AllSynth: Transiently Correct Network Update Synthesis Accounting for Operator Preferences

Kim Guldstrand Larsen[1], Anders Mariegaard[1], Stefan Schmid[2,3],
and Jiří Srba[1](✉)

[1] Aalborg University, Aalborg, Denmark
{kgl,am,srba}@cs.aau.dk
[2] TU Berlin, Berlin, Germany
[3] University of Vienna, Vienna, Austria
stefan_schmid@univie.ac.at

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 in the LTL logic), also during the reconfiguration process. In particular, in contrast to existing approaches, AllSynth symbolically computes and compactly represents *all* feasible solutions. At its heart, AllSynth relies on a novel, two-level and parameterized use of 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 often also in terms of runtime as documented by experiments on a benchmark of real-world network topologies.

1 Introduction

A more 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" [24]; many major Internet outages over the last years were caused by human errors [5,12,15]. 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 growing explosively but also features much temporal and spatial structure [4,6,48]; this introduces a significant potential to improve operational efficiency by rendering networks more adaptive towards the actual traffic patterns they serve.

Motivated by the vision of more automated networks [17], over the last years, great efforts were made in laying the foundations for automated network verification, and in designing synthesis tools [3,16,27,42,45]. Furthermore, motivated

Y. Aït-Ameur and F. Crăciun (Eds.): TASE 2022, LNCS 13299, pp. 344–362, 2022.
https://doi.org/10.1007/978-3-031-10363-6_23

by the benefits of more adaptive network operations, e.g., to improve availability and performance [28], automated tools for consistently updating network configurations have been developed [11,23,38,43,46] which overcome the limitations of existing hand-crafted algorithms [2,34,37]. However, the computation of provably consistent network update schedules remains challenging, due to the required performance and expressiveness. The performance requirements are multidimensional: network update schedules should not only be quickly computable but also account for operator preferences, like requiring that certain switches or routers are updated first. However, existing approaches only provide one update sequence that may not be preferred by the network operator.

Our Contributions. We present an automated network update synthesis tool, *AllSynth*, that computes and represents in a compact BDD form *all* correct update sequences that respect various logical properties expressible in linear temporal logic (LTL) [41] like reachability, waypointing and service chaining. *AllSynth* comes with formal correctness guarantees and for situations in which *provably* no simple update schedule exists, it can make suggestions for alternative solutions (where the same switch is updated multiple times).

Despite being more general, *AllSynth* significantly outperforms state-of-the-art tools in terms of runtime on all non-trivial real-world networks from the standard Topology Zoo benchmark [29]. The update synthesis problem solved by *AllSynth* is NP-hard, even if restricted to preserving the basic loop-freedom and waypointing properties [34]. To combat the complexity of the problem, *AllSynth* exploits a novel two-level use of binary decision diagrams (BDDs) [32] 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, 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 [31].

Related Work. Motivated by the benefits of adaptive and software-defined (i.e., programmable) communication networks [30], 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 [19].

In their seminal work, Reitblatt et al. [43] showed that a strong per-packet consistency can be achieved using packet versioning during reconfigurations. Their approach, which was subsequently studied intensively in the literature [8,10,20,25,26,33,40], 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 [37], 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 [2,14,21,34–36,47], however, it can only be used to provide a subset of the consistency properties of [43]. This in turn motivated hybrid approaches such as FLIP [46]. 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 light-weight and fully symbolic approach aiming at updating nodes multiple times.

The need for supporting more general or even customizable consistency properties [49] as well as more automated synthesis approaches [18,23,39] has already received attention in the literature as well. However, our approach is the first one that is using the BDD-based technology for the synthesis and representation of *all* correct network updates. The competing tool NetSynth [38] for update synthesis is relying on an incremental enumeration of candidates of update sequences that are then verified by external model checkers, like NuSMV [13], and the tool terminates as soon as the first correct update sequence is found.

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).

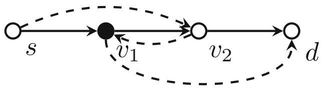


Fig. 1. Update synthesis problem

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 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.

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 multi-graph $G = (V, E, \mathsf{src}, \mathsf{tgt})$ 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, $0 \leq i \leq n$, where by convention $\mathsf{tgt}(e_{-1}) = v_0$ 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 $i \geq 0$ where as before $\mathsf{tgt}(e_{-1}) = v_0$. 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.

2.1 Routing Policies

We shall now define an LTL-based logic [41] that allows us to describe the policy of acceptable routings (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 freedom predicate. We now give the formal semantics of our logic, interpreted both on infinite and finite paths [22].

$$
\begin{aligned}
\mathsf{Reach}(d) &\equiv \mathsf{true}\, U\, d\\
\mathsf{Waypoint}(v,d) &\equiv \neg\mathsf{Reach}(d) \vee (\neg d\, U\, v \wedge \mathsf{Reach}(d))\\
\mathsf{MultiWaypoint}(W,d) &\equiv \bigvee_{v\in W} \mathsf{Waypoint}(v,d)\\
\mathsf{Service}(\omega,d) &\equiv \begin{cases} \mathsf{true} & \text{if } |\omega| = 0\\ \neg\mathsf{Reach}(d) \vee & \text{if } \omega = v \circ \omega'\\ \left(\bigwedge_{v'\in\omega'} \neg v' \wedge \neg d\right)\, U\, \left(v \wedge \mathsf{Service}(\omega',d)\right) & \text{where } v \in V \end{cases}
\end{aligned}
$$

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

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

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

We now formulate some standard routing policies as presented 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, $\mathsf{Service}(\omega, d)$ ensures that the sequence of waypoints in ω 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

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

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, meaning that each update changes the routing for a given node v from its initial routing $\rho_i(v)$ directly to its final routing $\rho_f(v)$.

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 $\rho_f(\mathsf{src}(u))$ 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 respect 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.

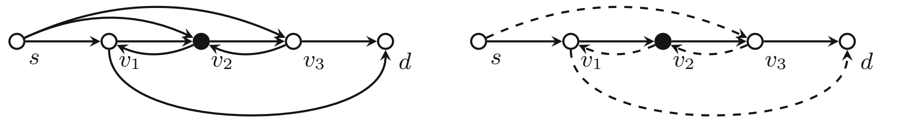
Definition 6 (Update Correctness). *An update sequence $w \in (E \cup V)^*$ on network topology G with initial routing configuration ρ is* correct *with respect to source node v_0 and a policy φ, if $\pi_{\rho^{w'}}(v_0) \models \varphi$ for any prefix w' of w.*

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 solutions 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). To see this, consider 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



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

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

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

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.

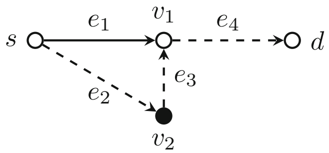


Fig. 4. Counter example for $\mathsf{Waypoint}(v_2, d)$; initial/final routing is in solid/dashed lines

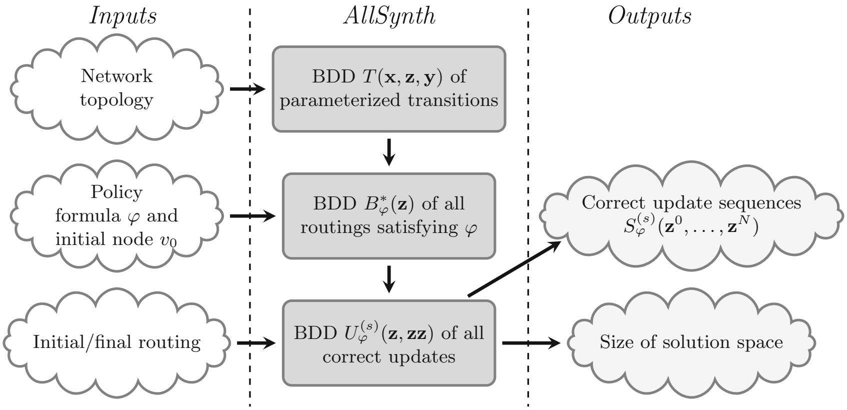

Inputs
AllSynth
Outputs
Network topology
BDD $T(\mathbf{x}, \mathbf{z}, \mathbf{y})$ of parameterized transitions
Policy formula φ and initial node v_0
BDD $B^*_\varphi(\mathbf{z})$ of all routings satisfying φ
Correct update sequences $S^{(s)}_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N)$
Initial/final routing
BDD $U^{(s)}_\varphi(\mathbf{z}, \mathbf{zz})$ of all correct updates
Size of solution space

Fig. 5. *AllSynth* workflow

The requirement in Lemma 1 that the policy must enforce at least the reachability of d is essential, as illustrated in Fig. 4 where $e_2 \circ e_3 \circ e_4$ is a correct update sequence preserving $\mathsf{Waypoint}(v_2, d)$. 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 $\mathsf{Waypoint}(v_2, d)$.

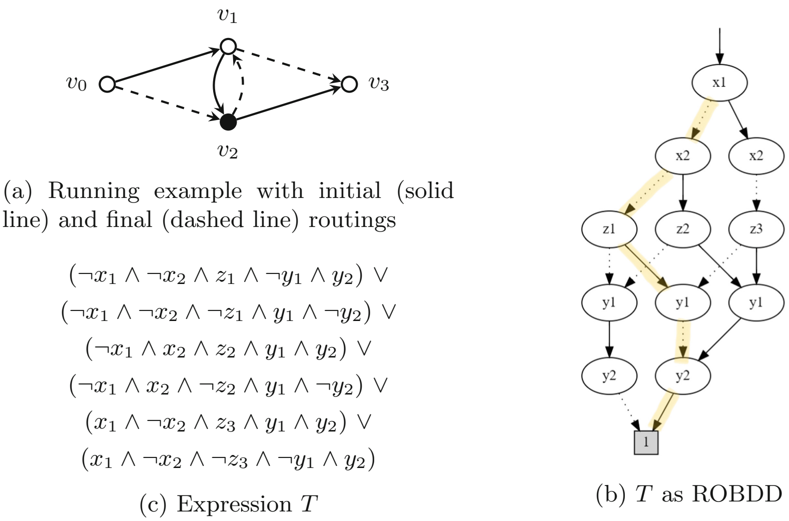
3 The AllSynth Tool and the Synthesis Algorithm

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 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.

We shall now present 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 [32] are data structures for the compact representation of a Boolean function. A BDD is a rooted directed acyclic graph (DAG), with nonleaf nodes labeled by Boolean variables, and leaf nodes labeled 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



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

$$
\begin{gathered}
(\neg x_1 \wedge \neg x_2 \wedge z_1 \wedge \neg y_1 \wedge y_2) \vee \\
(\neg x_1 \wedge \neg x_2 \wedge \neg z_1 \wedge y_1 \wedge \neg y_2) \vee \\
(\neg x_1 \wedge x_2 \wedge z_2 \wedge y_1 \wedge y_2) \vee \\
(\neg x_1 \wedge x_2 \wedge \neg z_2 \wedge y_1 \wedge \neg y_2) \vee \\
(x_1 \wedge \neg x_2 \wedge z_3 \wedge y_1 \wedge y_2) \vee \\
(x_1 \wedge \neg x_2 \wedge \neg z_3 \wedge \neg y_1 \wedge y_2)
\end{gathered}
$$

(c) Expression T

(b) T as ROBDD

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

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 [32] and later Bryant [9] 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.

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 $s_0, s_1, s_2, \dots s_{|S|-1}$ of S and it is based on $n = \lceil \log(|S|) \rceil$ Boolean variables $\mathbf{x} = x_1, x_2, \dots, x_n$. 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 $[\![t(\mathbf{x})]\!] = \{\, s_{n(\mu)} \mid \mu \text{ satisfies } t(\mathbf{x}) \,\} \subseteq S$.

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_v^d. 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 the possible transitions:

$$T(\mathbf{x}, \mathbf{z}_{v_0}, \ldots, \mathbf{z}_{v_k}, z_{v_0}^d, \ldots, z_{v_k}^d, \mathbf{y}) = \bigvee_{v \in V} \bigvee_{e \in E_v} \big(\mathbf{x}(v) \wedge \mathbf{z}_v(e) \wedge z_v^d \wedge \mathbf{y}(\mathsf{tgt}(e))\big)$$

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

Example 2. Reconsidering the network topology from Fig. 6a, we shall use three Boolean variables z_1, z_2, z_3 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 Example 1, the possible transitions between nodes are given by the Boolean expression T in Fig. 6c. The resulting unique ROBDD in Fig. 6b 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.

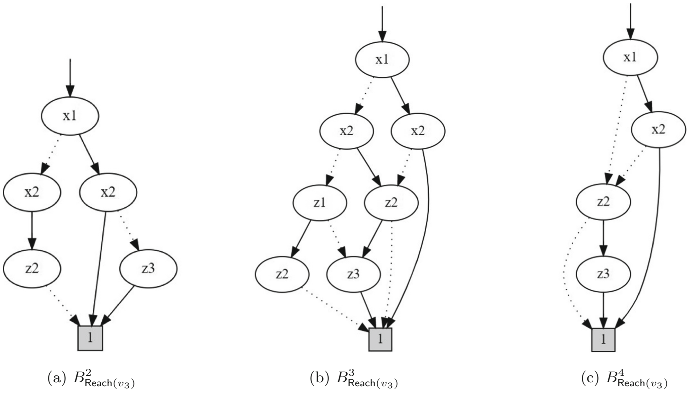
BDD Encoding of Routing Policies. Now let $G = (V, E, \mathsf{src}, \mathsf{tgt})$ be a network topology and let φ be a routing policy expressed in the LTL logic of Definition 3. Using Boolean variables $\mathbf{x}$ for encoding nodes and Boolean variables $\mathbf{z}$ for encoding routing configurations[2], 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 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:*

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

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

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



(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 $B^n_{\mathsf{Reach}(v_3)}$

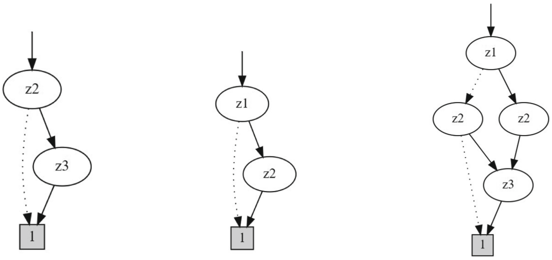
In the above definition we exploit that ROBDDs are closed under Boolean operations as well as Boolean quantification. In the cases NoLoop and $\varphi_1 \, U \, \varphi_2$, the changes of Boolean variables used in the parameter lists in the right-hand sides are obtained by simple substitution of variables, 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 minimal 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 $B^0_{\mathsf{NoLoop}} = 0$, and terminating when $B^{n+1}_{\mathsf{NoLoop}} = B^n_{\mathsf{NoLoop}}$.

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

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: $B^{n+1}_{\mathsf{Reach}(v_3)}(\mathbf{x}, \mathbf{z}) = \mathbf{x}(v_3) \vee \exists . \mathbf{y} . \big(T(\mathbf{x}, \mathbf{z}, \mathbf{y}) \wedge B^n_{\mathsf{Reach}(v_3)}(\mathbf{y}, \mathbf{z})\big)$ with $B^0_{\mathsf{Reach}(v_3)} = 0$. Figure 7 provides some of the approximants with $B^4_{\mathsf{Reach}(v_3)}$ found to be the least fixed point.

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.



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

Fig. 8. Encoding of different routing 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.

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

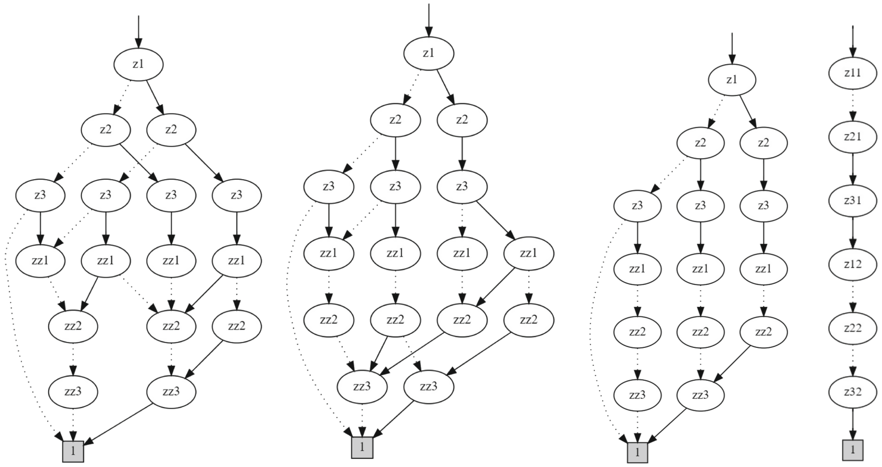
Note that in this simple update the routing configuration changes for exactly one node v_i from the setting in the initial configuration ρ_i, encoded as z_{v_i}, to the setting in final configuration ρ_f, encoded as $\neg zz_{v_i}$. In the general case, the update can change the setting of any node arbitrarily,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

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

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.*

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

$$R^s_\varphi(\mathbf{z}, \mathbf{zz}) \overset{\min}{=} \mathbf{z}(\rho_f) \vee \exists \mathbf{zzz}.\big(\, U^s_\varphi(\mathbf{z}, \mathbf{zz}) \wedge R^s_\varphi(\mathbf{zz}, \mathbf{zzz})\big) \qquad (1)$$



(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)

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 $\rho^w = \rho_f$.*

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:

$$S^s_\varphi(\mathbf{z}^0, \ldots, \mathbf{z}^N) = \mathbf{z}^0(\rho_i) \wedge \mathbf{z}^N(\rho_f) \wedge \bigwedge_{i=0}^{N-1} R^s_\varphi(\mathbf{z}^i, \mathbf{z}^{i+1}) \tag{2}$$

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 $\rho_N = \rho_f$.*

For the synthesis in the general case: simply replace U^s_φ in (1) with U_φ to get a ROBDD R_φ characterizing (general) update sequences leading to ρ_f. Now, replace R^s_φ with R_φ in (2) 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 Eq. (2) 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)[3]. Here the values suggested for the first three Boolean variables z_1^1, z_2^1, z_3^1 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_1^2, z_2^2, z_3^2 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 Implementation and Evaluation

Our tool *AllSynth* is implemented in Python and relies on a Cython wrapper [1] of the CUDD [44] package for manipulation of ROBDD. 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.

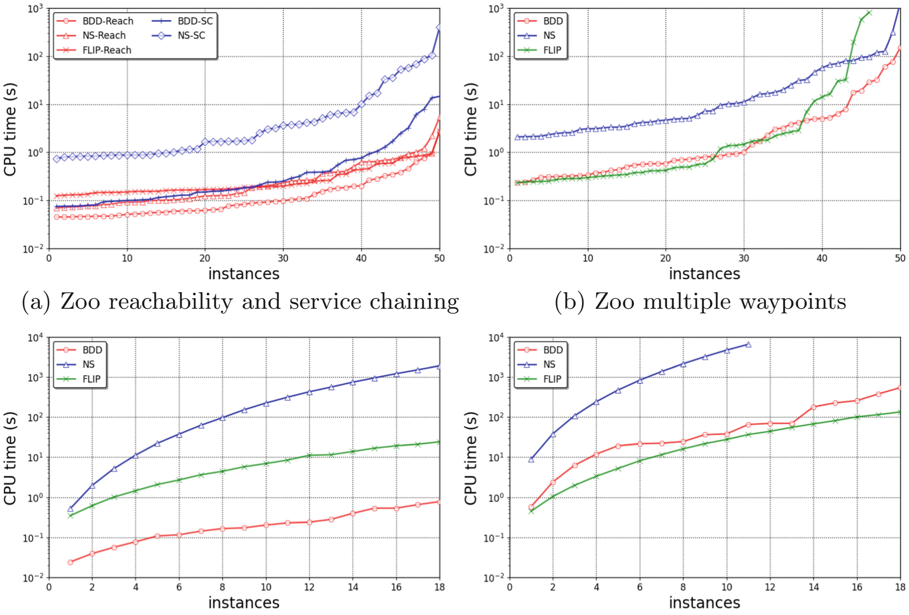
We evaluate *AllSynth* against two state-of-the-art update synthesis tools, NetSynth [38] and FLIP [46]. NetSynth can compute only a simple update sequence or inform the user that there is no solution; the synthesis of general update sequences is not supported. FLIP can synthesise sequences of steps (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, compare the running times only for simple updates.

All experiments are executed on Ubuntu 14.04 cluster with 2.3 GHz AMD Opteron 6376 processors with 2 h timeout and 14 GB memory limit. A reproducibility package is available in [31].

We consider a scalable synthetic topology and the standard benchmark of 261 real-world network topologies from the Topology Zoo dataset [29]. The class of synthetic topologies, referred to as *diamond* topologies, are overtaken from the NetSynth evaluation benchmark [38] 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 $5th$ node on both the

[3] Note that `zij` in the figure is to be read as the variable z_i^j.



(a) Zoo reachability and service chaining

(b) Zoo multiple waypoints

(c) Diamond reachability

(d) Diamond multiple waypoints

Fig. 10. Experimental results

initial and final path be included in W. For $\mathsf{Service}(\omega, d)$, the sequence ω is generated by including every $5th$ 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 it is able to directly express all these properties. On the other hand, the policy input to FLIP enumerates all admissible subpaths that are considered, in logical disjunction. The encoding of the 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* [7] in Fig. 10, 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. 10a and 10b, we notice that none of the tools has difficulty solving the synthesis of the plain reachability policy and it takes less than 10 s for all instances—here our approach has a slight margin. For waypointing, while FLIP is performing well on small instances, it shows a noticeable penalty once it reaches the most difficult problems where 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.

Results for diamond topologies are given in Figs. 10c and 10d. We observe that for reachability our computation of all solutions is almost one order of magnitude 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 we 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.

5 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 the LTL logic. 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.

Acknowledgements. The research is partly funded by the Vienna Science and Technology Fund (WWTF), project WHATIF (ICT19-045), the ERC Advanced Grant LASSO, the Villum Investigator Grant S4OS, DFF project QASNET as well as DIREC: Digital Research Centre Denmark.

References

1. dd python package (2021). https://github.com/tulip-control/dd
2. Akhoondian Amiri, S., Dudycz, S., Schmid, S., Wiederrecht, S.: Congestion-free rerouting of flows on DAGs. In: 45th International Colloquium on Automata, Languages, and Programming (ICALP), vol. 107, pp. 143:1–143:13. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
3. Anderson, C.J., et al.: NetKAT: semantic foundations for networks. ACM SIGPLAN Notices **49**(1), 113–126 (2014)
4. Avin, C., Ghobadi, M., Griner, C., Schmid, S.: On the complexity of traffic traces and implications. In: Proceedings of the ACM SIGMETRICS (2020)
5. Beckett, R., Mahajan, R., Millstein, T., Padhye, J., Walker, D.: Don't mind the gap: bridging network-wide objectives and device-level configurations. In: Proceedings of the 2016 ACM SIGCOMM Conference, pp. 328–341 (2016)
6. Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, pp. 267–280 (2010)
7. Brain, M.N., Davenport, J.H., Griggio, A.: 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 2017). CEUR, vol. 1974, pp. 1–15. CEUR-WS.org (2017)
8. Brandt, S., Förster, K.T., Wattenhofer, R.: On consistent migration of flows in SDNs. In: IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, pp. 1–9. IEEE (2016)
9. Bryant: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **C3-5**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
10. Canini, M., Kuznetsov, P., Levin, D., Schmid, S.: A distributed and robust SDN control plane for transactional network updates. In: 2015 IEEE Conference on Computer Communications (INFOCOM), pp. 190–198. IEEE (2015)
11. Černý, P., Foster, N., Jagnik, N., McClurg, J.: Optimal consistent network updates in polynomial time. In: Gavoille, C., Ilcinkas, D. (eds.) DISC 2016. LNCS, vol. 9888, pp. 114–128. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53426-7_9
12. Chirgwin, R.: Google routing blunder sent Japan's internet dark on friday (2017). https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/
13. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: a new symbolic model checker. Int. J. Softw. Tools Technol. Transf. **2**(4), 410–425 (2000). https://doi.org/10.1007/s100090050046
14. Dudycz, S., Ludwig, A., Schmid, S.: Can't touch this: consistent network updates for multiple policies. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 133–143. IEEE (2016)
15. Duluth News Tribune: Human error to blame in minnesota 911 outage (2018). https://www.ems1.com/911/articles/389343048-Officials-Human-error-to-blame-in-Minn-911-outage/
16. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.: Netcomplete: practical network-wide configuration synthesis with autocompletion. In: 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 2018), pp. 579–594 (2018)

17. Feamster, N., Rexford, J.: Why (and how) networks should run themselves. arXiv report (2017)
18. Finkbeiner, B., Gieseking, M., Hecking-Harbusch, J., Olderog, E.R.: Model checking data flows in concurrent network updates (full version). arXiv preprint arXiv:1907.11061 (2019)
19. Foerster, K., Schmid, S., Vissicchio, S.: Survey of consistent software-defined network updates. IEEE Commun. Surv. Tutor. **21**(2), 1435–1461 (2019)
20. Foerster, K.T.: On the consistent migration of unsplittable flows: upper and lower complexity bounds. In: 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA), pp. 1–4. IEEE (2017)
21. Foerster, K.T., Luedi, T., Seidel, J., Wattenhofer, R.: Local checkability, no strings attached:(a) cyclicity, reachability, loop free updates in SDNs. Theoret. Comput. Sci. **709**, 48–63 (2018)
22. Giacomo, G.D., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013), pp. 854–860. AAAI Press (2013)
23. Glavind, M., Christensen, N., Srba, J., Schmid, S.: Latte: improving the latency of transiently consistent network update schedules. In: Proceedings of 38th International Symposium on Computer Performance, Modeling, Measurements and Evaluation (PERFORMANCE) (2020)
24. Heller, B., et al.: Leveraging SDN layering to systematically troubleshoot networks. In: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, pp. 37–42 (2013)
25. Jin, X., et al.: Dynamic scheduling of network updates. In: ACM SIGCOMM Computer Communication Review, vol. 44, no. 4, pp. 539–550. ACM (2014)
26. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real time network policy checking using header space analysis. In: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013), pp. 99–111 (2013)
27. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: static checking for networks. In: Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 2012), pp. 113–126 (2012)
28. Kellerer, W., Kalmbach, P., Blenk, A., Basta, A., Reisslein, M., Schmid, S.: Adaptable and data-driven softwarized networks: review, opportunities, and challenges. In: Proceedings of the IEEE (PIEEE) (2019)
29. Knight, S., Nguyen, H.X., Falkner, N., Bowden, R.A., Roughan, M.: The internet topology zoo. IEEE J. Sel. Areas Commun. **29**(9), 1765–1775 (2011). https://doi.org/10.1109/JSAC.2011.111002
30. Kreutz, D., Ramos, F.M., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: a comprehensive survey. Proc. IEEE **103**(1), 14–76 (2014)
31. Larsen, K., Mariegaard, A., Schmid, S., Srba, J.: Reproducibility package for: the hazard value: a quantitative network connectivy measure accounting for failures, March 2022. https://doi.org/10.5281/zenodo.6534948
32. Lee, C.Y.: Representation of switching circuits by binary-decision programs. The Bell Syst. Tech. J. **38**(4), 985–999 (1959). https://doi.org/10.1002/j.1538-7305.1959.tb01585.x
33. Liu, H.H., Wu, X., Zhang, M., Yuan, L., Wattenhofer, R., Maltz, D.: zUpdate: updating data center networks with zero loss. In: ACM SIGCOMM Computer Communication Review, vol. 43, no. 4, pp. 411–422. ACM (2013)

34. Ludwig, A., Dudycz, S., Rost, M., Schmid, S.: Transiently secure network updates. ACM SIGMETRICS Perform. Eval. Rev. **44**(1), 273–284 (2016)
35. Ludwig, A., Marcinkowski, J., Schmid, S.: Scheduling loop-free network updates: it's good to relax! In: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, pp. 13–22. ACM (2015)
36. Ludwig, A., Rost, M., Foucard, D., Schmid, S.: Good network updates for bad packets: waypoint enforcement beyond destination-based routing policies. In: Proceedings of 13th ACM Workshop on Hot Topics in Networks (HotNets), p. 15. ACM (2014)
37. Mahajan, R., Wattenhofer, R.: On consistent updates in software defined networks. In: Proceedings of 12th ACM Workshop on Hot Topics in Networks (HotNets), p. 20. ACM (2013)
38. McClurg, J., Hojjat, H., Cerný, P., Foster, N.: Efficient synthesis of network updates. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 196–207 (2015). https://doi.org/10.1145/2737924.2737980
39. McClurg, J., Hojjat, H., Černỳ, P., Foster, N.: Efficient synthesis of network updates. In: ACM SIGPLAN Notices, vol. 50, no. 6, pp. 196–207. ACM (2015)
40. Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D.: Composing software defined networks. In: 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013), pp. 1–13 (2013)
41. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
42. Prabhu, S., Chou, K.Y., Kheradmand, A., Godfrey, B., Caesar, M.: Plankton: scalable network configuration verification through model checking. In: 17th USENIX Symposium on Networked Systems Design and Implementation ({NSDI} 2020), pp. 953–967 (2020)
43. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D.: Abstractions for network update. ACM SIGCOMM Comput. Commun. Rev. **42**(4), 323–334 (2012)
44. Somenzi, F.: CUDD: CU decision diagram package release 3.0.0. University of Colorado at Boulder (2015). http://vlsi.colorado.edu/~fabio/CUDD/
45. Steffen, S., Gehr, T., Tsankov, P., Vanbever, L., Vechev, M.: 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, pp. 750–764 (2020)
46. Vissicchio, S., Cittadini, L.: 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, 10–14 April 2016, pp. 1–9 (2016). https://doi.org/10.1109/INFOCOM.2016.7524419
47. Zerwas, J., et al.: AHAB: data-driven virtual cluster hunting. In: Proceedings of IFIP Networking (2018)
48. Zhang, Q., Liu, V., Zeng, H., Krishnamurthy, A.: High-resolution measurement of data center microbursts. In: Proceedings of the 2017 Internet Measurement Conference, pp. 78–85 (2017)
49. Zhou, W., Jin, D., Croft, J., Caesar, M., Godfrey, P.B.: Enforcing customizable consistency properties in software-defined networks. In: Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015), pp. 73–85 (2015)