the existence of a simple update u_k such that $\rho_{k+1} = \rho_k^{u_k}$ with φ preserved by both ρ_{k+1} and ρ_k. Hence the update sequence $w = u_0 u_1 \ldots u_{N-1}$ is a simple correct update sequence with respect to φ with the desired properties.

For the right to left direction, suppose there exists a simple correct update sequence w.r.t. φ and ρ_0, given by $w = u_0 u_1 \cdots u_{N-1}$, such that $\rho_{k+1} = \rho_k^{u_k}$ for all k with $0 \leq k < N$, $\rho_0 = \rho_i$ and $\rho_N = \rho_f$. Now, any suffix $w' = u_j \cdots u_{N-1}$ with $0 \leq j \leq (N-1)$ is a simple correct update sequence w.r.t. $\rho_0^{u_0 \cdots u_{j-1}}$. Thus, we can repeatedly apply Lemma 4 for all such suffixes, to conclude that $(\rho_0^{u_0 \cdots u_k}, \rho_0^{u_0 \cdots u_k u_{k+1}}) \in [\![R^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ for all $0 \leq k \leq (N-1)$. As $\rho_{k+1} = \rho_0^{u_0 \cdots u_k}$ we then have $(\rho_k, \rho_{k+1}) \in [\![R^s_\varphi(\mathbf{z}, \mathbf{zz})]\!]$ for all $0 \leq k \leq (N-1)$. We can now conclude that $(\rho_0, \rho_1, \ldots, \rho_N) \in [\![S^s_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N)]\!]$ as required. □

For the synthesis in the general case, we simply replace U^s_φ in (3) with U_φ to get a ROBDD R_φ characterizing (general) update sequences leading to ρ_f. We can now replace R^s_φ with R_φ in (4) to get a characterization of all correct (general) update sequences of length N.

Example 5. Consider again the network topology from Fig. 6a and the routing policies $W = \mathsf{Waypoint}(v_2, v_3)$ and $R = \mathsf{Reach}(v_3)$. The full sets of correct simple update-steps with respect to W, R and $W \wedge R$ are given by the ROBDDs R^s_W, R^s_R and $R^s_{W \wedge R}$ given in Fig. 9(a-c). Instantiating Equation (4) with these ROBDDs reveals that there are 3, 3 respectively 1 correct simple update sequences of length 3 with respect to the routing policies W, R respectively $W \wedge R$.

The unique simple update sequence for $W \wedge R$ (ignoring the initial and final routing configurations) is given by the ROBDD in Fig. 9(d). Here the values suggested for the first three Boolean variables z^1_0, z^1_1, z^1_2 indicate that the routing configuration after the first update is given by the edges $(v_0, v_2), (v_1, v_2), (v_2, v_3)$. Similarly, the values of the last three Boolean variables z^2_0, z^2_1, z^2_2 indicate the edges $(v_0, v_2), (v_1, v_3), (v_2, v_3)$ as the configuration after the second update. Note, that in case there is no correct (simple) update sequence the resulting ROBDD becomes empty (just consisting of the node false).

4. Synthesis with additional optimization criteria

During update synthesis, we may be interested in accounting for performance aspects in the update synthesis, which is of practical importance but has not been studied yet in the literature. To this end, we extend our model to weighted topologies representing quantities such as latency and hop-count [7,20,49]. In the following, we ignore loops as they can be detected and do not need to be considered for optimization.

Definition 9 *(Weighted topology and path weight).* A weighted network topology is a tuple $G = (V, E, \mathsf{src}, \mathsf{tgt}, \eta)$ where $(V, E, \mathsf{src}, \mathsf{tgt})$ is a network topology and $\eta \colon E \to \mathbb{N}$ is the edge *weight* function. The weight extends in a natural way to paths, by $\eta(e_1 e_2 \ldots e_n) = \sum_{i=1}^{n} \eta(e_i)$ where $e_1 e_2 \ldots e_n \in E^*$.

Definition 10 *(Optimal update synthesis problem).* For an update synthesis problem $P = (G, \rho_i, \rho_f, v_0, \varphi)$ and path valuation function f over some edge weight function η, the optimal synthesis problem is to find a solution $w^{opt} \in \mathsf{Sol}(P)$ such that

Fig. 10. Weighted topology with initial (solid) and final (dashed) routings.

$$w^{opt} = \operatorname*{argmin}_{w \in \mathsf{Sol}(P)} \max_{w' \text{ prefix of } w} \eta(\pi_{\rho_i^{w'}}(v_0)) \,.$$

Example 6. Consider a network update synthesis problem $(G, \rho_i, \rho_f, v_0, \mathsf{Reach}(v_2))$ where ρ_i (solid arrows) and ρ_f (dashed arrows) are depicted in Fig. 10 and edges are annotated by weights. Any of the two possible simple update sequences is a solution. For the sequence that first updates v_0 to use the expensive edge of weight 4 to v_1, there is an intermediate configuration of weight 7. However, the optimal solution is the update sequence that first updates v_1 followed by v_0 and creates a transient configuration of weight only 2.

We shall now extend our symbolic encoding of routing configurations, policies and (simple) update sequences to the weighted setting. In particular, we shall show how our encoding allows us to synthesize weight-optimal simple update sequences. Let $G = (V, E, \mathsf{src}, \mathsf{tgt}, \eta)$ be a weighted topology, and let $C \in \mathbb{N}$ be the maximum weight of any acyclic path in G. Using $k = \lceil \log(C) \rceil$ Boolean variables $\mathbf{c} = c_0, \ldots, c_{k-1}$ we may encode in binary any weight between 0 and C. As in before we shall encode routing configurations ρ by using Boolean variables $\mathbf{z}_v$ for each node v to encode the edge $\rho(v)$. For simplicity we shall assume that $\rho(v)$ is defined for any node v in the remainder of this section. Using Boolean variables $\mathbf{x}$ and $\mathbf{y}$ to encode source and target nodes and Boolean variables $\mathbf{c}$ to encode the weight of edges, the following Boolean expression T_c encodes the possible weighted transitions:

$$T_c(\mathbf{x}, \mathbf{z}_{v_0}, \ldots, \mathbf{z}_{v_k}, \mathbf{y}, \mathbf{c}) = \bigvee_{v \in V} \bigvee_{e \in E_v} \big(\mathbf{x}(v) \wedge \mathbf{z}_v(e) \wedge \mathbf{y}(\mathsf{tgt}(e)) \wedge \mathbf{c}(e)\big) \,.$$

As in Section 2.3, we shall assume that the routing policy φ considered enforces at least reachability, i.e. $\varphi = \varphi' \wedge \mathsf{Reach}(d)$ for some node d. Now let $R_d = R_d(\mathbf{x}, \mathbf{z}, \mathbf{c})$ be the minimal fixed point defined by

$$R_d(\mathbf{x}, \mathbf{z}, \mathbf{c}) \overset{\min}{=} \mathbf{x}(d) \vee \exists \mathbf{y}, \mathbf{c}', \mathbf{c}'', \mathbf{c}'''. \big[T_c(\mathbf{x}, \mathbf{z}, \mathbf{y}, \mathbf{c}') \wedge R_d(\mathbf{y}, \mathbf{z}, \mathbf{c}'') \wedge \mathsf{sum}(\mathbf{c}', \mathbf{c}'', \mathbf{c}''') \wedge \mathsf{leq}(\mathbf{c}''', \mathbf{c})\big] \,.$$

Here we use the existence of simple ROBDD sum and leq encoding addition and comparison (as ternary and binary predicates) of natural numbers such that $(n_1, n_2, n_3) \in [\![\mathsf{sum}]\!]$ iff $n_1 + n_2 = n_3$ and $(n_1, n_2) \in [\![\mathsf{leq}]\!]$ iff $n_1 \leq n_2$. Then $(v, \rho, c) \in [\![R_d]\!]$ if $\pi_\rho(v)$ ends in d with a total weight $\eta(\pi_\rho(v))$ not exceeding c.

The Boolean expression $R_d^*(\mathbf{z}, \mathbf{c})$ given by $\exists \mathbf{x}.\mathbf{x}(v_0) \wedge R_d(\mathbf{x}, \mathbf{z}, \mathbf{c})$ describes all pairs (ρ, c), where the weight of the path $\pi_\rho(v_0)$ does not exceed c. Now to ensure that the encoded routing configuration in addition satisfies the routing policy φ', we use the Boolean expression $B_\varphi^{*,c}(\mathbf{z}, \mathbf{c}) = B_{\varphi'}(\mathbf{z}) \wedge R_d^*(\mathbf{z}, \mathbf{c})$. Now, the expression

$$U_\varphi^{s,c}(\mathbf{z}, \mathbf{zz}, \mathbf{c}) = B_\varphi^{*,c}(\mathbf{z}, \mathbf{c}) \wedge B_\varphi^{*,c}(\mathbf{zz}, \mathbf{c}) \ \wedge \exists i. \big[z_{v_i} \wedge \neg zz_{v_i} \wedge \bigwedge_{j \neq i} z_{v_j} = zz_{v_j}\big]$$

encodes a single correctness preserving update between configurations whose accumulated weight is bounded by $\mathbf{c}$.

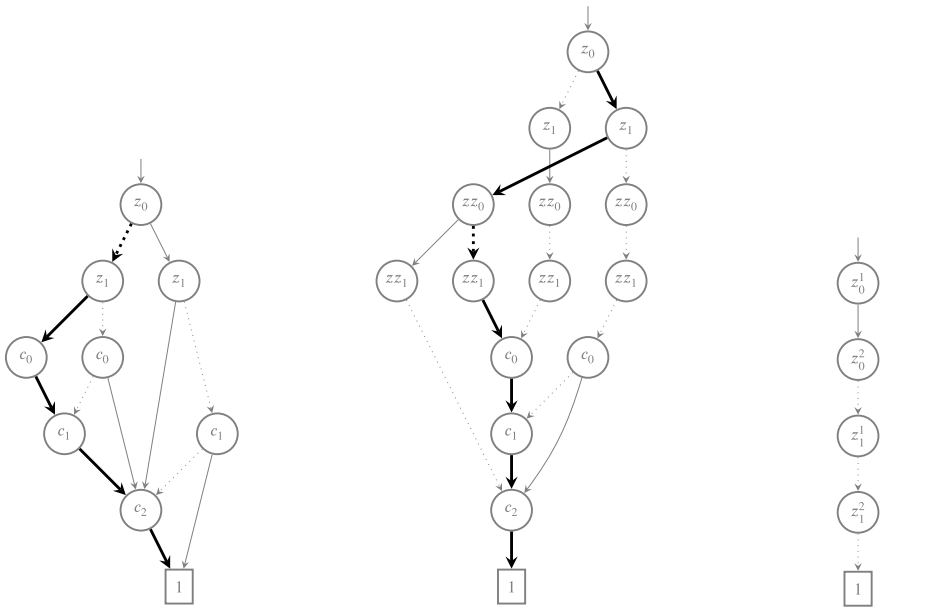
Thus, all correct, weight-bounded simple update sequences of length N may be characterized by the following Boolean expression S_φ^c, where $\mathbf{z}^i$ are (distinct) Boolean variables encoding the routing configuration after i updates:

$$S_\varphi^c(\mathbf{z}^0, \ldots, \mathbf{z}^N, \mathbf{c}) = \mathbf{z}^0(\rho_i) \wedge \mathbf{z}^N(\rho_f) \wedge \bigwedge_{i=0}^{N-1} U_\varphi^{s,c}(\mathbf{z}^i, \mathbf{z}^{i+1}, \mathbf{c}) \,.$$

Finally, the update sequences of length N solving the *optimal* synthesis problem are easily characterized by the following single expression O:

$$O_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N) = \exists \mathbf{c}. \Big[S_\varphi^c(\mathbf{z}^0, \ldots, \mathbf{z}^N, \mathbf{c}) \ \wedge \forall \mathbf{c}'. \big(S_\varphi^c(\mathbf{z}^0, \ldots, \mathbf{z}^N, \mathbf{c}') \rightarrow \mathsf{leq}(\mathbf{c}, \mathbf{c}')\big)\Big] \,.$$

Example 7. Recall the weighted topology from Fig. 6a. To encode the routing configurations, two Boolean variables z_0 and z_1 suffice (with the initial routing being encoded by $z_0 \wedge z_1$). Given that the maximum weight of a path is 7, three Boolean variables c_0, c_1, c_2 suffice to encode weight of any acyclic path. In this example, we consider the property $\varphi = \mathsf{Reach}(v_2)$. Now Fig. 11(a) is $B_\varphi^{*,c}(\mathbf{z}, \mathbf{c})$ encoding all pairs (ρ, c), where ρ is a correct routing wrt. φ with total reachability weight no more than c. The highlighted path is the encoding of the routing where the node v_0 uses the expensive edge (weight 4) to v_1 and v_1 is still using the initial routing. It can be seen that the total weight indicated by the path is 7. Now, Fig. 11(b) is $U_\varphi^{s,c}(\mathbf{z}, \mathbf{zz}, \mathbf{c})$, encoding all updates that are correct wrt. φ as well as their weight. Here the highlighted path



(a) $B_\varphi^{*,c}(\mathbf{z}, \mathbf{c})$ (b) $U_\varphi^{s,c}(\mathbf{z}, \mathbf{zz}, \mathbf{c})$ (c) $O_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N)$

Fig. 11. ROBDDs for Cost-Optimal Update Synthesis.

indicates the update from the initial routing to the "expensive" routing, again with the path indicating the weight 7. Finally, $O_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N)$ encodes the optimal (correct) update sequence, which first updates the routing for v_1 and only then the routing for v_0.

5. Implementation and evaluation

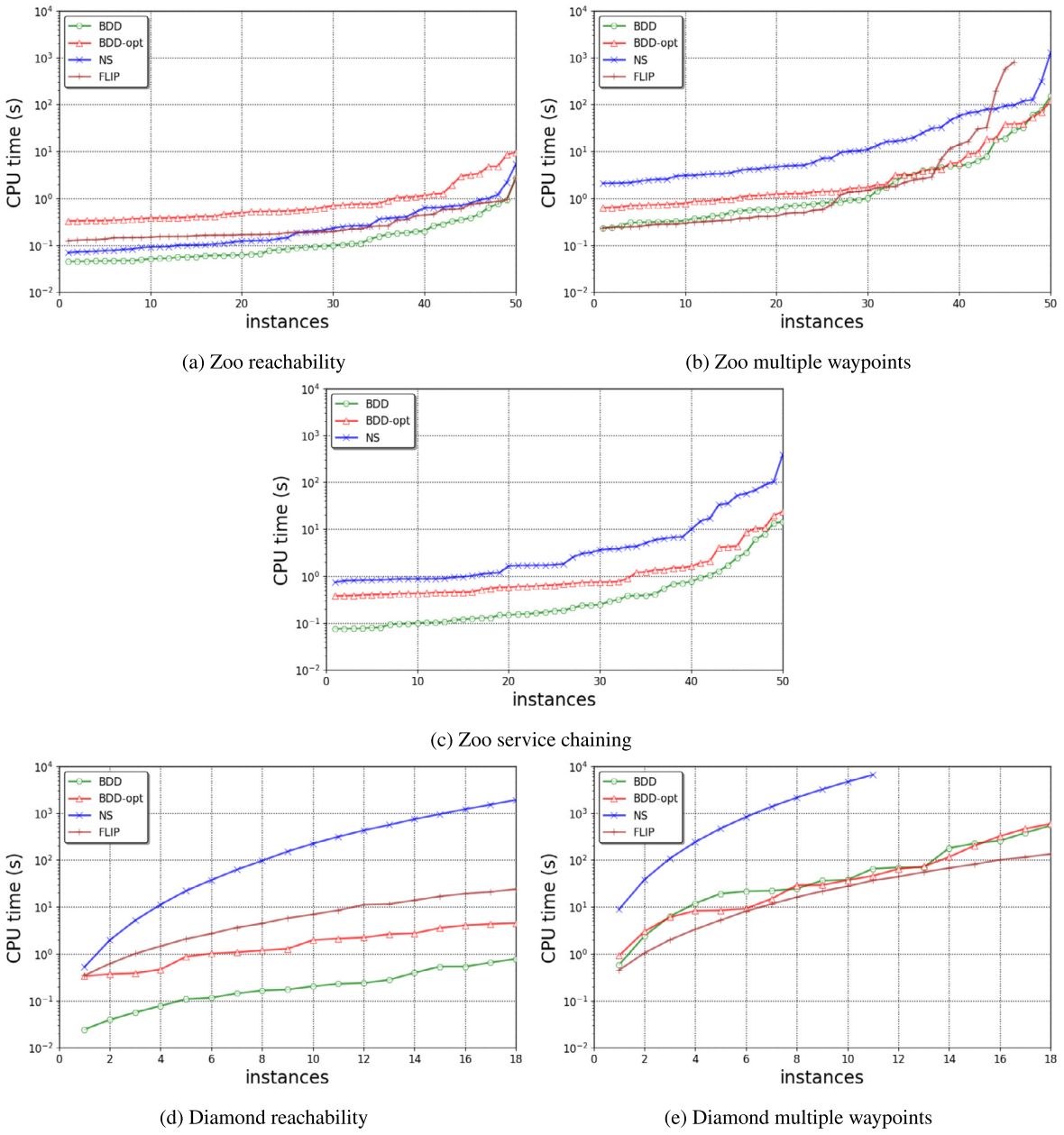
Our tool *AllSynth* for solving the update synthesis problem is implemented in Python and relies on a Cython wrapper [2] of the CUDD [46] package for manipulation of ROBDD. The overall tool architecture is given in Fig. 5. From a given network topology with the initial and final routing, the tool produces either a simple or general update sequence satisfying a given policy, as well as the information about the number of possible solutions. As all such correct solutions are symbolically represented in a compact way as an ROBDD, it is possible to generate alternative solutions without any additional computational effort.

We evaluate *AllSynth*, both with and without cost-optimization, against two state-of-the-art update synthesis tools, NetSynth [40] and FLIP [48]. NetSynth can compute only a simple update sequence or inform the user that there is no solution; the synthesis of general update sequences is not supported. FLIP can synthesise sequences of steps (groups of switches or routers) in which order the network can be updated, however, if such a sequence does not exist, the tool may introduce additional forwarding rules and use tagging of packets. As NetSynth and FLIP do not support general update sequences, the running times are only compared for simple update sequences.

All experiments are executed on Ubuntu 14.04 cluster with 2.3 GHz AMD Opteron 6376 processors with 2 hour timeout and 14 GB memory limit. To ensure the correctness of our implementation, we checked that for all instances, we agree with both NetSynth and FLIP on the existence/nonexistence of simple update sequences and we verified that the update sequences returned by NetSynth and FLIP are included in the ROBDD computed by *AllSynth*. A reproducibility package with the Python implementation and all benchmarks with scripts allowing to rerun the experiments is available in [33].

We consider a scalable synthetic topology and the standard benchmark of 261 real-world network topologies from the Topology Zoo dataset [30]. The class of synthetic topologies, referred to as *diamond* topologies, are taken from the NetSynth evaluation benchmark [40] and are formed by disjoint initial and final routing paths that only share the initial and final node. The size of the problem is defined to be the sum of the lengths of the two paths—we include instances of sizes up to 2000. The Topology Zoo instances are five times sequentially concatenated in order to obtain larger topologies where the size of the update problems ranges from 20 to 679. We display the 50 most difficult instances of the problem.

We consider three classes of update policies: $\mathsf{Reach}(d)$, $\mathsf{MultiWaypoint}(W, d)$ and $\mathsf{Service}(\omega, d)$. For $\mathsf{MultiWaypoint}(W, d)$, we let every 5*th* node on both the initial and final path be included in W. For $\mathsf{Service}(\omega, d)$, the sequence ω is generated by including every 5*th* node that is traversed by both the initial and final path. Because the diamond update problem consists of two disjoint paths, the service chaining policy is not considered here. The policy language of NetSynth is identical to our LTL-based specifications and hence we are able to directly express all these properties in this language. On the other hand, the policy input to FLIP enumerates all admissible subpaths that are considered, in logical disjunction. The encoding of the



(a) Zoo reachability

(b) Zoo multiple waypoints

(c) Zoo service chaining

(d) Diamond reachability

(e) Diamond multiple waypoints

Fig. 12. Experimental Results.

service chaining policy then entails an exhaustive enumeration of all paths that satisfy the service chaining policy and we therefore do not include FLIP in our service chaining experiments.

Results. The experiments are summarized in a number of so-called *cactus plots* [8] in Fig. 12, where for each method all instances of the problem are independently sorted from the fastest to the slowest one and plotted on the x-axis, and the y-axis (note the logarithmic scale) shows the increasing running time. If some curve does not reach to the right end of the plot, this means that the corresponding tool is not able to solve the remaining instances within the given timeout and memory limit. While cactus plots do not provide instance-to-instance runtime comparison, they provide an overall performance evaluation of the different tools.

For the experiments on the collection of real networks from the Topology Zoo presented in Figs. 12a to 12c, we notice that none of the tools has difficulty solving the synthesis of the plain reachability policy and it takes less than 10 seconds for all instances—here our approach without the cost optimization (BDD) has a slight margin, whereas the cost optimal algorithm (BDD-opt) is the slowest one (though solving a more general problem than the other ones). For waypointing,

while FLIP is performing well on small instances, it shows a noticeable decrease in performance once it reaches the most difficult problems: its running time quickly deteriorates and it is as the only tool not able to solve some of the largest instances. We maintain about one order of magnitude advantage over NetSynth (NS), which is the case also for service chaining. The overhead for computing the cost optimal solutions is less significant for the more complex policies.

Results for diamond topologies are given in Figs. 12d and 12e. We observe that for reachability our computation of all solutions is about one order of magnitude faster than FLIP (and even the cost optimal algorithm is faster than FLIP) and several orders of magnitude faster than NetSynth (both tools terminate as soon as they find the first correct update sequence). For waypointing, we still significantly outperform NetSynth and both BDD and BDD-opt are almost comparable with FLIP which shows better performance at the largest instances.

In conclusion, our experiments demonstrate that *AllSynth*, based on the symbolic BDD technology, not only significantly outperforms state-of-the-art tools on all non-trivial real-world networks, but also provides higher generality. Indeed, *AllSynth* computes *all* solutions, compared to only one solution returned by NetSynth or a more general sequence of update steps generated by FLIP. This aspect is important for the practical usage by network operators as it allows them to iteratively choose the most suitable update sequence. The additional optimization for computing cost optimal update sequences yields an acceptable overhead compared, especially for the more complex routing policies like waypointing and service chaining.

6. Conclusion

We presented an efficient approach for synthesizing correct update sequences for software-defined networks. In contrast to existing tools, our approach is fully symbolic and relies on BDD technology. As a result, we are able to represent *all* solutions to the update synthesis problem in a succinct binary tree, preserving generic routing policies (e.g., service chaining) that can be described in LTL. Our prototype implementation of *AllSynth* outperforms the state-of-the-art tools NetSynth and FLIP in many scenarios (e.g., on the real-world Internet topologies), while at the same time extending the generality.

Our experiments focused on the generation of simple update sequences (at most one update per flow per switch), similar to the methodology used in NetSynth and FLIP. *AllSynth* however also supports a novel generalization where a switch can be updated several times. This is particularly useful for the instances of the update synthesis problem that do not have any simple solution. In this case, NetSynth does not provide any alternative (and in fact does not terminate even on relatively small negative instances); FLIP may degrade to a two-phase commit strategy that is less preferable as it requires the duplication of forwarding rules as well as additional packet header space. *AllSynth* instead tries to suggest a general update sequence that does not require packet tagging. Moreover, our tool allows us to further select cost-optimal solutions with respect to any given cost function, representing for example the worst-case latency in any transient configuration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We want to thank the anonymous reviewers for their useful comments and suggestions. The research has been funded by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19045], the ERC Advanced Grant LASSO, the Villum Investigator Grant S4OS (37819) from Villum Fonden, DFF project QASNET as well as DIREC: Digital Research Centre Denmark.

Appendix A. Running the tool

We will in this example consider running *AllSynth* [33] for a small instance of the diamond class of models. Concretely, we consider a topology consisting only of two paths, where the nodes on the initial path are given by $v_0v_1v_2v_3v_4$ and the nodes on the final path are $v_0v_6v_7v_8v_4$. The policy of interest is $\mathsf{Reach}(v_4)$.

By running *AllSynth* with the following command, all solutions are synthesizes as one ROBDD:

Input

```
python3 run.py -t BDD -e diamond --index 5 --e-prop Reach
```

As a part of the output, the tool reports the time it takes to synthesize the set of all correct update sequences, as well as an indication of whether this set is empty:

Output

```
Policy: (Reach 4)
Initial path of nodes: [0, 1, 2, 3, 4]
Final path of nodes: [0, 6, 7, 8, 4]
Time: 0.006619453430175781
Solution: non-empty
```

To get a concrete update sequence, we add the `--some` argument:

Input

```
python3 run.py -t BDD -e diamond --index 5 --e-prop Reach
               --some
```

The synthesized update sequence is then given as a triple:

Output

```
Solution: (set(), [8, 6, 7, 0], {1, 2, 3, 4}).
```

The empty set indicates that no nodes were updated before the actual synthesis procedure executed. The middle sequence encodes that nodes v_8, v_7 and v_6 must be updated before v_0 to maintain reachability of v_4. The final set includes all trivial nodes that do not change routing or is only present on the initial path.

To fully use Lemma 1 reduction, we add the `--reduce` argument:

Input

```
python3 run.py -t BDD -e diamond --index 5 --e-prop Reach
               --some --reduce
```

As the initial and final path are disjoint, the critical middle part of the synthesized solution is now a singleton:

Output

```
Solution: ({8, 6, 7}, [0], {1, 2, 3, 4}).
```

The first set includes all nodes that may be updated (in any order) before synthesizing a correct update sequence for the remaining nodes (now only node v_0).

To count the number of solution, the `--count` argument can be provided instead of `--some`:

Input

```
python3 run.py -t BDD -e diamond --index 5 --e-prop Reach
               --reduce --count
python3 run.py -t BDD -e diamond --index 5 --e-prop Reach
               --count
```

If the reduction is used, the tool counts 1 solution and otherwise 6 as one may update v_6, v_7 and v_8 in any order as long as v_0 is updated at the end.

References

[1] S. Akhoondian Amiri, S. Dudycz, S. Schmid, S. Wiederrecht, Congestion-free rerouting of flows on dags, in: 45th International Colloquium on Automata, Languages, and Programming (ICALP), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018, 143.

[2] dd Python package, https://github.com/tulip-control/dd, 2021.

[3] C.J. Anderson, N. Foster, A. Guha, J.B. Jeannin, D. Kozen, C. Schlesinger, D. Walker, Netkat: semantic foundations for networks, ACM SIGPLAN Not. 49 (2014) 113–126.

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

[5] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, D. Walker, Don't mind the gap: bridging network-wide objectives and device-level configurations, in: Proceedings of the 2016 ACM SIGCOMM Conference, 2016, pp. 328–341.

[6] T. Benson, A. Akella, D.A. Maltz, Network traffic characteristics of data centers in the wild, in: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, 2010, pp. 267–280.

[7] P. Bonsma, The complexity of rerouting shortest paths, Theor. Comput. Sci. 510 (2013) 1–12.

[8] M.N. Brain, J.H. Davenport, A. Griggio, Benchmarking solvers, SAT-style, in: Proceedings of the 2nd International Workshop on Satisfiability Checking and Symbolic Computation Co-Located with the 42nd International Symposium on Symbolic and Algebraic Computation (ISSAC'17), 2017, pp. 1–15, https://ceur-ws.org/.

[9] S. Brandt, K.T. Förster, R. Wattenhofer, On consistent migration of flows in SDNs, in: IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE, 2016, pp. 1–9.

[10] Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput. C-35 (1986) 677–691, https://doi.org/10.1109/TC.1986.1676819.

[11] M. Canini, P. Kuznetsov, D. Levin, S. Schmid, A distributed and robust SDN control plane for transactional network updates, in: 2015 IEEE Conference on Computer Communications (INFOCOM), IEEE, 2015, pp. 190–198.

[12] P. Černý, N. Foster, N. Jagnik, J. McClurg, Optimal consistent network updates in polynomial time, in: International Symposium on Distributed Computing, Springer, 2016, pp. 114–128.

[13] R. Chirgwin, Google routing blunder sent Japan's Internet dark on Friday, https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/, 2017.

[14] A. Cimatti, E.M. Clarke, F. Giunchiglia, M. Roveri, NUSMV: a new symbolic model checker, Int. J. Softw. Tools Technol. Transf. 2 (2000) 410–425, https://doi.org/10.1007/s100090050046.

[15] S. Dudycz, A. Ludwig, S. Schmid, Can't touch this: consistent network updates for multiple policies, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2016, pp. 133–143.

[16] Duluth News Tribune, Human error to blame in Minnesota 911 outage, https://www.ems1.com/911/articles/389343048-Officials-Human-error-to-blame-in-Minn-911-outage/, 2018.

[17] A. El-Hassany, P. Tsankov, L. Vanbever, M. Vechev, Netcomplete: practical network-wide configuration synthesis with autocompletion, in: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), 2018, pp. 579–594.

[18] N. Feamster, J. Rexford, Why (and How) Networks Should Run Themselves, arXiv report, 2017.

[19] B. Finkbeiner, M. Gieseking, J. Hecking-Harbusch, E.R. Olderog, Model checking data flows in concurrent network updates (full version), arXiv preprint arXiv:1907.11061, 2019.

[20] K. Foerster, S. Schmid, S. Vissicchio, Survey of consistent software-defined network updates, IEEE Commun. Surv. Tutor. 21 (2019) 1435–1461.

[21] K.T. Foerster, On the consistent migration of unsplittable flows: upper and lower complexity bounds, in: 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA), IEEE, 2017, pp. 1–4.

[22] K.T. Foerster, T. Luedi, J. Seidel, R. Wattenhofer, Local checkability, no strings attached:(a) cyclicity, reachability, loop free updates in SDNs, Theor. Comput. Sci. 709 (2018) 48–63.

[23] G.D. Giacomo, M.Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13), AAAI Press, 2013, pp. 854–860.

[24] M. Glavind, N. Christensen, J. Srba, 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.

[25] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, et al., Leveraging sdn layering to systematically troubleshoot networks, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, 2013, pp. 37–42.

[26] X. Jin, H.H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, R. Wattenhofer, Dynamic scheduling of network updates, in: ACM SIGCOMM Computer Communication Review, ACM, 2014, pp. 539–550.

[27] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, S. Whyte, Real time network policy checking using header space analysis, Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13), 2013, pp. 99–111.

[28] P. Kazemian, G. Varghese, N. McKeown, Header space analysis: static checking for networks, Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), 2012, pp. 113–126.

[29] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, S. Schmid, Adaptable and data-driven softwarized networks: review, opportunities, and challenges, in: Proceedings of the IEEE (PIEEE), 2019.

[30] S. Knight, H.X. Nguyen, N. Falkner, R.A. Bowden, M. Roughan, The Internet topology zoo, IEEE J. Sel. Areas Commun. 29 (2011) 1765–1775, https://doi.org/10.1109/JSAC.2011.111002.

[31] D. Kreutz, F.M. Ramos, P.E. Verissimo, C.E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: a comprehensive survey, Proc. IEEE 103 (2014) 14–76.

[32] K. Larsen, A. Mariegaard, S. Schmid, J. Srba, 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), Springer, 2022, pp. 344–362.

[33] K. Larsen, A. Mariegaard, S. Schmid, J. Srba, Reproducibility package for: the hazard value: a quantitative network connectivity measure accounting for failures, https://doi.org/10.5281/zenodo.6534948, 2022.

[34] C.Y. Lee, Representation of switching circuits by binary-decision programs, Bell Syst. Tech. J. 38 (1959) 985–999, https://doi.org/10.1002/j.1538-7305.1959.tb01585.x.

[35] H.H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, D. Maltz, Zupdate: updating data center networks with zero loss, in: ACM SIGCOMM Computer Communication Review, ACM, 2013, pp. 411–422.

[36] A. Ludwig, S. Dudycz, M. Rost, S. Schmid, Transiently secure network updates, ACM SIGMETRICS Perform. Eval. Rev. 44 (2016) 273–284.

[37] A. Ludwig, J. Marcinkowski, S. Schmid, Scheduling loop-free network updates: it's good to relax!, in: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, ACM, 2015, pp. 13–22.

[38] A. Ludwig, M. Rost, D. Foucard, S. Schmid, Good network updates for bad packets: waypoint enforcement beyond destination-based routing policies, in: Proc. 13th ACM Workshop on Hot Topics in Networks (HotNets), ACM, 2014, p. 15.

[39] R. Mahajan, R. Wattenhofer, On consistent updates in software defined networks, in: Proc. 12th ACM Workshop on Hot Topics in Networks (HotNets), ACM, 2013, p. 20.

[40] J. McClurg, H. Hojjat, P. Cerný, N. Foster, Efficient synthesis of network updates, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, 2015, pp. 196–207.

[41] J. McClurg, H. Hojjat, P. Černỳ, N. Foster, Efficient synthesis of network updates, in: ACM Sigplan Notices, ACM, 2015, pp. 196–207.

[42] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, Composing software defined networks, in: 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13), 2013, pp. 1–13.

[43] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October – 1 November 1977, IEEE Computer Society, 1977, pp. 46–57.

[44] S. Prabhu, K.Y. Chou, A. Kheradmand, B. Godfrey, M. Caesar, Plankton: scalable network configuration verification through model checking, in: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020, pp. 953–967.

[45] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker, Abstractions for network update, ACM SIGCOMM Comput. Commun. Rev. 42 (2012) 323–334.

[46] F. Somenzi, CUDD: CU Decision Diagram Package Release 3.0.0, University of Colorado at Boulder, 2015, http://vlsi.colorado.edu/~fabio/CUDD/.

[47] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, M. Vechev, Probabilistic verification of network configurations, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, 2020, pp. 750–764.
[48] S. Vissicchio, 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.
[49] Y. Wang, Z. Wang, Explicit routing algorithms for Internet traffic engineering, in: Proceedings Eight International Conference on Computer Communications and Networks (Cat. No. 99EX370), IEEE, 1999, pp. 582–588.
[50] J. Zerwas, P. Kalmbach, C. Fuerst, A. Ludwig, A. Blenk, W. Kellerer, S. Schmid, Ahab: data-driven virtual cluster hunting, in: Proc. IFIP Networking, 2018.
[51] Q. Zhang, V. Liu, H. Zeng, A. Krishnamurthy, High-resolution measurement of data center microbursts, in: Proceedings of the 2017 Internet Measurement Conference, 2017, pp. 78–85.
[52] W. Zhou, D. Jin, J. Croft, M. Caesar, P.B. Godfrey, Enforcing customizable consistency properties in software-defined networks, in: Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15), 2015, pp. 73–85.