Characterizing Network Configuration Repair Spaces with Localized Subspecifications

Yongzheng Zhang
ShanghaiTech University
Shanghai, China
yongzheng@shanghaitech.edu.cn

Haoxian Chen
ShanghaiTech University
Shanghai, China
hxchen@shanghaitech.edu.cn

Abstract

Network misconfigurations are a common source of policy violations. Existing verification tools detect such violations but provide limited guidance for systematic repair, while repair tools typically generate correct but opaque fixes that bear little resemblance to configurations operators would write themselves. We propose a symbolic framework that characterizes the *repair space* of misconfigurations. Instead of generating a patch, we compute localized subspecifications by treating selected configuration locations as symbolic variables and deriving the correctness conditions as constraints on these variables. The resulting constraints accurately describe all updates that restore compliance at a given location. These constraints make the repair intent explicit and enable operators to select a fix within that space consistent with their operational practice. To improve scalability, we introduce a modularization strategy that leverages correct route propagation graphs through simulation as semantic interfaces, enabling device-level repair space computation under fixed routing contexts. We implement our approach by extending existing verification tools and demonstrate that it automatically derives interpretable and feasible repair spaces for representative misconfiguration scenarios.

CCS Concepts

• **Networks** → **Network manageability**; **Formal specifications**.

Keywords

network configuration, network repair, repair space

ACM Reference Format:
Yongzheng Zhang and Haoxian Chen. 2026. Characterizing Network Configuration Repair Spaces with Localized Subspecifications. In *The 10th Asia-Pacific Workshop on Networking (APNet '26), August 6–7, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3820441.3820471

1 Introduction

Network configurations are complex and frequently contain mistakes that lead to violations of network specifications, *e.g.,* reachability, isolation, load-balancing, access control. When such violations occur, operators must identify a repair and verify that the modified configuration restores the intended behavior. This process is time-consuming and error-prone due to the scale of networks and the complex interactions among routing protocols.

To help diagnose configuration errors, network verification [2–4, 13, 17, 19] checks whether the current configuration satisfies given specifications, and produces counterexamples when violations occur. While useful for diagnosis, such counterexamples typically expose only a single violating execution trace. Fixing that trace does not necessarily resolve the problem, as multiple factors may influence the network behavior. Network synthesis [5, 6, 9, 10, 18], on the other hand, generates correct configurations from scratch rather than repairing existing ones. As a result, neither approach directly guides operators in repairing misconfigurations within an existing deployment.

This motivates the problem of automatic configuration repair. S^{2}Sim [25] searches for intent-compliant configuration variants using selective symbolic simulation, while ACR [14] localizes faulty lines and generates fixes using templates, validating candidate repairs through incremental verification.

From patch synthesis to repair-space characterization. While these systems demonstrate the feasibility of automatic configuration repair, they typically produce *a single concrete repair*. As an artifact of automatic synthesis, the generated patches may not always align with operators' design rationale [1, 16] (*e.g.,* operators often prefer simpler policies using fewer route attributes). In this work, we explore a complementary question: can we *characterize the repair space* directly? Rather than returning an arbitrary correct patch, we aim to describe the space of valid repairs to guide operator analysis and interpretation.

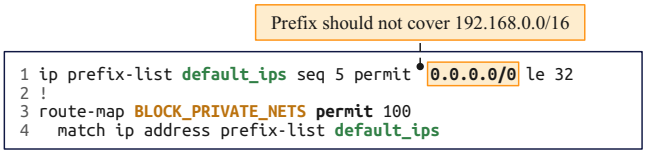

Figure 1: Misconfiguration and a repair space.

Figure 1 illustrates a simple example. The misconfiguration permits all IPv4 routes, thereby allowing a route to 192.168.0.0/16 to be advertised. Instead of proposing a single concrete fix *e.g.,* adding an explicit deny rule, our repair space characterizes the necessary condition on the prefix-list entry: any prefix that does not cover 192.168.0.0/16 satisfies the specification. An automated repair tool might instead suggest a specific prefix-list entry with a modified value (*e.g.,* 8.0.0.0/5), making it unclear why that particular value is

APNet '26, Singapore, Singapore

ACM ISBN 979-8-4007-2664-4/26/08
https://doi.org/10.1145/3820441.3820471

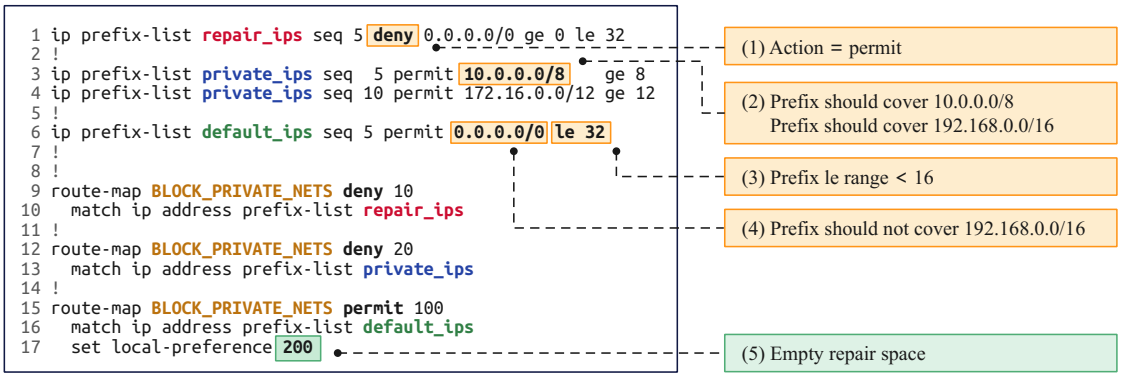

Figure 2: Misconfiguration under a specification requires blocking the private prefixes (10.0.0.0/8**,** 172.16.0.0/12**, and** 192.168.0.0/16**), and five repair spaces for relevant configuration locations, including one empty repair space.**

chosen. The constraint makes the intent explicit: the configuration must block this prefix.

Challenges. Characterizing the repair space raises several questions: (1) *Formulation.* How can we express the repair space so it is both flexible and interpretable? (2) *Feasibility.* Can we precisely compute the full repair space? (3) *Scalability.* How can we scale the computation to large networks?

Representing repair space via subspecification. We represent the repair space using *localized subspecifications* [15, 16]. Given a specification ϕ and a candidate configuration location h (*e.g.*, a configuration field, line, block), a subspecification captures the localized logical constraints over the configuration elements in h that characterize when ϕ holds. These constraints define the space of updates within location h that can repair the configuration and make it satisfy ϕ.

Figure 1 illustrates a subspecification requiring that the configured prefix must not cover 192.168.0.0/16. This constraint describes the repair space: any prefix satisfying it constitutes a valid repair.

Deriving repair space via constraint relaxation. To understand how a configuration can be repaired, we examine one configuration location at a time. For a candidate location h, we treat the config elements in h as unknown variables while keeping the rest of the configuration fixed. We then ask a simple question: *for what values of h would the specification ϕ hold?* If such values exist, the location h can potentially repair the configuration after simplification and constant propagation of relaxed constraints. By analyzing the resulting constraints, we obtain a compact description of all valid updates to h that would make the configuration satisfy ϕ.

Improving scalability via modularization. To improve scalability, we decompose the analysis across individual routers using a reference route propagation graph [25] as a semantic interface, which can be obtained from a previously correct configuration or an external routing analysis tool [24, 25]. This makes the analysis tractable: each repair space is conditioned on the specific reference route propagation graph, making the approach sound (the reference graph satisfies the specifications), but may be incomplete with respect to the full global specification, as alternative graphs that also satisfy the specification are not considered.

Contributions and organizations. In this paper, we present a symbolic and modular framework that automatically computes repair space for configurations that violate specifications. By decomposing analysis across devices and using a correct route propagation graph as a reference for the surrounding network, our approach enables scalable repair space reasoning.

The rest of the paper is organized as follows:

- Repair space characterization (Section 2). We introduce repair space as localized subspecifications around candidate configuration locations, capturing the conditions under which a specification can be restored.
- Repair framework. We first present the basic idea (Section 3), and then the modularization to scale to large networks (Section 4).
- Preliminary evaluation (Section 5). We implement a prototype and demonstrate its feasibility on representative network misconfiguration scenarios.

2 Motivating Example

Figure 2 shows a route-map intended to block private prefixes. Specifically, the specification requires that the private prefixes 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16 must not be installed or propagated through the device.

The routing policy `BLOCK_PRIVATE_NETS` includes three prefix-lists sequentially. The prefix list `private_ips` contains 10.0.0.0/8 and 172.16.0.0/12, but does not include the prefix 192.168.0.0/16. As a result, this prefix does not match any deny clause and eventually matches the final permit rule via `default_ips`, leading to a violation of the specification.

On the right are five candidate fix locations and their corresponding repair spaces. Note that one space is empty, indicating that no update at that location can repair the configuration. Each repair space is interpreted in isolation: a fix within that space is correct only if the rest of the configuration remains unchanged:

(1) Modifying the action deny in `repair_ips` to `permit`, causing it to match and block all IPv4 routes earlier.

(2) Modifying the prefix 10.0.0.0/8 in `private_ips` to include the prefixes 10.0.0.0/8 and 192.168.0.0/16. This repair space clearly

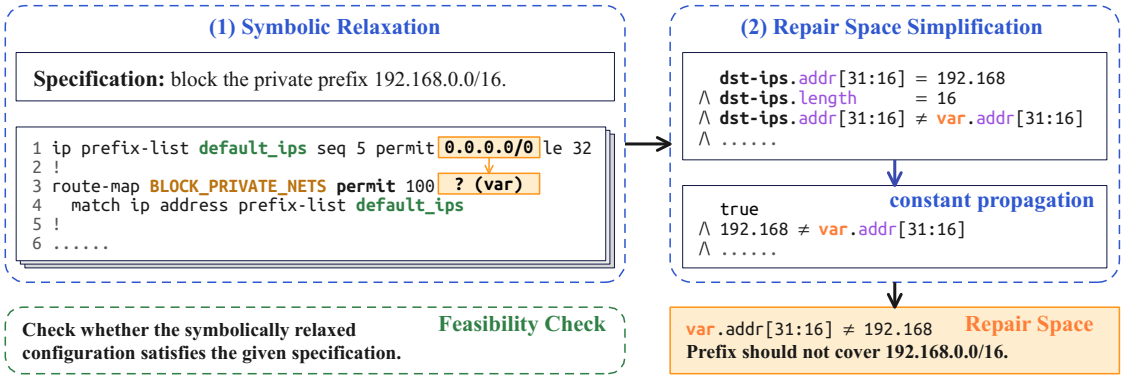

Figure 3: A general framework for repair space computation.

shows how to block prefix 192.168.0.0/16 without affecting the correct blocking of prefix 10.0.0.0/8.

(3) Modifying the le 32 in `default_ips` to a value smaller than 16, so that it no longer matches the prefix 192.168.0.0/16.

(4) Modifying the prefix 0.0.0.0/0 in `default_ips` to exclude 192.168.0.0/16, so that it no longer matches this prefix.

(5) An empty repair space for `set local-preference 200`, indicating that modifying this configuration location alone cannot repair the misconfiguration.

Correct patches may still conflict with unstated operator intent. Consider a candidate fix that changes the prefix 10.0.0.0/8 in prefix-list `private_ips` to 0.0.0.0/0. This modification eliminates the violation because it matches all prefixes. Without further specifications about the reachability of other prefixes, such a change could be a valid output of a repair tool. However, it goes beyond the apparent policy intent by blocking all IPv4 routes whose prefix length is greater than or equal to 8.

In practice, operator intents are often only partially specified, so a single synthesized patch may not align with unstated design considerations. Characterizing the repair space instead exposes alternative fixes and clarifies the rationale behind them.

Given these repair spaces, operators can examine the constraints to better understand the root cause of the misconfiguration and select a concrete fix consistent with their intent. Instantiating a repair from the space yields a configuration update that satisfies the specification while making other possible fixes explicit.

3 Repair Space Computation

Figure 3 presents a general framework for repair space computation [1]. At a high level, the process consists of two stages. First, we perform a feasibility check for each location by symbolically relaxing its configuration elements and determining whether any assignment can restore compliance with the specification. Second, for locations that are feasible, we simplify the resulting constraints to derive a localized subspecification that characterizes how the configuration at that location can be modified to satisfy the specification.

[1]We refer to this baseline approach as monolithic repair in the remainder of the paper, to distinguish it from modular repair, which is introduced in the next section.

Symbolic relaxation and feasibility check. Figure 3 shows a misconfiguration under a specification that requires blocking the private prefix 192.168.0.0/16. For each configuration location (*e.g.*, a configuration field, line, block), we symbolically relax it by replacing its concrete content with a symbolic variable while keeping the rest of the configuration fixed.

We encode the relaxed configuration together with the specification into a logical formula $\Phi(h)$, where h denotes the relaxed configuration location. Intuitively, this encoding asks: if the configuration at h were replaced by an arbitrary value, would there exist a replacement that makes the global specification satisfied? We answer this question by checking the satisfiability of $\Phi(h)$. If $\Phi(h)$ is satisfiable, the location h is considered repairable, and the satisfying constraints form the basis for computing its repair space. Otherwise, the location cannot independently repair the violation.

As shown in Figure 3, we selectively replace the prefix 0.0.0.0/0 in `default_ips` with a symbolic variable `var`. Through a feasibility check, we determine that this prefix corresponds to a repairable configuration location.

Repair space derivation via constraint simplification. For a repairable configuration location, we compute its repair space, which characterizes how the location can be modified so that the resulting configuration satisfies the given specification.

Starting from the formula $\Phi(h)$ obtained after symbolic relaxation, the formula contains constraints involving the symbolic configuration variable, other configuration elements, and internal network state variables introduced by the encoded control-plane semantics. These state variables capture the internal network state and the route propagation behavior, reflecting how configuration changes propagate through routing protocols and affect network behavior.

Like localized subspecifications [15, 16], we perform formula simplification and constant propagation to eliminate irrelevant internal network state variables and derive a localized constraint over the symbolic configuration variable. The simplification preserves satisfiability with respect to the symbolic variable, ensuring that the resulting constraint provides a sound characterization of the repair space.

As shown in Figure 3, the resulting constraints reduce to the condition `var.addr[31:16] ≠ 192.168`, which characterizes the

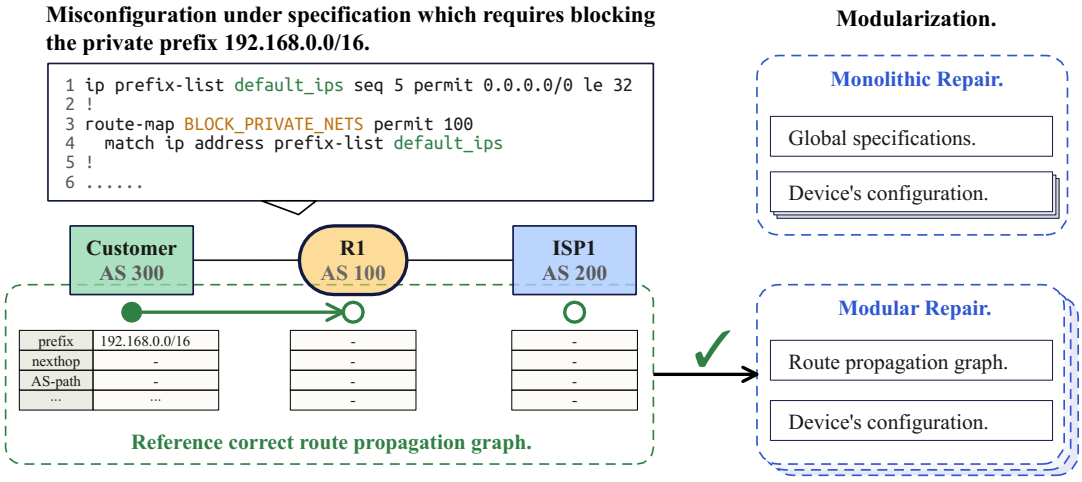

Figure 4: Misconfiguration and network topology, and reference correct route propagation graph to drive modular symbolic repair space computation.

repair space of the configuration location. Intuitively, this constraint means that the permitted prefix should not cover 192.168.0.0/16.

4 Modularization

The repair space computation described so far performs symbolic verification for each candidate configuration location against the global network specifications. While straightforward, this approach becomes expensive in large networks, as each location requires reasoning over the entire network configuration.

To improve scalability, we introduce a modular repair framework based on a reference route propagation graph [25]. The graph provides the routing context for each device by specifying the best routes received from neighbors and the route propagation decisions. Using this reference, repair space computation can be confined to device-level constraints, avoiding repeated reasoning over the entire network.

Because our modular reasoning is performed with respect to the reference route propagation graph rather than the global specifications, the approach may be incomplete. However, the reference graph is constructed from routing behaviors that satisfy the specifications. As a result, any repair derived under this reference preserves specification compliance, ensuring soundness.

In the following, we describe how the reference propagation graph enables modular repair space computation.

Reference correct route propagation graph. Network specifications often involve multiple devices and depend on the propagation of routes across the network. As a result, the behavior of a device cannot be reasoned about in isolation, since its routing decisions are influenced by routes learned from neighboring devices. This dependency makes it difficult to directly decompose the global repair problem into independent device-level reasoning tasks.

To enable modular reasoning, we therefore decouple specification enforcement from the monolithic global formula. Instead of embedding the full specification in each partitioned constraint, we preserve only its essential semantic components (e.g., target prefixes) and rely on a reference correct route propagation graph to provide the routing context. The reference graph can be obtained via simulation under a correct stable state [25] or from a previously verified configuration [11], and serves as an assume–guarantee interface for modular repair space computation.

As shown in Figure 4, each node in the route propagation graph represents a device's best route for the target prefix, and each directed edge represents a route propagation decision. Actual propagation may still be blocked by routing policies such as peer export or device import policies.

In Figure 4, the specification requires blocking the private prefix 192.168.0.0/16 originated by `Customer`. Due to a misconfiguration, `R1`'s import policy permits all IPv4 routes and propagates them to `ISP1`. In contrast, the reference route propagation graph reflects the intended behavior: only `Customer` originates the prefix, `R1` does not install it, and the route is not propagated to `ISP1`. The route propagation graph satisfies the specification and provides a concrete routing context that can replace the global specification for modular repair, focusing reasoning on relevant routing behavior.

Modular Repair Space Computation. As shown in Figure 4, a monolithic repair relies on a formula encoding that combines the global specifications with the entire network configuration, including the symbolically relaxed constraints for a repairable configuration location. Our framework can be extended to support the modular repair space computation by adopting device-level partitioned constraints and route propagation graph constraints.

In detail, a device-level partitioned constraint involves the configuration semantics around a single device, including the best routes received from peers, peer-to-device export policies, the device's import policies, the device's best route selection process, and the resulting best routes installed at the device. The encoding preserves the target prefix derived from the specifications, so that these elements together determine the device's routing behavior for that prefix.

Misconfiguration	Case		Can Repair	*Monolithic Repair*		*Modular Repair*	
				#RS (field / line)	Time	#RS (field / line)	Time
Single-field	prepend as-path	[ECMP]	✓	1 / 1	27s	1 / 1	3s
Single-field	prefix-list prefix	[ECMP]	✓	12 / 17	43s	12 / 17	5s
Single-field	prefix-list ge range	[REACHABILITY]	✓	29 / 30	105s	20 / 24	6s
Single-field	community-list action	[ECMP]	✓	2 / 2	29s	2 / 2	3s
Single-field	community-list community	[REACHABILITY]	✓	15 / 11	81s	4 / 4	3s
Single-line	prefix-list entry	[ECMP]	✓	11 / 17	43s	11 / 17	5s
Single-line	prefix-list entry	[REACHABILITY]	✓	28 / 30	104s	19 / 24	6s
Single-line	prefix-list entry	[ISOLATION]	✓	9 / 10	65s	8 / 10	3s
Single-line	community-list entry	[ECMP]	✓	1 / 2	28s	1 / 2	3s
Single-line	community-list entry	[REACHABILITY]	✓	15 / 11	80s	4 / 4	3s
Multiple-line	community-list + prepend as-path	[ECMP]	✗	0 / 0	26s	0 / 0	3s
Multiple-line	prefix-list + community-list	[ECMP]	✗	0 / 0	26s	0 / 0	4s
Multiple-line	two prefix-list entries	[REACHABILITY]	✓	27 / 30	104s	18 / 24	6s

Table 1: Repair space computation results under monolithic and modular repair strategies. We report the number of repair spaces (#RS) at field and line granularity, the computation time, and whether the misconfiguration is repairable.

The best routes and route propagation decisions therefore serve as interfaces between device-level constraint encodings. Since these behaviors are precisely captured, a reference correct graph provides the routing context at these interfaces. It can thus serve as an assume-guarantee interface, replacing the global specification to drive modular repair space computation. This ensures soundness, as the reference graph is derived from network behaviors satisfying the specifications.

5 Preliminary Evaluation

Our prototype extends Batfish [11] and Minesweeper [4] with 3,419 lines of Java code to support symbolic configuration locations, and implements the modular repair space plugin in 5,819 lines of Python.

Benchmark setup. We construct a benchmark based on common routing policy errors reported in ACR [14]. The benchmark includes misconfigurations in prefix-lists, community-lists, and AS-path prepending. The evaluation is conducted on a testbed consisting of 6 routers and 89 prefixes, comprising 289 lines of BGP routing policy configurations.

Our prototype currently supports single configuration location detection and repair (i.e., a configuration field or line). Using the benchmark, we perform a preliminary feasibility evaluation of repair space computation. Table 1 summarizes the computed repair spaces under both monolithic and modular repair strategies at field and line granularities, and indicates whether each misconfiguration can be successfully repaired.

Evaluation results. For single-field and single-line misconfigurations, all benchmark cases are successfully repaired, and multiple candidate repair spaces are often produced to help operators identify valid fixes. For multiple-line misconfigurations, only one benchmark can be repaired, due to the current limitation of our prototype, which primarily targets single-location detection and repair. The end-to-end computation time for repair spaces in modular repair ranges from 3s to 6s, while modular repair is on average 14× faster than monolithic repair, with improvements varying between 6× and 27×.

Our modular repair space computation is sound. In Table 1, the repair space computed by modular repair is always a subset of that computed by monolithic repair in terms of configuration locations.

6 Discussions

In this section, we discuss several limitations of our prototype and outline corresponding directions for future work.

Route attribute generality. Our prototype currently builds on Minesweeper [4] and inherits its support for symbolic route attributes (e.g., local-preference, MED, and communities). To support `prepend as-path` repair, we additionally encode AS-path length rather than the full AS-path list, since `prepend as-path` affects route preference only through path-length changes. However, this abstraction does not support `match as-path` configuration statement, as they depend on the full AS-path structure instead of its length-based representation.

Protocol generality. Our prototype currently focuses on IPv4 BGP configuration repair, as BGP exposes rich and widely used route-policy mechanisms. Since our approach is based on symbolic encoding and makes no assumption on the underlying routing protocols, it can be naturally extended to other protocols such as OSPF, and other configuration elements like ACLs.

Joint repair for multiple configuration locations. Our prototype currently computes repair spaces for a single configuration location. Extending the framework to support joint repair across multiple locations is conceptually straightforward. Specifically, configuration equality constraints and line enablement constraints can be relaxed into a MaxSMT optimization problem, allowing the solver to identify a minimum set of locations that must be jointly modified to restore correctness. The corresponding repair space can then be computed by treating the selected configuration variables as symbolic. A remaining challenge is that MaxSMT naturally returns a single minimum-change solution, making it harder to explore alternative combinations of repair locations that may better match operator preferences.

Additive repair for missing configurations. Our prototype currently supports modification and deletion repairs over existing configuration lines. To support additive repair, the symbolic relaxation framework can be extended with semantically neutral repair templates instantiated from protocol-specific configuration constructs whose line enablement variables are initially disabled. Missing configurations are then modeled as activating a subset of these template lines. By incorporating the corresponding enablement constraints into the MaxSMT formulation, the solver can determine the minimum set of additions required to restore correctness while simultaneously computing the associated repair space.

Repair space quality and explainability. Repair spaces characterize all valid updates that restore correctness for given configuration locations, allowing operators to choose repairs that best match their operational preferences. However, since specifications are often incomplete [26], a repair space may include changes that unnecessarily affect unrelated prefixes. To address this, repair space computation should preserve behavior outside the repair scope as much as possible, and explicitly identify any affected prefixes when such changes are unavoidable. To address this, repair space computation should preserve behavior outside the repair scope whenever possible and explicitly identify affected prefixes when such changes are unavoidable. This would help operators understand the operational impact of candidate repairs and select solutions that better align with deployment requirements.

Distinguishing verifier bugs from configuration errors. Our current formulation assumes that specification violations reported by the verifier accurately reflect underlying network behavior. In practice, however, violations may arise either from configuration errors or from bugs in the verifier itself [7]. A key distinction is that misconfigurations correspond to networks that are correctly encoded by the verifier but do not satisfy the given specification, whereas verifier bugs refer to cases where the verifier fails to correctly encode either the network configuration or the specification, leading to incorrect reasoning about the system. Distinguishing between these two causes remains an important challenge for automated repair systems. One possible direction is to leverage network emulation or simulation as an independent reference. By comparing verifier predictions with observed forwarding behavior, repair systems may be able to determine whether a reported violation originates from a configuration error or from incorrect verifier reasoning.

7 Related Works

Network verification. Minesweeper [4] encodes the entire network configuration symbolically and checks properties using SMT solving. Plankton [17] improves scalability via model checking and equivalence partitioning. Lightyear [19] further scales verification by adopting a modular approach based on local invariants. Differential network verification [22, 23, 26] moves beyond single-snapshot verification by tracking similarities and differences between pre-change and post-change network configurations, and supports "specification-less" verification.

Our work adopts the symbolic encoding strategies of Minesweeper [4], but focuses on the repair problem and computes repair spaces rather than verifying network properties.

Configuration repair. ACR [14] localizes faulty lines and generates fixes using templates, validating candidate repairs through incremental verification. Selectively symbolic simulation [24, 25] diagnoses and repairs routing configuration errors by symbolically relaxing selected configuration elements and comparing intent-compliant variants to the original configuration to derive patches.

In contrast, we compute the *repair space*: deriving localized constraints (subspecifications) that describe *all* valid updates at each candidate location, rather than a single patch. So operators need not iterate over candidate fixes; they can then choose one that matches their design rationale.

Configuration debugging. CEL [12] computes minimal correction sets to localize faulty lines. Campion [20] diagnoses configuration differences between two routers. There are also provenance-based approaches [8, 21, 27]. These works focus on pinpointing where faults lie or how configurations differ; we complement them by characterizing, at each candidate location, the space of valid repairs rather than only identifying that the location is faulty.

Explainable Program Synthesis. Subspecifications [15, 16] are introduced as localized logical constraints that describe what each part of a synthesized program must satisfy for the global specification to hold. We adopt the same principle and instead actively search for feasible fix locations and then use subspecifications to characterize the space of updates to restore correctness.

8 Conclusion

We propose a symbolic framework for characterizing repair spaces of misconfigurations. Instead of generating a single patch, our approach computes localized repair spaces via constraint relaxation and symbolic reasoning, and uses a reference route propagation graph to enable scalability computation. The framework provides interpretable guidance for network operators. We implemented a prototype, and preliminary results demonstrate its feasibility in representative network misconfiguration scenarios. And, we can extend the prototype with template-based repairs for missing configurations and multi-location symbolic relaxation, highlighting its potential to handle a wide range of misconfiguration scenarios and bridge the gap between verification and synthesis.

Acknowledgments

We sincerely thank the anonymous reviewers of APNet '26 for their valuable feedback on this paper. This work is supported by the ShanghaiTech Startup Fund.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 482–495.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 201–219.

[3] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2023. Modular control plane verification via temporal invariants. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 50–75.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 155–168.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 328–341.

[6] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 437–451.

[7] Rüdiger Birkner, Tobias Brodmann, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2021. Metha: Network verifiers need to be correct too!. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 99–113.

[8] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 115–128.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-wide configuration synthesis. In *International Conference on Computer Aided Verification*. Springer, 261–281.

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 579–594.

[11] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 469–483.

[12] Aaron Gember-Jacobson, Ruchit Shrestha, and Xiaolin Sun. 2022. Localizing router configuration errors using minimal correction sets. *arXiv preprint arXiv:2204.10785* (2022).

[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] Xu Liu, Peng Zhang, Anubhavnidhi Abhashkumar, Jiawei Chen, and Weirong Jiang. 2024. Automatic Configuration Repair. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. 213–220.

[15] Amirmohammad Nazari, Yifei Huang, Roopsha Samanta, Arjun Radhakrishna, and Mukund Raghothaman. 2023. Explainable program synthesis by localizing specifications. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 2171–2195.

[16] Amirmohammad Nazari, Yongzheng Zhang, Mukund Raghothaman, and Haoxian Chen. 2024. Localized Explanations for Automatically Synthesized Network Configurations. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. 52–59.

[17] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 953–967.

[18] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. 2021. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 33–49.

[19] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. 2023. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 94–107.

[20] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: debugging router configuration differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 748–761.

[21] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2014. Diagnosing missing events in distributed systems with negative provenance. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 383–394.

[22] Han Xu, Zachary Kincaid, Ratul Mahajan, and David Walker. 2026. Network Change Validation with Relational NetKAT. *Proceedings of the ACM on Programming Languages* 10, POPL (2026), 384–412.

[23] Xieyang Xu, Yifei Yuan, Zachary Kincaid, Arvind Krishnamurthy, Ratul Mahajan, David Walker, and Ennan Zhai. 2024. Relational network verification. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 213–227.

[24] Rulan Yang, Xing Fang, Lizhao You, Qiao Xiang, Hanyang Shao, Gao Han, Ziyi Wang, Zhihao Zhang, Jiwu Shu, and Linghe Kong. 2023. Diagnosing Distributed Routing Configurations Using Sequential Program Analysis. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*. 34–40.

[25] Rulan Yang, Gao Han, Hanyang Shao, Xiaoqiang Zheng, Xing Fang, Ziyi Wang, Lizhao You, Ruiting Zhou, Linghe Kong, Ennan Zhai, et al. 2026. Diagnosing and repairing distributed routing configurations using selective symbolic simulation. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 1635–1652.

[26] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. 2022. Differential network analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 601–615.

[27] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure network provenance. In *Proceedings of the twenty-third ACM symposium on operating systems principles*. 295–310.