# Taming the transient while reconfiguring BGP

**Conference Paper**

**Author(s):**
Schneider, Tibor; Schmid, Roland; Vissicchio, Stefano; Vanbever, Laurent

**Publication date:**
2023-09

**Permanent link:**
https://doi.org/10.3929/ethz-b-000612650

**Originally published in:**
https://doi.org/10.1145/3603269.3604855

**Funding acknowledgement:**
851809 - From Network Verification to Synthesis: Breaking New Ground in Network Automation (EC)

# Taming the transient while reconfiguring BGP

Tibor Schneider
ETH Zurich
sctibor@ethz.ch

Roland Schmid
ETH Zurich
roschmi@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@ucl.ac.uk

Laurent Vanbever
ETH Zurich
lvanbever@ethz.ch

## ABSTRACT

BGP reconfigurations are a daily occurrence for most network operators, especially in large networks. Yet, performing safe and robust BGP reconfiguration changes is still an open problem. Few BGP reconfiguration techniques exist, and they are either *(i) unsafe*, because they ignore transient states, which can easily lead to invariant violations; or *(ii) impractical*, as they duplicate the entire routing and forwarding states, and require special hardware.

In this paper, we introduce Chameleon, the first BGP reconfiguration framework capable of maintaining correctness throughout a reconfiguration campaign while relying on standard BGP functionalities and minimizing state duplication. Akin to concurrency coordination in distributed systems, Chameleon models the reconfiguration process with happens-before relations. This modeling allows us to capture the safety properties of transient BGP states. We then use this knowledge to precisely control the BGP route propagation and convergence, so that input invariants are provably preserved at any time during the reconfiguration.

We fully implement Chameleon and evaluate it in both testbeds and simulations, on real-world topologies and large-scale reconfiguration scenarios. In most experiments, our system computes reconfiguration plans within a minute, and performs them from start to finish in a few minutes, with minimal overhead.

## CCS CONCEPTS

• **Networks** → **Network management**; **Routing protocols**; *Network control algorithms*; Network reliability.

## KEYWORDS

Border Gateway Protocol (BGP), reconfiguration, network update, convergence, scheduling

**ACM Reference Format:**
Tibor Schneider, Roland Schmid, Stefano Vissicchio, and Laurent Vanbever. 2023. Taming the transient while reconfiguring BGP. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3603269.3604855

*ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA*

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604855

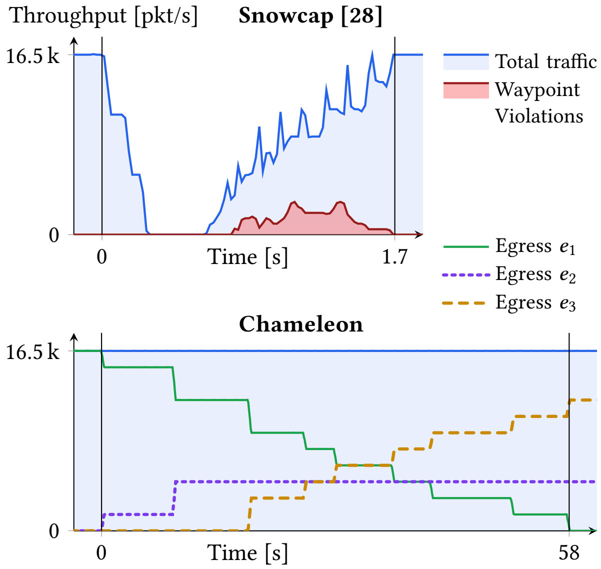


**Figure 1: BGP reconfigurations often lead to disruptions, even when using recent reconfiguration frameworks such as Snowcap [28]. Here, Snowcap transiently violates two invariants (reachability and waypointing). In contrast, Chameleon reconfigures the network without violating any.**

## 1 INTRODUCTION

Much has been written about network reconfigurations, their frequency [12, 20, 28, 32, 36] and their disruptiveness [18, 22, 28]. Yet, reconfiguration-induced downtimes *still* happen. In fact, Alibaba recently stated that the *majority* of their network outages resulted from configuration updates [22].

Among all reconfiguration scenarios, BGP ones are special because they are both particularly frequent and potentially highly disruptive. In large networks, for instance, operators reconfigure BGP up to 20 times a day on average [36]. Also, since BGP controls routing to (and from) remote destinations, reconfiguring it can have Internet-wide consequences. The recent Microsoft outage in January 2023 illustrates this perfectly: Microsoft Azure services were indeed unavailable for *90 minutes* due to a BGP reconfiguration [4].

Perhaps surprisingly, no existing network reconfiguration framework enables both *safe* (in a way that preserves network invariants) and *practical* (in a way that works operationally) BGP reconfigurations. Most previous works targeted other reconfiguration scenarios, such as networks running purely intra-domain routing protocols (OSPF or IS-IS) or SDN/OpenFlow [7]. A few techniques [2, 28, 36] did focus on BGP reconfiguration, but they suffer from fundamental limitations. Specifically, "Shadow Configurations" [2] and BGP

Ships-in-the-Night [36] perform the reconfiguration by duplicating the routing and forwarding states on every network device. Doing so is impractical though as it comes with significant overhead and is not supported by most routers [7]. In contrast, Snowcap [28] gradually modifies the BGP configuration in-place while ensuring that the network converges to a correct state. Snowcap, however, does *not* guarantee the correctness of any of the transient states explored as the network converges during the reconfiguration.

We illustrate the limitations of Snowcap in Fig. 1, which depicts the evolution of the throughput during a simple reconfiguration of the 11-router Abilene network (see §6). The reconfiguration should maintain two invariants: (i) *reachability* (i.e., traffic should not be dropped); and (ii) *waypointing* (i.e., traffic should always cross a firewall). Snowcap transiently violates *both* invariants during the reconfiguration for almost two seconds. For critical (e.g., SLA- or security-related) requirements, transient violations are problematic. Additional experiments for different scenarios show a similar trend, and even longer violations in some cases (cf. Fig. 12).

We present Chameleon, the first *in-place* BGP reconfiguration framework that guarantees correctness during the *entire* reconfiguration process for both the steady *and* transient states.

Intuitively, building a framework like Chameleon requires solving a concurrency problem. It indeed entails reasoning about the asynchronous computations and non-deterministic message exchanges of all BGP routers in the network. This is challenging, for at least three reasons. First, the problem requires a formal concurrency model of a converging BGP network which exceeds the existing BGP models—existing network verification models only reason about the steady state [10, 29] or whether a steady state will be reached [14]. Second, the safe BGP reconfiguration problem requires to design concurrency control mechanisms (i.e., synchronization techniques) that *provably* maintain network-wide correctness without controlling individual BGP message timings. Finally, it requires to design a runtime controller that can efficiently orchestrate the entire reconfiguration process.

We address the challenges in the following ways. First, we introduce a happens-before model which captures BGP-specific concurrency. Using this model, we show how to constrain BGP route propagation through synchronization barriers that provably maintain invariants for all possible concurrent executions. We then compile this schedule into a reconfiguration plan. This plan enforces the computed synchronization barriers by pushing standard BGP commands to the routers and checking their execution. Finally, Chameleon's runtime controller applies this reconfiguration plan to the live network.

Ensuring network correctness throughout the entire reconfiguration process does not come for free: the need to coordinate some of the BGP computation means that the reconfiguration will typically last longer. This cost is exemplified in Fig. 1, where Chameleon took 58 sec to complete instead of 1.7 sec for Snowcap. All in all, we think that such an increase in duration is a small price to pay for the increased correctness.

We implement Chameleon and evaluate it in both testbeds and simulations on real-world topologies and large reconfiguration scenarios. In most experiments, Chameleon computes reconfiguration plans in <1 min and performs them from start to finish in a few minutes, with minimal overhead.

**Temporal operators**

| | | | |
|---|---|---|---|
| $\phi$ | ::= | $\phi$ | *now* |
| | $\mid$ | $\mathbf{N}\,\phi$ | *next* |
| | $\mid$ | $\mathbf{G}\,\phi$ | *globally* |
| | $\mid$ | $\mathbf{F}\,\phi$ | *finally* |
| | $\mid$ | $\phi\,\mathbf{U}\,\phi$ | *until* |
| | $\mid$ | $\phi\,\mathbf{R}\,\phi$ | *release* |
| | $\mid$ | $\phi\,\mathbf{W}\,\phi$ | *weak **U*** |
| | $\mid$ | $\phi\,\mathbf{M}\,\phi$ | *mighty **W*** |

**Logical operators**

| | | | |
|---|---|---|---|
| $\phi$ | ::= | $\phi \land \phi$ | *conjunction* |
| | $\mid$ | $\phi \lor \phi$ | *disjunction* |
| | $\mid$ | $\neg\phi$ | *negation* |

**Propositional variables**

| | | | |
|---|---|---|---|
| $\phi$ | ::= | $reach(n)$ | *reachability* |
| | $\mid$ | $wp(n, n)$ | *waypointing* |
| $n$ | ::= | *node* | *node* |

**Figure 2: Grammar for constructing a specification $\phi$ for a single destination $d$. Chameleon guarantees that $\phi$ is satisfied during the entire reconfiguration, i.e., the sequence of forwarding states satisfies $\phi$.**

To sum up, our main contributions are:

- The first practical BGP reconfiguration technique capable of preserving correctness invariants throughout the entire reconfiguration campaign, including transient states.
- A concurrency model for BGP reconfigurations and its convergence process using happens-before relations.
- A complete implementation of Chameleon in Rust alongside an online application to explore reconfiguration scenarios (available at *https://bgpsim.github.io?s=example*).
- An extensive evaluation of Chameleon and its overhead.

## 2 OVERVIEW

We now provide an overview of Chameleon. We start with the problem statement and illustrate the challenges using a running example before describing Chameleon's high-level workflow.

### 2.1 Problem Statement

*Reconfiguration.* We refer to the problem of adapting the BGP configuration of one or more routers as a *BGP reconfiguration.* Examples of reconfigurations include: (i) *local* changes like adding or removing a BGP session or changing the local preference of routes learned from a specific neighbor; and (ii) *global* changes such as switching from an iBGP full-mesh to route reflection network-wide.

*Specification.* We refer to a *specification* $\phi$ as a set of network invariants that describe important forwarding properties for the network operators. Our specification language (cf. Fig. 2) combines individual properties with boolean operators and Linear Temporal Logic (LTL). Boolean operators allow users to express intricate properties on the forwarding state, such as isolating traffic from parts of the network. In contrast, temporal operators formulate constraints on the sequences of transient states explored during reconfigurations. For example, if a reconfiguration involves changing the egress router of a given destination prefix, operators can require routers to switch from the initial egress router to the final egress *once*, without switching back and forth multiple times.

We assume the initial and final configurations are correct and comply with $\phi$. Otherwise, there is little point in maintaining correctness *during* the reconfiguration. Similarly, we assume that the initial and final configurations eventually converge to a stable state [14, 15] to ensure that the semantics of input specifications is always sound.

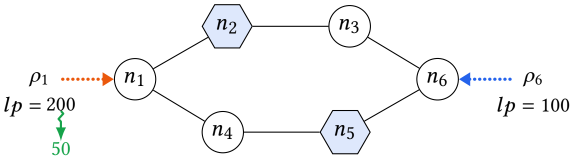


**Figure 3: Example network with six internal routers. Two routers $n_1$ and $n_6$ receive a route $\rho_1$ and $\rho_6$, respectively, for the same prefix. The depicted reconfiguration lowers the local preference of $\rho_1$ from 200 to 50, causing the network to shift from using $\rho_1$ to $\rho_6$.**

*Goals.* We aim to perform both local and global BGP reconfigurations in-place while guaranteeing the correctness of a given specification $\phi$ throughout the *entire* process. Specifically, we aim to achieve the following properties:

1. (**safety**) the reconfiguration process must guarantee that the input specification $\phi$ is satisfied throughout any transient state explored during the BGP convergence process;
2. (**practicality**) the reconfiguration process must limit the overhead it imposes on routers' resources, including routing and forwarding table sizes and reconfiguration time.

*Running example.* As an example BGP reconfiguration scenario, we consider the network in Fig. 3, which consists of six BGP routers, two of which ($n_2$ and $n_5$) operate as route reflectors [3].[1] Routers $n_1$ and $n_6$ each receive one BGP route *for the same prefix*. We refer to those routes as $\rho_1$ and $\rho_6$, respectively. The reconfiguration involves decreasing the local preference of $\rho_1$ from 200 to 50 so that all routers switch to using $\rho_6$. We suppose the operators aim at preserving reachability throughout the reconfiguration, which they can express as $\phi = \mathbf{G} \bigwedge_i reach(n_i)$ in our specification language.

Preferring $\rho_6$ will cause all routers to forward "to the right" instead of "to the left." Intuitively, reachability is violated whenever routers on the left-hand side of the network ($n_1$, $n_2$ or $n_4$) update their state before routers on the right ($n_3$, $n_5$ or $n_6$). Depending on the timing of the BGP messages, this can easily happen, especially since $n_3$ only learns $\rho_6$ from one of the two route reflectors, $n_2$ or $n_5$, i.e., *after* they have already selected $\rho_6$.

## 2.2 Chameleon

In a nutshell, Chameleon achieves safety by coordinating the forwarding state updates BGP makes during the reconfiguration. To practically coordinate the updates, Chameleon gradually introduces a temporary BGP configuration (which differs from both the initial and final configuration). This temporary configuration enforces a particular ordering of BGP messages that satisfies the specification.

Finding and implementing specific message orderings is far from trivial. The first challenge is capturing achievable orderings by introducing temporary BGP configurations. This is hard as many BGP-specific mechanisms—like route reflection [15]—limit route visibility. The second challenge is implementing a specific execution using temporary configurations; this requires both reasoning about the distributed routing state and synchronizing BGP computations.

*Workflow.* Chameleon solves those challenges by following the following three consecutive steps. Chameleon's workflow is visualized in Fig. 4 using the running example from Fig. 3.

1. The **analyzer** describes the space of concurrent convergence processes by analyzing the initial and final configuration (the input to Chameleon) and computing happens-before relations between routing states of different routers.
2. The **scheduler** explores the space of convergence processes spanned by the happens-before relations to find one that satisfies the specification. It describes this convergence process as a node schedule that captures which routes are selected at which time.
3. The **compiler** transforms this node schedule into a *reconfiguration plan*, that is, a sequence of temporary configuration commands and local conditions for synchronization.

Finally, Chameleon's runtime controller performs the reconfiguration on the live network by checking the local conditions and applying the commands, precisely following the compiled reconfiguration plan. In the following, we describe all three steps of Chameleon in more detail, using the running example from Fig. 3.

*Step 1: Analyzer §3.* The analyzer extracts happens-before relations between selected routes in the network, encoding the propagation path of routes. These relations define the space of convergence processes that are realizable just using temporary BGP configurations. We obtain these relations by simulating the network and analyzing the resulting network state in a way that generalizes to any BGP configuration.

In our example from Fig. 3, router $n_1$ propagates the initial route $\rho_1$ to both route reflectors $n_2$ and $n_5$, who then announce $\rho_1$ towards $n_3$, $n_4$, and $n_6$. Therefore, $n_4$ can choose $\rho_1$ as long as $n_2$ or $n_5$ select $\rho_1$. We repeat the same for $\rho_6$.

*Step 2: Scheduler §4.* We express the happens-before relations together with the specification as an Integer Linear Program (ILP). A solution to the ILP is a node schedule describing a BGP convergence process that meets the specification, in both steady and transient states. The scheduler allows concurrent updates if their relative order does not affect the specification. Further, Chameleon allows additional propagation paths to increase route visibility and ensure a solution exists. We maximize the number of concurrent updates (thus, minimizing the reconfiguration time) as a primary objective while reducing additional propagation paths as a secondary goal.

Chameleon schedules the example reconfiguration using four rounds, updating nodes from right to left. It also computes the rounds in which a router selects its old or new route. In Fig. 4, orange arrows represent the propagation of the old route $\rho_1$, whereas blue arrows depict the new one $\rho_6$.

*Step 3: Compiler §5.* Chameleon computes a reconfiguration plan to implement the calculated schedule, along with local conditions to synchronize the update and guarantee correctness. Each temporary command only rewrites routing preferences by modifying route attributes such as weight [25] or local preference. The conditions assert a router knows or selects a specific route and can be checked locally by inspecting that router's routing table.

[1] Concretely, this means that $n_2$ and $n_5$ redistribute the best BGP routes they receive to all other routers in the network.

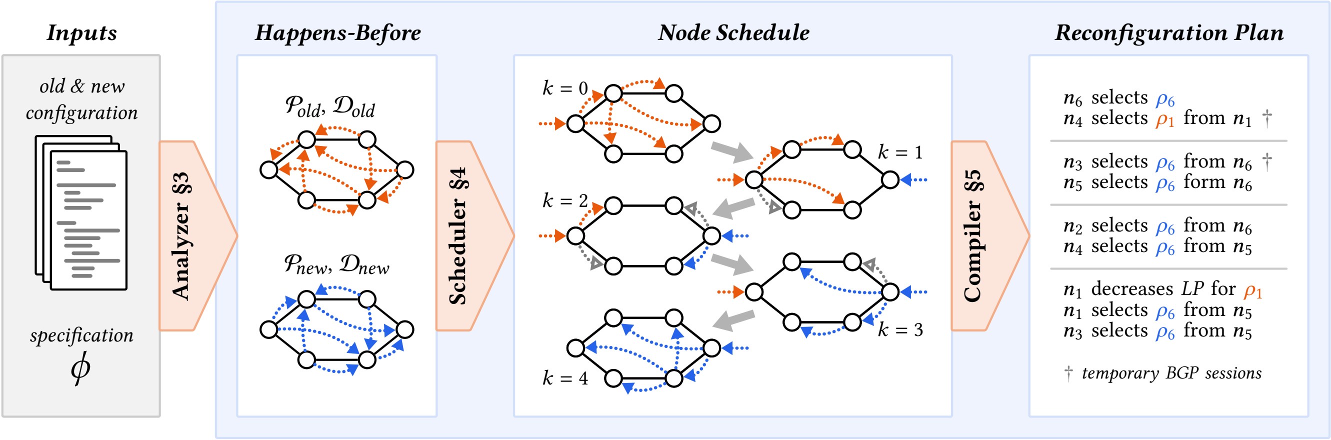

Inputs
old & new configuration
specification
$\phi$
Analyzer §3
Happens-Before
$\mathcal{P}_{old}$, $\mathcal{D}_{old}$
$\mathcal{P}_{new}$, $\mathcal{D}_{new}$
Scheduler §4
Node Schedule
k = 0
k = 1
k = 2
k = 3
k = 4
Compiler §5
Reconfiguration Plan
$n_6$ selects $\rho_6$
$n_4$ selects $\rho_1$ from $n_1$ †
$n_3$ selects $\rho_6$ from $n_6$ †
$n_5$ selects $\rho_6$ form $n_6$
$n_2$ selects $\rho_6$ from $n_6$
$n_4$ selects $\rho_6$ from $n_5$
$n_1$ decreases LP for $\rho_1$
$n_1$ selects $\rho_6$ from $n_5$
$n_3$ selects $\rho_6$ from $n_5$
† temporary BGP sessions

**Figure 4: Chameleon computes a reconfiguration plan to transition from the old to the new configuration while satisfying the specification. The illustrated instance corresponds to the example from Fig. 3. Explore this reconfiguration plan interactively at *https://bgpsim.github.io?s=example*. We provide a tutorial in App. E.**

Chameleon's reconfiguration plan for our example first updates the egress router $n_6$ to make $\rho_6$ available in the network. It then makes both $n_5$ and $n_3$ select $\rho_6$ using a temporary session between $n_3$ and $n_6$. In Round 3, it updates both $n_2$ and $n_4$. Chameleon introduces a temporary session between $n_4$ and $n_1$ to break $n_4$'s dependency on $n_2$. Finally, Chameleon finishes the reconfiguration by updating the local-pref on $n_1$, which causes $n_1$ to select $\rho_6$. Notice, that this reconfiguration can be performed without using temporary BGP sessions by reducing concurrent updates.

If the reconfiguration updates multiple destinations at a time, Chameleon runs the scheduler and the compiler for each destination separately and combines the generated schedules. This is only possible if all prefixes converge independently, which is valid case for most networks. However, there might exist networks for which some prefixes are dependent. We explain in §8 how the scheduler can be extended to support such networks.

## 3 ANALYZER

Chameleon models the routing state of the network as propagation paths of BGP routes. It then describes the reconfiguration process in terms of how these routes change over time and formulates happens-before relations capturing safety conditions on BGP transient states.

*Network & Routing Model.* We model the network as a graph $G = (N, E)$ with nodes $N$ connected by edges $E$. External networks advertise BGP routes towards a destination $d$ to multiple nodes; the egress routers of $G$. Nodes propagate routes over BGP sessions that form an overlay signaling graph, different from the physical topology. We define routes in terms of their *propagation path* rather than modeling a route's specific BGP attributes. If $\rho = [d, n_1, \ldots, n_i, n]$ is a route received at node $n$, then $\rho$ was first received by the egress router $e(\rho) = n_1$, and advertised to $n$ by its BGP neighbor $n_i = pre(\rho)$. The network's *configuration* determines:

(i) which route $\rho$ a node $n$ selects,
(ii) to which neighbors $n$ announces its selected route $\rho$,
(iii) how the egress $e(\rho)$ is mapped to the next hop.

The collection of all nodes and their next hop is called a *forwarding state*. For a single destination $d$, we express the forwarding state $nh : N \mapsto N \cup \{d, \emptyset\}$ as a mapping of each node to its next hop. Node $n$ drops packets if $nh(n) = \emptyset$, and forwards them to the external network if $nh(n) = d$.

A *routing state* $\mathcal{P}$ assigns each node $n$ its selected route $\mathcal{P}(n)$ for destination $d$. We call a routing state $\mathcal{P}$ *consistent* if

$$\forall n \in N : \mathcal{P}(n) = [d, n_1, \ldots, n_i, n] \Rightarrow \mathcal{P}(n_i) = [d, n_1, \ldots, n_i].$$

Observe that the routing state of any *converged* network, that is, a network in a fixed routing state with no more messages in flight, is always consistent. However, the consistency of a routing state only ensures that there exists a configuration such that the network converges to this routing state.

Using these terms, we define a *BGP reconfiguration* as the transition from an initial routing state $\mathcal{P}_{old}$ to the final routing state $\mathcal{P}_{new}$. A *reconfiguration process* $[\mathcal{P}_{old}, \mathcal{P}_1, \ldots, \mathcal{P}_{new}]$ is a sequence of routing states which contains all the intermediate routing states of the network during a reconfiguration. A reconfiguration process is *safe* if its corresponding sequence of forwarding states satisfies the specification that the operator expresses $[nh_{old}, nh_1, \ldots, nh_{new}] \models \phi$.

Our routing model considers a single destination $d$ at a time. However, one destination can represent an entire class of equivalent prefixes for which the network computes the same routing and forwarding state. If a BGP reconfiguration affects multiple destinations, Chameleon treats each prefix equivalence class separately: this is safe because BGP separately processes individual destinations.

*Methodology & Framework.* Chameleon controls each intermediate routing state during the reconfiguration to safely transition from $\mathcal{P}_{old}$ to $\mathcal{P}_{new}$. To that end, Chameleon synthesizes a sequence of routing states such that each state is consistent. Consequently, the BGP reconfiguration process never affects the forwarding state. Instead, we trigger each routing and forwarding state update by reconfiguring nodes.

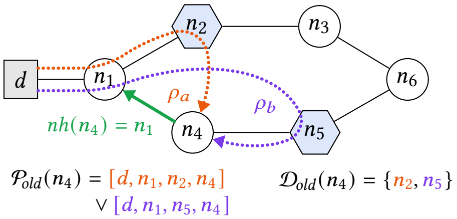

$d$
$n_1$
$n_2$
$n_3$
$n_4$
$n_5$
$n_6$
$\rho_a$
$\rho_b$
$nh(n_4) = n_1$
$\mathcal{P}_{old}(n_4) = [d, n_1, n_2, n_4] \vee [d, n_1, n_5, n_4]$
$\mathcal{D}_{old}(n_4) = \{n_2, n_5\}$

**Figure 5: Illustration of the BGP convergence model based on the example from Fig. 3 in the initial state.**

A naive approach to synthesizing a sequence of routing state updates is to make each node $n$ switch its selected route directly from the initial route $\mathcal{P}_{old}(n)$ to its final one $\mathcal{P}_{new}(n)$. This technique, however, does not work for most reconfiguration scenarios. Assume node $n$ learns both its initial and final route from $n_i = pre(\mathcal{P}_{old}(n)) = pre(\mathcal{P}_{new}(n))$. The naive approach cannot ensure the consistency of every routing state, as $n_i$ must switch before $n$. While $n_i$ is using $\mathcal{P}_{new}(n)$ and $n$ still selects $\mathcal{P}_{old}(n)$, the routing state is inconsistent. Chameleon uses two techniques to resolve this issue.

The first technique is to establish a temporary BGP session directly with the egress router, $e(\mathcal{P}_{old}(n))$ or $e(\mathcal{P}_{new}(n))$. This session allows $n$ to select the route $[d, n_1, n]$ directly. This technique has two significant downsides. First, the additional session causes state duplication. Second, the new route may have different BGP attributes that affect the route propagation. Thus, we must ensure no other node selects a route from $n$ while $n$ selects $[d, n_1, n]$.

The second technique to ensure each routing state's consistency is to leverage BGP redundancy. Most BGP configurations using route reflection have multiple reflectors. As a result, node $n$ may receive multiple *equivalent* routes that share the same BGP attributes. Choosing one equivalent route over another does not impact the forwarding state, but it affects the routing state. Therefore, simply switching between equivalent routes can violate routing state consistency while the update propagates through the network. We introduce a setup and cleanup phase, during which Chameleon chooses between equivalent routes, ensuring the forwarding state remains unchanged. Chameleon still enforces routing state consistency during the main update phase.

Our methodology results in each node $n$ updating its next hop exactly once from $nh_{old}(n)$ to $nh_{new}(n)$. Thus, Chameleon never exports any transient routes towards inter-domain neighbors. However, there may exist highly constraining specifications that Chameleon cannot enforce. We prove in App. B that our methodology is sufficient to perform any reconfiguration preserving reachability.

*Happens-Before Relations.* We model a BGP reconfiguration process within our framework using happens-before relations. First, we capture the set of all neighbors that advertise identical routes as the initial and the final route towards node $n$ using $\mathcal{D}_{old}(n)$ and $\mathcal{D}_{new}(n)$, respectively. $n$ cannot use its initial route $\mathcal{P}_{old}(n)$ for longer than any of $\mathcal{D}_{old}(n)$ also select their initial route. Likewise, $n$ can only select $\mathcal{P}_{new}(n)$ as soon as any of $\mathcal{D}_{old}(n)$ select their final route. The resulting happens-before relations for node $n$ are:

- $n$ selects $\mathcal{P}_{old}(n) \implies \exists m \in \mathcal{D}_{old}(n)$ that selects $\mathcal{P}_{old}(m)$.
- $n$ selects $\mathcal{P}_{new}(n) \implies \exists m \in \mathcal{D}_{new}(n)$ that selects $\mathcal{P}_{new}(m)$.

Fig. 5 illustrates the happens-before relations for the initial state using the running example introduced in Fig. 3. Node $n_4$ receives equivalent routes for $d$ with propagation paths $\rho_a$ and $\rho_b$ from route reflectors $n_2$ and $n_5$. Thus, $n_4$ can select route $\mathcal{P}_{old}(n_4)$ for as long as either $n_2$ or $n_4$ advertise it.

## 4 SCHEDULER

We now detail Chameleon's scheduler. The goal of the scheduler is to generate a node schedule, that is, the order in which routers switch from selecting their old route in $\mathcal{P}_{old}$ to their new route $\mathcal{P}_{new}$. We encode this schedule by assigning each node a round in which it will change its routing (and forwarding) decision. We formalize the problem of finding a schedule with at most $R$ rounds as an ILP. We increment $R$ in a loop as long as no solution exists, thus minimizing the number of rounds $R$ and, consequently, the reconfiguration time. Our ILP captures these three high-level properties:

1. *Happens-before relations*: The scheduler transforms the happens-before relations from §3 into constraints. We design the objective function to minimize the number of temporary BGP sessions.
2. *Concurrent Updates*: Allowing nodes to update concurrently reduces the number of required rounds. However, this concurrency can lead to different update orderings within a single round. Therefore, we require all routers to change their next hop in a round where *no* other router along its path also changes its forwarding decision. This constraint guarantees updates in the same round are *independent* concerning the forwarding state.
3. *Specification*: The resulting sequence of forwarding states must satisfy the specification $\phi$, expressed as LTL.

### 4.1 Happens-Before Relations

We enforce BGP propagation rules in the ILP model by using three symbolic integer variables for each node $n \in N$:

$$\bigwedge_{n \in N} r^n_{old} \le r^n_{nh} \le r^n_{new}. \tag{1}$$

We call the 3-tuple $\left(r^n_{old}, r^n_{nh}, r^n_{new}\right)$ the schedule of node $n$. Here, $r^n_{nh} \in \mathbb{N}$ describes the round in which $n$ changes its next hop. Further, $r^n_{old} \in \mathbb{N}$ capture the last round when $n$ receives the old route $\mathcal{P}_{old}(n)$, and $r^n_{new} \in \mathbb{N}$ is first round for $n$ to receive $\mathcal{P}_{new}(n)$.

Next, we enforce BGP route propagation as described by the happens-before model in §3. A neighbor $m \in \mathcal{D}_{old}(n)$ must exist that selects its old route $\mathcal{P}_{old}(m)$ for longer than $n$ selects $\mathcal{P}_{old}(n)$, i.e., $r^n_{old} < r^m_{old}$. Similarly, a neighbor $m \in \mathcal{D}_{new}(n)$ must exist that selects its new route before $n$ does, i.e., $r^n_{new} > r^m_{new}$. This yields

$$\bigwedge_{n \in N} r^n_{old} < \max_{m \in \mathcal{D}_{old}(n)} r^m_{old} \quad \wedge \quad \bigwedge_{n \in N} r^n_{new} > \min_{m \in \mathcal{D}_{new}(n)} r^m_{new}.$$

Whenever $r^n_{old} < r^n_{nh}$, node $n$ no longer knows its initial route $\mathcal{P}_{old}(n)$ as long as it must use its old next hop. Similarly, if $r^n_{nh} < r^n_{new}$, node $n$ does not know route $\mathcal{P}_{new}(n)$ when it should use its new next hop. In both cases, we must introduce a temporary BGP session. We minimize the number of such sessions as follows:

$$\min \sum_{n \in N} \left(1 \text{ if } r^n_{old} \ne r^n_{nh} \text{ else } 0\right) + \left(1 \text{ if } r^n_{nh} \ne r^n_{new} \text{ else } 0\right).$$

We encode all non-linear *min*, *max*, and *if-then-else* operators with linear constraints using the big-$M$ method [16].

## 4.2 Concurrent Updates

Our ILP allows multiple nodes to update their next hop in the same round. As a result, individual updates of the same round may occur in any order. Hence, the ILP must guarantee the specification is satisfied in all possible orderings for each round. To that end, Chameleon ensures all forwarding state updates in the same round are *independent.* We define two forwarding state updates of node $n_1$ and $n_2$ to be independent if the forwarding path of $n_1$ before and after the update does not traverse $n_2$, and vice versa—consequently, every forwarding path in the network experiences at most one change in each round.

We enforce the independence of all forwarding state updates in the same round recursively. For each node $n \in N$ and round $k$, we introduce the boolean variable $\delta_k^n \in \mathbb{B}$ to capture whether either $n$ or a different node along the forwarding path of $n$ changes its next hop in round $k$. We express $\delta_k^n$ in terms of its next hop:

$$\delta_k^n = \begin{cases} \delta_k^x & \text{if } r_{nh}^n > k \quad \text{$n$ uses its old route in round $k$} \\ 1 + \delta_k^x + \delta_k^y & \text{if } r_{nh}^n = k \quad \text{$n$ changes its route in round $k$} \\ \delta_k^y & \text{if } r_{nh}^n < k \quad \text{$n$ uses its new route in round $k$} \end{cases} \tag{2}$$
$$\text{where } x = nh_{old}(n) \text{ and } y = nh_{new}(n)$$

In case $r_{nh}^n \neq k$, $\delta_k^n$ directly depends on either the initial or the final next hop. However, in case $n$ changes its routing decision in round $r_{nh}^n = k$, then $\delta_k^n = 1 + \delta_k^x + \delta_k^y$. Since $\delta_k^n \in \{0, 1\}$, this constraint implies that $\delta_k^x = 0$ and $\delta_k^y = 0$, i.e., that there is no other update along either the old or the new forwarding path of $n$.

## 4.3 Specification

To encode the specification $\phi$ in the ILP, Chameleon generates a *syntax tree* according to the specification language. Each node in that graph is labeled with its production rule, as shown in Fig. 2.[2] We then generate a directed acyclic graph $G_\phi = (\Phi, E_\phi)$ by combining equivalent nodes with identical descendants. In other words, we simplify the tree by combining branches that result in redundant constraints. In the following, we call each node $\phi_i \in \Phi$ in this syntax graph an *expression.* For each expression $\phi_i \in \Phi$, we introduce symbolic boolean variables $\phi_{i,k} \in \mathbb{B}$. $\phi_{i,k}$ is true if and only if the expression $\phi_i$ is satisfied in round $k$. The following explains the constraints we add to the ILP for each kind of expression.

*Reachability.* Chameleon creates recursive constraints to ensure reachability. Essentially, we say a node $n$ satisfies reachability $reach(n)$ if its next hop does so. This recursive statement requires not only checking $n$ for reachability but also every other node $m \in N$ in the network. More formally, for each node $m \in N$ and each round $k \in \{1, \ldots, R\}$, we introduce a boolean variable $\phi_{reach,k}^m \in \mathbb{B}$ and constrain it in terms of $m$'s next hop $x$:

$$\phi_{reach,k}^m = \begin{cases} 1 & \text{if } x = d \\ 0 & \text{if } x = \emptyset \\ \phi_{reach,k}^x & \text{otherwise} \end{cases}$$
$$\text{where } x = nh_{new}(m) \text{ if } r_{nh}^m \leq k \text{ else } nh_{old}(m)$$

Here, $nh_{old}$ and $nh_{new}$ are the initial and final forwarding states, and $x$ is the next hop of node $m$ at round $k$. Notice that these constraints are sufficient if and only if there are no forwarding loops. We explain in §4.4 how we ensure the absence of loops.

*Waypointing.* Waypoints are treated similarly to reachability. Node $n$ satisfies $wp(n, w)$ if its next hop does so or if its next hop is $w$. For each waypoint target $w \in N$ in the specification, we introduce a boolean variable $\phi_{wp(w),k}^m$ for each node $m \in N$ and round $k \in \{1, \ldots, R\}$, constrained in terms of $m$'s next hop $x$:

$$\phi_{wp(w),k}^m = \begin{cases} 1 & \text{if } x = w \lor n = w \\ 0 & \text{if } x \in \{\emptyset, d\} \\ \phi_{wp(w),k}^x & \text{otherwise,} \end{cases}$$
$$\text{where } x = nh_{new}(m) \text{ if } r_{nh}^m \leq k \text{ else } nh_{old}(m).$$

*Logical and Temporal Modal Operators.* Chameleon encodes a logical or temporal modal expression $\phi_i$ by following its production rule and refering to the symbolic boolean variables of its subexpressions. We constrain the variables $\phi_{i,k} \in \mathbb{B}$ as follows:

- We implement the conjunction rule $\phi_i = \phi_a \land \phi_b$ at round $k$ using the big-$M$ method [16]: $\phi_{i,k} = \phi_{a,k} \land \phi_{b,k}$. The other logical operators ($\neg$ and $\lor$) are constructed identically.
- We encode the *globally* operator $\phi_i = \mathbf{G}\,\phi_a$ by asserting $\phi_a$ is satisfied indefinitely, starting from $k$: $\phi_{i,k} = \bigwedge_{j \geq k} \phi_{a,j}$.
- To encode the *until* operator $\phi_i = \phi_a \,\mathbf{U}\, \phi_b$, we assert the existence of a round $l \geq k$ until which $\phi_a$ is satisfied, and at which $\phi_b$ holds. To that end, we unroll the $\exists$ quantifier:
$$\bigvee_{l \geq k} \Big( \phi_{b,l} \land \bigwedge_{m<l} \phi_{a,m} \Big)$$
- We implement all other temporal modal operators similarly to $\mathbf{U}$ and $\mathbf{G}$ by unrolling logical quantifiers.

Ultimately, we assert the reconfiguration plan satisfies $\phi$ by asserting the root of the syntax tree $\phi_r$ holds in the first round $k = 1$:

$$\phi_{r,1} = 1.$$

## 4.4 Absence of loops

The recursive constraints of reachability and waypoint require the absence of forwarding loops. To illustrate, let us consider a forwarding loop between $n_1$ and $n_2$. While generating the reachability constraints, Chameleon only requires $\phi_{reach}^{n_1} = \phi_{reach}^{n_2}$, thus allowing the model to choose an arbitrary value for $\phi_{reach}^{n_1}$.

We ensure the absence of forwarding loops by enumerating all possible forwarding loops. To that end, we build a graph $G_{nh} = (N, E_{nh})$ by combining the old and the new forwarding states. For each node $n \in N$, $G_{nh}$ contains two edges from $n$ to its old and new next hop, $nh_{old}(n)$ and $nh_{new}(n)$, i.e., $(n, nh_{old}(n)) \in E_{nh}$ and $(n, nh_{new}(n)) \in E_{nh}$. Each simple cycle in $G_{nh}$ represents one possible forwarding loop. We enumerate all simple cycles $L$ of $G_{nh}$. For each cycle $l = (n_1, \ldots, n_j, n_1) \in L$ of length $j$ and each round $k$, we assert that not all nodes $n_i$ along the cycle can choose $n_{i+1}$ as next hop simultaneously:

[2] Our implementation ensures the uniqueness of this tree by surrounding each production rule with parenthesis.

$$j > \sum_{i \le j} \begin{cases} 1 & \text{if } n_{i+1} = nh_{old}(n_i) \wedge r^{n_i}_{nh} > k \\ 1 & \text{if } n_{i+1} = nh_{new}(n_i) \wedge r^{n_i}_{nh} \le k \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The constraints on concurrent updates (cf. §4.2 and Eq. (2)) implicitly ensure the absence of loops if the initial forwarding state is loop-free. We provide a proof in App. D. However, we notice the variance of the scheduling time of Chameleon decreasing when explicitly enumerating each possible loop, while the average time is unaffected. We, therefore, keep the explicit loop constraints in the model even though they are not strictly necessary.

## 5 COMPILER

Chameleon's compiler transforms the schedule of the routing and forwarding state into a reconfiguration plan. This reconfiguration plan consists of a sequence of commands and conditions, constituting synchronization to implement the schedule. Each command will only affect a specific entry of a single router's routing and forwarding table. Similarly, each conditions only need to check a particular entry of the routing or forwarding tables. The following explains how the compiler transforms the 3-tuples $(r^n_{old}, r^n_{nh}, r^n_{new})$, computed by the scheduler, into a reconfiguration plan.

To gradually transition the routing state as computed by the scheduler, Chameleon changes route attributes that stay local to that device without affecting other routers in the network. To that end, the compiled commands modify a route's weight [25] using route maps.

*Reconfiguration Phases.* The reconfiguration plan consists of three phases: setup, update, and cleanup. During the update phase, we forbid Chameleon to switch between equivalent routes from different neighbors (cf. §3). Instead, we ensure each node appropriately changes its BGP neighbor in the setup phase. Based on the happens-before relations, there must exist a neighbor $m^n_{old} \in \mathcal{D}_{old}(n)$ of node $n$ that advertises $\mathcal{P}_{old}(n)$ to $n$ for longer than $r^n_{old}$. Similarly, a neighbor $m^n_{new} \in \mathcal{D}_{new}(n)$ advertises $\mathcal{P}_{new}(n)$ to $n$ before round $r^n_{new}$. The setup phase ensures each node $n$ prefers the route learned from $m^n_{old}$. Further, the setup phase establishes all temporary BGP sessions. Similarly, the cleanup phase removes all route preferences and temporary sessions.

We divide the update phase into $R$ rounds. Chameleon ensures synchronization by *(i)* using pre- and post-conditions for commands and *(ii)* transitioning between rounds when all post-conditions are satisfied. Pre-conditions check a route's availability, while a post-condition ensures it is selected. Chameleon generates those commands and conditions using rules shown in Table 1 based on the 3-tuple $(r^n_{old}, r^n_{nh}, r^n_{new})$ of each node $n$. These rules guarantee node $n$ selects $\mathcal{P}_{old}(n)$ until round $r^n_{old}$, updates its next hop in round $r^n_{nh}$, and then selects $\mathcal{P}_{new}(n)$ starting from round $r^n_{new}$.

**Table 1: Compilation rules for the update phase.**

$r^n_{old} = r^n_{nh} = r^n_{new}$:

- In round $r^n_{nh}$, make $n$ prefer $\mathcal{P}_{new}(n)$ from $m^n_{new}$.
    - *pre-condition*: $n$ knows $\mathcal{P}_{new}(n)$.
    - *post-condition*: $n$ selects $\mathcal{P}_{new}(n)$.

$r^n_{old} < r^n_{nh} = r^n_{new}$:

- In round $r^n_{old}$, make $n$ prefer the route from $e\big(\mathcal{P}_{old}(n)\big)$ using a temporary BGP session.
    - *post-condition*: $n$ selects the route from $e\big(\mathcal{P}_{old}(n)\big)$.
- In round $r^n_{nh}$, make $n$ prefer $\mathcal{P}_{new}(n)$ from $m^n_{new}$.
    - *pre-condition*: $n$ knows $\mathcal{P}_{new}(n)$.
    - *post-condition*: $n$ selects $\mathcal{P}_{new}(n)$.

$r^n_{old} = r^n_{nh} < r^n_{new}$:

- In round $r^n_{nh}$, make $n$ prefer the route from $e\big(\mathcal{P}_{new}(n)\big)$ using a temporary BGP session.
    - *post-condition*: $n$ selects the route from $e\big(\mathcal{P}_{new}(n)\big)$.
- In round $r^n_{new}$, make $n$ prefer $\mathcal{P}_{new}(n)$ from $m^n_{new}$.
    - *pre-condition*: $n$ knows $\mathcal{P}_{new}(n)$.
    - *post-condition*: $n$ selects $\mathcal{P}_{new}(n)$.

$r^n_{old} < r^n_{nh} < r^n_{new}$:

- In round $r^n_{old}$, make $n$ prefer the route from $e\big(\mathcal{P}_{old}(n)\big)$ using a temporary BGP session.
    - *post-condition*: $n$ selects the route from $e\big(\mathcal{P}_{old}(n)\big)$.
- In round $r^n_{nh}$, make $n$ prefer the route from $e\big(\mathcal{P}_{new}(n)\big)$ using a temporary BGP session.
    - *post-condition*: $n$ selects the route from $e\big(\mathcal{P}_{new}(n)\big)$.
- In round $r^n_{new}$, make $n$ prefer $\mathcal{P}_{new}(n)$ from $m^n_{new}$.
    - *pre-condition*: $n$ knows $\mathcal{P}_{new}(n)$.
    - *post-condition*: $n$ selects $\mathcal{P}_{new}(n)$.

*Original Reconfiguration Commands.* Applying the reconfiguration plan must transition the network from the initial to the final configuration. Chameleon interleaves its temporary commands with the original reconfiguration commands. Let $N^* \subseteq N$ contain all nodes subject to those commands. For each node $n \in N^*$, we apply the original commands $c^*$ for $n$ immediately before or after $r^n_{nh}$. If $c^*$ makes $n$ deny route $\mathcal{P}_{old}(n)$, then we execute $c^*$ after $r^n_{nh}$. Otherwise, we schedule $c^*$ before $r^n_{nh}$. Thus, we guarantee $n$ knows $\mathcal{P}_{old}(n)$ until round $r^n_{nh}$, and $\mathcal{P}_{new}(n)$ starting from $r^n_{nh}$.

Chameleon treats each destination separately. To support reconfigurations that affect multiple destinations, Chameleon performs the update phase for all destinations in parallel and aligns their execution along the original commands. For instance, assume that the original command $c^*$ is applied before round 3 for destination $d_1$ and after round 4 for $d_2$. We perform the update phase for $d_1$ until round 2 and $d_2$ until round 4. Then, we apply the $c^*$ and proceed with the update phase of both $d_1$ and $d_2$. However, such an alignment may not exist if multiple nodes are subject to the original reconfiguration. In fact, the schedule for $d_1$ could require the original command $c^*_1$ to be applied after $c^*_2$, whereas $d_2$ expects the reverse order for $c^*_1$ and $c^*_2$. In that case, we use Snowcap to split the reconfiguration into commands that targeting individual nodes. We then apply each of them one by one in an ordering computed by Snowcap.

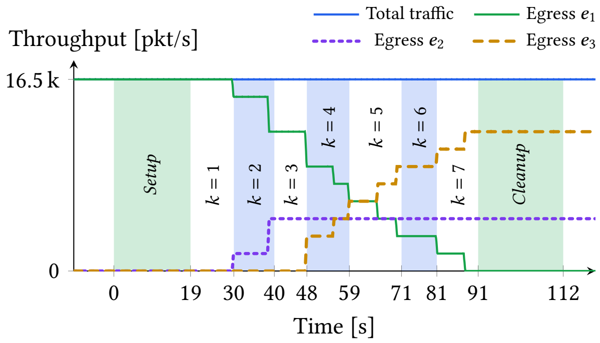


**Figure 6: Most rounds take between 10 and 12 seconds to execute. The highlighted regions visualize the setup or cleanup phase, as well as each round $k$ in the update phase. Most time is spent waiting for the routers to update their route maps. Explore the reconfiguration plan interactively at *https://bgpsim.github.io?s=abilene*.**

## 6 CASE STUDY

We demonstrate the effectiveness of Chameleon in a testbed. The testbed consists of three Cisco Nexus 7000 devices, each running four virtual routers. We connect the resulting 12 virtual routers to a Barefoot Tofino programmable switch, enabling us to emulate any network with 12 routers. We simulate the physical distances between routers using a server that delays all packets all link. We also peer the egress routers with external BGP speakers.

We use the above setup to emulate the *Abilene* network from Topology Zoo [21], a network with 11 nodes. We configure the routers to run OSPF [24] and connect them in an iBGP route reflection topology with three route reflectors. Then, we generate three external networks, each injecting routes towards 1024 individual prefixes at nodes $e_1$, $e_2$, and $e_3$. All nodes prefer routes from $e_1$ over $e_2$ and $e_3$ and decide between $e_2$ and $e_3$ based on the shortest IGP path. The emulated reconfiguration removes the BGP session between $e_1$ and its external peer, forcing all routers to change their routing decision during the reconfiguration. The network treats all prefixes identically; thus, Chameleon considers only a single prefix equivalence class.

During the reconfiguration, we require that all destinations remain reachable. Furthermore, each router must first use $e_1 = e(\mathcal{P}_{old}(n))$ as egress and then switch, only once, to its final egress $e_n = e(\mathcal{P}_{new}(n))$ (either $e_n = e_2$ or $e_n = e_3$). More precisely,

$$\phi = \bigwedge_{n \in N} \mathbf{G}\; reach(n) \land wp(n, e_1)\; \mathbf{U}\; \mathbf{G}\; wp(n, e_n). \tag{4}$$

To validate if Chameleon satisfies the specification, we inject traffic at a constant rate at each node towards $d$ and measure the egress router where they leave the network.

*Comparison with Snowcap.* Snowcap applies the reconfiguration command directly to the network as the reconfiguration affects only a single line in the configuration. Fig. 1 in the introduction shows the comparison between Chameleon and Snowcap. Snowcap' reconfiguration takes 1.7 seconds to finish. However, during this time, the network drops around 15 k packets for almost one second of traffic. Further, around 1.3 k packets violate the waypoint constraints. In contrast, Chameleon performs the reconfiguration without dropping any packets and permanently preserving waypoint requirements. Chameleon reconfigures the network in around 112 seconds, of which 72 seconds are spent in the main update phase. Fig. 6 shows Chameleon's phases and rounds. We run the same experiment on five different topologies and show the results in the Appendix (cf. Fig. 12). All experiments show a similar outcome: reconfiguring the network with Snowcap causes transient black holes and violations of waypoint constraints, while Chameleon performs all reconfigurations safely.

## 7 EVALUATION

In this section, we evaluate the overhead of Chameleon in three dimensions. First, we analyze the *scheduling time* (§7.1) and show that it typically takes Chameleon a few minutes to schedule challenging BGP reconfigurations that affect the routing state of the entire network. We additionally capture the two major factors impacting Chameleon's scheduling time with metrics that we call reconfiguration and specification complexity; we show that the former has a heavier impact than the latter. Second, we analyze the *reconfiguration time* (§7.2) and show that Chameleon can perform large reconfigurations in a few minutes while guaranteeing safety during the entire process. Third, we measure *routing table sizes* (§7.3) and compare the memory requirement of Chameleon with previous work: Snowcap does not need any extra memory, SITN roughly double, and Chameleon only around 10%.

*Implementation.* We implement Chameleon using around 7 k lines of Rust code[3] that includes the analyzer, scheduler, compiler, and runtime controller, and that supports specifications as defined in Fig. 2. We use our BGP simulator (≈ 30 k lines of Rust code) to obtain the initial and final routing state and to validate Chameleon's reconfiguration plan for experiments that are too large to run directly on our testbed. We use COIN-OR CBC [8] to solve the ILP. We run all experiments on a server with 32 CPU cores and 64 GB of memory. COIN-OR CBC is allowed to use all available cores.

*Methodology and Reconfiguration Scenario.* We use 106 topologies from Topology Zoo [21]: their size ranges from a few routers to 754. For each topology, we randomly select three different nodes $e_1$, $e_2$, and $e_3$, and connect them to three external networks. All external networks advertise the same destination $d$, and all internal nodes prefer routes received by $e_1$ over those from $e_2$ and $e_3$. We then elect three random nodes to be BGP route reflectors; each other router is a client of all three reflectors.

We choose a reconfiguration scenario that affects the routing state of all routers in the network to evaluate Chameleon in a challenging setting. Adding or removing individual BGP sessions are reported to be the most common BGP reconfigurations [36], but they rarely affect all routers in the network. Instead, we simulate adding a route map to the initially most preferred egress point $e_1$: the route map denies the route towards $d$ from its external peer, forcing all routers in the network to change their selected route during the reconfiguration.

[3] The source code is available at *https://github.com/nsg-ethz/chameleon* (GPLv2 license)

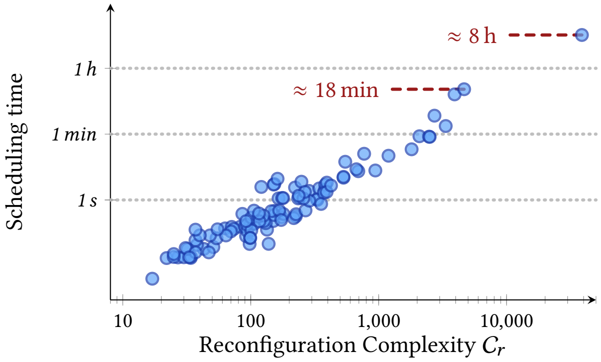


**Figure 7: This figure shows a strong correlation between reconfiguration complexity $C_r$ and the scheduling time. Chameleon usually finds a schedule in a few minutes, and up to 8 hours for the largest scenario, in which 660 routers change their next hop. A point corresponds to one of 106 scenarios, and both the $x$- and $y$-axis have a logarithmic scale.**

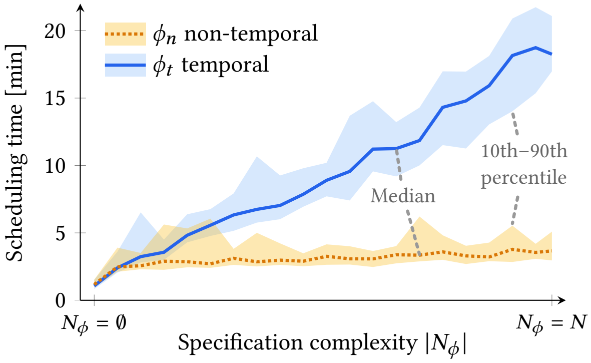


**Figure 8: Temporal expressions have a more significant impact on Chameleon's scheduling time than non-temporal constraints. This plot measures the scheduling time for the same scenario using specifications of different complexity. At $N_\phi = \emptyset$, the specification only includes reachability, while the specification at $N_\phi = N$ contains temporal (blue) or non-temporal (orange) waypoint constraints for all nodes.**

## 7.1 Scheduling Time

We first evaluate Chameleon's scheduling time. Perhaps surprisingly, the time spent solving the ILP model is not directly proportional to topological metrics such as the network size (see Table 2 in the Appendix).

We investigate the most important factors impacting Chameleon's scheduling time. We identify two orthogonal dimensions: reconfiguration complexity and specification complexity. Intuitively, the reconfiguration complexity relates to the number of nodes each router can reach during a reconfiguration, irrespective of the invariants to be preserved (i.e., the specification). In contrast, the specification complexity models the complexity of invariants to be preserved during a reconfiguration, irrespectively of the network topology, routing, and forwarding.

In the following, we separately evaluate how Chameleon's scheduling time varies with respect to reconfiguration and specification complexity. To do so, we evaluate the impact of reconfiguration complexity by keeping the same specification and simulating different reconfigurations across the 106 networks in our dataset (Figure 7). We then assess the impact of specification complexity by considering specifications of increasing complexity for the same network and reconfiguration scenario (Figure 8).

*Reconfiguration Complexity.* Some reconfigurations only affect a few nodes, while some affect nodes located far away from each other. Other scenarios, however, can update multiple nodes along a single forwarding path, thus introducing dependencies. As a result, Chameleon may schedule one scenario in a few milliseconds, while it needs seconds or even minutes to schedule a different scenario, even if both reconfigure the same network.

We measure the number of dependencies between nodes by defining the reconfiguration complexity $C_r$. Intuitively, $C_r$ counts for each node $n$ the number of different nodes that $n$ could reach during the reconfiguration. To determine $C_r$, we first construct a directed graph $G_{nh} = (N, E_{nh})$ as the union of the initial and final forwarding state $nh_{old}$ and $nh_{new}$ (as described in §4.4). For all nodes $n \in N$, $E_{nh}$ contains edges $(n, nh_{old}(n))$ and $(n, nh_{new}(n))$. Further, let $N_{nh} = \{n \in N \mid nh_{old}(n) \neq nh_{new}(n)\}$ be the set of all nodes that eventually change their next hop, and let $reachable(G_{nh}, n)$ be the set of all reachable nodes in $G_{nh}$ from $n$. The reconfiguration complexity $C_r$ is given as:

$$C_r = \sum_{n \in N_{nh}} \left| reachable(G_{nh}, n) \cap N_{nh} \right|.$$

Note that $C_r$ only depends on the initial and final forwarding states $nh_{old}$ and $nh_{new}$ and does not depend on the specification $\phi$.

Fig. 7 displays a log-log plot that shows the strong correlation between the reconfiguration complexity $C_r$ and the scheduling time of Chameleon. Each point represents one reconfiguration scenario. Chameleon finds a solution in a few minutes despite the non-trivial specification $\phi$ from Eq. (4) and the network-wide reconfiguration scenario. In the worst case, it takes our system less than 8 hours to schedule a reconfiguration in which 754 nodes update their routing state, and 660 nodes change their next hop. For the vast majority of scenarios, Chameleon finds a schedule in less than a minute.

We explain the strong correlation between $C_r$ and the scheduling time based on the semantics of constraints in our ILP. Constraints related to the specification (§4.3) and concurrent updates (§4.2) are recursive: a variable $x_k^n$ associated with node $n$ at round $k$ depends on the next hop of $n$ at round $k$. Consequently, $x_k^n$ depends on the next hop of all nodes along all possible forwarding paths of $n$ that also change their next hop, which $C_r$ precisely captures.

*Specification Complexity.* The specification language from Fig. 2 captures specifications ranging from reachability to complex waypoint constraints that change over time. We evaluate how the complexity of the specification influences the scheduling time to determine the impact of *(i)* recursive properties like waypoints and *(ii)* temporal operators.

To this end, we define two different specifications: $\phi_t$ does contain temporal operators in the form $\mathbf{U}\,\mathbf{G}$, and $\phi_n$ asserts the same properties throughout the reconfiguration. We construct $\phi_t$ and $\phi_n$ as follows: Both specifications require reachability for all nodes and waypoint constraints for *some* nodes $N_\phi$. Increasing the number of nodes in $N_\phi$ influences the complexity of the specification. Each node $n \in N_\phi$ must use either its initial egress router $e_1 = e(\mathcal{P}_{old}(n))$ or its final one $e_n = e(\mathcal{P}_{old}(n))$ (cf. §6). In addition, the temporal specification $\phi_t$ further requires that node $n \in N_\phi$ switches *once* from $e_1$ to $e_n$. More precisely,

$$\phi_n = \bigwedge_{n \in N} \mathbf{G}\, reach(n) \wedge \bigwedge_{n \in N_\phi} \mathbf{G}\left( wp(n, e_1) \vee wp(n, e_n)\right)$$

$$\phi_t = \bigwedge_{n \in N} \mathbf{G}\, reach(n) \wedge \bigwedge_{n \in N_\phi} wp(n, e_1)\, \mathbf{U}\,\mathbf{G}\, wp(n, e_n)$$

Fig. 8 shows the scheduling time for a varying number of nodes in $N_\phi$. The figure refers to reconfigurations on the *CogentCo* topology, the network with 197 nodes for which both $C_r$ and the scheduling time are the second largest of all scenarios (i.e., the scenario with a scheduling time of 18 minutes in Fig. 7). We run every experiment 20 times for each $N_\phi$ and plot the median scheduling time, as well as the 10th and 90th percentiles for both $\phi_n$ (orange) and $\phi_t$ (blue).

The figure highlights a significant difference between the non-temporal specification $\phi_n$ and the temporal one $\phi_t$: While adding more waypoints has a small impact on the scheduling time of Chameleon, increasing the number of temporal operators affects the scheduling time significantly. This disparity is because waypoint constraints are recursive; constraints for a single waypoint target are added for all nodes, even if only one node must use that waypoint target. In contrast, the temporal operators in $\phi_t$ demand variables and constraints for each additional node in $N_\phi$.

*Takeaways.* Chameleon's scheduling time depends on both reconfiguration and specification complexity, two orthogonal dimensions. However, the scheduling time is much more sensitive to the reconfiguration complexity. In fact, the specification complexity only affects scheduling time by 20× at most, while the reconfiguration complexity has an impact of over four orders of magnitude.

## 7.2 Reconfiguration Time

We show that Chameleon applies large-scale reconfigurations in a few minutes, while satisfying the specification in all transient states.

*Quantifying Reconfiguration Time.* Our testbed includes only 12 virtual routers. To estimate the reconfiguration time of Chameleon in larger networks, we run simulations.

The time for Chameleon to apply a single round depends on two factors: the time to apply the reconfiguration and the convergence time itself. From our measurements in Fig. 6, we determine that routers in our testbed take between 8 and 12 seconds to modify BGP route maps (presumably to anticipate additional changes, such that updates can be applied in batches). In contrast, the convergence time is negligible. This is because Chameleon enforces the happens-before relations: updating the selected route of one node will never change the selected route of a different node. Consequently, any BGP update will be ignored, and the convergence time depends on the distance between nodes rather than on the network size.

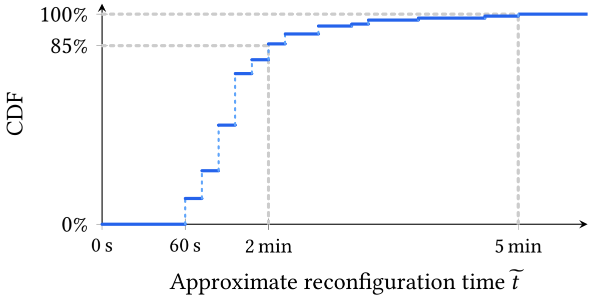


**Figure 9: The Cumulative Distribution Function (CDF) of the reconfiguration time $\widetilde{t}$ for all 106 scenarios. We approximate Chameleon to reconfigure 85% of scenarios in less than two minutes.**

We approximate the running time $\widetilde{t}$ of a reconfiguration plan by counting its number of rounds $R$, including the setup and cleanup phase. To that end, we define $\widetilde{t}_{rm}$ as the approximation of the time to apply a single round of the reconfiguration plan. We approximate the reconfiguration time as

$$\widetilde{t} = \widetilde{t}_{rm}\,(2 + R)\,.$$

The reconfiguration time heavily depends on the specific hardware. Our case study suggests $\widetilde{t}_{rm} \approx 12$ s for Cisco Nexus 7000 routers. Further, we measure no significant impact of routing table size on $\widetilde{t}_{rm}$, as we have compared $\widetilde{t}_{rm}$ for table sizes between 1 and 16 k. In the following, we use $\widetilde{t}_{rm} = 12$ s.

*Results.* We estimate the running time for all 106 scenarios and find that Chameleon's approximate reconfiguration time is less than 2 minutes for 85% of the scenarios while preserving the specification from Eq. (4). For the largest reconfiguration scenario, we estimate that Chameleon takes 5 minutes to reconfigure all 754 nodes. We provide a cumulative distribution function of the approximated reconfiguration time across all 106 scenarios in Fig. 9.

## 7.3 Routing Table Size

In the following, we compare Chameleon with related work regarding additional routing table entries needed during the reconfiguration process. On average, Chameleon replicates about 11% of the routing state to preserve invariants during the transient state, whereas related work duplicates the entire routing process to achieve similar guarantees. For Chameleon, the routing table size increases depending directly on the number of temporary BGP sessions—the scheduler can minimize the required number of temporary sessions to account for networks or routers with specific memory limitations.

*Methodology.* While simulating a reconfiguration, we measure the maximum number of routing table entries in the network at any given time during the process. We then compare this distributed routing table size of Chameleon with the baseline of directly reconfiguring the network (as Snowcap would also do) and SITN [36].

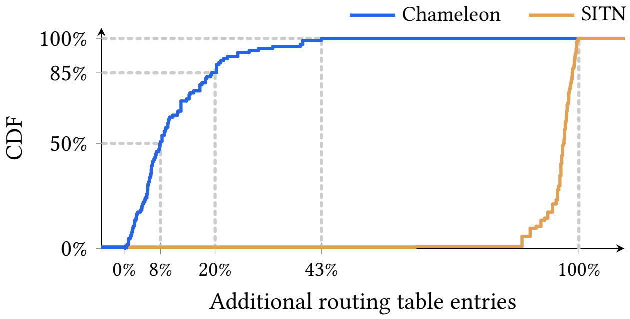


**Figure 10: In the median case, Chameleon requires 8% more routing table entries compared with the baseline, while SITN requires 96%. This figure shows the CDF of the additional routing table entries required by Chameleon and SITN, normalized by the maximum table size used for the baseline.**

*Results.* Chameleon requires, on average, 11% more routing table entries than the baseline, whereas SITN requires 96% more entries. Fig. 10 shows detailed statistics for all 106 scenarios.

We note that temporary BGP sessions exclusively cause state replication in Chameleon's reconfigurations. Hence, Chameleon can customize its reconfiguration plans according to specific memory limitations of operational networks. For example, if a given node $n$ is operating at its memory's limit, Chameleon can generate a reconfiguration plan in which $n$ only uses existing BGP sessions by including a constraint $r^n_{old} = r^n_{nh} = r^n_{new}$.

## 8 LIMITATIONS AND DISCUSSION

We now discuss the main limitations of Chameleon. They pertain to how Chameleon manages constraining specifications, dependencies between prefixes, and unplanned external events.

*Specifications.* We design Chameleon to guarantee that for each BGP destination, every router always selects its best BGP route in the initial configuration or the final one. We choose to avoid transiently leaking intermediate routes to neighboring networks, preventing unnecessary interdomain churn.

The drawback of this choice is that Chameleon may not be able to compute a safe reconfiguration for highly constrained specifications. We prove that Chameleon *always* guarantees reachability (App. B). However, it may be unable to carry out a safe reconfiguration in the presence of many waypointing invariants. This is because *no safe reconfiguration actually exists* in some settings if routers are only allowed to use their initial and final BGP routes. In those cases, Chameleon notifies the user that it cannot perform the reconfiguration safely.

We expect that in practice, Chameleon cannot perform reconfigurations rarely. In all our experiments (§7), we only encountered two examples of unsolvable specifications: they try to enforce waypointing requirements in a star-shaped topology.

We plan to extend Chameleon so that operators can trade consistency of routes announced to neighboring networks for the feasibility of highly constrained reconfigurations. This would require explicitly supporting *routing invariants*, that is, constraints on the BGP routes selected by routers during reconfigurations. Supporting routing invariants within Chameleon's design should be straightforward: in addition to updating the specification language (see Fig. 2), it would involve formulating new happens-before relations for routing invariants and translating them into ILP constraints.

*Dependencies between prefixes.* Chameleon treats all prefixes independently. If a BGP reconfiguration impacts multiple prefixes, Chameleon computes reconfiguration plans for each prefix, solving the resulting smaller problems in parallel and performing per-prefix reconfiguration concurrently. The above approach is consistent and, hence, correct whenever eBGP routes are not modified in iBGP, as is often suggested by current best practices.

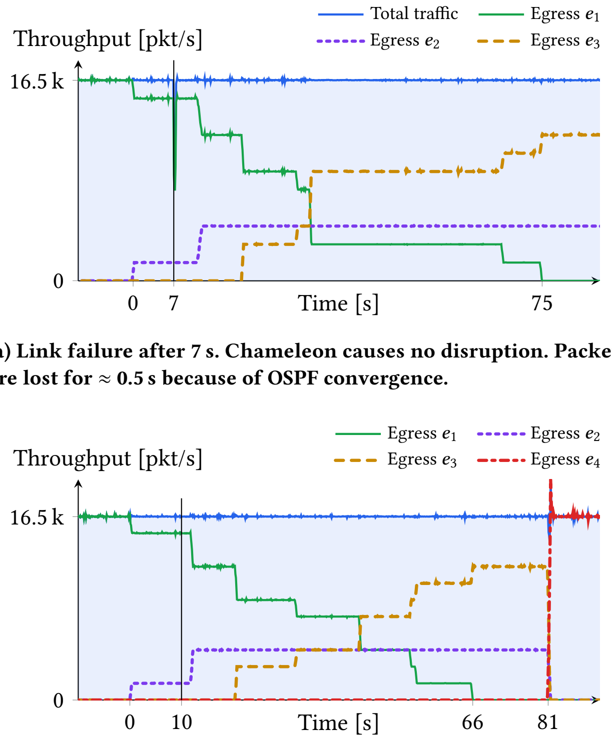
More advanced BGP configurations can, however, create dependencies between prefixes, such as route aggregation/de-aggregation or iBGP policies.

- Most BGP speakers support *route aggregation*, where nodes aggregate multiple routes into a single one and advertise the summary to neighboring routers. We note that only routes aggregated *within* a network create dependencies between prefixes. Typically, routes are aggregated only at the network border to either reduce the number of routes handled in iBGP or to announce a single eBGP route to external networks [5]. In those common cases, it is still correct to treat prefixes independently since all internal routers know either all individual routes or only the summarized ones. Similar considerations hold when prefixes are de-aggregated in sub-prefixes, for example, for traffic engineering [26].
- Routers may apply arbitrary *iBGP policies* using route maps, i.e., modifications of BGP routes when crossing internal BGP sessions. Some Internet networks do configure iBGP policies [35]. Depending on the configured route maps, this practice can create dependencies between prefixes. For example, if route maps discard a route on internal BGP sessions, different routers may forward the same packet by matching it to more or less specific prefixes, depending on which routes they receive.

Chameleon's scheduler can be extended to support these cases by jointly considering multiple prefixes. Supporting route aggregation requires generalizing our current happens-before relations. Additionally, modeling the absence of some routes at specific nodes can be encoded in the next hop computation of the ILP model.

*External Events.* Chameleon guarantees network correctness throughout the reconfiguration process in the absence of external events, such as link failures or external route changes. Stated differently, Chameleon might (transiently) violate some of the invariants if external events happen during the reconfiguration.

Because of the commands applied by Chameleon, *only the withdrawals of the best BGP routes* can break Chameleon's correctness guarantees. By definition, non-best routes are not used by any router and are, therefore, irrelevant. New best routes that appear during the reconfiguration are also not problematic since transient states installed by Chameleon make routers ignore the new routes until the end of the reconfiguration process. In contrast, Snowcap is vulnerable to all kinds of BGP events.



**(a) Link failure after 7 s. Chameleon causes no disruption. Packets are lost for ≈ 0.5 s because of OSPF convergence.**

**(b) New route announced at 10s is ignored during the update phase. At 81 s, when we restore the original preferences, all nodes converge to the new route from $e_4$.**

**Figure 11: Chameleon is resilient against many events.**

As an illustration, we simulate the effect of unplanned external events during reconfiguration in our testbed (cf. §6). Fig. 11a shows the effect of a link failure that causes OSPF to reconverge during the reconfiguration. OSPF takes around half a second to reconverge, causing routers to drop packets during that time. The effect of the link failure would have been similar if it had occurred before or after the reconfiguration. Fig. 11b shows that the network reacts to the appearance of a new, more preferred route advertised to $e_4$ only after Chameleon restores the original route preferences. That is, routers choose a sub-optimal path for about 70 seconds, but safety is preserved throughout the reconfiguration.

We envision that Chameleon's runtime controller can additionally track the network state and external events, especially BGP withdrawals, and react to them as follows:

1. If the event does not impact correctness, we can delay its effect by simply ignoring it, similar to Fig. 11.
2. If the event causes a suboptimal routing state, we can quickly compute a new reconfiguration plan to reduce the time spent in a suboptimal routing state. This is enabled by Chameleon's scheduling efficiency (cf. §7.1).
3. If the event triggers long-term anomalies, Chameleon immediately commits to the final configuration so that it can restore connectivity as quickly as possible.

## 9 RELATED WORK

An extended volume of proceeding work from academia and industry has studied the general network update problem [7]. While some operate on traditional (distributed) networks, the majority of systems [19, 23, 27] extend SDN, in which a central controller modifies the forwarding tables of all nodes. They can give strong guarantees like per packet consistency or congestion avoidance by leveraging direct control of the forwarding decision.

As for traditional network update systems, we identify two broad categories. The first category of tools [2, 18] duplicates the control and data planes on all routers, allowing them to run both the initial and the final configurations in parallel. They then gradually instruct routers to forward traffic according to the final configuration in an order that avoids forwarding loops [34, 36]. While useful, these tools tend not to be used in practice due to their overhead.

The second category of tools [6, 9, 28] performs the reconfiguration "in-place" by partitioning the configuration changes into smaller units and gradually applying them to the network while preserving correctness. Compared to the first category, "in-place" reconfiguration imposes virtually no overhead, nor does it require router support. Prior techniques like [6, 9] enable avoiding forwarding loops during IGP reconfigurations but are not applicable to BGP. Snowcap [28] is the only in-place system to reconfigure BGP. However, it provides correctness guarantees only in steady states.

The literature has proposed many models for distributed routing protocols, particularly BGP, to verify properties of the network's steady state [10, 15, 29]. Most notably, the Stable Paths Problem [14] formulates conditions for a network to converge to a unique state. However, they cannot describe *how* the network converges. In contrast, Chameleon models transient states during convergence.

Based on previous network models, many promising verification systems have emerged [1, 11, 13, 30]. These systems are complementary to Chameleon by verifying the final configuration. Similarly, network management tools [22, 31, 33] can use our system safely reconfigure the network.

Finally, research has proposed extensions to BGP [17] to guarantee the absence of forwarding loops during convergence. Unfortunately, they have seen little adoption due to additional control logic introduced to BGP speakers.

## 10 CONCLUSION

We presented Chameleon, the first system to perform BGP reconfiguration while maintaining correctness throughout the entire reconfiguration process. We proposed a framework to describe the convergence process of BGP and a technique to generate a reconfiguration plan that seamlessly transitions the network from the old to the new configuration. We demonstrated how Chameleon schedules and performs large-scale reconfigurations in real networks within a few minutes while satisfying complex specifications.

*Ethical issues.* This work does not raise any ethical issues.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and helpful feedback. The research leading to these results was supported by an ERC Starting Grant (SyNET) 851809.

# REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation.*

[2] Richard Alimi, Ye Wang, and Y. Richard Yang. 2008. Shadow Configuration as a Network Management Primitive. *SIGCOMM Comput. Commun. Rev.* 38, 4 (Aug. 2008), 111–122.

[3] Tony Bates, Enke Chen, and Ravi Chandra. 2006. *RFC 4456: Bgp route reflection: An alternative to full mesh internal bgp (ibgp).* Technical Report.

[4] Ann Bednarz. 2023. Global Microsoft cloud-service outage traced to rapid BGP router updates. (Jan. 2023). https://www.networkworld.com/article/3686531/global-microsoft-cloud-service-outage-traced-to-rapid-bgp-router-updates.html

[5] Cisco. 2013. Understand Route Aggregation in BGP. (2013). Accessed: June 2023.

[6] Francois Clad, Stefano Vissicchio, Pascal Mérindol, Pierre Francois, and Jean-Jacques Pansiot. 2014. Computing minimal update sequences for graceful router-wide reconfigurations. *IEEE/ACM Transactions on Networking* 23, 5 (2014), 1373–1386.

[7] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. 2018. Survey of consistent software-defined network updates. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1435–1461.

[8] John Forrest, Ted Ralphs, and Haroldo Gambini Santos. [n. d.]. coin-or/Cbc 2.10.8. (May [n. d.]).

[9] Pierre Francois, Mike Shand, and Olivier Bonaventure. 2007. Disruption free topology reconfiguration in OSPF networks. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications.* IEEE, 89–97.

[10] Lixin Gao and Jennifer Rexford. 2001. Stable Internet Routing without Global Coordination. *IEEE/ACM Trans. Netw.* 9, 6 (2001), 681–692.

[11] 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.* 300–313.

[12] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. 2015. Management plane analytics. In *Proceedings of the 2015 Internet Measurement Conference.* 395–408.

[13] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 958–973.

[14] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. 2002. The stable paths problem and interdomain routing. *IEEE/ACM Transactions On Networking* 10, 2 (2002), 232–243.

[15] Timothy G Griffin and Gordon Wilfong. 2002. On the correctness of IBGP configuration. *ACM SIGCOMM Computer Communication Review* 32, 4 (2002), 17–29.

[16] Igor Griva, Stephen G Nash, and Ariela Sofer. 2009. *Linear and nonlinear optimization.* Vol. 108. Siam.

[17] Nikola Gvozdiev, Brad Karp, Mark Handley, et al. 2013. LOUP: The Principles and Practice of Intra-Domain Route Dissemination. In *NSDI.* 413–426.

[18] Gonzalo Gomez Herrero and Jan Antón Bernal Van der Ven. 2011. *Network Mergers and Migrations: Junos Design and Implementation.* John Wiley & Sons.

[19] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM.* 15–26.

[20] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. 2011. The evolution of network configuration: a tale of two campuses. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference.* 499–514.

[21] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. 2011. The Internet Topology Zoo. *Selected Areas in Communications, IEEE Journal on* 29, 9 (october 2011), 1765 –1775.

[22] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic life cycle management of network configurations. In *Proceedings of the Afternoon Workshop on Self-Driving Networks.* 29–35.

[23] Ratul Mahajan and Roger Wattenhofer. 2013. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks.* 1–7.

[24] John Moy. 1998. *RFC 2328: OSPF Version 2.* Technical Report.

[25] William R Parkhurst. 2001. *Cisco BGP-4 command and configuration handbook.* Number 2969. Cisco Press.

[26] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure, and S. Uhlig. 2003. Interdomain traffic engineering with BGP. *IEEE Communications Magazine* 41, 5 (2003), 122–128. https://doi.org/10.1109/MCOM.2003.1200112

[27] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. 2011. Consistent updates for software-defined networks: Change you can believe in!. In *Proceedings of the 10th ACM workshop on hot topics in networks.* 1–6.

[28] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. 2021. Snowcap: Synthesizing Network-Wide Configuration Updates. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21).* New York, NY, USA, 33–49.

[29] Joao Luis Sobrinho. 2003. Network routing with path vector protocols: Theory and applications. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications.* 49–60.

[30] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. 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.* 750–764.

[31] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A network-state management service. In *Proceedings of the 2014 ACM Conference on SIGCOMM.* 563–574.

[32] Yu-Wei Eric Sung, Sanjay Rao, Subhabrata Sen, and Stephen Leggett. 2009. Extracting network-wide correlated changes from longitudinal configuration data. In *Passive and Active Network Measurement: 10th International Conference, PAM 2009, Seoul, Korea, April 1-3, 2009. Proceedings 10.* Springer, 111–121.

[33] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. 2016. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference.* 426–439.

[34] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. 2011. Seamless network-wide IGP migrations. In *Proceedings of the ACM SIGCOMM 2011 Conference.* 314–325.

[35] Stefano Vissicchio, Luca Cittadini, and Giuseppe Di Battista. 2015. On IBGP Routing Policies. *IEEE/ACM Trans. Netw.* 23, 1 (feb 2015), 227–240.

[36] Stefano Vissicchio, Laurent Vanbever, Cristel Pelsser, Luca Cittadini, Pierre Francois, and Olivier Bonaventure. 2012. Improving network agility with seamless BGP reconfigurations. *IEEE/ACM Transactions on Networking* 21, 3 (2012), 990–1002.

| *Topology* | $\lvert N\rvert$ | $C_r$ | *scheduling time* |
|---|---|---|---|
| *Deltacom* | 113 | 1810 | 23.4 s |
| *Ion* | 125 | 2731 | 189.0 s |
| *Pern* | 127 | 162 | 3.7 s |
| *TataNld* | 145 | 3343 | 99.2 s |
| *Colt* | 153 | 1194 | 15.6 s |
| *UsCarrier* | 158 | 2505 | 50.5 s |
| *Cogentco* | 197 | 4657 | 978.4 s |

**Table 2: The reconfiguration complexity $C_r$ shows stronger correlation with scheduling time than the number of nodes in the network.**

# APPENDIX

In the following, we provide supporting material that has not been peer-reviewed.

## A NETWORK SIZE VS. RECONFIGURATION COMPLEXITY

Table 2 shows the network size and reconfiguration complexity for seven different topologies from TopologyZoo, together with Chameleon's time to schedule the same reconfiguration as in §7.1 (Table 2 shares the same data as Fig. 7). We find stronger correlation between the reconfiguration complexity $C$ and scheduling time than with the number of nodes $|N|$ in the network. For instance, the topology *Pern* has more nodes than *Ion*, but its scheduling time is smaller by almost two orders of magnitude. However, *Ion* has a reconfiguration complexity that is around 20 times larger than the one of *Pern*.

## B EXISTENCE OF A SOLUTION

The scheduler presented in §4 only allows a router to change its forwarding decision once from the initial to the final next hop. In the following, we prove that this technique is sufficient for maintaining reachability throughout the reconfiguration.

Theorem 1. *Let $N$ be a set of nodes and $\phi = \bigwedge_{n \in N} reach(n)$ be the specification. Further, let $nh_{old}$ and $nh_{new} : N \mapsto N \cup \{\emptyset, d\}$ be the initial and final forwarding state. Let $s : N \mapsto \{1, \ldots, R\}$ be a node schedule, and let $nh_k$ be the forwarding state at round $k$ according to $s$. If $nh_{old} \models \phi$ and $nh_{new} \models \phi$, then there exists a node schedule $s$ such such that $\forall k \in \{1, \ldots, R\} : nh_k \models \phi$.*

Proof. We proof Theorem 1 by constructing a $s$, such that $\forall k : nh_k \models \phi$. We do so using a breath-first traversal of $nh_{new}$, as shown in Alg. 1. For each round $k$, the algorithm keeps a set of nodes $N_k \subseteq N \cup \{d\}$ have updated its state at round $k$, i.e., $\forall n \in N_k, s(n) \leq k$. In each round $k$, the algorithm picks one node $n \in N_x$ that has $n \notin N_{k-1}$ and $nh_{new}(n) \in N_k$ to migrate at this round $s(n) = k$.

Notice, that $N_k$ is always constructed from $N_{k-1}$ by adding a single node $n \in N_x$ for which $nh_{new}(n) \in N_{k-1}$. Therefore, at each step $k$, nodes within $N_k$ form a tree rooted at $d$ within $nh_k$. Thus, in every round $k$, $\forall n \in N_k : nh_{new}(n) \in N_k$, and that all nodes $n \in N_k$ eventually reach $d$ using $nh_k$.

**Algorithm 1** Building a schedule $s$ for transitioning from $nh_{old}$ to $nh_{new}$. This algorithm is used to proof Theorem 1.

```
 1: procedure Solve($N$, $nh_{old}$, $nh_{new}$)
 2:     $\phi \leftarrow \mathbf{G} \bigwedge_{n \in N} reach(n)$
 3:     $\forall n \in N : s(n) \leftarrow 0$
 4:     $k \leftarrow 1$
 5:     $N_k = \{d\}$                         ▷ Set of nodes already updated.
 6:     $N_x \leftarrow \{n \in N \setminus N_k \mid nh_{new}(n) \in N_k\}$
 7:     while $N_x \neq \emptyset$ do
 8:         $n \leftarrow pick\ N_x$
 9:         $s(n) \leftarrow k$
10:         $k \leftarrow k + 1$
11:         $N_k \leftarrow N_{k-1} \cup \{n\}$                 ▷ $N_k \leftarrow \{\}$
12:         $N_x \leftarrow \{n \in N \setminus N_k \mid nh_{new}(n) \in N_k\}$
13:     end while
14:     return $s$
15: end procedure
```

Next, we show that we eventually schedule all nodes, i.e., that $\forall n \in N, s(n) > 0$. For the sake of contradiction, let $N^*$ be the set of nodes for which $s(n) = 0$. Since $nh_{new} \models \phi$, each node $n \in N^*$ can reach $d$ in the final state. Let $n \in N^*$ be the node with the shortest path towards $d$ in $nh_{new}$. Since $s(n) = 0$ we know that $n \notin N_x$ at Line 14. However, this is impossible since $nh_{new}(n) \in N_k$ as $n$ is the node in $N^*$ with the shortest path towards $d$.

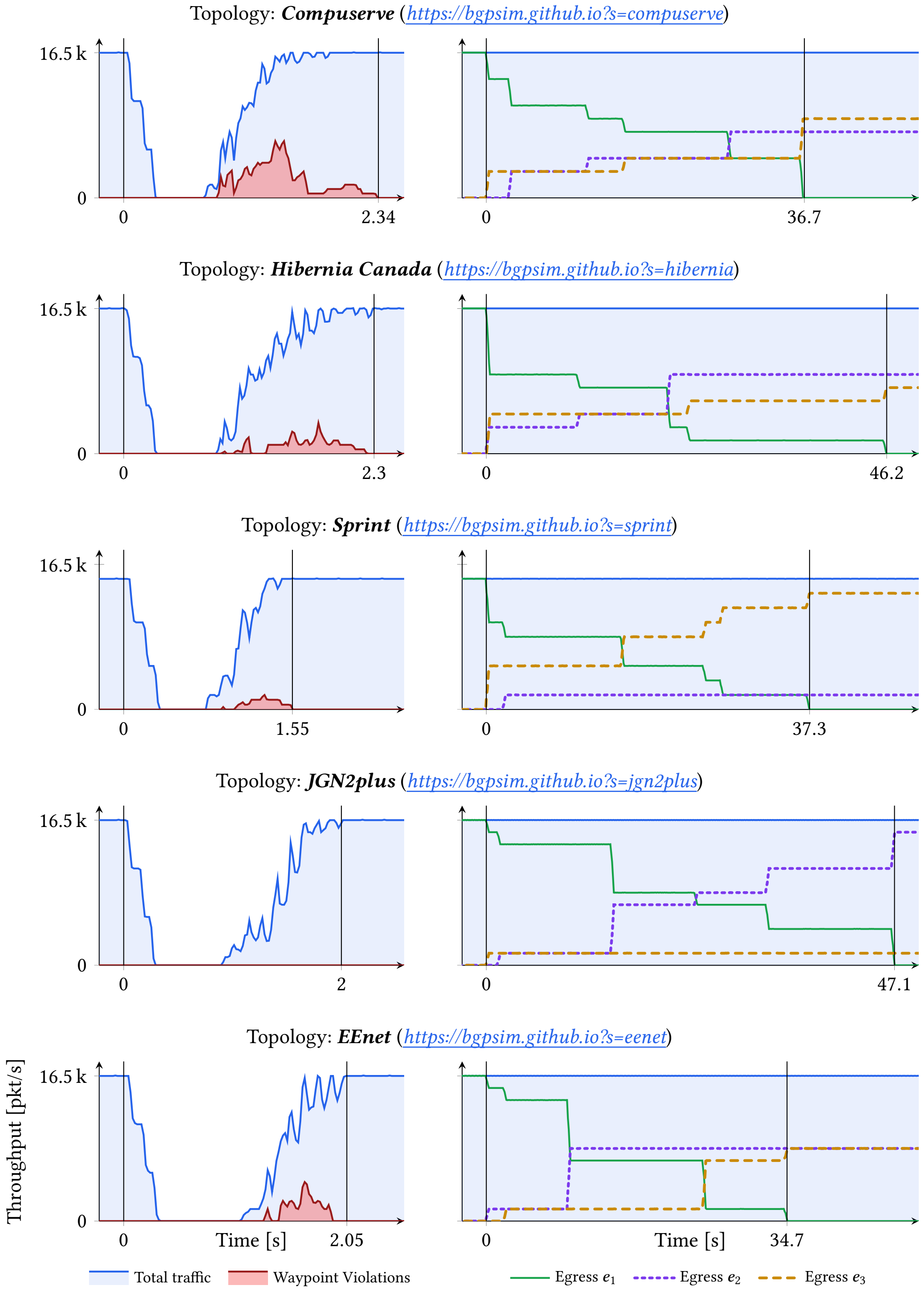
We now prove that $\forall k : nh_k \models \phi$. To that end, consider any node $n \in N$ and any round $k$. Let $N_k = \{n \in N \mid s(n) \leq k\}$ be the set of all nodes already updated in round $k$. We have already shown that all nodes in $N_k$ eventually reach $d$ in $nh_k$. Thus, let $n \notin N_k$, and let $p$ be the path of $n$ in $nh_k$. If $p \cup N_k = \emptyset$, then $p$ is the same as the path of $n$ in $nh_{old}$, which must end at the destination since $nh_{old} \models \phi$. Otherwise, if $p \cup N_k \neq \emptyset$, then the path $p$ eventually enters $N_k$, and thus, $nh_k \models \phi$. This concludes the proof of Theorem 1. □

## C SUPPLEMENTARY CASE STUDIES

The case study in §6 only considers the *Abilene* network from Topology Zoo [21]. We also run the same scenario for 5 additional topologies: *Compuserve*, *Hibernia Canada*, *Sprint*, *JGN2plus*, and *EEnet* that contain 11 or 12 internal routers.

Similar to §6, we configure each topology with random OSPF link weights, and we elect three random routers as BGP route reflectors. We simulate three external networks $e_1$, $e_2$, $e_3$, advertising the same 1024 prefixes to three internal routers, while the routes towards $e_1$ have a shorter AS path length than the ones from $e_2$ and $e_3$. The reconfiguration involves removing the eBGP session towards $e_1$, causing all routers to select either the routes from $e_2$ or $e_3$, depending on the shorter IGP path. We use the specification from Eq. (4) that requires reachability and contains temporal waypoints.

We reconfigure the network both with Snowcap [28] (which simply applies the reconfiguration command) and Chameleon. We find that Snowcap causes black holes for one to two seconds in all scenarios. Snowcap additionally violates waypoint invariants in four out of five scenarios. Chameleon, however, reconfigures the networks while satisfying the specification. It takes Chameleon less than 1 minute to perform the reconfiguration in all five scenarios.



Topology: ***Compuserve*** (*https://bgpsim.github.io?s=compuserve*)

Topology: ***Hibernia Canada*** (*https://bgpsim.github.io?s=hibernia*)

Topology: ***Sprint*** (*https://bgpsim.github.io?s=sprint*)

Topology: ***JGN2plus*** (*https://bgpsim.github.io?s=jgn2plus*)

Topology: ***EEnet*** (*https://bgpsim.github.io?s=eenet*)

**Figure 12: Chameleon consistently and safely reconfigures a network, while Snowcap [28] triggers black holes, and violates waypoint specifications. We repeat a similar experiment to Fig. 1 on five additional topologies from Topology Zoo [21], all containing 11 or 12 routers. The reconfiguration causes routers to change their selected route from $e_1$ to either $e_2$ or $e_3$. The left column shows the total traffic (and violations) for Snowcap, while the right shows the throughput for Chameleon**

# D ABSENCE OF LOOPS

In the following, we proof that the constraints from Eq. (2) also ensure the absence of forwarding loops in all rounds. We further evaluate the impact of the explicit constraints from Eq. (3) on the scheduling time of Chameleon.

Theorem 2. *Let $N$ be a set of nodes (routers), and let $nh_{old} : N \mapsto N \cup \{\emptyset, D\}$ and $nh_{new} : N \mapsto N \cup \{\emptyset, D\}$ be the initial and final forwarding state. For each node $n \in N$, let $r^n_{nh}$ be the round in which $n$ changes from its old next hop $nh_{old}(n)$ to $nh_{new}(n)$. If $nh_{old}$ is free of forwarding loops, and if Eq. (2) is satisfied for each node $n$ and round $k$ (i.e., all updates of the same round are independent), then the resulting forwarding state in each round $k$ is free of forwarding loops.*

Proof. We prove this statement by contradiction. Let $k$ be the *first* round in which a forwarding loop $L = (n_1, \ldots, n_i, \ldots, n_l, n_1)$ appears. Further, let $n_i \in L$ be a node in the loop that changes its next hop to $n_{i+1}$ in round $k$. Node $n_i$ must exist, since round $k-1$ is loop-free. Further, since all updates in round $k$ are independent, no other node $n_j \in L \setminus \{n_i\}$ updates its next hop in round $k$. Thus, $\delta^{n_j}_k = \delta^{n_{j+1}}_k$ for all $j \neq i$. Repeatedly applying Eq. (2) yields

$$\delta^{n_i}_k \geq 1 + \delta^{n_{i+1}}_k = 1 + \delta^{n_{i+2}}_k = \ldots = 1 + \delta^{n_i}_k .$$

This is equivalent to $0 \geq 1$, and, by contradiction, proves that Eq. (2) ensures the absence of forwarding loops. □

Even though the constraints from Eq. (2) implicitly ensure loop-freedom, we explicitly add constraints as described in §4.4. We do this because we notice the variance of the scheduling time decreasing when using explicit loop constraints. Fig. 13 shows the difference of solving the same problem with and without explicit constraints. In the most extreme case, the scheduling time with implicit constraints is around 40× larger than solving the same problem with explicit constraints.

# E WEB SIMULATOR TUTORIAL

Along with this publication, we provide an application to visualize Chameleon's reconfiguration plans. The simulator is available directly in the browser (*https://bgpsim.github.io*). The application simulates BGP events, visualizes the current (transient) state, and verifies forwarding policies. We prepare Chameleon's reconfiguration plan for both the running example and the case study:

- The example (cf. Fig. 3) without Chameleon: *https://bgpsim.github.io?s=example-baseline*.
- The example (cf. Fig. 3) with a reconfiguration plan: *https://bgpsim.github.io?s=example*.
- The case study (cf. §6) without Chameleon: *https://bgpsim.github.io?s=abilene-baseline*.
- The case study (cf. §6) with a reconfiguration plan: *https://bgpsim.github.io?s=abilene*.

## E.1 Visualization

The application visualizes the network using four different layers, two of which show the current state of the network while the other two show the configuration. The button 1 on the top left switches between those layers.

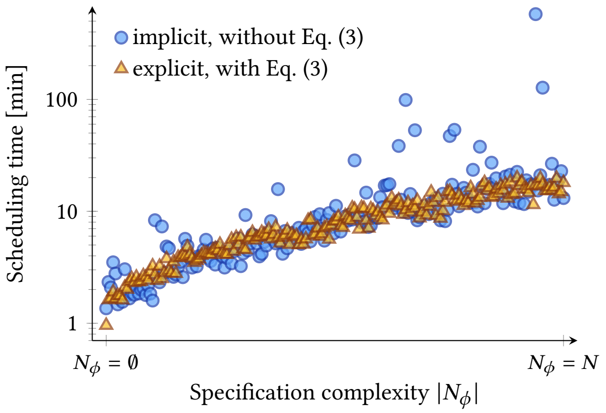


**Figure 13: Adding explicit constraints to avoid forwarding loops with Eq. (3) significantly reduces the variance of the scheduling time of Chameleon. This plot shows the scheduling time of Chameleon for the same scenario using different specifications arranged on the *x*-axis (as for Fig. 8). The red points show the scheduling time without explicit loop constraints, while the green points show the scheduling time with explicit constraints. The *y*-axis has a logarithmic scale.**

The *Data Plane* layer displays the forwarding state. Hovering over nodes highlights their forwarding path to reach the destination. Similarly, the *Control Plane* layer visualizes the propagation of BGP routes as arrows. Hovering over a yellow arrow will display the propagated route attributes, while hovering over a node will show the current routing table of that node, along with its selected route.

On the other hand, both the *IGP Config* and *BGP Config* layers visualize parts of the current configuration. Selecting a node will show its complete BGP configuration on the right side of the screen.

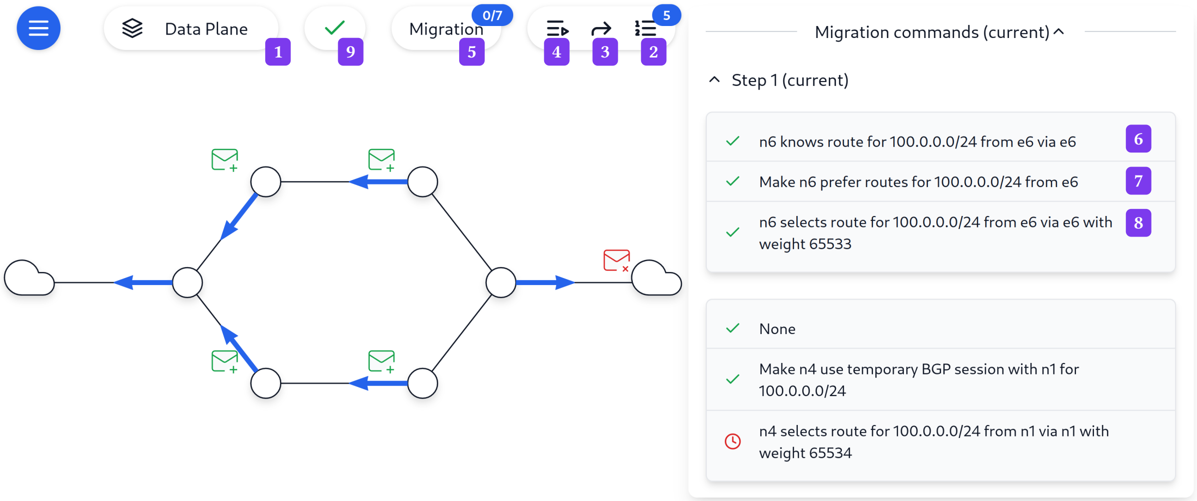
## E.2 Event-Based Simulation

The application is based on an event-based simulator. Each BGP message is an event, enqueued in a FIFO queue. Once executed, the event has an immediate effect and might enqueue new events. The simulator does not execute events automatically, but when the user clicks on 3. Clicking the button on the top right 2 displays all currently enqueued events on the right side. This list allows the user to trigger an event explicitly. Finally, 4 executes the entire queue until a violation occurs or the queue is empty.

## E.3 Reconfiguration Plan

Clicking the button 5 shows the current reconfiguration plan on the right, grouped into the three phases (collapsed by default). Expanding a phase (and a round of the update phase) reveals the pre-condition 6, the command 7, and its post-condition 8. The left side indicates whether the conditions are satisfied and if the command is already applied. Clicking on a command will execute it, assuming that the pre-conditions are satisfied.

The application continuously verifies the conditions. The current status is shown left of the reconfiguration plan 9 Clicking the icon reveals a list of all satisfied or violated invariants.



Migration commands (current)

Step 1 (current)

- ✓ n6 knows route for 100.0.0.0/24 from e6 via e6 6
- ✓ Make n6 prefer routes for 100.0.0.0/24 from e6 7
- ✓ n6 selects route for 100.0.0.0/24 from e6 via e6 with weight 65533 8

- ✓ None
- ✓ Make n4 use temporary BGP session with n1 for 100.0.0.0/24
- 🕓 n4 selects route for 100.0.0.0/24 from n1 via n1 with weight 65534

**Figure 14: Screenshot of the web application (*https://bgpsim.github.io*) to visualize a reconfiguration plan.**