Relational Network Verification

Xieyang Xu[w] Yifei Yuan[a] Zachary Kincaid[p] Arvind Krishnamurthy[w]
Ratul Mahajan[w] David Walker[p] Ennan Zhai[a]

[w]University of Washington [a]Alibaba Cloud [p]Princeton University

ABSTRACT

Relational network verification is a new approach for validating network changes. In contrast to traditional network verification, which analyzes specifications for a single network snapshot, it analyzes specifications that capture similarities and differences between two network snapshots (e.g., pre- and post-change snapshots). Relational specifications are compact and precise because they focus on the flows and paths that change between snapshots and then simply mandate that all other network behaviors "stay the same", without enumerating them. To achieve similar guarantees, single-snapshot specifications would need to enumerate all flow and path behaviors that are not expected to change in order to enable checking that nothing has accidentally changed. Such specifications are proportional to network size, which makes them impractical to generate for many real-world networks.

We demonstrate the value of relational reasoning by developing Rela, a high-level relational specification language and verification tool for network changes. Rela compiles input specifications and network snapshot representations to finite state automata, and it then verifies compliance by checking automaton equivalence. Our experiments using data from a global backbone with over 10^3 routers find that Rela specifications need fewer than 10 terms for 93% of the complex, high-risk changes. Rela validates 80% of the changes within 20 minutes.

CCS CONCEPTS

• **Networks** → **Network reliability**; **Network management**; • **Theory of computation** → **Regular languages**; **Automated reasoning**;

KEYWORDS

Network verification, domain-specific language, relational specification, regular language, network changes, reliability

ACM Reference Format:
Xieyang Xu, Yifei Yuan, Zachary Kincaid, Arvind Krishnamurthy, Ratul Mahajan, David Walker, and Ennan Zhai. 2024. Relational Network Verification. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3651890.3672238

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08.
https://doi.org/10.1145/3651890.3672238

1 INTRODUCTION

Changing a running network, for instance, to alter its security posture, optimize resource usage, or add capacity, is one of the riskiest network management activities today. Outages can occur during changes because of incorrect change implementation (e.g., accidentally blocking traffic) or latent bugs (e.g., traffic starts traversing a longstanding filter). When changes go wrong, banks go offline, airlines stop flying, emergency services become unreachable, and businesses lose millions of dollars [4, 27–29, 31, 32, 35]. Since changing a network is unavoidable, we must make changes safer to make networks more reliable.

The last decade has seen remarkable progress toward verification technologies that can reason about large, real-world networks. These technologies typically tell a user whether a *single* network snapshot N satisfies specification S. The snapshot may represent an updated network configuration that engineers wish to deploy, and the specification may demand that DNS traffic is never blocked or that external traffic always traverses a firewall before reaching the high-security zone. Indeed, many large networks use these technologies today [5, 13, 21, 38].

Single-snapshot verification tools, while valuable, do not suffice for keeping networks running reliably as they are updated. Consider a common network change that moves all traffic on link A to link B as a precursor to shutting A for maintenance. To validate this change, the engineer would want to ensure that all traffic on link A is moved, that it is moved to link B and nowhere else, and that no other traffic is impacted. Using single-snapshot verification for this validation requires that the engineer (1) discover all traffic classes on link A, (2) create a specification asserting that the discovered traffic classes traverse link B in the new network, (3) discover *all other traffic classes* and *all their current paths*, exactly, (4) create a specification asserting all such other traffic classes continue to follow these discovered paths.

Unless the network is configured using high-level intents (e.g., Robotron [30]), which is rare today [13], creating such specifications is almost impossible. One challenge is *scale*: The specification needed scales with the size of the network, and modern networks are enormous and continue to grow. The network in our experiments has on the order of 10^3 routers, 10^4 routes per router, and 10^6 classes of flows with distinct forwarding paths, with up to 10^4 classes impacted by typical changes. Then there is an additional challenge of *incomplete information*: Networks evolve organically over the years, and their size and complexity mean an engineer may have only partial knowledge of a network's behavior. Creating precise, detailed specifications in these circumstances and maintaining them through successive changes requires otherworldly effort.

The upshot is that while single-snapshot verification helps ensure coarse, long-term invariants, it is not helpful when it comes to the fine details of many network updates. Yet network engineers must check such details to prevent congestion-induced outages, security breaches, or performance issues. Lacking appropriate tools, network engineers today rely on manually inspecting the impact of changes. Unsurprisingly, manual inspection is time-consuming, tedious, and error-prone, sometimes taking weeks to check even simple-seeming changes. See §2 for an example.

We introduce *relational network verification* and investigate if it can make network changes more reliable, more efficient, and less dependent on manual audits. Rather than reasoning about the behavior of a *single* snapshot in isolation, relational network verification reasons about the similarities and differences (i.e., the *relationships*) in the behavior of *two* network snapshots.

Relational specifications make it easy to specify "no change" for the behaviors that engineers do not want to modify (and may not even know about). Indeed, the size of a relational network specification is proportional to the complexity of the change rather than that of the network as a whole. If a desired network change is small (e.g., changing link A to link B), the relational specification will also be small. It is no wonder that engineers already informally use such ideas to specify intent in change request tickets. In a sense, relational specifications formally capture the kind of thinking that engineers use, but in a way that enables automatic checking.

Realizing relational network verification requires (1) a language to compactly specify the intent of a network change, and (2) a decision procedure to verify that the pre- and post-change network snapshots adhere to the specification. We develop a tool called Rela with these capabilities. Network engineers use a new high-level source language to specify change intent, such as adding or removing parts of a path. Rela compiles this user-friendly language to a new low-level, regular intermediate representation (RIR). RIR is a general language for describing regular languages and relations [20] and can encode specifications for a wide range of network changes. Rela combines the generated RIR with data from the pre- and post-change network snapshots, and it checks that the pair of snapshots satisfies the RIR specification by reducing the problem to equivalence-checking for finite state automata. The final result is either a "thumbs up" (if the network satisfies the specification) or a set of counterexample flows and paths.

We evaluate Rela using all complex, high-risk changes to a global backbone network over the last seven months. These changes were involved enough that each was reviewed (manually) by a committee of experts. Despite that, Rela specs are compact—93% of the changes need fewer than 10 terms in the language. And even though the network has over 10^3 routers, validation takes under 20 minutes for 80% of the changes. We are currently integrating Rela into the change pipeline of this network.

The primary contribution of this paper is to consider network verification in the context of a new kind of semantic model, which analyzes two networks simultaneously rather than a single network. It develops a high-level language with simple abstractions that are useful for expressing network changes. It also develops a specialized language (RIR) for describing relational specifications in terms of classical operations on finite automata and transducers. Although individual components of the language are well understood, the challenge was identifying the combination of features necessary for describing network changes. Our decision procedure, which we show can scalably reason about network paths and path changes, is based on a novel reduction of change specifications to automata-theoretic machinery.

2 NETWORK CHANGES TODAY

Implementing network changes requires translating high-level change intents into low-level device configuration changes. Unfortunately, errors in this translation are common. Using a change from Alibaba Cloud's backbone, we illustrate the difficulty of making even seemingly simple changes and how incomplete information and scale limit the effectiveness of existing network analysis tools.

2.1 An Example Change

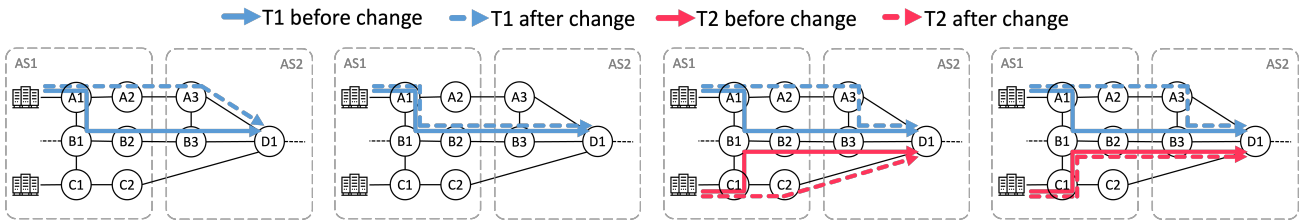
Figure 1a shows a change in the global backbone of Alibaba Cloud. The part of the backbone shown has two BGP autonomous systems, $AS1$ and $AS2$, each enclosed by a grey box and comprising many routers. Each circle denotes a group of routers that fulfill the same functionality. An AS spans multiple geographic regions, encoded using the prefix letter of router groups. So, $A1$ and $A2$ are in the same region, which is different from that of $B1$ and $B2$.

The goal of the change is to prevent $T1$, which goes from region A to region D, from traversing region B. That is, all traffic on the path $A1$-$B1$-$B2$-$B3$-$D1$ (solid line) should move to $A1$-$A2$-$A3$-$D1$ (dotted line). Importantly, no other WAN traffic should be impacted. Despite the simplicity of this abstract picture, it took network engineers *four iterations across three weeks* to devise a working implementation of the change.

First iteration. The engineers' first iteration (Figure 1b) changed the configuration of $A2$ routers. They added $T1$ prefixes to an allow-list on $A2$, with the hope that $A1$ would pick the shorter path $A1$-$A2$ over $A1$-$B1$-$B2$. However, on inspecting the impact of the change (using the process in §2.3), the engineers found it ineffective: The $T1$ traffic followed the same path as before! Investigation revealed that the routers in region B were configured to announce T1 prefixes with a high local preference. Since local preference overrides path length in BGP, $A1$ continued to prefer the route through $B1$ over $A2$. This failure illustrates the challenge of *incomplete information*: The engineers who implemented this configuration change on $A2$ were not necessarily familiar with configuration details for $B2$ routers, as each configuration is accumulated from years of changes by different individuals.

Second iteration. The engineers' second iteration (Figure 1c) reconfigured $A2$ to increase the local preference of $T1$ prefixes. As a fail-safe, they also configured routers in region B to lower the local preference for these prefixes. This time, the engineers observed that $T1$ had indeed moved from $B2$ to $A2$. However, it turned out that the implementation caused *collateral damage*: the path of traffic T2, which should not have been impacted, changed. Debugging revealed that the root cause was a typo in the import policy at $B2$.

Third iteration. The next iteration (Figure 1d) fixed the typo. Upon testing, the engineers saw that it fixed the collateral damage but found another issue. While T1 traffic had indeed moved away from



(a) Change intent (b) v1: Unchanged path (c) v2: Collateral damage (d) v3: Still incorrect path

Figure 1: An example network change in a global WAN. T1 and T2 denote aggregate traffic bundles.

$B2$, it was bouncing back to $B3$ due to an old configuration bug that made the link costs of $A3-B3-D1$ lower than those of $A3-D1$. This undesirable behavior was present in the previous iteration as well, but the engineers missed it amidst the information overload created by the collateral damage. This failure illustrates the challenge of *scale*: It is not enough to focus on one small set of paths because even small configuration changes can impact many paths at once.

Fourth iteration. The fourth iteration finally achieved the intended behavior after three weeks of labor.

Changes like this one are common in the backbone's daily operation. While some errors are caught prior to deployment, some make it to the network and have a widespread impact.

2.2 Just Verify It?

Readers familiar with network verification might ask: Do the backbone network engineers have access to a verification tool; and does it help find errors in changes? The answers are: Yes, they have a verification tool [37]; and while it does flag some types of errors, it does not uncover the types of errors above. We explain why.

Abstractly, existing network verification methods check whether a network configuration C satisfies a specification S. We call this method *single-snapshot verification* because it analyzes a single network configuration against the specification. Typically, one analyzes the new (post-change) network configuration, and the original (pre-change) network configuration is not used.

Single-snapshot verifiers are used to check coarse properties that hold over long periods of time, such as "never block DNS traffic" and "always block ssh from outside." These verifiers can validate that such properties are not violated by a proposed change. However, to catch finer-grained problems, such as the problems with specific paths (described in the previous subsection), finer-grained specifications are needed. Unfortunately, with 10^6 traffic classes, creating a detailed specification for all of them is an insurmountable barrier to even getting started with single-snapshot network verification. Said differently, the cost of creating network-wide single-snapshot specifications is proportional to the size of the network. To make verification worthwhile, we need to create specifications at a lower cost, ideally proportional to the size of the change.

A naive tactic is to generate small but highly incomplete single-snapshot specifications. For instance, if a network engineer wishes to replace a path $P1$ with $P2$, they might verify that $P2$ exists in the new network and $P1$ does not. But this tactic omits a key property: all other traffic should remain unchanged, and hence does not help identify accidental collateral damage.

Thus, while single-snapshot verification has a role in ensuring network reliability, it is insufficient for precise change validation. So the backbone's network engineers resort to manual inspection, which we discuss next.

2.3 Back to Manual Inspection

The primary change validation method that engineers use today is manual inspection. Its workflow is:

(1) use a simulator [18, 37] to compute the network's forwarding state $N1$, based on the current configuration;
(2) compute the network's after-change forwarding state $N2$, based on planned changes to the configuration;
(3) use $N1$ and $N2$ to compute the before- and after-change forwarding paths for all flows that traversed the network over the last hour;[1] a *flow* is a packet that starts at a particular point in the network;
(4) aggregate flows into *flow equivalence classes* that have flows with identical paths in the current configuration and in the planned configuration;
(5) manually inspect the *path diff*, which contains all equivalence classes whose paths differ for the two configurations, and check that all expected changes have occurred and no unexpected changes have occurred.

Manual auditing is tedious and subject to human error. The path diffs can have anywhere from tens to tens of thousands of differences. Experienced engineers can audit only tens of classes per day, which makes a complete audit intractable for some changes. Engineers may then resort to sampling, increasing the risk of missing problems. Further, while it is relatively easy (but still hard) to spot undesired path changes by inspecting the path diff, ensuring that all desired path changes occur is harder.

3 A NEW APPROACH: RELATIONAL NETWORK VERIFICATION

Relational network verification is inspired by today's manual approach and relational program verification research from the formal methods community (see Barthe [6] for an introduction). Relational methods reason about similarities and differences in *two* versions of

[1] NetFlow [33] monitoring provides this data. Engineers prefer it over considering all possible flows (i.e., symbolic analysis) because it reduces information they need to inspect and helps focus on flows that matter.

a system, rather than considering one version in isolation. Because network changes involve two network configurations, one old and one new, these methods naturally apply.

Relational verification can be more effective at validating network changes than single-snapshot verification when changes have precise and compact specifications, even if the network itself is enormous and the change moves many flows. Compact specifications are possible because network changes often involve a small number of path (not necessarily flow) changes. To specify that a path P_1 should be replaced by P_2, a relational specification can declare that (a subset of) flows traversing P_1 in the old network should traverse P_2 in the new network and that all other traffic should follow the same path(s) in both networks. This change intent can be expressed in just a few lines of code in part because specifying "no change" (i.e., old equals new) is trivial with relational methods, though next to impossible with single-snapshot specifications.

4 RELA BY EXAMPLE

Rela's specifications describe the relationship between the forwarding behavior of two network snapshots. It focuses on networks with stateless forwarding, the same context that was targeted by the first wave of single-snapshot verification tools [10]. We defer extending Rela to networks with richer forwarding and to non-forwarding behavior changes (e.g., route attributes) to future work.

This section introduces Rela using the change in §2.1. The next section formalizes its syntax and semantics. Recall that the intent of the change in §2.1 has three elements: (1) only impact the traffic from region A to D that traverses $A1$ and $D1$; (2) change the forwarding sub-paths of this traffic from $A1$-$B1$-$B2$-$B3$-$D1$ to $A1$-$A2$-$A3$-$D1$, while preserving the sub-paths before $A1$ and after $D1$; (3) no other traffic should be impacted.

Change Zones. In Rela, the first step in defining a change intent is to define the *change zone*. Informally, change zones allows users to create a focus area for the impact of a change and ignore behaviors outside of that focus. Users define change zones using *path patterns*, which are regular expressions over network locations.

A *network location* identifies one hop in a forwarding path. In Rela, forwarding paths and locations can be viewed at different levels of granularity, including at the interface level, the router level and the router group level. Users choose the level that suits their needs. Our example uses router-level locations because we do not care which interfaces are used for forwarding as long as they belong to the correct router.

Rela is used in concert with a database that stores information about all locations available in the network. Users can refer to a set of locations within the same entity (such as a router group or a tier) by issuing "where" queries to select locations from the database and return the union of them. We define below `a1` and `d1` as, respectively, sets of routers with group attributes $A1$ and $D1$.

```
regex a1 := where(group=="A1")
regex d1 := where(group=="D1")
```

Regular expressions `a1` and `d1` can now be used in the rest of the Rela specification. For instance, the regex `a1.*d1` denotes the set of paths that starts from any location in $A1$ and ends at any location in $D1$ after traversing zero or more (any) intermediate locations.

Change specifications. An atomic *change specification* is written *zone* : *modifier*. Roughly speaking, such a specification indicates that paths in the zone should be changed according to the modifier. When desired, such specifications may be named as follows for reuse and composition with other change specifications.

```
spec name := { zone : modifer; }
```

Path modifiers describe the sets of paths to add, remove, replace, or preserve between old and new network snapshots. For example, the following code presents one implementation of the second element of our example change intent.

```
spec pathRepl := {
    a1.*d1 : replace(a1b1b2b3d1, a1a2a3d1);
}
```

This spec has zone `a1.*d1`, which cares about the subset of pre-change forwarding paths that start from `a1` and end in `d1`. Its modifier demands that if one or more paths in the zone are matched by regex `a1b1b2b3d1`, then all paths in regular path set `a1a2a3d1` should appear in the new network (assuming symbols `a2`, `b1`, etc., have all been defined earlier as the union of routers in the corresponding router group). The semantics of **`replace`** also demands that (1) if any path in regular path set `a1a2a3d1` appears in the old network, it continues to appear in the new network, and (2) paths that belong to zone `a1.*d1` but not belongs to regex `a1b1b2b3d1` should remain the same throughout the change.

The **`replace`** modifier demands *all* paths in `a1a2a3d1` appear in the new network snapshot if any path in `a1b1b2b3d1` appears. This may be what the user wants in some cases, but it may not be in others. After all, `a1a2a3d1` represents the Cartesian product of four router groups and contains a large number of possible paths—does the user want all such paths to be present in the new network? The initial informal English specification we gave is actually mute on this issue; it simply says "change it." Indeed, we have found that working with Rela requires thinking carefully about desired semantics, and typically, there are corner cases to consider. Because the specifications are short (as well as reuseable and executable), one can afford to think carefully about their consequences.

Rela provides several built-in modifiers if **`replace`** is not the right one. If the traffic should move to *some* (any) path in `a1a2a3d1`, the **`any`**`(regex1)` modifier can be used as follows.

```
spec pathShift := {
    a1.*d1 : any(a1a2a3d1);
}
```

Recall that traffic in our change zone may start upstream of $A1$ routers and continue downstream of $D1$ routers. The spec above has not expressed changes expected for these starting and ending *sub-paths*. The user may not even know all the paths leading to this part of the network. In other systems, specifying a change accurately with such incomplete information is challenging, or perhaps impossible. But because Rela is *compositional* as well as relational, users can stitch change specifications of different kinds for different subpaths to construct an end-to-end specification. In this case, to specify that the beginnings and ends of the paths should not change, they can use change specifications with the **`preserve`** modifier as follows.

```
spec e2e := {
    a* : preserve;
    pathShift;
    d* : preserve;
}
```

This spec, which concatenates three atomic specs, defines the change zone as "a* (a1.*d1) d*". The first sub-spec's zone is a*, which denotes arbitrary length paths within region *A*. Even though users may not know the details of sub-paths in this zone, they do understand that these sub-paths are expected to remain unchanged, and the **preserve** modifier does the trick. We then reuse pathShift , which was defined earlier, to specify the sub-path changes in the middle. The spec of the third and last sub-path is similar to the first one. Rela thus allows end-to-end specs to be expressed compositionally, even when users do not know some sub-paths.

Up to this point, we have a spec that defines which paths should change and how they should change. Our third and final task is to specify that no other paths are affected by the network change. Rela makes this task easy via composition using the >> operator:

```
spec nochange := { .* : preserve; }
spec change := e2e >> nochange
```

All traffic that does not match the first spec will fall through to the next spec chained by >>. Thus, all existing traffic except those matched by e2e will be required to comply with nochange, which demands preservation for everything.

Summary. Rela specifications describe relations between a pair of network snapshots—that is, the paths that are added, removed, replaced or preserved when a network is updated. It allows change zones to be defined at a level of location granularity appropriate to their task. Once a zone of interest is defined, one may craft atomic change specifications that describe the relation between old and new networks for (sub-)paths in a zone. Users may draw on a collection of pre-defined modifiers to define relations of interest. Finally, complex change specs may be built out of simple ones through the use of Rela's composition operators.

5 FORMALIZING RELA SPECIFICATIONS

This section specifies the formal syntax of Rela and provides its semantics via translation to an intermediate representation with *regular relations*, which we call the RIR. While the RIR is more expressive than Rela's surface language, it is low-level, making it harder to use by network engineers. Indeed, Rela was created with the goal of making it easier to write relational specifications for networking use cases. Still, an expert user may use the RIR directly if they choose.

5.1 Rela Syntax

Figure 2 presents Rela's formal syntax, which includes sub-languages for (regular) sets of paths (r), modifiers (m), simple specifications (s), header constraints (h) and guarded specifications (g). The paths are built from a set of locations Σ, which includes a single special location *drop*—that place (akin to /dev/null) where intentionally dropped packets go. We use a to range over elements of the set Σ.

Path Sets	r	::=	a
		|	$(r_1 \mid r_2)$
		|	$r_1 r_2$
		|	$r*$
Modifiers	m	::=	**preserve**
		|	**add**(r)
		|	**remove**(r)
		|	**replace**(r_1,r_2)
		|	**drop**
		|	**any**(r)
Simple Specs	s	::=	$r : m$
		|	$s_1 s_2$
		|	s_1 >> s_2
Header Constraints	h	::=	*true*
		|	**dst in** *prefix*
		|	**src in** *prefix*
		|	**dscp** == *digits*
		|	$(h_1 \mid h_2)$
		|	h_1 & h_2
		|	!h
Guarded Specs	g	::=	s
		|	**if**(h) {g}
		|	**if**(h) {g_1} **else** {g_2}

Figure 2: The syntax of Rela's front-end language.

This syntax omits named definitions **spec** *name* := { g } and **regex** *name* := { r }, which are easily inlined. It also excludes **where** queries to select locations from database, which are implemented as a prepass.

We saw several of the modifiers (m) in the previous section. One that we did not see is **drop**, which replaces old paths with a new path that drops a packet. Each modifier is defined by a straightforward translation into the RIR. While our experiments suggest that we have developed a useful set of modifiers, new ones can be added by encoding their semantics in the RIR.

A simple specification (s) defines an expected change between a set of old paths and a set of new paths in a network. Simple specs include path modifiers ($r : m$), concatenation of simple specs ($s_1 s_2$), and prioritized union of simple specs (s_1 >> s_2).

Sets of packets are described by header constraints (h). In our current implementation, programmers may use the source IP (e.g., **src in** *prefix*), destination IP, or DSCP (a field in the IP header that encodes the priority of the packet) to describe packet sets.

Guarded specifications (g) describes changes for the paths taken by different subsets of packets. For example, the conditional statement **if**(h) {s} specifies that when packets described by h take paths M in the old network and N in the new network, M and N should be related by s. This specification says nothing about the paths taken by packets outside the set described by h. More concretely, consider the common change of decommissioning an IP prefix. For this change, we want to remove the paths used by traffic to the target prefix, but preserve the paths for all other traffic, even if the other traffic traverses similar or identical sets of paths. We can

encode this decommissioning requirement for address 10.0.0.0/24 using the following specification.

```
spec deallocP :=
  if (dst in 10.0.0.0/24) {
    .* : remove(.*);
  } else {
    .* : preserve;
  }
```

5.2 Regular IR (RIR)

The Rela RIR is an intermediate language for defining *regular sets* of paths and *regular relations* between paths. A regular set is a set created through the usual operations on regular languages (concatenation, union, and Kleene star). Likewise, regular relations are binary relations between paths (i.e., sets of pairs of paths), also constructed with the usual operations on regular languages. Since all RIR-expressible sets and relations are regular, we are able to make use of known, efficient constructions and decision procedures from automata theory as the basis of a decision procedure for RIR.

Figure 3 presents the syntax of the RIR, which contains four sub-languages. The language of path sets (P) describes regular sets of paths over the alphabet Σ (as defined in §5.1). The path sets a, 0, and 1 denote sets with a single one-hop path, no paths at all, and a single 0-length path (written ϵ). The special symbol `PreState` denotes the set of paths in the pre-change network. Similarly, `PostState` denotes the set of paths in the post-change network. The expressions $P_1|P_2$, P_1P_2, and P^* denote union, concatenation, and Kleene star operations over path sets. Finally, $P \triangleright R$ denotes the *image*, the path set derived by applying relation R to paths recognized by P. In other words, $P \triangleright R$ describes the set of paths that are related (via R) to *some* path recognized by P.

Figure 4 (a) presents the evaluation functions that define the semantics of path sets. These equations have the form $\mathscr{P}[\![P]\!](M, N) \triangleq S$, meaning that P describes the set of paths S when M is the pre-change set of forwarding paths and N is the post-change set of forwarding paths.

Relations in Figure 3 denote regular relations, which are sets of pairs of paths. Alternately, a relation may be viewed as a map from each path in the domain to zero or more related paths in the image. The cross-product relation $P_1 \times P_2$ denotes the relation that associates every path in P_1 with all paths in P_2. The identity relation $\mathtt{I}(P)$ associates every path in P with itself; paths not in P are not related to any other path by $\mathtt{I}(P)$. The symbols 0 and 1 denote the empty relation and the relation associating ϵ with itself. $R_1|R_2$, R_1R_2, R^*, and $R_1 \circ R_2$ denote union, concatenation, Kleene star, and composition of relations. (Rational relations are closed under all of these operations [16].)

Figure 4 (b) shows the semantics of relations. The equations have the form $\mathscr{R}[\![R]\!](M, N) \triangleq T$, meaning that R describes a set of pairs of paths T when M is the pre-change set of forwarding paths and N is the post-change set of forwarding paths.

Simple specs (Figure 3) may be equations ($P_1 = P_2$) or subset relations ($P_1 \subseteq P_2$). As an example, consider this spec:

$$\mathtt{PreState} \triangleright R = \mathtt{PostState}$$

Assuming the relation R is an intended transformation of network forwarding paths, the spec says that if one applies the transformation R to the pre-change path set, then one should obtain a result that equals the post-change path set. Our translation from Rela's surface language into the RIR uses this sort of idiom.

Figure 4 (c) presents the semantics of simple specs. The satisfaction relation has the form $M, N \models S$, which may be read as "pre-change forwarding paths M and post-change forwarding paths N satisfy S"

Finally, guarded specs (Figure 3) allow programmers to specify that the paths travelled by different sets of packets should change in different ways. Guarded specs (G) include simple specs (S) as well as restricted specs ($h \mapsto G$) and conjunctions thereof.

Figure 4 (d) shows the semantics of guarded specs in terms of *traffic equivalence classes* (TECs), which are triples of the form (T, M, N) where T is a set of packets, M is a set of paths that T follows in the pre-change network, and N is a set of paths that T follows in the post-change network. Flow equivalence classes used in manual inspection (§2.3) are a refinement of TECs where every string in M and N originates at the same starting location.

The semantics in Figure 4 (d) depends upon a function $\mathscr{H}$, which maps header constraints to the set of headers that satisfy it. For example, the constraint `dst in 10.0.0.0/24` specifies the set of packets with destination IP prefix in the given range. We omit the full definition of $\mathscr{H}$ for brevity. In general, the satisfaction relation for guarded specs has the form $T, M, N \models G$, which may be read as "traffic equivalence class (T, M, N) satisfies the guarded specification G." A class (T, M, N) satisfies a spec G whenever T is an empty set of packets, and if T is non-empty, satisfaction is defined by induction on the structure of G.

5.3 Compilation from Rela to RIR

In this subsection, we show how to compile any Rela simple spec expression (s) into an RIR simple spec (S) and any Rela guarded spec (g) into an RIR guarded spec (G).

Ultimately, a simple specification s is translated into an RIR expression of the following form.

$$\mathtt{PreState} \triangleright \mathscr{R}_{pre}[\![s]\!] = \mathtt{PostState} \triangleright \mathscr{R}_{post}[\![s]\!]$$

In this process, we generate two relation expressions from s. The first relation ($\mathscr{R}_{pre}[\![s]\!]$) transforms the pre-change paths, and the second relation ($\mathscr{R}_{post}[\![s]\!]$) transforms the post-change paths.

In what follows, we show how to compute relations for some of the key modifiers in the Rela language.

Encoding path preservation. Consider the translation of the path preservation modifier "`D: preserve`". Intuitively, this change specification says that all paths that appear in the zone D in the pre-state should also appear in the post-state. If the pre- and post-relations are as follows:

$$\mathscr{R}_{pre}[\![D : \mathtt{preserve}]\!] \triangleq \mathtt{I}(D)$$
$$\mathscr{R}_{post}[\![D : \mathtt{preserve}]\!] \triangleq \mathtt{I}(D)$$

then our overall translation will be:

$$\mathtt{PreState} \triangleright \mathtt{I}(D) = \mathtt{PostState} \triangleright \mathtt{I}(D)$$

$$
\begin{array}{lll}
P \in \textit{Path Set} & ::= & a \mid 0 \mid 1 \mid \texttt{PreState} \mid \texttt{PostState} \mid (P_1|P_2) \mid P_1P_2 \mid P^* \mid P_1 \cap P_2 \mid P_1 \setminus P_2 \mid \overline{P} \mid P \triangleright R \\
R \in \textit{Relation} & ::= & P_1 \times P_2 \mid \texttt{I}(P) \mid 0 \mid 1 \mid (R_1|R_2) \mid R_1R_2 \mid R^* \mid R_1 \circ R_2 \\
S \in \textit{Simple Spec} & ::= & P_1 = P_2 \mid P_1 \subseteq P_2 \\
G \in \textit{Guarded Spec} & ::= & S \mid h \mapsto G \mid G_1 \wedge G_2
\end{array}
$$

Figure 3: RIR syntax

(a) Path Sets

$$
\begin{aligned}
\mathscr{P}[\![a]\!](M,N) &\triangleq \{a\} \\
\mathscr{P}[\![0]\!](M,N) &\triangleq \emptyset \\
\mathscr{P}[\![1]\!](M,N) &\triangleq \{\epsilon\} \\
\mathscr{P}[\![\texttt{PreState}]\!](M,N) &\triangleq M \\
\mathscr{P}[\![\texttt{PostState}]\!](M,N) &\triangleq N \\
\mathscr{P}[\![P_1 \mid P_2]\!](M,N) &\triangleq \mathscr{P}[\![P_1]\!](M,N) \cup \mathscr{P}[\![P_2]\!](M,N) \\
\mathscr{P}[\![P_1P_2]\!](M,N) &\triangleq \{p_1p_2 \mid p_1 \in \mathscr{P}[\![P_1]\!](M,N), \\
&\qquad p_2 \in \mathscr{P}[\![P_2]\!](M,N)\} \\
\mathscr{P}[\![P^*]\!](M,N) &\triangleq \{p_1 ... p_n \mid p_1, ..., p_n \in \mathscr{P}[\![P]\!](M,N)\} \\
\mathscr{P}[\![P_1 \cap P_2]\!](M,N) &\triangleq \mathscr{P}[\![P_1]\!](M,N) \cap \mathscr{P}[\![P_2]\!](M,N) \\
\mathscr{P}[\![P_1 \setminus P_2]\!](M,N) &\triangleq \mathscr{P}[\![P_1]\!](M,N) \setminus \mathscr{P}[\![P_2]\!](M,N) \\
\mathscr{P}[\![\overline{P}]\!](M,N) &\triangleq \Sigma^* \setminus \mathscr{P}[\![P]\!](M,N) \\
\mathscr{P}[\![P \triangleright R]\!](M,N) &\triangleq \{q : \exists p.\ \langle p,q\rangle \in \mathscr{R}[\![R]\!](M,N) \\
&\qquad \wedge\ p \in \mathscr{P}[\![P]\!](M,N)\}
\end{aligned}
$$

(b) Relations

$$
\begin{aligned}
\mathscr{R}[\![P_1 \times P_2]\!](M,N) &\triangleq \{\langle p_1,p_2\rangle \mid p_1 \in \mathscr{P}[\![P_1]\!](M,N), \\
&\qquad p_2 \in \mathscr{P}[\![P_2]\!](M,N)\} \\
\mathscr{R}[\![\texttt{I}(P)]\!](M,N) &\triangleq \{\langle p,p\rangle \mid p \in \mathscr{P}[\![P]\!](M,N)\} \\
\mathscr{R}[\![0]\!](M,N) &\triangleq \emptyset \\
\mathscr{R}[\![1]\!](M,N) &\triangleq \{(\epsilon,\epsilon)\} \\
\mathscr{R}[\![R_1|R_2]\!](M,N) &\triangleq \mathscr{R}[\![R_1]\!](M,N) \cup \mathscr{R}[\![R_2]\!](M,N) \\
\mathscr{R}[\![\texttt{I}(P)]\!](M,N) &\triangleq \{\langle p,p\rangle \mid p \in \mathscr{P}[\![P]\!](M,N)\} \\
\mathscr{R}[\![R_1R_2]\!](M,N) &\triangleq \{\langle p_1p_2, q_1q_2\rangle \mid \langle p_1,q_1\rangle \in \mathscr{R}[\![R_1]\!](M,N), \\
&\qquad \langle p_2,q_2\rangle \in \mathscr{R}[\![R_2]\!](M,N)\} \\
\mathscr{R}[\![R^*]\!](M,N) &\triangleq \{\langle p_1 ... p_n, q_1 ... q_n\rangle \mid \\
&\qquad \langle p_1,q_1\rangle, ..., \langle p_n,q_n\rangle \in \mathscr{R}[\![R]\!](M,N)\} \\
\mathscr{R}[\![R_1 \circ R_2]\!](M,N) &\triangleq \{\langle x,z\rangle \mid \exists y.\langle x,y\rangle \in \mathscr{R}[\![R_1]\!](M,N), \\
&\qquad \langle y,z\rangle \in \mathscr{R}[\![R_2]\!](M,N)\}
\end{aligned}
$$

(c) Simple Specs

$$
\begin{aligned}
M,N \models P_1 = P_2 &\iff \mathscr{P}[\![P_1]\!](M,N) = \mathscr{P}[\![P_2]\!](M,N) \\
M,N \models P_1 \subseteq P_2 &\iff \mathscr{P}[\![P_1]\!](M,N) \subseteq \mathscr{P}[\![P_2]\!](M,N)
\end{aligned}
$$

(d) Guarded Specs

$$
\begin{aligned}
T,M,N \models S &\text{ if } T = \emptyset, \text{ and otherwise:} \\
T,M,N \models S &\iff M,N \models S \\
T,M,N \models h \mapsto G &\iff T \cap \mathscr{H}[\![h]\!], M, N \models G \\
T,M,N \models G_1 \wedge G_2 &\iff T,M,N \models G_1 \text{ and } T,M,N \models G_2
\end{aligned}
$$

Figure 4: RIR semantics.

which is equivalent to the equation:

$$(\texttt{PreState} \cap D) = (\texttt{PostState} \cap D)\ ,$$

as desired.

Encoding path additions. Consider adding the paths P when the pre-change network contains a path in D.[2] Our goal now is to preserve all of the paths in the zone (and also P, if it exists) from the pre-state into the post-state. In other words, we would like to apply the identity relation $\texttt{I}(D \mid P)$. In addition, we would like a relation that adds the path P. We can use the relation $D \times P$ to do so. Overall, our pre-relation is the combination of those two relations.

[2]The Rela surface language can not express the addition of a path in D when the pre-change network contains no path in D. Such "unconditional" path additions can be expressed in the RIR, however. For instance, the equation $\texttt{PostState} = \texttt{PreState}|P$ expresses that exactly the set of paths recognized by P are added to the network.

Hence, we generate the following equations.

$$
\begin{aligned}
\mathscr{R}_{pre}[\![D : \texttt{add}(P)]\!] &\triangleq \texttt{I}(D \mid P) \mid (D \times P) \\
\mathscr{R}_{post}[\![D : \texttt{add}(P)]\!] &\triangleq \texttt{I}(D \mid P)
\end{aligned}
$$

Encoding path removals. Next, consider path removals using the modifier "`D: remove(P)`". This modifier expresses that the paths in D in the pre-state should be preserved in the post-state, except the paths in P that should be removed. Hence, our relations are as follows.

$$
\begin{aligned}
\mathscr{R}_{pre}[\![D : \texttt{remove}(P)]\!] &\triangleq \texttt{I}(D \setminus P) \\
\mathscr{R}_{post}[\![D : \texttt{remove}(P)]\!] &\triangleq \texttt{I}(D)
\end{aligned}
$$

Encoding non-deterministic path replacement. The modifier "`D: any(P)`" demands that (1) if there is any path in $D \mid P$ in the pre-state, there must be some path in P in the post-state and (2) all paths in $D \mid P$ in the post-state must be in P. To encode this

condition, we use a relation for the pre-state that replaces paths in $D \mid P$ with a symbol #. Likewise, the relation for the post-state replaces all paths in P with #, while also retaining the paths in $D \setminus P$. Since paths in $D \setminus P$ are *not* retained in the pre-state relation, this relation encodes that there are no paths in $D \setminus P$ in the post-state network. Together, the two relations enforce the desired condition.

$$\mathscr{R}_{pre}[\![D : \texttt{any}(P)]\!] \triangleq (D \mid P) \times \#$$
$$\mathscr{R}_{post}[\![D : \texttt{any}(P)]\!] \triangleq (P \times \#) \mid \mathtt{I}(D \setminus P)$$

Encoding prioritized union. A prioritized union "$s_1 >> s_2$" should apply the change specification s_1 to s_1's zone and s_2 to everything else in s_2's zone. To achieve this specification in the RIR, we need to explicitly extract s_1's zone. We do so with an auxiliary function $\mathcal{Z}[\![D : \texttt{modifier}]\!]$. See Figure 5 for the full definition of $\mathcal{Z}[\![\cdot]\!]$.

To translate "$s_1 >> s_2$", we first translate s_1, and then take the union with the translation of s_2 applied exclusively to the complement of the zone of s_1.

Summary of simple specs. See Figure 5 for the complete translation for simple specs.

Translation for guarded specs. A guarded spec is simply an if-then-else structure that applies different header constraints to different simple specs. Based on the translation of simple specs, we define the following evaluation functions for guarded specs ($\mathcal{G}[\![\cdot]\!]$)

$$\mathcal{G}[\![s]\!] \triangleq \mathcal{S}[\![s]\!]$$
$$\mathcal{G}[\![\texttt{if}(h)\ \{g\}]\!] \triangleq h \mapsto \mathcal{G}[\![g]\!]$$
$$\mathcal{G}[\![\texttt{if}(h)\ \{g_1\}\ \texttt{else}\ \{g_2\}]\!] \triangleq h \mapsto \mathcal{G}[\![g_1]\!] \wedge\ !h \mapsto \mathcal{G}[\![g_2]\!]$$

6 DECISION PROCEDURE

Rela's decision procedure models a network change as a set of traffic equivalence classes (TEC). Given an RIR spec and a network change, the decision procedure determines whether each TEC in the change meets the spec. If not all TECs meet the spec, an exhaustive list of counterexamples will be provided in the form of specific packets and paths that violate the spec.

6.1 Checking Guarded Specs

To validate a guarded spec against a TEC, we may follow the rules described in §5.2. Thus, we are left with the problem of checking whether a pair of path sets M, N satisfies a simple spec.

6.2 RIR to Finite-State Automaton (FSA)

To validate an RIR simple spec regarding two sets of paths, we first construct a finite-state automaton (FSA) from each *Path Set* and *Rel* expression in the simple spec. Per Kleene's Theorem, every regular language can be represented by an FSA that moves from one state to another in response to the input sequence of symbols. Similarly, every regular relation can be represented as a finite-state transducer (FST) [16].

An FST is essentially an FSA that uses two tapes. It may be viewed as a "translating machine" that reads from an input tape and writes to the output tape. The following diagram presents an

$$\mathcal{S}[\![s]\!] \triangleq \texttt{PreState} \triangleright \mathscr{R}_{pre}[\![s]\!] = \texttt{PostState} \triangleright \mathscr{R}_{post}[\![s]\!]$$

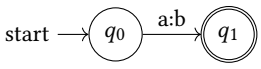
$$\begin{aligned}
\mathscr{R}_{pre}[\![D : \texttt{preserve}]\!] &\triangleq \mathtt{I}(D)\\
\mathscr{R}_{pre}[\![D : \texttt{add}(P)]\!] &\triangleq \mathtt{I}(D \mid P) \mid (D \times P)\\
\mathscr{R}_{pre}[\![D : \texttt{remove}(P)]\!] &\triangleq \mathtt{I}(D \setminus P)\\
\mathscr{R}_{pre}[\![D : \texttt{replace}(P_1, P_2)]\!] &\triangleq \mathtt{I}((D \mid P_2) \setminus P_1)\\
&\quad \mid ((D \cap P_1) \times P_2)\\
\mathscr{R}_{pre}[\![D : \texttt{drop}]\!] &\triangleq (D \mid drop) \times drop\\
\mathscr{R}_{pre}[\![D : \texttt{any}(P)]\!] &\triangleq (D \mid P) \times \#\\
\mathscr{R}_{pre}[\![s_1 s_2]\!] &\triangleq \mathscr{R}_{pre}[\![s_1]\!]\ \mathscr{R}_{pre}[\![s_2]\!]\\
\mathscr{R}_{pre}[\![s_1 >> s_2]\!] &\triangleq \mathscr{R}_{pre}[\![s_1]\!]\\
&\quad \mid \left(\mathtt{I}(\overline{\mathcal{Z}[\![s_1]\!]}) \circ \mathscr{R}_{pre}[\![s_2]\!]\right)
\end{aligned}$$

$$\begin{aligned}
\mathscr{R}_{post}[\![D : \texttt{preserve}]\!] &\triangleq \mathtt{I}(D)\\
\mathscr{R}_{post}[\![D : \texttt{add}(P)]\!] &\triangleq \mathtt{I}(D \mid P)\\
\mathscr{R}_{post}[\![D : \texttt{remove}(P)]\!] &\triangleq \mathtt{I}(D)\\
\mathscr{R}_{post}[\![D : \texttt{replace}(P_1, P_2)]\!] &\triangleq \mathtt{I}(D \mid P_2)\\
\mathscr{R}_{post}[\![D : \texttt{drop}]\!] &\triangleq \mathtt{I}(D \mid drop)\\
\mathscr{R}_{post}[\![D : \texttt{any}(P)]\!] &\triangleq (P \times \#) \mid \mathtt{I}(D \setminus P)\\
\mathscr{R}_{post}[\![s_1 s_2]\!] &\triangleq \mathscr{R}_{post}[\![s_1]\!]\ \mathscr{R}_{post}[\![s_2]\!]\\
\mathscr{R}_{post}[\![s_1 >> s_2]\!] &\triangleq \mathscr{R}_{post}[\![s_1]\!]\\
&\quad \mid \left(\mathtt{I}(\overline{\mathcal{Z}[\![s_1]\!]}) \circ \mathscr{R}_{post}[\![s_2]\!]\right)
\end{aligned}$$

$$\begin{aligned}
\mathcal{Z}[\![D : \texttt{preserve}]\!] &\triangleq D\\
\mathcal{Z}[\![D : \texttt{add}(P)]\!] &\triangleq D \mid P\\
\mathcal{Z}[\![D : \texttt{remove}(P)]\!] &\triangleq D\\
\mathcal{Z}[\![D : \texttt{replace}(P_1, P_2)]\!] &\triangleq D \mid P_2\\
\mathcal{Z}[\![D : \texttt{drop}]\!] &\triangleq D \mid drop\\
\mathcal{Z}[\![D : \texttt{any}(P)]\!] &\triangleq D \mid P\\
\mathcal{Z}[\![s_1 s_2]\!] &\triangleq \mathcal{Z}[\![s_1]\!]\ \mathcal{Z}[\![s_2]\!]\\
\mathcal{Z}[\![s_1 >> s_2]\!] &\triangleq \mathcal{Z}[\![s_1]\!] \mid \mathcal{Z}[\![s_2]\!]
\end{aligned}$$

Figure 5: Rela to RIR translation for simple specs ($\mathcal{S}[\![\cdot]\!]$).

FST for relation $a \times b$, which translates path a into path b.



The label a:b on the arc means a should be read from the input tape and b should be written to the output tape.

From small, simple FSAs, like the one above, we can build larger, more complex ones using standard automaton composition algorithms. In what follows, we sketch some of the algorithms used to construct Rela-specific symbols and operators. Most other aspects of the compilation strategy are well-established and are thus omitted (see Thompson's construction [34], for instance).

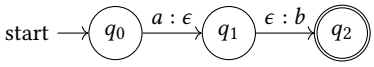
Packet	Pre-change paths	Post-change paths	Cause of violation
$(dst_1,\ src_1,\ dscp_1)$	$\{x_1A_1B_1B_2B_3D_1y_1\}$	$\{x_1A_1A_2A_3B_3D_1y_1\}$	e2e: $\{x_1A_1A_2A_3D_1y_1\} \neq \{x_1A_1A_2A_3B_3D_1y_1\}$
$(dst_2,\ src_2,\ dscp_2)$	$\{x_2C_1B_1B_2B_3D_1y_2\}$	$\{x_2C_1C_2D_1y_2\}$	nochange: $\{x_2C_1B_1B_2B_3D_1y_2\} \neq \{x_2C_1D_2D_1y_2\}$

Figure 6: A subset of counterexamples generated by Rela when verifying the change implementation in Figure 1c. The first row shows a flow in traffic class T1, and the second row shows a flow in T2.

`PreState` and `PostState` symbols. Conceptually, the input to the decision procedure contains two sets of forwarding paths that correspond to `PreState` and `PostState` respectively. In practice, however, the number of ECMP forwarding paths may explode in a large network. This problem is prominent when forwarding behavior is modeled at the interface level, because the network may employ 10s of parallel links between any two hops to increase capacity. Indeed, we recorded a flow with 10^8 interface-level ECMP paths for our backbone, and it takes several hours just to deserialize the paths from file input. To address this challenge, Rela defines a graph format to represent the interface-level input path set. Each vertex in the graph denotes a router that appears as a forwarding hop for this traffic, and each directed edge denotes a physical link that is used to forward this traffic between the two hops. There is also extra metadata to identify all source vertices and sink vertices (start and end locations of paths) in the DAG. With this format, the 10^8 paths of the aforementioned traffic class can be encoded with a DAG with 38 vertices and 50K edges.

Constructing an FSA for `PreState` and `PostState` from the forwarding graph is straightforward: We turn vertices and edges in the DAG into FSA states and transitions, respectively. If the user has specified a coarser granularity than interface-level (e.g., router level), we do granularity conversion in this step by merging vertices that belong to the same coarser entity. Next, we add an initial state q_0 and draw an ϵ-transition from q_0 to each source node identified by the metadata. Finally, we set all sink nodes to be accepting states of the FSA.

$P_1 \times P_2$ **relation.** The FST for $P_1 \times P_2$ may be obtained by (1) translating the FSA for P_1 into an FST that accepts P_1 on its first tape and ϵ on its second, (2) translating the FSA for P_2 to a FST that accepts P_2 on its second tape and ϵ on its first, and (3) concatenating the results. An illustration of this construction for $a \times b$ can be found in figure below (recall that ϵ is the empty string).



$\mathtt{I}(P)$ **relation.** The FST of $\mathtt{I}(P)$ is the same as the FSA of P except that each FST transition has an output symbol that duplicates the input symbol on the same transition.

$P \triangleright R$ **image.** To obtain the FSA of $P \triangleright R$, one may compute the composition of relations $\mathtt{I}(P)$ and R, and then compute the output projection of $\mathtt{I}(P) \circ R$. The output projection of an FST removes the input symbols from all transition arcs (while keeping the output symbols) to derive an FSA from an FST. The composition of two regular relations $R_1 \circ R_2$ may be compiled using standard FST algorithms [16].

6.3 Compliance Checking

Once we have the FSA representation of both sides of an equation ($P_1 = P_2$ or $P_1 \subseteq P_2$ in simple specs), we can check for equality (or inclusion) using standard automaton equivalence algorithms.

6.4 Counterexample Generation

If previous steps found any TEC that violates an RIR specification, then we generate an exhaustive list of counterexamples, where each entry contains a traffic equivalence class (TEC) and a reason that explains the failure. Figure 6 shows a subset of counterexamples reported by Rela when verifying the change implementation in Figure 1c using the change spec in §4. The two entries indicate incorrect path changes for traffic T1 and collateral damage for T2.

The forwarding paths that violate a simple spec are derived by extracting paths from the difference of two FSAs. Recall that a Rela simple spec s is translated to an equation of the form $P_1 = P_2$ in RIR, where $P_1 = \mathtt{PreState} \triangleright \mathcal{R}_{pre}[\![s]\!]$ and $P_2 = \mathtt{PostState} \triangleright \mathcal{R}_{post}[\![s]\!]$. The difference $P_1 \setminus P_2$ represents the expected forwarding paths that are missing from the observed post-change network, and $P_2 \setminus P_1$ represents the unexpected paths in the post-change network.

For each violating TEC, we generate a reason to help understand why it failed the spec. For simple specs that are composed using the >> operator, we can find the exact sub-spec that failed a flow by matching the flow with the zone of each sub-spec. We then apply $\mathcal{R}_{pre}$ and $\mathcal{R}_{post}$ of this sub-spec to the flow's pre- and post-change path set, respectively. The difference between the two derived sets explains the failure of set equation and inclusion assertions made by the spec. For special symbols introduced by rewriting in the compilation process, we rewrite them back to their original forms to make the counterexamples more human-readable. For example, the before paths in the first row of Figure 6 yield $\{x_1\#y_1\}$ when applying $\mathcal{R}_{pre}$, where "#" rewrites $A_1A_2A_3D_1$. After undoing this rewriting, the "Cause of violation" column clearly shows that the flow failed the sub-spec `e2e`, which expected the path set to be $\{x_1A_1A_2A_3D_1y_1\}$ after the change. This set is not equal to the observed path set $\{x_1A_1A_2A_3B_3D_1y_1\}$.

7 IMPLEMENTATION

We implemented Rela with 6,000 lines of Python code. Rela and RIR are implemented as domain-specific languages embedded in Python. The decision procedure uses the OpenFST library [2] and the Python bindings provided by HFST [24] to construct and compose finite state automata and transducers. We implemented certain automaton operations, such as the product relation ($P_1 \times P_2$), ourselves using low-level HFST APIs that manipulate automata directly.

For each traffic equivalence class (TEC), Rela reads the before and after forwarding paths from file input, which is produced by

the network simulator described in §2.3. We modified the simulator to output forwarding paths in the Rela-defined graph format and to enable efficient FSA construction for `PreState` and `PostState` expressions (§5.3). Each TEC is processed in parallel.

The Rela source code is publicly available [15].

8 QUALITATIVE EVALUATION

We evaluated Rela qualitatively and quantitatively. This section presents the qualitative evaluation, which focuses on user experience. The next section presents a quantitative evaluation of Rela's expressiveness and performance. For the qualitative evaluation, we ran Rela on historical changes to Alibaba Cloud's backbone. Our workflow shared the first four steps with the current workflow in §2.3: simulate pre- and post-change networks, compute forwarding paths, and aggregate them into equivalence classes. The final step is different: the forwarding data is given to Rela as input, along with a spec, and we analyze all equivalence classes rather than just the diff. We describe how this process played out for the change in §2.1, compared to the original experience of the engineers, and then draw lessons from our experience.

8.1 Revisiting the Example Change

For each proposed change (i.e., "iteration"), we used Rela to check the change against a relational specification.

First iteration. We invoked Rela with the change implementation v1 (Figure 1b) and the `change` spec in §4. For this implementation, the path diff of the manual inspection tool had 17 flow equivalence classes. Engineers investigated each class and discovered that none of them corresponded to the desired path change, and all of them stemmed from either issues with the simulation tool or benign side effects of the change. The allow-list change on *A*2 routers caused unexpected but acceptable traffic changes.

Rela produced 17 counterexamples for `nochange` and 15 for `e2e`. The 15 violations for `e2e` clearly signaled that the change failed to move T1 traffic, as the pre-change and post-change paths were still the same for such flows. The counterexamples for `nochange` are the same as those reported by the path diff tool. To automatically exclude such benign violations in future iterations (and avoid triaging them again), we extended the spec with a new component called `sideEffects`, to explicitly permit such changes.

Second iteration. In the second iteration, we provided the change implementation v2 (Figure 1c) and the refined spec. For this implementation, the current path diff tool produced a path diff with 46 flow equivalence classes. Engineers waded through them to discover the collateral damage and, because of information overload, missed that the change to T1 traffic was incorrect.

Rela produced 15 counterexamples for `e2e`, 24 for `nochangeand` 0 for `sideEffects`. The violations signaled that changes to T1 traffic were wrong and that there was collateral damage as well. The refined `sideEffects` spec helped suppress benign differences.

Final iteration. Because Rela discovered two errors at the same time, we skipped the third iteration (which was needed during the original manual analysis), and jumped straight to the final iteration. In this iteration, we supply the correct change implementation to Rela and the refined spec. Rela validated the change automatically and completely. In contrast, in their original debugging effort, the engineers had to manually inspect the path diff to certify the change.

8.2 Lessons Learned

Based on our experience with Rela, we draw these lessons:

(1) Rela's categorization of violations based on which sub-spec is violated speeds up error diagnosis and reduces the number of iterations. Errors are quickly diagnosed because the violated sub-spec provides strong hints about the root cause; the types of errors that violate `nochange` are different from those that violate `e2e`. The number of iterations is reduced because multiple errors in an implementation are easier to spot, especially when spread across different sub-specs. With manual inspection, when analyzing a big bag of path diffs, it is hard to spot multiple errors.

(2) Rela specs may need refinement because the original change intent (in natural language) is under-specified or the network is not configured as expected. Under-specification and unexpected behaviors are common for large networks. However, while the effort put into a manual audit is hard to reuse, the effort put into refining a Rela spec pays off. The refined spec saves work during future iterations of the same change or other similar changes. Multiple changes of the same type are a common occurrence for production networks.

(3) When a change (sub) spec does not match an implementation, there is less data to analyze. Change implementations are often partially correct, and Rela produces only violations. The current path diff contains both compliant and non-compliant changes. The engineers must analyze both to find violations.

(4) When the change spec matches the implementation, the engineers need to do nothing. They can have greater confidence in the change compared to manually inspecting the path diff.

9 QUANTITATIVE EVALUATION

To quantify the expressiveness and performance of Rela, we apply it to a set of real network changes in the global backbone of Alibaba Cloud. This dataset has all changes that were reviewed by the network's technical committee from Jun 2023 to Jan 2024. The committee reviews all high-risk, complex changes. There are 10s of changes in the dataset. We omit the exact count for confidentiality. Appendix A lists a subset of these changes and their Rela specs.

9.1 Expressiveness

We used Rela to specify the engineers' intent for each change in the dataset. We determined the intent by examining change tickets, which contain a description of the intent in natural language as well as a change implementation plan. The tickets describe change intents pertaining to the network data plane as well as those of other aspects, such as configuration settings and route attributes. Change 3 in Appendix A, which modifies BGP communities, is an example change whose full intent includes aspects beyond the data plane intent. We focused on data plane change intents. All changes in our dataset have a data plane change intent, and three in four have only data plane change intents.

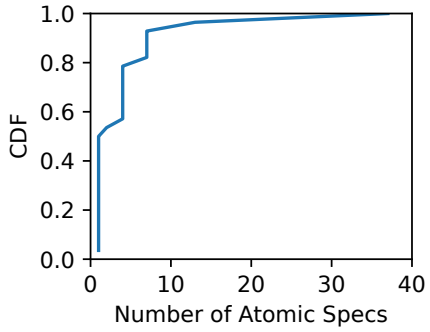


Figure 7: Distribution of spec size in our dataset.

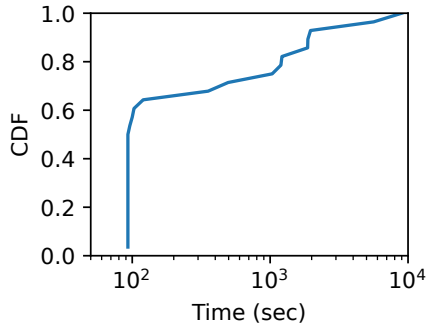


Figure 8: Time (log scale) to validate changes with Rela.

We found that Rela can specify the intended data plane change for 97% of the changes in our dataset. That Rela can support this high a fraction of high-risk changes in a large, complex network is a highly encouraging indicator of its expressiveness.

For the remaining 3% of the changes, Rela could only partially specify the intended data plane change. Rela's key current limitation is a lack of support for path counting: In addition to path shape, users sometimes want to limit the number of paths that a flow can take. For example, because of router hardware limits, one might not want the number of ECMP (equal cost multipath routing) paths for a flow to exceed 128. We will explore supporting such intents in the future by generalizing the **any** modifier to include a path count.

To assess the compactness of specifications, we quantify their size as the number of atomic Rela specs (of the form `r: m`). This analysis excludes any spec refinement that may be needed to accommodate benign side effects (§8.1); we do not have the data to make that determination. Figure 7 shows a cumulative distribution function (CDF) of the number of atomic specs needed across all changes. The vast majority of the changes (93%) can be expressed with fewer than 10 atomic specs. The outliers correspond to infrequent changes to the backbone's routing architecture in which significant traffic carried by the network is shifted.

Half the changes require only one atomic spec, corresponding to no expected impact on the forwarding behavior. It may seem odd at first that so many high-risk changes fall into this bucket. But fully preserving forwarding behavior while something is changed under the hood (e.g., modifying the routing policy to replace concrete routes with aggregate routes or standardizing on community tags) is common. It is also high-risk. Indeed, there are changes in our data where no behavior change was expected but the path diff revealed forwarding changes that could have led to an outage.

9.2 Performance

We benchmark Rela's performance using the time to validate changes in our dataset, including the time to deserialize the forwarding path data, FSA/FST construction and equivalence checking. This experiment was done on a computer with 96 CPU cores and 768 GB DRAM. Because we did not have access to the precise data plane states of historical changes, we ran all specs on the same data plane state produced by a recent snapshot. We release a subset of this data plane state [11]. Appendix B has details on its content.

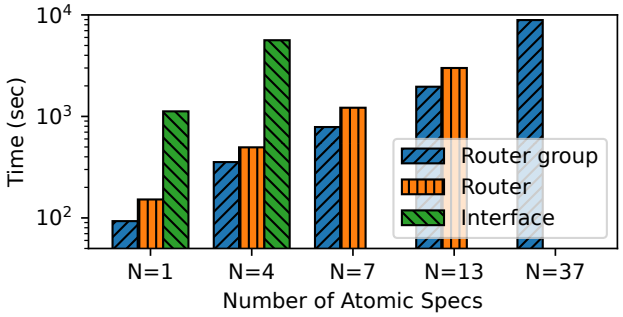


Figure 9: Rela's validation time (log scale) for different spec sizes and location granularity.

Figure 8 shows a CDF of the validation time. Half of the changes take 93 seconds, which is the time to check the "no change" spec. Four in five changes need less than 20 minutes, and the most complex change needs 150 minutes. For context, we observe that it takes 140 minutes to simulate both network snapshots and compute forwarding paths. We conclude from these results that the performance of Rela is acceptable for the backbone network, especially considering how long manual inspection takes today.

Diving deeper into Rela's performance, we find that the two most important factors are the size of the spec (number of atomic specs) and the location granularity. Figure 9 shows this impact by running specs in our data at different granularities. (Figure 8 used the granularity indicated by the change intent, so it has a mix.) We exclude granularity-size combinations that need over 3 hours.

We see that validation time grows with the spec size, and finer granularity analysis takes more time (as expected). The impact of going from the router group level to the router level is small, but the impact of going to the interface level is substantial (10x), due to the substantially higher number of paths at the interface level. Fortunately, under 4% of the changes in our data require interface-level granularity. 7% require device-level.

10 RELATED WORK

Our work builds on the foundation laid by single-snapshot verification tools [1, 3, 9, 17–19, 21–23, 25, 26, 36, 37]. The application

of these tools to real-world networks has improved reliability and provided insights into the problems they do and do not solve. We act upon one such insight: that many large, real-world networks are difficult to specify accurately in their entirety. Without such single-snapshot specifications, engineers need different kinds of tools to help them validate network changes automatically.

Differential network analysis (DNA) [39] shares our perspective on network verification—that it is crucial to track similarities and differences between pre- and post-change networks. It simulates the pair of pre- and post-change control planes efficiently to generate differences in their data plane states. (Rela makes no contributions to control plane simulation.) In addition to showing path diffs, DNA can generate differences in single-snapshot invariants, e.g., "A can reach B in the pre-change network but not the post-change network." Engineers must manually inspect the path and invariant diffs to determine whether or not they indicate errors. In contrast, Rela specifications characterize what constitutes an error, and our decision procedures check these specifications automatically. Importantly, Rela's specifications can be perfectly precise, more precise than "A can reach B"—any specific path or regular set of paths may be specified. This precision takes manual audits completely out of the loop when changes are conformant.

Batfish supports differential analysis as well [8]. It independently analyzes two snapshots and formats the output such that the differences are easier to analyze. Like DNA, it requires humans to certify correctness and does not have a relational spec. Once again, Rela improves on this situation using a relational specification language and deciding the validity of specifications without human auditing.

Our language design is inspired in part by NetKAT [3], which has shown that using regular languages (Kleene algebra) is an effective way to specify network behavior. Rela builds on this idea and uses regular relations in addition to regular languages to express differences and similarities between pairs of networks.

Researchers have explored relational verification for ordinary programs many times in the past [7, 12, 14]. The archetypal goal here is to verify that the two programs are equivalent. At least superficially, the techniques for relational program verification differ from those in Rela. A common method is to consider a "product" program that combines two input programs and verify the safety properties of this product. An interesting avenue for future work is to consider whether specific relational program verification techniques can help us verify networks more efficiently or vice versa.

11 SUMMARY

We develop the concept of relational network verification and realize it in the Rela tool for validating network changes. Our key observation is that relational specifications can compactly and precisely capture change intents; they need only express what is expected to change, which is often a miniscule fraction of the overall network, and simply say "no change" for the rest. For a global backbone with over 10^3 routers, 93% of the high-risk changes need fewer than 10 terms and 80% of them can be validated in under 20 minutes.

While we focus on change verification, we expect that relational reasoning for networks will prove effective in other contexts as well. For instance, it could help verify if two parts (e.g., two geographic regions) of the *same* snapshot are similar modulo a few exceptions. It could also compactly describe a large network for the purposes of synthesis by describing a baseline network behavior and local modifications to the behavior. We look forward to exploring other applications of relational network analysis.

Acknowledgements. We thank the SIGCOMM reviewers and our shepherd Dave Maltz for feedback that helped improve this paper. This work was supported in part by NSF awards 2007073, 2219862, and 2219863 and by the partners of UW FOCI.

Ethics. This work does not raise any ethical issues.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *Proceedings of NSDI 20*. USENIX Association, 201–219.
[2] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A General and Efficient Weighted Finite-State Transducer Library: (Extended Abstract of an Invited Talk). In *Implementation and Application of Automata: 12th International Conference, CIAA 2007, Praque, Czech Republic, July 16-18, 2007, Revised Selected Papers 12*. Springer, 11–23.
[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of POPL '14*. ACM, 113–126.
[4] Mae Anderson. 2014. Time Warner Cable Says Outages Largely Resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved. (2014). Retrieved June 23, 2021 from http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved
[5] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. 2019. Reachability analysis for AWS-based networks. In *International Conference on Computer Aided Verification*. Springer, 231–241.
[6] Gilles Barthe. 2020. An introduction to relational program verification. (2020). Retrieved Feb 2, 2024 from https://software.imdea.org/~gbarthe/__introrelver.pdf
[7] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods*, Michael Butler and Wolfram Schulte (Eds.).
[8] batfish-differential 2022. Differential Questions. (2022). Retrieved Feb 2, 2024 from https://batfish.readthedocs.io/en/latest/notebooks/differentialQuestions.html
[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of SIGCOMM '17*. ACM, 155–168.
[10] Ryan Beckett and Ratul Mahajan. 2020. Capturing the state of research on network verification. (2020). Retrieved Feb 2, 2024 from https://netverify.fun/2-current-state-of-research/
[11] Anonymized benchmarking data. [n. d.]. ([n. d.]). Retrieved June 8, 2024 from https://github.com/alibaba/rela/tree/main/dataset
[12] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *POPL*.
[13] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. 2023. Lessons from the Evolution of the Batfish Configuration Analysis Tool. In *Proceedings of SIGCOMM '23*. ACM, 122–135.
[14] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational verification using reinforcement learning. In *OOPSLA*.
[15] Rela Source Code. 2024. (2024). Retrieved June 3, 2024 from https://github.com/alibaba/rela
[16] C. C. Elgot and J. E. Mezei. 1965. On relations defined by generalized finite automata. *IBM J. Res. Dev.* 9, 1 (jan 1965), 47–68. https://doi.org/10.1147/rd.91.0047
[17] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *Proceedings of (OSDI 16*. USENIX Association, 217–232.
[18] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of NSDI 15*. USENIX Association, 469–483.
[19] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of SIGCOMM '16*. ACM, 300–313.
[20] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[21] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating Datacenters at Scale. In *Proceedings of SIGCOMM '19*. ACM, 200–213.

[22] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of NSDI 12*. USENIX Association, 113–126.

[23] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of NSDI 13*. USENIX Association, 15–27.

[24] Krister Lindén, Miikka Silfverberg, and Tommi Pirinen. 2009. Hfst tools for morphology–an efficient open-source package for construction of morphological analyzers. In *State of the Art in Computational Morphology: Workshop on Systems and Frameworks for Computational Morphology, SFCM 2009, Zurich, Switzerland, September 4, 2009. Proceedings*. Springer, 28–47.

[25] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *Proceedings of SIGCOMM '11*. ACM, 290–301.

[26] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *Proceedings of NSDI 20*. USENIX Association, 953–967.

[27] Steve Ragan. 2016. BGP errors are to blame for Monday's Twitter outage, not DDoS attacks. (2016). Retrieved June 23, 2021 from https://www.csoonline.com/article/3138934/security/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html

[28] Deon Roberts. 2018. It's been a week and customers are still mad at BB&T. (2018). Retrieved June 23, 2021 from https://www.charlotteobserver.com/news/business/banking/article202616124.html

[29] Mike Robuck. 2020. Due to a router misconfiguration, Cloudflare suffers short outage on Friday. (2020). Retrieved Feb 23, 2022 from https://www.fiercetelecom.com/telecom/due-to-a-router-misconfiguration-cloudflare-suffers-short-outage-friday

[30] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of SIGCOMM '16*. ACM, 426–439.

[31] Yevgeniy Sverdlik. 2014. Microsoft Says Config Change Caused Azure Outage. (2014). Retrieved Feb 23, 2022 from https://www.datacenterknowledge.com/archives/2014/11/20/microsoft-says-config-change-caused-azure-outage

[32] Yevgeniy Sverdlik. 2017. United Says IT Outage Resolved, Dozen Flights Canceled Monday. (2017). Retrieved June 23, 2021 from https://www.datacenterknowledge.com/archives/2017/01/23/united-says-it-outage-resolved-dozen-flights-canceled-monday

[33] Cisco Systems. 2021. Overview of Netflow. (2021). Retrieved Feb 2, 2024 from https://www.cisco.com/c/dam/en/us/td/docs/routers/asr920/configuration/guide/netmgmt/fnf-xe-3e-asr920-book.html

[34] Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (jun 1968), 419–422. https://doi.org/10.1145/363347.363387

[35] Zach Whittaker. 2020. T-Mobile hit by phone calling, text message outage. (2020). Retrieved June 23, 2021 from https://techcrunch.com/2020/06/15/t-mobile-calling-outage/

[36] Hongkun Yang and Simon S. Lam. 2016. Real-time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. Netw.* 24, 2 (April 2016), 887–900.

[37] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of SIGCOMM '20*. ACM, 599–614.

[38] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proceedings of NSDI 14*. USENIX Association, 87–99.

[39] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. 2022. Differential network analysis. In *Proceedings of NSDI 22*. USENIX Association, 601–615.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A EXAMPLE CHANGES AND RELA SPECS

We present ten example changes, drawn from the dataset in §9, and their Rela specs. Various aspects of these specifications, such as locations and IP addresses, have been anonymized.

Change 1: Disabling certain internal forwarding paths. Network engineers want to disable forwarding paths inside the backbone network from region A to region B for certain prefixes, such that traffic to these prefixes will exit the backbone network via `exit2` in region A and take another path to region B, as requested by users. No other traffic should be affected. The implementation of the change modifies the configurations on the iBGP route reflectors in region B, such that the iBGP announcements from region B to region A with related prefixes will be denied by routing policies.

The intent of this change can be expressed in Rela as:

```
regex a := where(region=="A")
regex b := where(region=="B")
spec change1 := if (dst in 1.1.1.0/24) {
    a*      : preserve;
    b*exit1 : replace(.*, exit2);
    >>
    .*      : preserve;
} else {
    .*      : preserve;
}
```

The granularity of this spec is device group level.

Change 2: Expanding a device group. Network engineers want to expand a device group from n routers to $2n$ routers, such that traffic that traversed one or more pre-existing routers should now go through one or more routers in the new group. No other traffic should be affected.

The intent of this change can be expressed in Rela as:

```
regex dg_old := r_1|...|r_n
regex dg_new := r_1|...|r_2n
spec change2 := {
    .*     : preserve;
    dg_old : any(dg_new);
    .*     : preserve;
    >>
    .*     : preserve;
}
```

The granularity of this spec is device level.

Change 3: Adding community tag to certain BGP prefixes. Network engineers want to add a new community tag to certain BGP routes. The implementation involves changing the export routing policy in the configurations of border routers. Despite the changes to routing policies, engineers want to ensure that no existing flow changes its paths at the device level.

The intent of this change can be expressed in Rela as:

```
spec change3 := .* : preserve;
```

The granularity of this spec is device level.

Change 4: Publishing a new internal IP prefix. Network engineers want to establish connectivity from certain source datacenters (S1, *etc.*) to a particular destination datacenter (D) for a newly assigned IP prefix (e.g., 1.1.1.0/24).

The intent of this change can be expressed in Rela as:

```
regex s := where(datacenter=="S1") | ...
regex d := where(datacenter=="D")
spec change4 := if (dst in 1.1.1.0/24) {
    .* : any(s.*d);
} else {
    .* : preserve;
}
```

The granularity of this spec is device group level.

Change 5: Shifting traffic to new interconnection routers. Network engineers want to shift traffic that leaves three regions (R1, R2, R3) from an old set of interconnection routers (i1, i2, i3), which denote the device group that serves this purpose in the three regions respectively, to a new set of router groups (i1', i2', i3'). The traffic between these three regions should be shifted from interconnection links such as i1-i2 to new links such as i1'-i2', while traffic to regions other than R1, R2 and R3 should exit via the same border routers as before. As always, traffic in other parts of the backbone should not be affected by this change.

The intent of this change can be expressed in Rela as:

```
spec change5 := {
    region1   : preserve;
    i1        : replace(i1, i1');
    border1.* : preserve;
    >>
    region1  : preserve;
    i1i2     : replace(i1i2, i1'i2');
    region2* : preserve;
    >>
    region1  : preserve;
    i1i3     : replace(i1i3, i1'i3');
    region3* : preserve;
    >>
    ...
    /*
    3 specs for R2->R2, R2->R1, R2->R3 flows
    3 specs for R3->R3, R3->R1, R3->R2 flows
    */
    ...
    >>
    .* : preserve;
}
```

The granularity of this spec is device group level.

Change 6: Shrinking edge links. Network engineers want to shrink the capacity of an edge link connected to an external peer.

This is implemented by disconnecting some of the parallel links connected to this peer.

The intent of this change can be expressed in Rela as:

```
spec change6 := {
    .*         : preserve;
    exits_old  : any(exits_new);
}
```

The granularity of this spec is interface level. But our evaluation uses a device-level version because the network simulation tool does not retain information about exit interfaces.

Change 7: Upgrading router firmware. Network engineers want to upgrade the firmware of a router. After the upgrade, the configuration syntax was updated by the manufacturer, and thus engineers updated the configuration content accordingly. In such cases, it is critical to check that the forwarding behaviors are equivalent before and after the change.

```
spec change7 := .* : preserve;
```

The granularity of this spec can be at any level. Finer-grained specs may catch more errors, but will have a higher computational cost.

Change 8: Shifting local traffic. Network engineers want to disable the link between a high-tier router group H1 and its low-tier neighbor router group L1, and they would like the H1-L1 traffic to detour via L1's sibling group L2, *i.e.*, H1-L2-L1.

The intent of this change can be expressed in Rela as:

```
spec change8 := {
    .*    : preserve;
    H1L1  : any(H1L2L1);
    .*    : preserve;
    >>
    .*    : preserve;
    L1H1  : any(L1L2H1);
    .*    : preserve;
    >>
    .*    : preserve;
}
```

The granularity of this spec is device level.

Change 9: Fixing IGP cost. Network engineers want to lower the IGP (interior gateway protocol) cost of the link between R1 and R2, such that traffic will go through direct R1-R2 link instead of detouring via other routers.

The intent of this change can be expressed in Rela as:

```
spec change9 := {
    .*      : preserve;
    R1.*R2  : replace(R1.*R2, R1R2);
    .*      : preserve;
    >>
    .*    : preserve;
}
```

The granularity of this spec is device level.

Change 10: Modifying topology. Network engineers aim to modify the topology such that a low-tier router group L is connected to mid-tier router group M. The router group L was originally connected to a high-tier router group H before the change.

The intent of this change can be expressed in Rela as:

```
spec change10 := {
    .*   : preserve;
    LHM  : any(LM);
    .*   : preserve;
    >>
    .*   : preserve;
    LH   : any(LMH);
    .*   : preserve;
    >>
    .*   : preserve;
    MHL  : any(ML);
    .*   : preserve;
    >>
    .*   : preserve;
    HL   : any(HML);
    .*   : preserve;
    >>
    .*   : preserve;
}
```

The granularity of this spec is device group level.

B PERFORMANCE EVALUATION DATASET

We released a subset of the data plane generated by simulation of the same backbone network as studied in §9 [11]. It contains 22,000 traffic equivalent classes (TECs) and the the set of packets used for simulation, the before-change forwarding paths, and the after-change forwarding paths of each TEC. Router names, interface names, and IP addresses in the dataset have been anonymized for confidentiality. This dataset works out-of-the-box with the open-sourced Rela tool [15] and can be used to benchmark the performance of Rela or other change validation tools. But because it contains only a subset of TECs compared to what we used in §9, while it can be used to benchmark per-TEC verification performance, it will produce the same results as §9. Total verification time grows in proportion to the number of TECs in the change.