Self-evolving network verifiers

Ioannis Protogeros
ETH Zürich
Zürich, Switzerland
iprotogeros@ethz.ch

Tibor Schneider
ETH Zürich
Zürich, Switzerland
sctibor@ethz.ch

Laurent Vanbever
ETH Zürich
Zürich, Switzerland
lvanbever@ethz.ch

ABSTRACT

Symbolic network verifiers can reason about correctness across vast spaces of routing inputs and failures, but only for the protocols and features an expert has encoded by hand. Creating and maintaining a faithful model of the control plane is both difficult and never-ending, since no written source specifies perfectly what a network does: vendor implementations deviate from the RFCs, and behaviour shifts with releases. The burden of constant upkeep ultimately keeps verification out of many networks that need it.

We argue that the model should instead evolve automatically to faithfully capture the actual network behaviour. To achieve that, we leverage the only source that specifies it unambiguously: the router software itself. In a counterexample-guided loop, a coding agent proposes extensions to the verifier's symbolic encoding, while a trusted oracle (e.g., emulated routers) supplies the ground-truth routing state. The agent iteratively refines the network model using each disagreement with the oracle.

As early evidence, a prototype of this system taught a 3,000-line SMT-based verifier three features it did not support: OSPF areas, BGP route reflection, and L3VPN over EVPN, converging autonomously on models that match the oracle, even noticing vendor-specific behaviour. Automating model growth shifts the hard problem from writing verification systems to systematically testing them; we propose a research agenda for trusting and harnessing automatically evolved verifiers.

CCS CONCEPTS

• **Networks** → **Network reliability**; **Network manageability**; • **Software and its engineering** → *Software evolution*; *Formal software verification*; • **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

Network verification, Control-plane verification, SMT, CEGIS, LLM agents, Differential testing

ACM Reference Format:
Ioannis Protogeros, Tibor Schneider, and Laurent Vanbever. 2026. Self-evolving network verifiers. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16–17, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3847290.3859935

Please use nonacm option or ACM Engage class to enable CC licenses



This work is licensed under a Creative Commons Attribution 4.0 International License.

HotNets '26, November 16–17, 2026, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859935

1 INTRODUCTION

Network verification is remarkably powerful: given only a configuration, modern tools can reason about network behaviours across all possible routing inputs and link failures [4, 6]—at least, in theory.

In practice, though, a decade of academic success has not translated (yet) into (vast) industry adoption. In a recent survey of network operators, the most frequently stated deterrent is that existing tools do not support the protocols and features their networks run [21]. As a result, most networks remain unverified and continue to suffer the outages that verification was built to prevent [33].

The reason can be traced to a single artifact: the network control-plane *model*, which is today built entirely by hand. Existing verifiers indeed support only the features their authors have manually encoded, which is only a subset of what operators actually run. This is not surprising: accounting for all the different protocols and vendor implementations is a Sisyphean task. (The maintainers of Batfish already report a decade's worth of extending vendor and protocol support and of chasing undocumented semantics in router software [8].) Worse, extending models is often disproportionately hard because of how protocols interact. For instance, a BGP verifier cannot "only" model the BGP computation across vendors and implementations; it also needs to model the IGP computation logic, again across vendors and implementations, as the BGP's route selection depends on the IGP costs [4].

These limits have pushed recent works to make radical proposals: in [21], the authors argue that *"models will always lag in their ability to faithfully represent the actual control plane"*. They propose abandoning control plane models altogether, and instead running the configuration on an emulated network, verifying the resulting data plane directly. Such

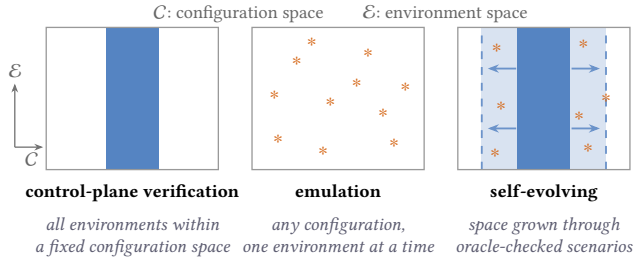


Figure 1: A self-evolving verifier keeps the symbolic model's guarantees and grows their coverage, guided by ground-truth observations.

model-free approaches overcome the limitations of hand-crafted models: any feature the router image supports is covered, the image is always up-to-date, and there is no encoding to extend.

Foregoing models comes with a high cost, though: the loss of widely applicable guarantees. For a single converged data plane, we can easily reason about properties without using a model; we just run the configuration and then verify correctness over the packet space [19, 20]. But whether the network *stays* correct across environments (any link failure, any external advertisement) is a question that a single data plane cannot answer. Most of the time, the environment space is huge and combinatorial, making it intractable to enumerate every possible data plane. Relying on a (symbolic) model [6] is precisely what enables control-plane verification to reason about all environments at once.

In this paper, we argue that the problem is *not* relying upon a model but that humans write it. To address this, we describe a vision in which symbolic models *automatically* evolve to faithfully represent the behaviour of the network they describe (Fig. 1). In this world, models automatically cover new features, adapt to new vendors or software updates, without any expert labour. More importantly, they do so without surrendering the vast guarantees they provide.

Realizing this vision is, quite unsurprisingly, difficult. Yet, we believe the recent advances in network emulation, coupled with the advances in LLM-based code generation, make it possible. Concretely, we treat the network model like software under test: we continually compare what the model admits against what the network computes, and let an LLM coding agent repair the model wherever the two disagree.

Any procedure that changes the model automatically needs a criterion for when a change is correct [16]. In networking, deriving such a specification is exceedingly difficult because no written source exists that specifies what a network actually does. Protocol implementations diverge from the RFCs [35], vendor documentation is incomplete [8], protocols interact in ways no standard describes [7], and behaviour shifts across even minor releases. The one artifact that fully determines a network's behaviour is the router software, and unlike a written specification, it can be executed: emulation can therefore answer, for any configuration in one concrete environment, exactly what the model should say.

We therefore propose to use this executable ground truth to drive a counterexample-guided synthesis loop: for each configuration and concrete environment, we check that the symbolic model is consistent with the routing state the network computed. Every disagreement becomes a counterexample, and an off-the-shelf coding agent turns counterexamples into candidate repairs (current LLMs have been shown to be effective in various adjacent domains [22, 23, 25, 34]). The specification, in short, is the network you run, not the one the standards describe: in our experiments (§3.1), the agents initially followed RFC 4456's route-selection order [2], but the ground truth followed a different order [13]; counterexamples guided the agents to it.

We present early evidence that our vision is feasible. On a 3,000-line Rust SMT verifier, this pipeline autonomously added three mechanisms the model did not support: OSPF areas, BGP route reflection, and L3VPN over EVPN, each for a few tens of dollars and with no expert in the loop beyond inspecting the end result.

This approach aims to yield a rigorously tested (though not verified) extended verifier. It drastically reduces the cost of obtaining a *possibly correct* network model, but the hard problem shifts from writing verifiers to validating them systematically: what makes a corpus of tests adequate, and how far can we trust a verifier beyond the tested region? Our early work shows that the loop works end-to-end on three features, but also surfaces failure modes: a verifier that passed all tests still failed on new inputs, and encodings that agreed on every test differed by orders of magnitude in solving time. The need for systematic testing is not specific to our approach. Verifiers today are spot-checked by hand, with no notion of coverage, and even widely used ones contain bugs [7, 21]. Knowing which parts of the configuration and environment space have been validated would already be a step forward. Scaling this vision to more difficult scenarios requires tackling fundamental problems that we discuss in §4: defining and achieving coverage of the scenario space when each test checks a single environment, and making encodings tractable without sacrificing correctness.

2 SELF-EVOLVING VERIFIER

This section makes the vision concrete: we state the problem (§2.1), show how to test a symbolic model against an Oracle (§2.2), close the loop that repairs it (§2.3), and state what a passing run does (and does not) certify (§2.4).

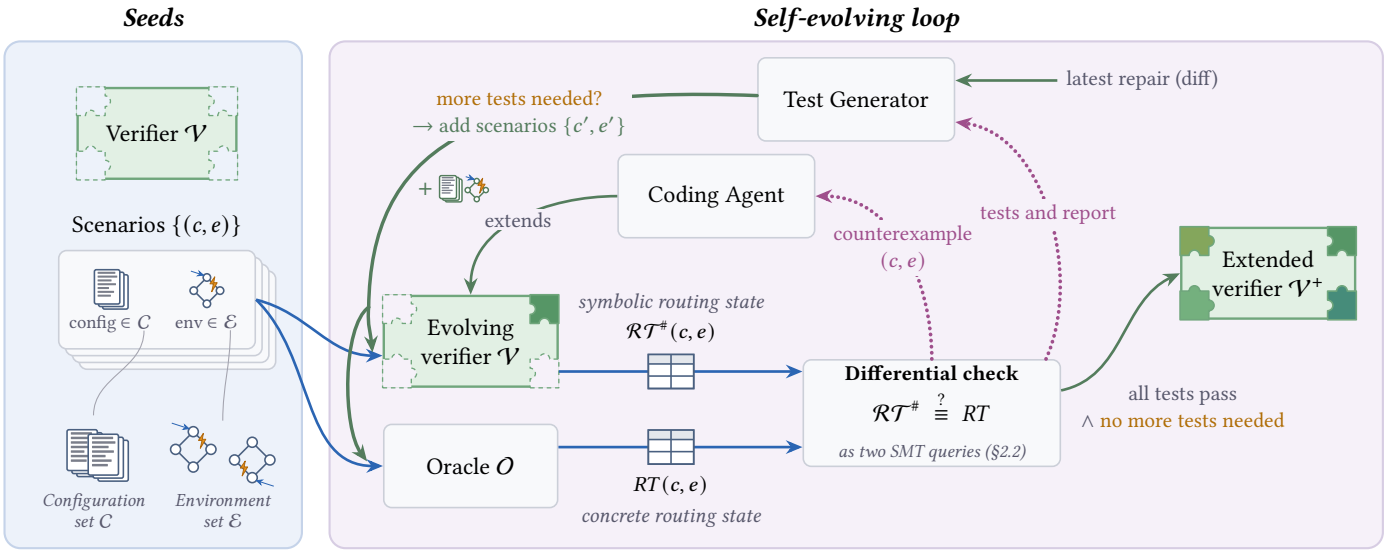


Figure 2: Every disagreement between the symbolic model and the Oracle becomes a counterexample that drives the next repair; the loop ends only when the whole corpus passes *and* the Test Generator can grow it no further.

2.1 Problem statement

We are given a set of configurations C, the environments $\mathcal{E}$ they may face—link failures, external advertisements—and a verifier $\mathcal{V}$ that cannot reason correctly about every *scenario* (c, e): a configuration paired with one concrete environment. We also have an Oracle O we trust (a simulator, emulator, or the network itself), which, given a scenario, returns the routing state the network converges to. For simplicity, we consider only networks where this state is unique (otherwise, every notion of agreement below would range over all possible stable routing states [14, 15]).

Our goal is to produce an *extended* verifier $\mathcal{V}^{+}$ that agrees with the Oracle on all scenarios. More concretely, the verifier's model must accept the routing state the Oracle produces and must accept no other. As with any tested system, if a large and varied corpus cannot surface any difference between the model and the Oracle, we trust the model across all configurations in C and environments in $\mathcal{E}$.

2.2 Differentially testing an SMT verifier

Even for a concrete configuration and environment, a symbolic verifier does not compute a routing table. It produces constraints over symbolic routes $r^{\#}$, and every assignment that satisfies them is a routing state the model considers possible. A correct model admits exactly the state the Oracle observes. We therefore check the Oracle's table RT against the constraints in both directions, for a given c and e:

Soundness: Does the generated SMT model reject a routing state that is observed through the Oracle O? We can effectively check soundness by crafting an SMT query that asks *"Given the environment e, is it possible for the routes in the SMT model to be the same as the ones produced by the Oracle?"*. If we denote as r each route in the routing table calculated by the Oracle $RT = O(c, e)$, and $r^{\#}$ its respective symbolic route under the SMT model, then that query would be

$$e \wedge \left(\bigwedge_{r \in RT} r = r^{\#} \right) \rightsquigarrow (\textit{should be } \textsc{sat})$$

Here $r = r^{\#}$ asserts, for a concrete route r and its symbolic counterpart at the same (router, destination) slot, agreement on availability and on every route attribute (next hop, local preference, AS-path, and so on). A SAT result here exhibits a model assignment matching reality; UNSAT means the encoding forbids the real state.

Completeness: Does the generated SMT model accept a routing state that cannot be observed through the Oracle O? Conversely, we check completeness by querying whether the symbolic encoding can disagree with the Oracle's state.

$$e \wedge \left(\bigvee_{r \in RT} r \neq r^{\#} \right) \rightsquigarrow (\textit{should be } \textsc{unsat})$$

A SAT result here would mean that the model accepts a spurious state, revealing a bug (or an omission) in the encoding.

As stated in §2.1, these queries assume a unique stable state. With several, the model must admit exactly all the

stable states that can appear. We can check whether a given state is stable (so admitting an unstable one would be a counterexample). However, deciding whether a routing state can be reached given a configuration is much more difficult: it requires exploring the huge space of routing states the configuration can produce; *guided state exploration* [28] does this efficiently, but relies on a partial model of BGP. For properties that must hold in every converged state, such as reachability or loop freedom, admitting stable but unreachable states is a sound overapproximation.

2.3 The evolution loop

The differential testing approach above turns a single scenario into a verdict on the current encoding, but a verdict alone does not repair the model. Two agents close the loop (Fig. 2): a *Coding Agent* that continually refines the verifier's symbolic encoding, and a *Test Generator* that extends the scenario corpus. The proposed workflow using the two agents is not set in stone; other workflows, including agents dynamically orchestrating verifier implementation and test generation, could also produce an extended verifier. What must stay fixed is that whoever implements the verifier neither controls the tests that judge it nor decides when testing is finished.

The loop follows counterexample-guided inductive synthesis (CEGIS) [32], with an LLM agent as the inductive synthesizer [25], and with the specification that would ordinarily bound the search replaced by the Oracle (a form of OGIS [18]). We seed the corpus with scenarios (c, e) and run the differential test on each of them. A scenario in which either query fails is a counterexample: a configuration and environment in which the encoding and the Oracle disagree. We hand it to the Coding Agent along with the failing query and the Oracle's routing state; the agent edits the encoding, we recompile, and the entire corpus is re-run so that a fix for one feature cannot regress another. Iteration continues until every scenario passes.

Passing the corpus certifies the encoding only on the scenarios it contains, so it is not, on its own, a stopping condition. We therefore ask the Test Generator whether any untested region of the scenario space remains. If it can produce a new scenario that the encoding and Oracle have not yet been checked against, that scenario enters the corpus, and the loop resumes; only when all tests pass, and no further scenario is needed, do we emit the extended verifier $\mathcal{V}^+$.

2.4 What a passing run means

A successful test run certifies that, for every scenario in the corpus, the verifier encoding admits the Oracle's routing state, and no other. Any claim beyond the corpus is inductive, so naturally, there can still be uncaught errors; untested regions of the joint configuration and environment space may still hide modelling bugs. The corpus is thus an artifact in its own right, as it can specify the region in which the verifier is empirically known to agree with the network.

While we recognize that such empirical agreement is weaker than a proof, hand-written encodings do not come with proofs either; they are validated with sample tests, expert review, and hand-maintained comparisons against ground truth (e.g., Batfish against emulated networks [8], Alibaba's Hoyan continuously against observed scenarios in its live WAN [35, 36]), and prominent verifiers ship fidelity bugs regardless [7, 21]. Irrespective of how a verifier is created, its testing needs to be systematized, and cheap emulation makes coverage of the scenario space a realistic goal.

3 EARLY RESULTS

We implemented and ran the loop of §2 on an SMT-based control-plane verifier (~3000 lines of Rust using Z3 [11]) that models BGP over a flat shortest-path IGP. We gave it three extension tasks for mechanisms it did not model: **(i) OSPF areas**, **(ii) BGP route reflection** (RFC 4456), including cluster lists, originator IDs, and their effect on best-path selection; and **(iii) L3VPN over EVPN**, a feature family the verifier had no notion of. The oracle is hot-swappable, and we used two oracles of different fidelity: *Batfish* [12] for (i) and (ii), and emulated *FRRouting* instances (via containerlab) for (iii), where L3VPN support is more complete. A general *Coding Agent* (Claude Opus 4.8 or Sonnet 5) is tasked with implementing the update to the existing symbolic verifier; the *Test Generator* is a separate agent (Opus 4.8 throughout) that reads the implementer's code and existing test cases each round and proposes new scenarios.

Table 1 summarizes the runs. For tens of dollars and ~100–250 agent steps per feature, every run converged to a verifier that fully agreed with its oracle on a test corpus adversarially grown by the Test Generator. Not every model we tried sufficed: runs with Haiku 4.5 as the Coding Agent failed to converge on any of the three tasks. Model capability alone did not suffice either: where the ground truth deviates from the RFCs or behaves in ways they do not specify (§3.1), the agents reached the correct encoding only through the Oracle's counterexamples.

3.1 What the agents modelled

Figure 3 distills each converged encoding into the rules the agent effectively added, as faithful abridgements of the agents' Rust/Z3 code.

OSPF areas (Fig. 3i). Both agents independently converged on the same formulation. They defined symbolic per-area distances and composed them into inter-area distances dependent on link availability. Correctly modelling shortest-path

	OSPF areas		Route refl.		L3VPN	
Coding Agent model	Opus	Sonnet	Opus	Sonnet	Opus	Sonnet
Tests (seed→final)	316→544	316→648	484→600	484→600	508→596	508→582
Rounds (failing)	7 (3)	9 (1)	5 (1)	4 (0)	13 (3)	11 (2)
Agent steps	95	91	200	173	218	254
Total cost ($)	41.8	35.8	30.1	18.7	37.2	25.0
Lines added	288	275	85	152	1162	1060
Final passing tests	544/544	648/648	600/600	600/600	596/596	582/582

Table 1: Every run converged to full agreement with its oracle on an implementation-aware testing suite.

calculation under the partitioning of Area 0 was a subtlety that required both agents to refine their code through counterexamples.

Route reflection (Fig. 3ii). Both agents converged on FRR's decision process, even on a detail that deviates from the route reflection RFC: in *Batfish*, FRR compares cluster-list length right after the IGP-metric step, ahead of the originator-ID comparison [3], whereas RFC 4456 §9 prescribes the opposite order. Originally, the agents implemented an SMT model from their knowledge of the protocol, so the route decision process followed the RFC. They adapted to the FRR logic only through counterexamples authored by the Test Generator.

L3VPN over EVPN (Fig. 3iii). Both agents converged on the encoding shown, including all eight terms of the decision process $\prec$. There were, however, subtleties in the network that the RFCs do not specify: a prefix advertised from two sites has a different best route at its origin than in the rest of the network, because a router privately prefers its own routes through an attribute that never leaves the router. The router software had to reveal such behaviour [13].

(i) OSPF areas

DEFINE per-area distances $d_a(u, v)$: symbolic shortest paths over the area-a links that are up in the environment

DEFINE $D(s,t) = \min_{\substack{x \in \mathrm{ABR}(a_s) \\ y \in \mathrm{ABR}(a_t)}} d_{a_s}(s,x) + d_0(x,y) + d_{a_t}(y,t)$

↪ Despite leading to the exact same result, Opus considered the min over a much vaster space (§3.3)

ASSERT each path segment is reachable within its area

ASSERT an intra-area route, when one exists, beats every inter-area route, regardless of cost

(ii) Route reflection

DEFINE per-route attributes: cluster list CL, originator ID OID

DEFINE $CL \leftarrow cid(r) :: CL$ ▷ *reflector r prepends its cluster ID*

DEFINE $OID \leftarrow$ *sender* if unset

ASSERT router t admits a route iff $OID \neq t \wedge cid(t) \notin CL$ ▷ *loop rejection*

DEFINE lexicographic minimization along ▷ *best-path selection*
$\cdots \succ$ IGP cost $\succ |CL| \succ OID \succ$ peer ID

(iii) L3VPN over EVPN

GIVEN routers R, weighted core links L, EVPN sessions E, VRF attachments, originated prefixes O

DEFINE $d(u, v)$: shortest paths after deleting failed nodes and links

DEFINE $act(u, v) \Leftrightarrow (u, v) \in E \wedge$ both up $\wedge\ d(u, v) < \infty$
▷ *EVPN session liveness*

DEFINE $o \rightsquigarrow t$: RFC 4456 reflection reachability over act ▷ *overlay RR*

DEFINE $rt(v)$ = ASN:VNI if auto, $rd(v)$ = routerID:vrfID if auto
▷ *route target / distinguisher*

DEFINE a type-5 route is $\langle rd(o),\ p,\ RT(o),\ \mathcal{A}(o) \rangle$ ▷ *RD keys identity*

DEFINE $\prec = (\neg weight,$ LP, |ASpath|, origin, MED, eBGP$\prec$iBGP, IGP cost, router ID$)$ ▷ *full decision process*

DEFINE $ebest(r, rd, p) = \min_{\prec} \{\ o : rd(o) = rd \wedge d(r, o) < \infty \wedge o \rightsquigarrow r\ \}$
▷ *EVPN RIB: selection per (RD, prefix),* not *across RDs*

DEFINE $cand(r, v, p) = \{\ ebest(r, rd, p) : rt^{\mathrm{imp}}(r, v) \in RT(o) \wedge rd(o) \neq rd(v)\ \}$ ▷ *leak: import filter + self-RD rejection*

DEFINE $best(r, v, p) = \min_{\prec} cand(r, v, p)$
▷ *tenant v4 RIB: cross-RD competition resolved here*

Figure 3: The two agents converged to semantically equivalent encodings.

3.2 Tests must co-evolve with the model

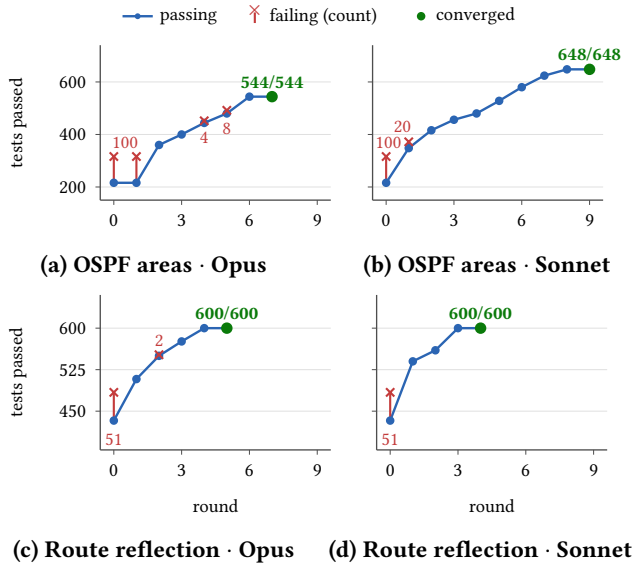
Figure 4 traces how a run's corpus and passing checks move together: in each round, the Test Generator tries to grow the corpus with adversarial examples, and each red cross is a round that ended with counterexamples. The runs share a common structure. The verifier's initial state fails a large fraction of the seed corpus—unsurprisingly so, since a lot of the seed scenarios exercise the very feature being added (e.g., 100 of 316 for OSPF areas)—and the agent repairs against those counterexamples until the seed passes. Had the loop stopped there, the model would *look* finished. Yet in every run, the Test Generator, reading the converged code, invented scenarios the seed never stressed, and some promptly failed. Figure 5 follows one such trajectory in detail.

Passing the current corpus is thus a fixed point of *repair*, not of correctness: each repaired model has behaviour that only fresh, code-aware tests can probe. Our Test Generator finds such tests, but greedily and without guarantees: nothing tells us how far "the Test Generator ran out of ideas" is from "no counterexample exists." Closing that gap, e.g. by generating adversarial scenarios systematically, with some notion of coverage over a targeted scenario space, is, we argue, a core open problem for self-evolving verifiers (§4).

3.3 Performance is important too!

Consistency with the oracle is the main signal that drives the loop of §2. But a verifier has (at least) two different requirements: *correctness*, a hard constraint that admits no tradeoff, and *performance*, a soft objective the model can satisfy to wildly varying degrees. The oracle enforces only the first on the test corpus; computational tractability is invisible to it.

We benchmarked each pair of converged network models on the same query—proving reachability under all single-link failures—over topologies at growing sizes. Each pair yields the same verdict, yet their costs diverge by orders



(a) OSPF areas · Opus (b) OSPF areas · Sonnet

(c) Route reflection · Opus (d) Route reflection · Sonnet

Figure 4: An evolved verifier that passes all tests in a single round might still fail on different inputs.

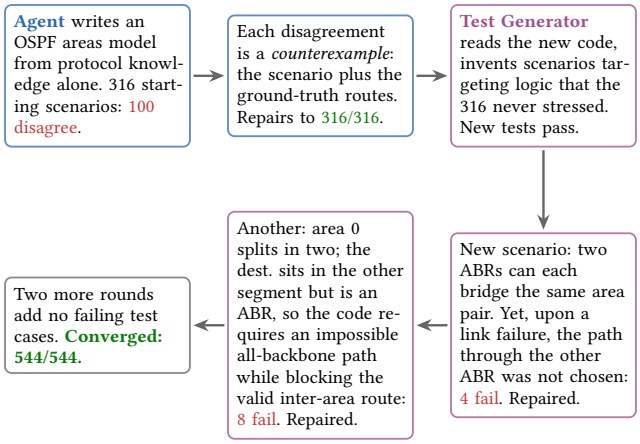


Figure 5: The Test Generator proposes adversarial scenarios until it cannot invent further ones.

of magnitude. For OSPF areas, Opus's encoding considers vastly more candidate inter-area paths than are actually possible, while Sonnet derives each area's border routers from the configuration and admits only paths through them. The redundant computations compound with scale: at just 18 routers, Opus's encoding cannot answer the query within 20 minutes, while Sonnet's takes 0.16 s. For L3VPN at 17 routers, Sonnet's model takes 143 s, whereas Opus's takes 0.9 s (all measurements are means over 10 runs).

To test whether the loop can also pursue the performance objective, we resumed the converged Opus OSPF run with a single generic instruction (the encoding must also scale) and a cheap size signal (SMT DAG nodes), with no hint at the underlying anti-pattern. In two rounds the agent independently reinvented the ABR restriction: correctness held at every step (544/544 against the oracle), and the once-intractable query now completes in 0.4 s. A tangible gap remains, however: Sonnet's encoding is still over 2× faster on the same proof, so even within verifiers with similar network models and the same end results, performance may vary significantly.

4 DISCUSSION & RESEARCH AGENDA

Our prototype deliberately uses an off-the-shelf agent in a straightforward CEGIS loop to show that the running network, used as an oracle, can guide the synthesis of non-trivial control-plane extensions. Scaling this approach to enterprise configurations with denser protocol interactions and tests that are even more difficult to derive raises fundamental problems. This early work aims to draw attention to them; for each, we outline why it is hard and a direction forward.

Systematic testing of verifiers. The system in §2 drastically cuts the cost of writing a verifier, leaving only the task of rigorously testing one. The need is not ours alone: the operators of Alibaba's production verifier explicitly call for an automated framework that tests a verifier against vendor implementations and diagnoses the causes of divergence [36]. We see the goal as a certificate: an explicit set of hypotheses $\mathcal{H}$ under which a verifier has been validated against the network. If a certified verifier is still wrong, some hypothesis in $\mathcal{H}$ fails, which tells us where to look. While our Test Generator does expose failures by probing logic that the previous test suite never stressed, it cannot guarantee that every corner case is covered.

Discriminating tests are hard to find. Scenarios that surface a discrepancy between the model and the network are sparse: validating the order of a decision process at step k requires routes with equal attributes on all previous steps, as in the route-reflection tie-break of §3.1. The environment space adds a second dimension, since each test checks a single environment while the verifier's guarantee covers all of E. Our Test Generator greedily finds such scenarios. Recent work [37] shows adversarial inputs uncover correctness and performance bugs in AI-evolved programs (e.g., [24, 26]) that their original evaluation suites miss; an evolved verifier likewise needs adversarial scenarios.

A natural approach towards structurally covering the scenario space is partitioning the joint configuration and environment space into equivalence classes (EC), and validating each one with a single scenario. The hypothesis in $\mathcal{H}$ in this case is that the network treats all members of an EC uniformly, so that agreement on one tested member implies agreement on the class. The difficulty is where the classes

come from. Symbolic execution [9] derives them from a program's structure, but classes derived from the verifier inherit its blind spots: a feature the model omits yields a class whose members the network treats differently. The classes must therefore come from the implementation, yet symbolically executing production routing daemons does not scale [23, 31], and environments such as link failures are network-wide, so analysing a single daemon does not suffice. One promising direction is to restrict the analysis to the slice of the daemon exercised by the configuration features the verifier supports. A weaker measure is also available: in model checking, a part of the model counts as covered only if mutating it changes the verdict of some test [10, 17]. Applied to an evolved verifier, it reveals logic the tests never exercise, though not logic the model lacks.

Optimizing for performance. The oracle enforces correctness on the corpus, yet computational tractability is invisible to it, and §3.3 showed that verifiers that pass the same tests can differ in efficiency by orders of magnitude. The scalability of the starting verifier does not hinder the loop: each differential test fixes a concrete environment, so its queries stay cheap even for encodings that cannot answer a universal query at scale. The loop can therefore first converge to a faithful but slow model, and optimize it afterwards. The main lever is the boundary between what can be precomputed from the configuration (e.g., an area's border routers) and what must stay symbolic (distances under failures) [5]. That boundary moves with the query: a reachability check on a fixed topology has no need for symbolic distances. Taken to its limit, concretizing everything is simply enumeration (one emulation per environment), which for spaces like all single-link failures may well beat a symbolic query, and definitely beats an unfortunate SMT encoding. If deriving a model is cheap, a verifier need not fix this boundary once; it could choose symbolic reasoning or enumeration per calculation.

Automating such optimizations needs two signals. For correctness, a first converged verifier can serve as the reference for faster ones, as a correct but slow program does in *superoptimization* [27]. Known optimizations such as Minesweeper's *hoisting* and *slicing* [4] could be validated this way. For performance, a reliable signal is still missing: encoding size let the agent remove the anti-pattern of §3.3, but left an over-2× gap to the faster encoding. Finding a signal that predicts solving time remains open.

Reasoning about more properties. The loop evolves toward whatever its oracle tests against, here the converged routing state (§2.2). However, operators also care about transient states [29], and performance properties still largely remain out of reach: general formal models for network performance scale poorly [1], and verifying specific properties, such as worst-case link loads, requires modelling and optimizing entire systems by hand [30]. Emulation already observes transient states, link loads, and convergence times, so the question remains: how can we use these signals to effectively steer the creation of a faithful and tractable network model?

Universal or ad-hoc verifiers? If a faithful network model costs days and tens of dollars instead of months of expert labour, what should the community maintain: one "universal" verifier, or a small verifier per network, covering exactly its configuration and re-evolving as it changes? Our results favour the latter, since each verifier automatically adapts to vendor-specific behaviour, whereas general configuration analysis tools like *Batfish* must encode all possible vendor-dependent behaviours and select the applicable one per configuration. What the universal verifier offers instead is more comprehensive validation: every scenario tests the same code, whereas a per-network verifier is tested only in its own domain. Yet a hard scenario found by one network's loop is an oracle-checked, reusable test for any verifier that covers the same features. A community corpus of such scenarios would give ad-hoc verifiers the shared validation that universal ones enjoy today.

ACKNOWLEDGMENTS

We thank our shepherd, Behnaz Arzani, for her feedback, which substantially improved this paper.

REFERENCES

[1] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. 2023. Formal Methods for Network Performance Analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 645–661. https://www.usenix.org/conference/nsdi23/presentation/tahmasbi

[2] Tony Bates, Enke Chen, and Ravi Chandra. 2006. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456. https://doi.org/10.17487/RFC4456

[3] Batfish Project. 2025. Batfish v2025.07.07, `BgpRib.java`. https://github.com/batfish/batfish/blob/v2025.07.07/projects/batfish/src/main/java/org/batfish/dataplane/rib/BgpRib.java.

[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 (SIGCOMM '17)*. 155–168. https://doi.org/10.1145/3098822.3098834

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/3230543.3230583

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2020. Abstract Interpretation of Distributed Network Control Planes. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 42 (2020).

[7] Rudiger 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). USENIX Association, 99–113. https://www.usenix.org/conference/nsdi21/presentation/birkner

[8] 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 the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3603269.3604866

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[10] Hana Chockler, Orna Kupferman, and Moshe Vardi. 2003. *Coverage Metrics for Formal Verification*. Vol. 8. 111–125. https://doi.org/10.1007/978-3-540-39724-3_11

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. *Z3: An Efficient SMT Solver*. Vol. 4963. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[12] 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 the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 469–483.

[13] FRRouting Project. 2026. FRRouting 10.6.1, `bgpd/bgp_route.c`, function `bgp_path_info_cmp`. https://github.com/FRRouting/frr/blob/frr-10.6.1/bgpd/bgp_route.c.

[14] Lixin Gao and J. Rexford. 2001. Stable Internet routing without global coordination. *IEEE/ACM Transactions on Networking* 9, 6 (2001), 681–692. https://doi.org/10.1109/90.974523

[15] T.G. Griffin, F.B. Shepherd, and G. Wilfong. 2002. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking* 10, 2 (2002), 232–243. https://doi.org/10.1109/90.993304

[16] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4 (07 2017), 1–119. https://doi.org/10.1561/2500000010

[17] Y. Hoskote, T. Kam, Pei-Hsin Ho, and Xudong Zhao. 1999. Coverage estimation for symbolic model checking. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*. 300–305. https://doi.org/10.1109/DAC.1999.781330

[18] Susmit Jha and Sanjit A. Seshia. 2016. A Theory of Formal Synthesis via Inductive Learning. arXiv:1505.03953 [cs.AI] https://arxiv.org/abs/1505.03953

[19] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[20] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 15–27. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[21] Alexander Krentsel, Oliver Ye, Anthony Tafoya, Xuqian Ma, Sylvia Ratnasamy, and Anees Shaikh. 2025. Towards Accessible Model-Free Verification. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks* (UMD Campus, College Park, MD, USA) *(HotNets '25)*. Association for Computing Machinery, New York, NY, USA, 210–217. https://doi.org/10.1145/3772356.3772380

[22] Lezhi Ma, Shangqing Liu, Yi Li, Qiong Wu, Han Wang, and Lei Bu. 2026. SpecSyn: LLM-based Synthesis and Refinement of Formal Specifications for Real-world Program Verification. arXiv:2604.21570 [cs.SE] https://arxiv.org/abs/2604.21570

[23] Rajdeep Mondal, Rathin Singha, Todd Millstein, George Varghese, Ryan Beckett, and Siva Kesava Reddy Kakarla. 2026. Eywa: Automating Model-Based Testing using LLMs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 777–800. https://www.usenix.org/conference/nsdi26/presentation/mondal

[24] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131 [cs.AI] https://arxiv.org/abs/2506.13131

[25] Pedro Orvalho, Mikoláš Janota, and Vasco M. Manquinho. 2025. Counterexample guided program repair using zero-shot learning and MaxSAT-based fault localization. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'25/IAAI'25/EAAI'25)*. AAAI Press, Article 73, 9 pages. https://doi.org/10.1609/aaai.v39i1.32046

[26] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2023. Mathematical discoveries from program search with large language models. *Nature* (2023). https://doi.org/10.1038/s41586-023-06924-6

[27] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[28] Tibor Schneider, Jean Mégret, and Laurent Vanbever. 2025. Guided Exploration of Control-Plane Routing States. In *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*. 1–12. https://doi.org/10.1109/ICNP65844.2025.11192328

[29] Tibor Schneider, Roland Schmid, Stefano Vissicchio, and Laurent Vanbever. 2023. Taming the transient while reconfiguring BGP. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 77–93. https://doi.org/10.1145/3603269.3604855

[30] Tibor Schneider, Stefano Vissicchio, and Laurent Vanbever. 2025. Verifying maximum link loads in a changing world. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 1269–1287. https://www.usenix.org/conference/nsdi25/presentation/schneider

[31] Rathin Singha, Rajdeep Mondal, Ryan Beckett, Siva Kesava Reddy Kakarla, Todd Millstein, and George Varghese. 2024. MESSI: Behavioral Testing of BGP Implementations. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1009–1023. https://www.usenix.org/conference/nsdi24/presentation/singha

[32] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Ph. D. Dissertation. USA. Advisor(s) Bodik, Rastislav.

[33] 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 (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 748–761. https://doi.org/10.1145/3452296.3472925

[34] Hongshu Wang, Xinyue Zuo, Yuhan Sun, Qin Li, Yamine Ait Ameur, and Jin Song Dong. 2026. Event-B Agent: Towards LLM Agent for Formal Model Synthesis and Repair. arXiv:2605.17475 [cs.SE]

[35] 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 the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 599–614. https://doi.org/10.1145/3387514.3406217

[36] Yifei Yuan, Fangdan Ye, Yifan Li, Jingkai Zhang, Mengqi Liu, Yuyang Sang, Ruizhen Yang, Duncheng She, Zhiqing Ye, Tianchen Guo, Xiaobo Zhu, Xinji Tang, Li Jia, Zhongyu Guan, Lingpeng Su, Ci Wang, Ruiyang Feng, Shuo Wu, Zhonghui Xie, Cheng Jin, Peng Zhang, Qing Ma, Xianlong Zeng, Dennis Cai, and Ennan Zhai. 2025. New Evolution of Hoyan: Enhancing Scalability, Usability, and Accuracy for Alibaba's Global WAN Verification. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 809–825. https://doi.org/10.1145/3718958.3754343

[37] Yajie Zhou, Ao Li, Ashwin Silla, Zaoxing Liu, and Vyas Sekar. 2026. AIChilles: Automatically Uncovering Hidden Weaknesses in AI-Evolved Systems. arXiv:2606.15834 [cs.AI] https://arxiv.org/abs/2606.15834