AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks

Peter Gjøl Jensen
Aalborg University
Denmark

Dan Kristiansen
Aalborg University
Denmark

Stefan Schmid
Faculty of Computer Science
University of Vienna
Austria

Morten Konggaard Schou
Aalborg University
Denmark

Bernhard Clemens Schrenk
Faculty of Computer Science
University of Vienna
Austria

Jiří Srba
Aalborg University
Denmark

ABSTRACT

We present an automated what-if analysis tool AalWiNes for MPLS networks which allows us to verify both logical properties (e.g., related to the policy compliance) as well as quantitative properties (e.g., concerning the latency) under multiple link failures. Our tool relies on weighted pushdown automata, a quantitative extension of classic automata theory, and takes into account the actual dataplane configuration, rendering it especially useful for debugging. In particular, our tool collects the different router forwarding tables and then builds a pushdown system, on which quantitative reachability is performed based on an expressive query language. Our experiments show that our tool outperforms state-of-the-art approaches (which until now have been restricted to logical properties) by several orders of magnitude; furthermore, our quantitative extension only entails a moderate overhead in terms of runtime. The tool comes with a platform-independent user interface and is publicly available as open-source, together with all other experimental artefacts.

CCS CONCEPTS

• **Networks** → Network reliability; **Network algorithms**.

KEYWORDS

Network Verification, Network Performance, Tools

ACM Reference Format:
Peter Gjøl Jensen, Dan Kristiansen, Stefan Schmid, Morten Konggaard Schou, Bernhard Clemens Schrenk, and Jiří Srba. 2020. AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3386367.3431308

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12.
https://doi.org/10.1145/3386367.3431308

1 INTRODUCTION

While communication networks are a critical infrastructure of our digital society, their correct configuration and operation is complex, requiring operators to become "*masters of complexity*" [21]. As many recent network outages were caused by human errors, e.g., [8, 13, 14], we currently witness major research efforts toward more automated and formal approaches to operate and verify networks [5, 7, 15, 16, 24, 25, 29, 34, 39].

A particularly critical but challenging task for human operators is to *reason about failures* (in this paper referred to as *what-if analysis*). In order to meet their stringent dependability requirements, most modern communication networks come with fast recovery mechanisms which revert traffic to alternative paths [11, 12, 28, 31]. While this is attractive, already a single link failure can lead to unintended network behaviors which are easily overlooked and may violate the network policy [8]. Especially multiple link failures, which are more likely to occur in large networks and can be caused e.g. due to shared risk link groups [6, 17, 30], may threaten network dependability.

It is often insufficient to only ensure the logical correctness (e.g., policy compliance) of the network behavior under failures. A dependable network also needs to satisfy quantitative properties. For example, traffic should be rerouted along *short* paths, e.g., regarding link latency (offering a low latency) or number of hops (reducing load), even under a certain number of link failures.

We are particularly interested in networks based on Multiprotocol Label Switching (MPLS) [2]. MPLS networks are widely deployed in the Internet today, especially in IP networks and for traffic engineering purposes. The study of MPLS networks is also interesting from a theoretical perspective, as the header size in these networks is not fixed; rather, additional labels may be pushed on the header while rerouting packets around failed links, creating "tunnels". This makes the employment of formal methods particularly challenging as we must deal with a possibly unbounded set of packet headers.

Our Contributions. We present a what-if analysis tool for MPLS networks, AalWiNes[1], which supports a fully automated and fast verification of the network behavior under failures. In particular, AalWiNes relies on an expressive query language and allows us to test both logical properties (such as the policy compliance) as

[1] AALborg WIen NEtwork verification Suit

well as quantitative properties (such as the latency, number of hops, required label stack size resp. tunneling depth, or number of failed links), and in polynomial-time using an over- and under-approximation technique for an arbitrary number of link failures (and with a low number of inconclusive answers). AalWiNes operates directly on the dataplane forwarding tables, allowing to debug issues not visible in the control plane.

At the heart of AalWiNes lies a *weighted* pushdown automaton, a quantitative generalization of classical automata: based on the router forwarding tables and a query (the input to the tool), we build a weighted pushdown system and then perform a quantitative reachability analysis. To improve performance, AalWiNes uses novel algorithms tailored to this use case.

We offer an optimized C++ implementation of AalWiNes and report on its platform-independent user interface. Considering a case study in cooperation with NORDUnet, a major network operator, we show that our tool outperforms state-of-the-art approaches (only applicable to verification of logical properties) by several orders of magnitude in terms of runtime. We also demonstrate that our quantitative extension only entails a reasonable overhead. As a contribution to the research community and in order to ensure reproducibility, we released our tool as open source and we also shared all our experimental artefacts [23].

Related work and novelty. The problem of how to render networks more automated and formally verifiable has recently received much interest, both for specific networks and protocols, such as BGP [38], OpenFlow [5, 25], or MPLS [34] networks, as well as for networks which are protocol agnostic [24]. The different systems rely on different approaches, including e.g., algebraic approaches [5], static verification based on geometric approaches [24], or automata-theoretic approaches [22]. We specifically consider MPLS networks and use an automata-theoretic approach. Whereas some recent work focus on verification of the control plane configurations [4, 7, 18–20, 32], we directly verify the router forwarding tables, which allows us to also catch errors in the data plane [36].

We focus on *polynomial-time* verification via a suitable over- and under-approximation, even under failures: many existing approaches in the literature do not consider failure scenarios explicitly (and still exhibit a super-polynomial runtime, e.g., due to SAT solving [29]), and/or have to solve NP-hard problems when modelling failures scenarios, e.g. SMT solving [7]. Furthermore, most existing approaches target different network types [4, 7, 19, 20, 32] and do not support arbitrary header sizes, which however arise in the context of MPLS networks: by representing MPLS networks symbolically as pushdown automata, we hence achieve an exponential speedup compared to the direct encoding of all possible sequences of header symbols.

To the best of our knowledge, our tool is the first to consider what-if analysis of quantitative aspects as well, and we are not aware of any applications of weighted automata theoretical results in this context. The few weighted solutions that exist do not consider failure scenarios and have an exponential runtime [27]. The paper closest to ours is P-Rex [22], a polynomial-time approach to verify logical properties of MPLS networks, also accounting for possible failures by using approximative analysis. As we demonstrate in our evaluation, our tool not only adds the novel quantitative dimension, but also outperforms P-Rex by several orders of magnitude. We further contribute an interactive user interface and make a leap forward regarding applicability for network operators.

Finally, we note that what-if analyses tools have been developed also various other networking contexts, such as in CDNs, to predict the effects of possible configuration and deployment changes [37].

2 MPLS NETWORK MODEL

This section introduces our MPLS model and query language.

2.1 Network definition

An MPLS network consists of a topology and forwarding rules.

Definition 1. A *network topology* is a directed multigraph (V, E, s, t) where V is a set of *routers*, E is a set of *links* between routers, $s : E \to V$ assigns the *source router* to each link, and $t : E \to V$ assigns the *target router*.

We assume that links in the network can fail. This is modelled by a set $F \subseteq E$ of *failed* links. A link is *active* if it belongs to $E \setminus F$. We assume asymmetric link failures that can be caused e.g., by congestion in one direction, resulting in packet drops that can also appear as a link failure.

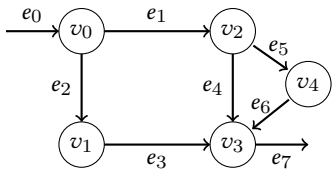
Let L be a nonempty set of MPLS labels used in packet headers. We define the set of MPLS operations on packet headers as $Op = \{\texttt{swap}(\ell) \mid \ell \in L\} \cup \{\texttt{push}(\ell) \mid \ell \in L\} \cup \{\texttt{pop}\}$. Given an alphabet A, let A^* denote the set of all finite words over the elements of A, including the empty word ϵ.

Definition 2. An *MPLS network* is a tuple $N = (V, E, s, t, L, \tau)$ where (V, E, s, t) is a network topology, $L = L_M \uplus L_M^{\perp} \uplus L_{IP}$ is a finite set of labels partitioned into (1) the MPLS label set L_M, (2) the set of MPLS labels with the bottom of the stack bit (S) set to true $L_M^{\perp}$, (3) a set of IP addresses L_{IP}, and $\tau : E \times L \to (2^{E \times Op^*})^*$ is the routing table.

The routing table, for every link $e \in E$ and a top (left-most) packet label ℓ, returns a sequence of traffic engineering groups $\tau(e, \ell) = O_1 O_2 \ldots O_n$ where each traffic engineering group is a set of the form $\{(e_1, \omega_1), \ldots, (e_m, \omega_m)\}$ where e_j is the outgoing link such that $t(e) = s(e_j)$ and $\omega_j \in Op^*$ is a sequence of operations to be performed on the packet header. In a given traffic engineering group, the router can nondeterministically (e.g. pseudorandomly) select any active link and forward the packet via that link while applying the corresponding sequence of MPLS operations. This allows us to abstract away from various routing policies that facilitate e.g. splitting of a flow along multiple shortest paths. The group O_i has a higher priority than O_{i+1} and during the forwarding, and the router always selects the traffic engineering group with the highest priority and at least one active link.

2.2 Valid MPLS headers

The MPLS labels can be nested only in a specific way. For a given network $N = (V, E, s, t, L, \tau)$, we define the set of valid headers $H \subseteq L^*$ by $H = L_{IP} \cup \{\alpha \ell_1 \ell_0 \mid \alpha \in L_M^*, \ \ell_1 \in L_M^{\perp}, \ \ell_0 \in L_{IP}\}$. Hence the on top of the IP label there can be one label with the



(a) Network topology

Router	e_{in}	Label	Priority	e_{out}	Operation
v_0	e_0	ip_1	1	e_1	push($s20$)
	e_0	ip_1	1	e_2	push($s10$)
	e_0	$s40$	1	e_1	swap($s41$)
v_1	e_2	$s10$	1	e_3	swap($s11$)
v_2	e_1	$s20$	1	e_4	swap($s21$)
	e_1	$s41$	1	e_5	swap($s42$)
	e_1	$s20$	2	e_5	swap($s21$) ∘ push(30)
v_3	e_3	$s11$	1	e_7	pop
	e_4	$s21$	1	e_7	pop
	e_6	$s43$	1	e_7	swap($s44$)
	e_6	$s21$	1	e_7	pop
v_4	e_5	30	1	e_6	pop
	e_5	$s42$	1	e_6	swap($s43$)

(b) Routing table

$\sigma_0 = (e_0, ip_1)\,(e_1, s20 \circ ip_1)\,(e_4, s21 \circ ip_1)\,(e_7, ip_1)$
$\sigma_1 = (e_0, ip_1)\,(e_2, s10 \circ ip_1)\,(e_3, s11 \circ ip_1)\,(e_7, ip_1)$
$\sigma_2 = (e_0, ip_1)\,(e_1, s20 \circ ip_1)\,(e_5, 30 \circ s21 \circ ip_1)(e_6, s21 \circ ip_1)\,(e_7, ip_1)$
$\sigma_3 = (e_0, s40 \circ ip_1)\,(e_1, s41 \circ ip_1)\,(e_5, s42 \circ ip_1)\,(e_6, s43 \circ ip_1)\,(e_7, s44 \circ ip_1)$

(c) Examples of traces

Query	Witness traces
$\varphi_0 =$ ⟨ip⟩ [.#v_0] .* [v_3#.] ⟨ip⟩ 0	σ_0, σ_1
$\varphi_1 =$ ⟨ip⟩ [.#v_0] [^ v_2#v_3]* [v_3#.] ⟨ip⟩ 2	σ_1, σ_2
$\varphi_2 =$ ⟨$s40$ ip⟩ [.#v_0] .*[v_3#.] ⟨smpls ip⟩ 0	σ_3
$\varphi_3 =$ ⟨$s40$ ip⟩ [.#v_0] .*[v_3#.] ⟨mpls+ smpls ip⟩ 1	no trace exists
$\varphi_4 =$ ⟨smpls? ip⟩ [.#v_0] . . . .*[v_3#.] ⟨smpls? ip⟩ 1	σ_2, σ_3

(d) Network queries

Figure 1: A small network example

bottom of stack bit S set to true and an arbitrary number of other MPLS labels. The MPLS operations can manipulate the label-stack by modifying only the topmost label so that the result of operations performed on a valid header is itself a valid header.

Definition 3. The semantics of MPLS operations is a partial *header rewrite function* $\mathcal{H} : H \times Op^* \hookrightarrow H$ where $\omega, \omega' \in Op^*, \ell \in L, h \in H$ and ϵ is the empty sequence of operations:

$$\mathcal{H}(\ell h, \omega) = \begin{cases} \ell h & \text{if } \omega = \epsilon \text{ and } \ell \in L \\ \mathcal{H}(\ell' h, \omega') & \text{if } \omega = \texttt{swap}(\ell') \circ \omega' \text{ and } \ell' h \in H \\ \mathcal{H}(\ell' \ell h, \omega') & \text{if } \omega = \texttt{push}(\ell') \circ \omega' \text{ and } \ell' \ell h \in H \\ \mathcal{H}(h, \omega') & \text{if } \omega = \texttt{pop} \circ \omega' \text{ and } \ell \in L_M \cup L_M^{\perp} \\ \textit{undefined} & \textit{otherwise}\,. \end{cases}$$

Let $L_M = \{30, 31\}$, $L_M^{\perp} = \{s20, s21\}$ and $L_{IP} = \{ip_1\}$. We use here and in what follows the convention that all labels that are on the bottom of the MPLS label stack (have the bottom of stack bit S set to true) are prefixed with small s. Then $\mathcal{H}(30 \circ s20 \circ ip_1,\ \texttt{pop} \circ \texttt{swap}(s21) \circ \texttt{push}(31)) = 31 \circ s21 \circ ip_1$.

2.3 Example network

An example of a simple network topology is given in Figure 1a together with the forwarding table described in Figure 1b. The example defines two label switching paths for IP-packet routing from v_0 to v_3, either via the links e_1 and e_4, or the links e_2 and e_3. The respective path can be selected nondeterministically. Moreover, packets arriving via the link e_0 with the service label $s40$ (agreement with the neighboring network operator) are routed via the links e_1, e_5, e_6 and leave the network on the link e_7.

Every forwarding rule for the router v is represented by a line in the table and depending on the incoming link e_{in} (where $t(e_{in}) = v$) and the top of the stack label, it determines the outgoing link e_{out} (where $s(e_{out}) = v$) and a sequence of stack operations that replace the top label. Each such rule has a priority that is depicted by the priority column in the middle of the table. In our example, it is only the router v_2 that has more than one priority group associated to its forwarding table in order to protect the link e_4. If a packet arrives via the link e_1 with label $s20$ on top of the stack then it is primarily forwarded via the link e_4 while the label is swapped with $s21$. Only if the link e_4 fails, a backup rule with priority 2 is used so that it forwards the traffic via the link e_5, first swapping the top label with $s21$ and then pushing a new label 30 on top of the label stack. The router v_4 then pops the label and the packet arrives to v_3 with the same label as if the link e_4 did not fail.

2.4 Network traces

We now define valid traces in an MPLS network $N = (V, E, s, t, L, \tau)$ under the assumption that the links in the set $F \subseteq E$ failed. For a traffic engineering group $O = \{(e_1, \omega_1), (e_2, \omega_2), \ldots, (e_m, \omega_m)\}$ we let $E(O) = \{e_1, e_2, \ldots, e_m\}$ denote the set of all links in the group. The group O is *active* if it contains at least one active link, i.e. $E(O) \setminus F \neq \emptyset$. Further, we define $\mathcal{A}(O_1 O_2 \ldots O_n) = \{(e, \omega) \in O_j \mid e \text{ is an active link}\}$ where j is the lowest index such that O_j is an active traffic engineering group, and we let $\mathcal{A}(O_1 O_2 \ldots O_n) = \emptyset$ if no such j exists. The set $\mathcal{A}(\tau(e, \ell))$ so contains all the currently available output links and the corresponding label-stack operations to be performed on a packet arriving on the link e with the top-most label ℓ. A trace in a network is a routing of a packet in the network and consists of a sequence of active links together with the corresponding label-stack headers.

Definition 4. A *trace* in a network $N = (V, E, s, t, L, \tau)$ with a set of failed links $F \subseteq E$ is any finite sequence

$$(e_1, h_1)(e_2, h_2) \ldots (e_n, h_n) \in \big((E \setminus F) \times H\big)^*$$

of link-header pairs where $h_{i+1} = \mathcal{H}(h_i, \omega)$ for some $(e_{i+1}, \omega) \in \mathcal{A}(\tau(e_i, head(h_i)))$ for all i, $1 \leq i < n$, where $head(h_i)$ is the top (left-most) label of h_i.

Examples of network traces for our running example are provided in Figure 1c. The traces σ_0 and σ_1 describe two possible traces for routing a packet arriving to v_0 with the destination IP

ip_1, under the assumption that $F = \emptyset$. The trace σ_2 shows a failover protection of the link between v_2 and v_3 in case that $F = \{e_4\}$. Finally, the trace σ_4 encodes a label switching path for packets arriving to v_0 with the service label $s40$ and it is a valid trace for example for the set of failed links $F = \{e_2, e_3\}$.

2.5 Query language

We present a powerful query language that allows us to specify regular trace properties, both regarding the initial and final label-stacks as well as the link sequence in the trace.

Definition 5. A reachability *query* for an MPLS network $N = (V, E, s, t, L, \tau)$ is of the form $\langle a \rangle\ b\ \langle c \rangle\ k$ where a and c are regular expressions over the set of labels L, b is a regular expression over the set of links E, and $k \geq 0$ specifies the maximum number of failed links to be considered.

We assume here a standard syntax for regular expressions and by $Lang(a)$, $Lang(b)$ and $Lang(c)$ we understand the regular language defined by the expressions a, b and c, respectively. For specifying labels in the regular expressions a and c we use the abbreviations:

- `ip` = $[ip_0, \ldots, ip_n]$ where $L_{IP} = \{ip_0, \ldots, ip_n\}$,
- `mpls` = $[\ell_0, \ldots, \ell_n]$ where $L_M = \{\ell_0, \ldots, \ell_n\}$, and
- `smpls` = $[\ell_0^\perp, \ldots, \ell_n^\perp]$ where $L_M^\perp = \{\ell_0^\perp, \ldots, \ell_n^\perp\}$.

We further use the following notation for specifying links in the network. If v and u are routers, then $[v\#u]$ matches any link e from v to u such that $s(e) = v$ and $t(e) = u$. If in_1 is an interface on router v that uniquely identifies the outgoing link e, and in_2 identifies the incoming interface on router u for the link e, then $[v.in_1\#u.in_2]$ matches exactly the link e. The dot-syntax is used to denote any link in the network and it is extended to match also any router so that $[v\#\cdot] = \bigcup_{u \in V}[v\#u]$ and $[\cdot\#u] = \bigcup_{v \in V}[v\#u]$.

Problem 1 (Query Satisfiability Problem)**.** Given an MPLS network N and a query $\varphi = \langle a \rangle\ b\ \langle c \rangle\ k$, decide if there exists a trace $\sigma = (e_1, h_1) \ldots (e_n, h_n)$ in the network N for some set of failed links F such that $|F| \leq k$ where $h_1 \in Lang(a)$, $e_1 \ldots e_n \in Lang(b)$, and $h_n \in Lang(c)$. If this is the case, the query φ is *satisfied* and we call σ a *witness trace*.

Examples of queries are provided in Figure 1d. The query φ_0 asks about the existence of a trace that starts and ends with the label-stack containing only the IP header, such that the first link is incoming to the router v_0, followed by zero or more hops via unspecified links, and ending with link that leaves the router v_3, all under the assumption of no link failures. The traces σ_0 and σ_1 satisfy the query, however, even though the trace σ_2 has the required form as well, it does not satisfy φ_0 as it requires that the link e_4 fails. The next query φ_1 expresses a similar property as φ_0 with the exception that we allow for up to 2 link failures and the inner path may not contain any link between v_2 and v_3 (the symbol $\hat{}$ stands for complement of regular expressions). The traces σ_1 and σ_2 satisfy this query. The query φ_2 asks about a possible routing path between v_0 and v_3 where the header of the initial packet contains the label $s40$ on top of an IP header and leaves the network with an arbitrary MPLS label (where the bottom of the stack bit is set to true) on top of the IP header. Indeed, the trace σ_3 has this property and it is a valid trace even in case of no link failures. The next query φ_3 checks the transparency of the routing from v_0 to v_3 by asking whether a packet with the service label $s40$ can leave our network with at least one additional MPLS label on top of the service label. Should this be the case then our network leaks internal MPLS labels to the neighboring networks, which is not desirable. Even in case of one link failure, the query is not satisfied. Finally, the query φ_4 asks whether in case of one link failure there is an IP routing, with an optional MPLS label on the top of the IP label, with three or more hops between the incoming and outgoing links, and this is indeed the case as documented by the witness traces σ_2 and σ_3. In case of no link failures, the query is satisfied only by the trace σ_3.

3 QUANTITATIVE EXTENSION

After describing our network model and the query language used in our tool, we now present a novel extension of the framework which allows us to account for quantitative aspects, like latency, number of hops, tunnels (label stack size), number of failures, and linear combinations of these measures.

For a given network query, there can be several network traces that satisfy the query and for some queries there exist even infinitely many witness traces. From the user perspective, it is hence essential that when debugging the reasons why a certain query holds, we can impose quantitative constrains on the traces and specify what kind of witness traces we wish to visualize. For traffic engineering purposes we may want to find a trace that has the lowest latency or the smallest number of hops. We may be interested in finding a trace that minimizes tunneling depth or the number of failed links required to execute a given trace, or we may wish to find a trace that balances several such measures simultaneously.

We shall start by defining atomic quantiative properties of network traces. Let $N = (V, E, s, t, L, \tau)$ be an MPLS network and let $F \subseteq E$ be the set of failed links. An *atomic quantity* is a function $p : ((E \setminus F) \times H)^* \to \mathbb{N}_0$ that for a given trace σ evaluates to a non-negative integer $p(\sigma)$ representing the quantitative measure of the trace. In our tool, we support the following atomic quantities of a network trace $\sigma = (e_1, h_1) \ldots (e_n, h_n)$:

- $Links(\sigma) = n$ is the length of the trace,
- $Hops(\sigma) = |\{e \in \{e_1, \ldots, e_n\} \mid s(e) \neq t(e)\}|$ is the number of hops, where we avoid counting links that are self-loops,
- $Distance(\sigma) = \sum_{i=1}^{n} d(e_i)$ is the distance for any distance function $d : E \to \mathbb{N}_0$, e.g., the geographical distance, latency or e.g. inverse bandwidth capacity,
- $Failures(\sigma) = \sum_{i=1}^{n-1} |failed(i)|$ where $failed(i) = \{e \mid (e, \omega) \in O_k,\ 1 \leq k < j\}$, where $\tau(e_i, head(h_i)) = O_1 O_2 \ldots O_m$, and where j, is the lowest index such that O_j is an active traffic engineering group, and
- $Tunnels(\sigma) = \sum_{i=1}^{n-1} \max(0,\ |h_{i+1}| - |h_i|)$ is the number of pushes of new MPLS labels on the existing label-stack.

The atomic quantity $Failures(\sigma)$ measures the minimal number of failed links which are necessary at each router in order to enable the feasibility of the trace σ. The function $Tunnels(\sigma)$ measures the positive increase in the label-stack height during the trace σ that corresponds to the number of tunnels created during the trace.

Consider again the traces for our running example from Figure 1c. We have $Hops(\sigma_0) = Links(\sigma_0) = 4$ and $Hops(\sigma_3) = Links(\sigma_3) =$

5. We also observe that $Failures(\sigma_2) = 1$ while $Failures(\sigma_3) = 0$. Finally, we can see that e.g. $Tunnels(\sigma_1) = 1$, $Tunnels(\sigma_2) = 2$ and $Tunnels(\sigma_3) = 0$.

We can now combine the atomic quantities in order to define composed criteria for trace weight specification, by constructing linear expressions of the form

$$expr ::= p \mid a * expr \mid expr_1 + expr_2$$

where p is an atomic quantity and $a \in \mathbb{N}$. A vector of linear expressions $(expr_1, expr_2, \ldots, expr_n)$ allows us to specify trace properties by priorities, so that $expr_1$ has a higher priority than $expr_2$ etc. For a trace σ, there is a natural evaluation of linear expressions to nonnegative integers and for a vector of linear expressions, we assume the lexicographical ordering $\sqsubseteq$ on vectors of nonnegative integers, by abuse of notation extended to traces.

Problem 2 (Minimum Witness Problem)**.** For a network, a query that is satisfied in the network and a vector of linear expressions $(expr_1, expr_2, \ldots, expr_n)$, we want to find a witness trace σ such that $\sigma \sqsubseteq \sigma'$ for any other witness trace σ'.

Consider the query φ_4 in our running example from Figure 1 where we want to find witness traces that will minimize the vector $(Hops, Failures + 3 \cdot Tunnels)$. The query has two witness traces σ_2 and σ_3 and when we evaluate them on the minimization vector, we get the pair $(5, 1+3\cdot 2) = (5, 7)$ for σ_2 and $(5, 0+3\cdot 0) = (5, 0)$ for σ_3. As lexicographically $(5, 0) \sqsubseteq (5, 7)$, the answer to the minimum witness problem is the trace σ_3. In general, there can be several minimum witness traces, and we may return any of those, or add another minimization criterion to the vector of linear expressions.

4 TOOL IMPLEMENTATION

We now give an overview of the tool architecture, its theoretical foundation and integration with the dataplane configuration. The front end of our tool provides a web-browser based visualization. The graphical interface allows us to load a number of predefined networks from the Internet Topology Zoo [1], the operator's network used in the experiments as well as the running example used in this tool paper. In the interface we can specify the query, including an online help for router names with interfaces as well as the sets of labels tested at each router. In options, we can set different parameters for the tool and graphically create the vector of linear expressions for the minimum trace specification. If a witness trace is discovered, the GUI visualizes the trace including the operations performed at each router. A screenshot in Figure 2 shows how to specify the minimization vector $(Hops, Failures + 3 \cdot Tunnels)$ and the corresponding witness trace. The GUI is written in JavaScript and the source code is available under the GPL3 license. The backend verification engine is running on a web server at https://demo.aalwines.cs.aau.dk/ and there is also a packaged version of the tool that can be run locally without the use of a web server (and allows to input additional MPLS networks created by the user).

4.1 Verification methodology

Our tool is based on automata-theoretic approach that leverages a translation from the query satisfiability (in an MPLS network) to a reachability analysis of a pushdown automaton (with potentially infinitely many reachable configurations). As reachability in pushdown automata is decidable in polynomial time [9, 10], this approach has the potential of scaling to large networks.

Figure 2: Running example loaded in the AalWiNes GUI

The connection between MPLS networks and pushdown automata was first noticed in [22, 34] where the authors provide a command-line prototype implementation in Python with encouraging experiments showing the feasibility of the approach. They use a state-of-the-art pushdown model checker Moped [3, 35] for reachability checking on pushdown automata and show that they can verify complex network queries on network topologies with 20-30 routers in a matter of hours. However, the work in [22] is a purely qualitative approach and does not provide any support for quantitative analysis. In order to deal with quantitative aspects, we extend the approach from [22] and suggest a novel translation from the query satisfiability problem with minimization criteria for witness traces, into the framework of *weighted* pushdown automata [26]. The theoretical foundations for the verification of weighted pushdown automata have been developed in the area of dataflow analysis [33] where polynomial-time algorithms are known even for the weighted extension. The basic observation behind this automata-theoretic approach to reachability analysis of weighted pushdown automata is that the set of all reachable configurations in a pushdown system forms a regular language that can be effectively represented by a nondeterministic finite automaton (of polynomial size) with transitions annotated by weights. The length of the shortest path to reach a pushdown configuration then corresponds to the shortest accepting path in the finite automaton under that configuration. As the Moped tool employed in [22] cannot handle weighted pushdown automata, we develop a new weighted pushdown automata C++ library AalWiNes (available at https://github.com/DEIS-Tools/AalWiNes) to replace Moped. Our experiments show a significant (several orders of magnitude) speedup due to the novel translation method with optimized reduction methods as well as due to our efficient implementation of the backend engine.

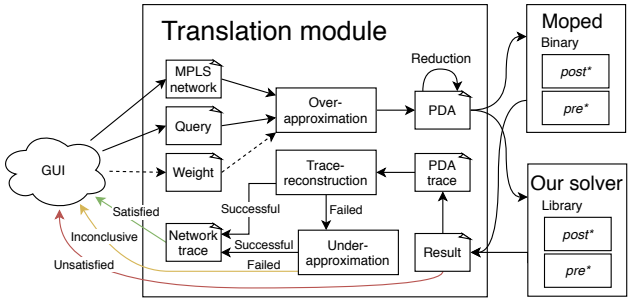


Figure 3: Tool Architecture

4.2 Tool architecture

The details about the architecture of our tool are given in Figure 3. First we obtain a dataplane snapshot of the routing forwarding tables (including the failover rules) as described in Appendix A. If the network configuration changes, we need to obtain a new dataplane snapshot. The graphical user interface allows us to load the MPLS network, a query and possibly also a weight expression. The tool then constructs a pushdown automaton by means of over-approximation as the exact analysis requires to enumerate all of the (exponentially many) failure scenarios. Intuitively, over-approximation assumes that up to k links can fail at any router. This clearly includes all failure scenarios of up to k globally failed links but it may include additional traces that contain more than k failed links in total.

After this, the tool performs a series of reductions (based on static analysis that overapproximates the possible top-of-stack symbols in every given control state) on the constructed (weighted) pushdown automaton by removing redundant rules in order to decrease its size. The reduced pushdown is then sent either to the Moped engine (possible only if the weight requirements are not specified) or to our solver that accepts both weighted and unweighted pushdown automata. If the verification result says that the query is not satisfied, we achieve a conclusive answer and report it to the GUI. Otherwise, the produced network trace must be verified for its feasibility and the fact that it does not exceed in total k link failures (for a fixed trace this can be done in polynomial time). If the trace reconstruction succeeds, we have a witness trace (possibly one of the minimal ones in case the weight objective is given) and we can report that a query is satisfied. Otherwise, our tool constructs an under-approximating pushdown automaton where we add a global failed link counter and use this counter to guarantee that the total number of failed links is not exceeded. This produces only an under-approximation as in case of traces with loops, we may count the same failed link twice. If a valid trace is generated by the under-approximation, we can return it as a witness trace. Otherwise the answer is inconclusive. In our experiments on a real operator network, the answer was inconclusive for 8 out of 6,000 queries (0.13% of the total)—a more expensive analysis is then needed.

5 PERFORMANCE EVALUATION

We evaluate the performance of our tool on a real-world network operator NORDUnet (http://www.nordu.net/) with 31 routers and more than 250.000 forwarding rules. The operator uses an advanced MPLS routing in its network, including numerous service labels by which it communicates with neighboring networks. In order to increase the variety of different types of networks, we create several variants of networks from Internet Topology Zoo [1] (having on average 84 routers and 240 routers at the largest instance) with label switching paths between any two edge routers and with local fast failover protection by introducing tunnels based on shortest paths; the queries are like in Table 1 and in our running example. The experiments were run on our cluster with AMD EPYC 7551 processors running at 2.55 GHz with boost disabled, using 16GB memory limit and 10 minutes timeout. A reproducibility artifact [23] includes the specific queries used in our experiments.

Query	Moped	Dual	Failures
⟨smpls ip⟩ [· #R6] ·* [· #R4] ⟨smpls ip⟩ 1	9.57	0.82	41.23
⟨smpls ip⟩ [· #R2] ·* [· #R18] ⟨(mpls* smpls)? ip⟩ 1	9.29	0.86	31.76
⟨ip⟩ [· #R0] ·* [· #R4] ⟨ip⟩ 0	0.88	0.01	0.02
⟨[$449550] ip⟩ [· #R0] ·* [· #R5] ·* [· #R1] ⟨ip⟩ 0	1.66	0.02	0.03
⟨[$449550] ip⟩ [· #R0] ·* [· #R5] ·* [· #R1] ⟨ip⟩ 1	6.08	0.05	0.06
⟨smpls? ip⟩ ·* ⟨· smpls ip⟩ 0	89.37	14.73	432.66

Table 1: Query verification time (in seconds)

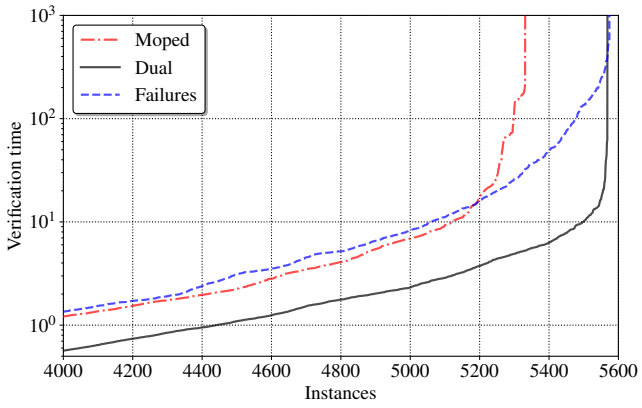


Figure 4: Comparison on Topology Zoo networks

The operator asked us to verify a number of specific queries, including those in Table 1. The table shows the verification time for using Moped as the backend engine, our own engine (called Dual as it combines the over- and under-approximation approach) and our weighted verification engine that uses the *Failures* atomic quantity. Both the unweighted (Dual) and weighted (Failure) engine is a part of our AalWiNes verification suite. The results show that for three queries our weighted engine is on average about 4 times slower that Moped and for other three queries it is about 70 times faster than Moped. Our unweighted engine is always faster and has a 53 times speed up on average compared to Moped. The overhead of performing quantitative analysis is hence reasonable as the performance is comparable with the state-of-the art unweighted tool. Noticeably, the last query in the table takes significantly more verification time for all three engines. The reason is that its path constraint is very unspecific (allows for any sequence of routers)

and the created pushdown system hence becomes significanly larger. We also note that [22] reports that the unweighted verification of similar queries on a network of comparable size took between 28 minutes (for the simpler queries) and up to 109 minutes (for the more complex ones). This shows an improvement of several orders of magnitude and makes it possible to perform MPLS verification interactively for human operators, in particular in combination with our GUI ([22] is a command-line tool).

Finally, the plot in Figure 4 shows a comparison (note the logarithmic scale) of the verification times (in seconds) between Moped, our Dual unweighted approach and our weighted engine with the quantity *Failures* (we also run the experiment for the other quantitative measures and the verification times did not differ significantly). The plot includes over 5602 experiments on different queries on the networks from the Internet Topology Zoo database, ordered by their verification times. As the input format of all three engines is the same, all experiments were run with the same set of network topologies and the same queries. Again, we outperform Moped by almost an order of magnitude by using our unweighted engine. An interesting phenomenon can be observed for our weighted engine that behaves similarly as Moped on the smaller instances; however, on the difficult instances it is able to verify 6 more cases than our unweighted implementation. This is due to the fact that the guided search for shortest traces, that minimize the number of failures, allows us to find witness traces that are otherwise not discovered by the unweighted search; this highlights the benefits of quantitative analysis. This is further confirmed by the percentage of inconclusive answers that corresponds to 0.57% (32 inconclusive answers out of 5568) for the Dual unweighted approach and only 0.04% (2 inconclusive answers out of 5574) for the weighted engine optimizing the number of failures. In vast majority of cases we can hence use our approximation approach that is guaranteed to run in polynomial time.

6 CONCLUSION

We presented an MPLS what-if analysis tool that not only provides an unprecedented performance in theory but also in practice, as demonstrated in our case study with a major network operator. We regard our contribution concerning the automated analysis of quantitative aspects as a first step, and believe that our paper opens interesting avenues for future research. We are currently improving the expressiveness of the query language.

Acknowledgements. We thank Henrik T. Jensen from NORDUnet for providing us with configuration data. The research is supported by DFF project QASNET and WWTF project ICT19-045.

REFERENCES

[1] *The Internet Topology Zoo.* http://www.topology-zoo.org/. Visited: 19/04/2020.
[2] *Introduction to MPLS.* https://www.cisco.com/c/dam/global/fr_ca/training-events/pdfs/Intro_to_mpls.pdf. Visited: 19/05/2020.
[3] *Moped - A Model-Checker for Pushdown Systems.* http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/. Visited: 03/03/2020.
[4] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 201–219.
[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.* 49, 1 (2014), 113–126.
[6] Alia K Atlas and Alex Zinin. 2008. Basic specification for IP fast-reroute: loop-free alternates. IETF RFC 5286. (2008).
[7] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. 2017. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*. ACM, 155–168.
[8] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. 2016. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *Proc. ACM SIGCOMM*. ACM, 328–341.
[9] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*. Springer, 135–150.
[10] J Richard Büchi. 1964. Regular canonical systems. *Archiv für mathematische Logik und Grundlagenforschung* 6, 3-4 (1964), 91–111.
[11] Marco Chiesa, Andrzej KamisiÅĎski, Jacek Rak, GÃąbor RÃľtvÃąri, and Stefan Schmid. 2020. Fast Recovery Mechanisms in the Data Plane. (5 2020). https://doi.org/10.36227/techrxiv.12367508.v1
[12] Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrović, Andrei Gurtov, Aleksander Madry, Michael Schapira, and Scott Shenker. 2016. On the resiliency of static forwarding tables. *IEEE/ACM Transactions on Networking* 25, 2 (2016), 1133–1146.
[13] Richard Chirgwin. 2017. Google routing blunder sent JapanâĂŹs Internet dark on Friday. In *https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/*.
[14] Duluth News Tribune. 2018. Human error to blame in Minnesota 911 outage. In *https://www.ems1.com/911/articles/389343048-Officials-Human-error-to-blame-in-Minn-911-outage/*.
[15] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-wide configuration synthesis. In *Proc. International Conference on Computer Aided Verification (CAV)*. Springer, 261–281.
[16] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *Proc. 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 579–594.
[17] Theodore Elhourani, Abishek Gopalan, and Srinivasan Ramasubramanian. 2014. IP fast rerouting for multi-link failures. In *Proc. IEEE INFOCOM*. ACM, 2148–2156.
[18] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis using a Succinct Control Plane Representation. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 217–232.
[19] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 469–483.
[20] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 300–313.
[21] Brandon Heller, Colin Scott, Nick McKeown, Scott Shenker, Andreas Wundsam, Hongyi Zeng, Sam Whitlock, Vimalkumar Jeyakumar, Nikhil Handigol, James McCauley, et al. 2013. Leveraging SDN layering to systematically troubleshoot networks. In *Proc. ACM SIGCOMM Workshop HotSDN*. ACM, ACM, 37–42.
[22] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. 2018. P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 217âĂŞ227. https://doi.org/10.1145/3281411.3281432
[23] P.G. Jensen, D. Kristiansen, S. Schmid, M. Konggaard Schou, B.C. Schrenk, and J. Srba. 2020. Artifact for "AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks". (Oct. 2020). https://doi.org/10.5281/zenodo.4056504
[24] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 113–126.
[25] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *Proc. USENIX NSDI*. 15–27.
[26] Werner Kuich and Arto Salomaa (Eds.). 1985. *Semirings, Automata, Languages*. Springer-Verlag, Berlin, Heidelberg.
[27] Kim G. Larsen, Stefan Schmid, and Bingtian Xue. 2016. WNetKAT: A Weighted SDN Programming and Verification Language. In *Proc. 20th International Conference on Principles of Distributed Systems (OPODIS)*. LIPICS.
[28] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. 2013. Ensuring connectivity via data plane mechanisms. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 113–126.
[29] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. In *ACM SIGCOMM Computer Communication Review*, Vol. 41 (4). ACM, 290–301.
[30] Michael Menth, Michael Duelli, Ruediger Martin, and Jens Milbrandt. 2009. Resilience analysis of packet-witched communication networks. *IEEE/ACM transactions on Networking (ToN)* 17, 6 (2009), 1950–1963.
[31] P. Pan, G. Swallow, and A. Atlas. 2005. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. In *Request for Comments (RFC) 4090*.

[32] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 953–967.
[33] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58, 1-2 (2005), 206–263.
[34] Stefan Schmid and Jiří Srba. 2018. Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 1–9.
[35] Stefan Schwoon. 2002. *Model-Checking Pushdown Systems*. Ph.D. Thesis. Technische Universität München. http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/schwoon-phd02.pdf
[36] A. Shukla, S.J. Saidi, S. Schmid, M. Canini, T. Zinner, and A. Feldmann. 2020. Towards Consistent SDNs: A Case for Network State Fuzzing. In *IEEE Transactions on Network and Service Management (TNSM)*. 668–681.
[37] Mukarram Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. 2008. Answering what-if deployment and configuration questions with wise. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*. 99–110.
[38] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. 2012. FSR: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking (ToN)* 20, 6 (2012), 1814–1827.
[39] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 87–99.

A APPENDIX

By default, our tool accepts a generic and vendor agnostic XML input format for a network. The input is split into a *topology* definition and a *routing* definition and examples are given below.

topo.xml

```
<network>
  <routers>
    <router name="R0">
      <interfaces>
        <interface name="ae1.11"/>
        <interface name="ae5.0"/>
        ...
      </interfaces>
    </router>
    ...
  </routers>
  <links>
    <sides>
      <shared_interface interface="et-3/0/0.2"
                        router="R0"/>
      <shared_interface interface="et-1/3/0.2"
                        router="R3"/>
    </sides>
  </links>
  ...
</network>
```

route.xml

```
<routes>
  <routings>
    <routing for="R0">
      <destinations>
        <destination from="ae1.11" label="$300292">
          <te-group>
            <routes>
              <route to="et-1/1/0.0">
                <actions>
                  <action arg="$300050"
                          type="swap"/>
                  <action arg="496505"
                          type="push"/>
                </actions>
              </route>
            </routes>
          </te-group>
          ...
        </destination>
        ...
      </destinations>
    </routing>
    ...
  </routings>
</routing>
```

A.1 IS-IS input

Our tool accepts topological description and routing tables exported directly from an IS-IS system; to do so we run the following commands on each router in the network:

```
show isis adjacency detail | display xml
show route forwarding-table family mpls extensive |\
     display xml
show pfe next-hop | display xml
```

To correctly reconstruct the network configuration, an additional mapping file has to be constructed. Each line in the mapping file corresponds to a single logical routing entity and is given in the form `<aliases>:<adj.xml>:<route-ft.xml>:<pfe.xml>`. Edge routers can also be defined by omitting the xml-files. In the case of edge routers, the routing-table is assumed empty, and such routers will act as sink-nodes in the network. An example of such a mapping file is given below.

```
192.0.0.1,R1:R1-adj.xml:R1-route.xml:R1-pfe.xml
192.0.0.2,10.10.0.2,E1
...
```

A network given as an extract from an IS-IS system can be turned into the vendor agnostic format by calling directly our binary and providing the `--write-topology topo.xml` and `--write-routing route.xml` options.

A.2 Location data

To correctly visualize the network in the GUI, an additional location mapping has to be provided giving latitude and longitude to each router. This information is also used for computing the physical distance between routers used in the minimum trace specification. An example is given below.

```
{ "R0": { "lat": 46.5,"lng": 7.3}, ...  }
```