Towards Learned Switch Behavior Modeling

Daqian Ding
The University of Hong Kong
Hong Kong, China
daqian.ding0@gmail.com

Ziqian Liu
The University of Hong Kong
Hong Kong, China
liuziqian@connect.hku.hk

Zhixiong Niu
Microsoft Research
Beijing, China
zhniu@microsoft.com

Ziyu Mao
The University of Hong Kong
Hong Kong, China
zerrymore@gmail.com

Jingyu Wang
Zhongguancun Laboratory
Beijing, China
wangjy@zgclab.edu.cn

Yongqiang Xiong
Microsoft Research
Beijing, China
yqx@microsoft.com

Yiming Qiu
The University of Hong Kong
Hong Kong, China
yimingq@hku.hk

Abstract

Commodity switches are ubiquitous in production networks, yet their fixed-function packet-processing behavior remains largely opaque: vendors provide coarse documentation and control interfaces, but not an executable specification of how packets are processed under interacting feature compositions. Operators can probe hardware to answer individual questions, but these efforts do not scale and rarely yield a reusable semantic contract. This paper proposes Lumen, a neuro-symbolic approach to automatically construct a high-fidelity behavior model of a blackbox switch. Our key insight is to cast switch modeling as a behavior-space learning problem with counterexample-guided alignment: first *grow* a model hypothesis to cover an increasingly large portion of switch behavior through incremental synthesis and model-centric alignment, then *explore* the long tail with hardware-centric alignment and constrained global repair. The resulting behavior model provides an executable specification that enables configuration preflight validation, semantic diffing across devices and versions, and precise failure diagnosis. We instantiate Lumen in an initial prototype and report preliminary validation results that demonstrate feasibility.

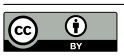
CCS Concepts

• **Networks** → **Network experimentation**; *Network manageability*; • **Software and its engineering** → *Software verification and validation.*

Keywords

black-box switch modeling, network experimentation, P4, program synthesis, network verification

ACM Reference Format:
Daqian Ding, Ziqian Liu, Zhixiong Niu, Ziyu Mao, Jingyu Wang, Yongqiang Xiong, and Yiming Qiu. 2026. Towards Learned Switch Behavior Modeling. In *The 10th Asia-Pacific Workshop on Networking (APNet '26), August 6–7, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3820441.3820465



APNet '26, Singapore, Singapore

ACM ISBN 979-8-4007-2664-4/26/08
https://doi.org/10.1145/3820441.3820465

1 Introduction

Network switches are the backbone of modern networking, underpinning communication in data centers [1, 12, 50, 64], edge deployments [45, 60], and the public Internet [34, 46, 59]. Yet in most production settings, the switches that carry this traffic are commodity, fixed-function ASICs whose packet-processing semantics are largely opaque to their users [8, 37, 38]. Vendors expose documentation and controller APIs, but these interfaces rarely constitute an executable specification of how packets are parsed, matched, modified, and forwarded—and they also fail to precisely describe how control-plane rules interact with the data plane packets under complex conditions. Programmable white-box switches such as Tofino exist [2, 20], but they are not the dominant installed base across heterogeneous operational networks.

This opacity creates a persistent semantic gap: operators are forced to infer switch behavior empirically, a process that is combinatorially complex, poorly interpretable, and inherently heterogeneous [25, 28, 66]. While the hardware can serve as a ground-truth oracle for individual experiments, it does not provide a reusable semantic contract: the outcome of one debugging session rarely translates into a check that can be applied to future configurations, different devices, or firmware versions [17, 24]. As a result, operational knowledge accumulates as fragile, informal rules of thumb and workarounds, and each change risks invalidating prior assumptions and reopening the same uncertainty. What's missing is a user-available, executable behavior specification of the deployed switch—one that enables validation of configurations, semantic diffing across devices or versions, and reproducible diagnoses.

A natural way to provide such an executable specification is to express switch semantics in programmable switch languages (e.g., P4 [6, 43], NPL [7]), which directly captures packet processing behavior while remaining executable and compatible with an established ecosystem of compilation and validation toolchains [13, 47, 65]. Indeed, vendors and prior research have used programmable

switch specifications as executable references when validating fixed-function switch implementations during development [2, 26, 41, 42]. However, these models rarely reach the users: they are typically vendor-internal artifacts [2], and most legacy or commodity fixed-function switches ship without any public, faithful behavior model. More importantly, even if a model does exist, producing and maintaining it is still a manual, per-device effort that does not scale to heterogeneous fleets and continuously evolving ASIC pipelines. Thus, the bottleneck is not testing a model; it is acquiring and maintaining the model.

In this paper, we argue that programmable switch specifications are the right vehicle for fixed-function switch modeling, but they should be coupled with a user-centric, automated workflow that can distill and reconcile messy, incomplete sources of truth into a faithful model without months of manual reverse engineering. Our goal is to automatically construct a high-fidelity *semantic twin* of a fixed-function switch, such that given the same control-plane state and ingress packets, the switch twin produces the same externally observable outcomes as the hardware: whether the packet is forwarded or dropped, which egress port is chosen, and how headers are rewritten. Rather than claiming universal equivalence, we seek fidelity across broad configurations and traffic through an alignment loop that progressively expands covered behaviors and shrinks residual disagreement with the hardware.

At first glance, this may appear to be a daunting active learning task [11, 14, 33, 53, 54, 61]. Fortunately, unlike general-purpose software, a network switch features a highly specialized packet processing pipeline: its data plane behavior is largely a restricted, ordered composition of header parsing and table match-actions, parameterized by control-plane rule state. This restriction creates two key opportunities. On the one hand, the pipeline admits a natural *decomposition with clear boundaries*: each processing stage operates on a well-defined set of header fields and metadata, and the controller API explicitly exposes the schema of matches, priorities, and actions. On the other hand, since the semantics are bounded and discrete, switch behavior modeling is unusually amenable to *automated reasoning*: candidate models can be systematically exercised and refuted by compact counterexamples (i.e., specific packets and rules) that pinpoint semantic mismatches [44, 62, 63]. Together, these properties make it plausible to use a neuro-symbolic workflow: an AI agent can propose and patch behavior models, while symbolic reasoning and refinement keep the synthesis grounded in the hardware's observable behavior.

Building on this opportunity, we organize switch twinning as a learning roadmap that progressively narrows the gap between the behavior space of the hardware switch and that of the current model hypothesis. The first phase, *incremental synthesis*, focuses on expanding the hypothesis so that it covers an increasingly large portion of the switch's functionality. It grows the behavior model in a guided order, extending parsers and stages along the natural boundaries of packet processing. This keeps each intermediate hypothesis executable and testable, and prevents mistakes from cascading across features. Once the model has grown to cover the major pipeline, the remaining discrepancies tend to concentrate in long-tail interactions that are hard to infer. The second phase, *global repair*, is designed to close this residual gap. It performs refinement on the existing model hypothesis with a small set of bounded edits, so that each fix has a limited blast radius and can be validated in isolation. In effect, incremental synthesis builds a broad, faithful sketch of switch behavior, while global repair uses targeted refinements to close the long-tail gaps that matter to users.

To make the above roadmap work, the central challenge is to *align* the model with the hardware in a way that actively guides both phases of learning. Switch behavior is highly combinatorial: subtle mismatches may only surface under specific combinations of packet headers, pipeline conditions, and control-plane rules. We therefore build a phased alignment engine that systematically searches the switch behavior space and turns discrepancies into actionable guidance for synthesis and repair. During incremental synthesis, we use *model-centric* alignment to steer growth: for the current model hypothesis, we generate tests that systematically exercise its reachable execution paths and check them against the hardware early, so that large semantic gaps are corrected before they propagate into later stages. For global repair, we shift to *hardware-centric* alignment that treats the switch as the oracle, replaying representative packets and rules drawn from real traces to expose long-tail feature interactions that initial AI-based exploration may not encounter. When this global alignment detects a mismatch, we invoke targeted differential localization, where counterexamples are systematically mutated under semantic constraints to isolate the minimal trigger and pinpoint the responsible pipeline condition. This localized diagnosis is then fed back as a precise repair request, closing a tight loop from verification to localization to refinement.

The rest of this paper outlines our technical roadmap, and presents initial results as validation.

2 Motivation

2.1 Understanding switch behavior is hard

Operating switches is a crucial aspect of modern network management, yet understanding switch behavior remains challenging. A myriad of factors come into play.

Opaque sources of truth. For commodity fixed-function switches [8, 38], the only sources of truth available to operators are text documentation plus a controller API schema that describes which features exist and how to configure their knobs. What is often missing is the semantics that actually determine behavior: the precise packet-processing order, match precedence, table-miss handling, implicit gating conditions, and header rewrite details that decide whether a packet is dropped, forwarded, or modified [6, 36, 51]. This opacity is harmful because it prevents operators from accurately predicting the behavior of the switch, forcing them to rely on blind empirical probing with no reliable specification to validate against.

Combinatorial behavior. Observed switch behavior is determined by how multiple features compose along an ordered pipeline under a particular combination of data-plane packets and control-plane rules [17, 27, 52]. This yields a large combinatorial space in which discrepancies may appear only under rare combinations. Since manual preflight validation covers only a small subset of runtime behaviors, most improper combinations on the long tail often remain latent until they trigger failures in realistic deployments. Such mismatches often arise when an implicit dependency changes the effective precedence of processing stages or rule matches.

Weak interpretability. When failures occur, a hardware switch behaves like an oracle that returns an output packet or a drop decision, but exposes little about the internal decisions that produced it [24, 25, 47]. Operators typically cannot observe which pipeline stage took effect or which rule matched, making it difficult to localize the root cause beyond coarse hypotheses. As a result, reproducing a failure becomes a manual, iterative process, and the postmortem often ends as a one-off workaround rather than a reusable check that can be enforced before the next rollout.

Heterogeneous fleets. Operational networks rarely consist of a single switch model; they span vendors, ASIC generations, and firmware versions whose semantics can differ and evolve over time [39]. As a result, lessons learned from one device or release do not reliably generalize, forcing operators to revalidate assumptions. Worse, heterogeneity introduces interaction failures: packets processed by one switch become the inputs to the next, so small differences in processing can compound and manifest as end-to-end surprises that are difficult to attribute to any single device.

2.2 State-of-the-art and opportunities

The last decade has seen the *maturation of network behavior modeling* in both scope and rigor. A large body of work on control-plane verification has made it increasingly practical to reason about network-wide properties by analyzing configurations and intent [3, 10, 15, 16, 18, 19, 22, 23, 30]. As these techniques have become mainstream, major operators have reported a corresponding shift in the dominant causes of failures: from simple misconfigurations to intrinsic hardware and software bugs in the switch stack [2, 31, 57]. This shift has motivated efforts to bring behavior modeling closer to the data plane by using programmable switch languages as an executable semantic specification.

For example, SwitchV [2] leverages P4 specifications to simulate forwarding behavior and expose switch bugs, illustrating the promise of P4-based modeling for validating fixed-function switches. At the same time, existing P4 testing, synthesis, and debugging toolchains [13, 27, 36, 47, 51, 52, 65] provide increasingly strong *verification* and *validation* capabilities once an executable model is available, but they either target programmable data planes or assume that faithful models already exist [41, 42, 49], leaving model synthesis for fixed-function switches a missing topic.

Opportunity 1. Network modeling and validation have largely matured. If we can synthesize models for fixed-function switches, we can close the synthesis-verification loop and make switch behavior modeling largely automated.

Learned models and AI-driven emulation have become increasingly common across networking and systems, with applications spanning mobile networks [4, 55, 56], congestion control [21, 58], and cloud infrastructure [5, 29, 32, 35]. Recent work on learned system emulators [5] further suggests a promising direction: use AI-assisted code synthesis to translate coarse sources of truth into executable emulation logic, and then refine the emulator under validation feedback. Switches are a particularly compelling target for this trend because their packet-processing behavior is structured and bounded, and its semantics can be expressed in an executable form with clear inputs and outputs. This structure creates an unusually strong foundation for coupling AI-driven synthesis with symbolic alignment, turning learned models from approximate predictors into high-fidelity behavioral twins.

Opportunity 2. Learned emulation has reached a turning point in which AI-based synthesis can be coupled with symbolic reasoning; switches are ideal for this vision due to grounded packet semantics.

2.3 Learned switch behavior modeling

Our project, Lumen, aims to realize a neuro-symbolic workflow that learns the behavior model of a fixed-function switch. It takes as input switch documentation, controller API schema, and observed traces, and outputs an executable model that captures the switch's data and control plane behavior.

Figure 1 summarizes Lumen's workflow and its behavior-space progress. Lumen starts with *knowledge extraction* to distill a coarse, machine-usable view of switch semantics. Guided by this extracted knowledge, Lumen incrementally constructs a partial model hypothesis via *incremental synthesis*. Each synthesized increment is immediately subjected to *model-centric alignment*. This loop is designed to *grow the behavior space*: each successful increment adds new reachable behaviors that can be exercised and validated. When mismatches arise, model-centric alignment produces concrete counterexamples that incremental synthesis can use to update or expand the hypothesis, steadily growing the portion of switch behavior that the model can explain.

Once incremental synthesis produces a global hypothesis, Lumen shifts to *complete the behavior space*. It enters the other loop driven by *hardware-centric alignment* on historical traces, using the switch as an oracle to surface behaviors that were missed by incremental exploration. When a discrepancy is observed, Lumen invokes differential localization to isolate the minimal conditions that characterize the missing behavior. Lumen then applies constrained *global repair* to incorporate this uncovered behavior into the hypothesis through bounded edits. Conceptually, this loop discovers long-tail behaviors under realistic workloads and folds them into the model in a controlled manner, so that the residual disagreement between the hypothesis and the hardware steadily shrinks over iterations.

3 Solution Sketch

3.1 Knowledge extraction

Lumen extracts a *structured constraint scaffold* for downstream synthesis and alignment tasks. As shown in Figure 2, Lumen derives this information from three inputs.

Vendor documentation. Most modern pipeline specifications (e.g., SAI [39] or OF-DPA [9]) provide macro-level structure such as logical processing order of their features, for example that L2 resolution precedes L3 routing. Since much of this information appears as text, Lumen treats documentation as best-effort priors and uses LLM-aided parsing to extract usable structure into a normalized form, including dependency constraints, header formats and field references. These priors are non-authoritative and are later validated and refined through subsequent alignment.

Controller API schema. Unlike documentation, the API schema provides a structured and authoritative view of the switch behavior. Lumen converts the schema into a uniform representation that includes the set of configurable features, along with their match

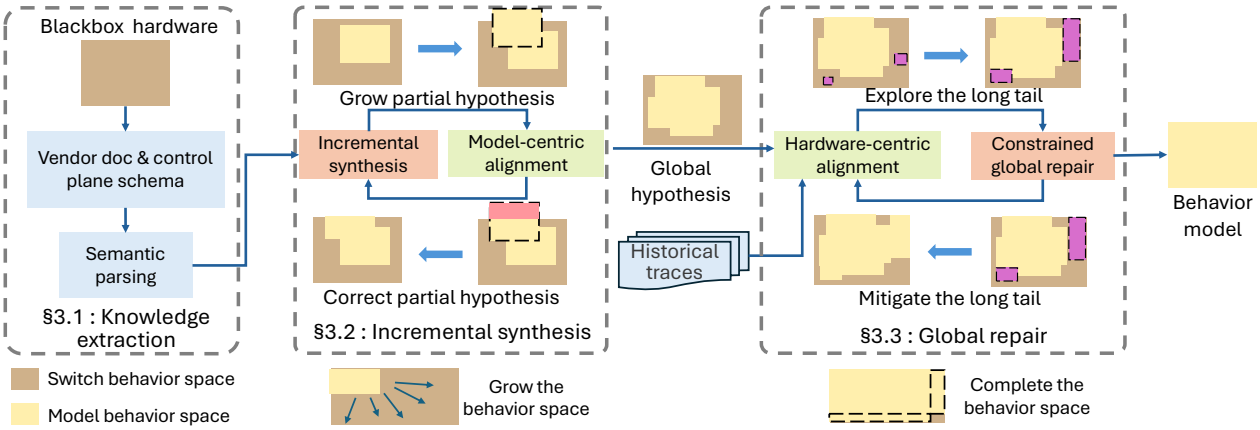


Figure 1: The envisioned technical roadmap of Lumen.

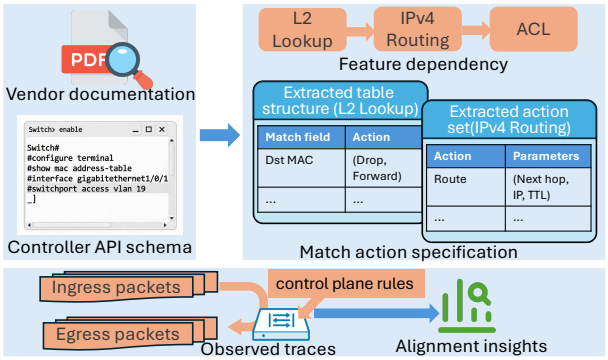


Figure 2: Sources of truth for fixed-function switches

keys, actions, and priorities. To handle diverse switch stacks, Lumen uses a *learned symbolic parser* that symbolically parses schemas, while learning the mappings to canonicalize vendor conventions.

Observed traces. Lumen leverages *historical traces* obtained from the user environment as a key realism anchor for alignment. Critically, these traces include both packets and the corresponding controller rules, making them high-value seeds for hardware-centric alignment. In addition to historical traces, Lumen also generates *targeted traces* by executing the model hypothesis and exploring its execution paths during model-centric alignment. Both traces are executed on both the hardware and the model to extract outcomes.

3.2 Incremental synthesis

A central observation behind Lumen is that switch modeling is not an unconstrained code-generation problem. This is because switch components are constrained by a natural *partial order*: header dependencies specify when a component can become active, while the API reveals what header fields a feature matches and acts on, and whether those actions alter context in ways that constrain subsequent stages. Lumen uses such ordering information to replace one-shot synthesis with an *incremental synthesis* roadmap.

Task decomposition. Lumen converts the resulting partial order into an *assembly roadmap* by distinguishing between components with *fixed dependency order* and those with *bounded placement flexibility*. The former are pinned by hard producer-consumer relations. For example, a component that matches on IPv4 destination fields must follow the parser states that expose those fields, and a component that consumes a derived forwarding identifier must follow the stage that produces it. The latter are not globally fixed, but can only be placed within a constrained region determined by what they read and what their actions may invalidate downstream. For example, a policy table may be inserted only after the metadata it matches on has been established, yet before stages whose behavior its actions could suppress or redirect. By making these bounds explicit, Lumen derives a structured decomposition in which each synthesis block has a well-defined interface to the rest of the model and can be integrated and aligned incrementally.

Partial hypothesis synthesis. The central challenge in AI-based synthesis is choosing the right *unit of growth*. In Lumen, that unit is the smallest feature slice that introduces a new externally observable behavior. This choice is crucial because system-by-system growth (e.g., synthesizing the entire parser state machine first, then moving onto the processing stages) does not work well in our setting. A parser fragment without the corresponding processing logic often has no visible effect at the model interface, while a processing fragment without the parser support needed to expose its inputs cannot be exercised meaningfully. Lumen therefore grows the hypothesis feature by feature, adding the parser states, match-action logic, and output handling needed for each new behavior to become immediately visible and testable.

Each new slice is then grafted onto the current hypothesis as a self-contained increment. The assembly roadmap restricts growth to slices whose dependencies are already satisfied, so integration never invalidates the structural assumptions of the existing model. For example, adding an L3 forwarding feature requires not only the lookup and action logic, but also the parser support needed to expose the relevant header fields and the output-side handling needed to make the forwarding decision observable. Because each increment is complete enough to produce a visible effect, the alignment phase can attribute any new discrepancy to the newly added slice, keeping failures localized and allowing the model's behavior space to grow in controlled, checkable steps.

Model-centric alignment. Incremental growth is only useful if each newly added feature slice is aligned before further synthesis proceeds. Lumen therefore places *model-centric alignment* directly inside the growth loop, using it to validate the newly reachable behaviors of the current hypothesis. After a slice is integrated, Lumen symbolically executes the updated model to explore the execution paths introduced or affected by that slice, and leverages SMT solving to generate targeted test cases that exercise those paths under concrete packets and rules. These tests are then replayed on the hardware, and any discrepancy between the model's and the switch's behavior is recorded as a counterexample. Because the hypothesis has grown by only one self-contained slice, such counterexamples are naturally localized: they expose either missing logic in the new slice or an incorrect interaction between that slice and the existing hypothesis.

These counterexamples are then fed back into synthesis as guidance, allowing Lumen to refine the new slice before admitting it into the hypothesis. In this way, model-centric alignment acts as the mechanism that keeps growth bounded and interpretable. Rather than reasoning about the full combinatorial space of the switch at once, Lumen repeatedly expands the hypothesis by one observable increment, aligns the newly covered paths, and preserves the already aligned behavior of the rest of the model. The outcome of this phase is a structurally *sound* model hypothesis that captures the core features and behaviors of the hardware switch.

3.3 Global repair

The incremental synthesis is inherently limited by the model hypothesis: it can only exercise behaviors that the model already exposes. As a result, residual gaps often remain in the long tail, including feature interactions, corner conditions, and hidden behaviors that were never triggered during incremental growth. The goal of global repair is to incorporate these residual behaviors without destabilizing the already-aligned structure.

Hardware-centric alignment. To expose the remaining gap, Lumen shifts to *hardware-centric alignment*, where it leverages operators' historical traces (including both packets and rules) actually processed by the hardware switch to query the model hypothesis under realistic packet-rule combinations. Any discrepancy between the model and the switch signals behavior that the current hypothesis does not yet capture correctly. Unlike the incremental phase, however, the scope of such a discrepancy is initially *global*: once the model is exercised, a mismatch could in principle originate from many interacting components. Lumen therefore performs *differential localization* to recover a bounded repair scope. Starting from the failing counterexamples, Lumen systematically mutates packets and rules under protocol and schema constraints, and compares how those perturbations affect the model and the hardware. The result is a minimal triggering condition, together with a localized diagnosis of which parser states, tables, match conditions, actions, or inter-stage dependencies are implicated by the mismatch.

Constrained repair. Once localization has reduced a global discrepancy to a bounded diagnosis, Lumen moves onto *constrained repair*. Rather than allowing unrestricted rewrites, repair is expressed through an explicit *edit surface*: a fixed set of operations, including adding, deleting, or updating parser states and transitions, tables, match keys, actions, and control-flow links between stages. A learned repair module proposes edits only through this interface, and a compiler materializes the resulting patch into the hypothesis. This design serves two purposes. First, it keeps the repair process interpretable and bounded: every patch has a small, explicit scope. Second, it limits blast radius by ensuring that global repair refines the existing model rather than replacing large portions of it. After each patch, Lumen re-runs both model-centric and hardware-centric alignment to confirm that the targeted discrepancy is resolved and that aligned behavior remains intact.

Through this global repair loop, Lumen gradually absorbs long-tail behaviors into the behavior model. While Lumen does not claim universal semantic completeness, it aims to cover the behaviors exercised by the user's observed historical traces, yielding strong *operational coverage* over the packet-rule combinations that matter.

4 Preliminary Validation

In this section, we evaluate the Lumen prototype and report preliminary evidence on the feasibility of its behavior modeling workflow. To make the evaluation meaningful, we first need a ground truth that is both blackbox to the model learning process, and quantitatively comparable to the learned model. Direct evaluation on a commercial fixed-function switch is insufficient for this purpose: while such hardware is the ultimate deployment target, its hidden implementation provides no way to measure model coverage, attribute discrepancies, or compare intermediate synthesis progress against a complete semantic reference. For evaluation, a controlled P4-based testbed is therefore the only rational ground truth we can target, because it allows the learned model and the target pipeline to be compared under the same executable semantics and metrics. It also lets us vary pipeline complexity in a controlled manner, which reveals how Lumen behaves as functionality scales.

Importantly, the reliance on white-box target does not reduce the challenge of the modeling task: during the learning process, we ask AI agents to mask away the target details and expose only the same sources of truth assumed by Lumen, namely documentation, controller API schema, and observed traces. For this preliminary validation, we establish a controlled testbed using a simplified fabric.p4 [40] program running on the BMv2 software switch. We use P4Testgen [48] to generate observed traces for both model-centric and hardware-centric alignment. The target pipeline includes several common switch features derived from fabric.p4, ranging from fundamental L2/L3 lookups to more complex MPLS label processing and ACL-based traffic policing. This setup gives us a quantitatively comparable target, controllable functional complexity, and a clean way to evaluate whether Lumen can synthesize and refine a behavior model under realistic blackbox constraints.

Overall effectiveness. We first evaluate the two design choices at the core of Lumen: incremental synthesis and alignment. Figure 3 shows that their roles are complementary and that both are required for robust scaling. The full Lumen pipeline sustains high success rate across all target sizes, indicating that bounded growth together with symbolic alignment can keep model quality stable even as the switch behavior becomes more complex. In contrast, removing alignment causes errors to appear early and accumulate as the

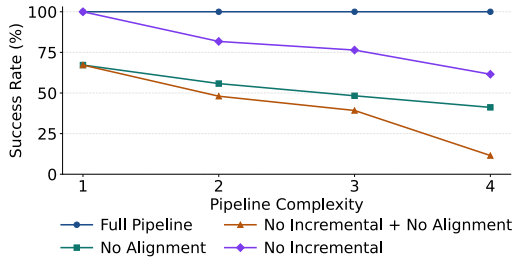


Figure 3: Overall effectiveness. Success rate across different pipeline complexities (from 1 to 4 represent L2, L2+L3, L2+L3+ACL, and L2+L3+ACL+MPLS).

pipeline complexity grows, while removing incremental synthesis preserves strong performance only on the simple targets before degrading as the search scope becomes larger. Eliminating both mechanisms leads to the most severe collapse. Taken together, these results suggest a clear division of labor: alignment is essential for eliminating local errors, while incremental synthesis is essential for keeping the search space manageable as complexity scales. Only their combination yields consistently high effectiveness.

End-to-end workflow. Figure 4 illustrates the pass rate and test case volume over a complete modeling lifecycle. The end-to-end workflow starts with incremental synthesis (round 1-16), then moves onto global repair (round 17-20).

Incremental Synthesis. The system progressively grows and tests feature combinations, starting from foundational L2 forwarding and extending to more complex capabilities such as MPLS processing. The colored bars in Figure 4 indicate the feature increments added during model-centric alignment. Because Lumen grows the model in semantically isolated slices, early errors are corrected quickly and remain localized rather than propagating into later stages. As a result, each increment converges rapidly under model-centric tests, showing that the growth-and-alignment loop can build a stable model hypothesis feature by feature.

Global Repair. Once the global model hypothesis is built, the workflow shifts to hardware-centric testing to expose long-tail behaviors missed by model-centric exploration. Starting from round 17, we synthesize simulated historical traces by executing P4Testgen against the ground truth pipeline. These traces trigger long tail behaviors and cause a sharp drop in pass rate. This drop is expected and, in fact, desirable: it demonstrates that model-centric alignment alone is insufficient, and that hardware-centric testing is necessary to uncover residual semantic gaps. After localization, the pass rate recovers through global repair, indicating that the long-tail discrepancies can be turned into actionable repair signals. In the current prototype, this repair step is still implemented in a lightweight heuristic manner guided by counterexample-guided diagnoses rather than by a full differential localization engine. Even so, the recovery trend suggests that the overall workflow is viable, while also highlighting constrained global repair as the most important remaining step toward a truly scalable system.

5 Discussion

Goal and non-goal. Lumen is designed to construct an executable *behavior model* of fixed-function switch packet processing under

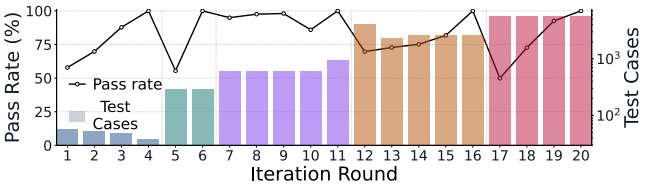


Figure 4: End-to-end workflow. Bar colors denote the added features (blue: L2, green: L3, purple: ACL, orange: MPLS) during incremental synthesis. The final red bars represent the global repair phase.

control-plane rule state. Its goal is to reproduce externally observable functional outcomes, including whether a packet is forwarded or dropped, which egress port is selected, and how packet headers are transformed. It is *not* a performance model: Lumen does not currently attempt to model queueing, buffering, scheduling, contention, timing, or throughput-related effects. These behaviors are important in their own right, but they require a different abstraction boundary and different evaluation signals than the functional packet-processing semantics targeted in this work.

Fidelity. Lumen does not claim universal semantic completeness over the full behavior space of the hardware. Instead, it targets strong *operational fidelity*: the learned model is expected to align with the behaviors exercised by observed historical traces and by the additional queries generated during alignment. In this sense, the current model should be viewed as a progressively refined subset of hardware behavior, while the hardware may still remain a proper superset if some rare or undocumented behaviors are never exposed by the history or exploration. Lumen could potentially adopt an iterative refinement loop, where fidelity improves as new discrepancies are discovered and absorbed into the model.

Stateful and probabilistic behaviors. The current design focuses on the large class of switch behaviors that can be explained on a per-packet basis under a given control-plane state, which already covers many common fixed-function network functions. We do not yet model stateful packet-processing behaviors such as dynamic MAC learning, whose outcomes depend on temporal evolution. We also do not yet model probabilistic primitives such as hash functions used for features like ECMP. Extending Lumen to such behaviors is a natural next step, and it will likely require richer synthesis and alignment mechanisms that reason not only about isolated queries, but also about hidden computation and, in some cases, packet sequences and state transitions across multiple packets.

6 Conclusion

This paper argues that fixed-function switches need not remain semantic black boxes to their users. Lumen shows a path forward by casting switch twinning as a behavior-space learning problem: it first grows an executable behavior model through incremental, observable synthesis, then explores the long tail through hardware-centric alignment and bounded repair. Our preliminary results suggest that this roadmap is both feasible and promising, and point to a broader opportunity to turn opaque switches into user-available executable specifications for validation, comparison, and diagnosis.

References

[1] Dennis Abts, Michael R Marty, Philip M Wells, Peter Klausler, and Hong Liu. 2010. Energy proportional datacenter networks. In *Proceedings of the 37th annual international symposium on Computer architecture*. 338–347.

[2] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. 2022. SwitchV: automated SDN switch validation with P4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 365–379.

[3] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software* 86, 8 (2013), 1978–2001.

[4] Anurag Bambardekar, Prasanthi Maddala, Sreeram Mandava, Narayan B Mandayam, and Ivan Seskar. 2024. SDR-Based Emulation of Machine Learning-Enabled Spectrum Sharing in 5G Private Networks. In *2024 IEEE Future Networks World Forum (FNWF)*. IEEE, 572–577.

[5] Archit Bhatnagar, Yiming Qiu, Sarah McClure, Sylvia Ratnasamy, and Ang Chen. 2025. A Case for Learned Cloud Emulators. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*. 69–76.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[7] Broadcom. 2025. NPL: Network Programming Language. https://nplang.org/ Accessed: 2026.

[8] Broadcom and community contributors. 2019. Whitebox Switches: Programmable VS Fixed ASICs. https://bm-switch.com/2019/06/24/whitebox_basics_programmable_fixed_asics Accessed: 2026.

[9] Broadcom Corporation. [n. d.]. OpenFlow Data Plane Abstraction (OF-DPA) API Guide and Reference Manual: OF-DPA Software Overview. https://broadcom-switch.github.io/of-dpa/doc/html/OFDPA_OVERVIEW.html. Accessed: 2026-03.

[10] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A {NICE} way to test {OpenFlow} applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 127–140.

[11] Haoxian Chen, Chenyuan Wu, Andrew Zhao, Mukund Raghothaman, Mayur Naik, and Boon Thau Loo. 2023. Synthesizing Formal Network Specifications From Input-Output Examples. *IEEE/ACM Transactions on Networking* 31, 3 (2023), 994–1009. doi:10.1109/TNET.2022.3208551

[12] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. 2013. OSA: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Transactions on networking* 22, 2 (2013), 498–511.

[13] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2020. bf4: towards bug-free P4 programs. 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*. 571–585.

[14] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 762–774.

[15] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*. 1–7.

[16] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: probabilistic inference for networks. *ACM SIGPLAN Notices* 53, 4 (2018), 586–602.

[17] Shixian Guo, Kefei Liu, Yulin Lai, Yangyang Bai, Ziwei Zhao, Songlin Liu, Jianghang Ning, Gen Li, Jianwei Hu, Yongbin Dong, et al. 2025. ByteTracker: An Agentless and Real-time Path-aware Network Probing System. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 541–553.

[18] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 735–749.

[19] Alex Horn, Ali Kheradmand, and Mukul R Prasad. 2019. A precise and expressive lattice-theoretical framework for efficient network verification. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 1–12.

[20] Intel Corporation. 2020. *P4_16 Intel® Tofino™ Native Architecture – Public Version*. Technical Report. Intel (formerly Barefoot Networks). https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf Accessed: 2026.

[21] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A deep reinforcement learning perspective on internet congestion control. In *International conference on machine learning*. PMLR, 3050–3059.

[22] Ali Kheradmand. 2020. Automatic inference of high-level network intents by mining forwarding patterns. In *Proceedings of the Symposium on SDN Research*. 27–33.

[23] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2012. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*. 49–54.

[24] Ralf Kundel, Fridolin Siegmund, Rhaban Hark, Amr Rizk, and Boris Koldehofe. 2022. Network testing utilizing programmable network hardware. *IEEE Communications Magazine* 60, 2 (2022), 12–17.

[25] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. 2012. A soft way for openflow switch interoperability testing. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 265–276.

[26] Seyyidahmed Lahmer, Nikita Tyunyayev, and Tom Barbette. 2025. OpenDesc: From Static NIC Descriptors to Evolvable Metadata Interfaces. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*. 271–279.

[27] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*. 490–503.

[28] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, et al. 2024. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 554–567.

[29] LocalStack Team. 2025. LocalStack: A fully functional local AWS cloud stack. https://www.localstack.cloud/localstack-for-aws Accessed: 2025-06-29.

[30] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 290–301.

[31] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*. 393–407.

[32] Microsoft Corporation. 2025. Use the Azurite emulator for local Azure Storage development. https://learn.microsoft.com/en-us/azure/storage/common/storage-use-azurite?tabs=visual-studio%2Cblob-storage Accessed: 2025-07-02.

[33] Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. 2025. Active Learning of Symbolic NetKAT Automata. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1119–1142.

[34] Pablo Molinero-Fernández and Nick McKeown. 2003. Performance of circuit switching in the Internet. *Journal of Optical Networking* 2, 4 (2003), 83–96.

[35] Moto Team. 2025. Moto: Mock AWS Services. https://docs.getmoto.org/en/latest/ Accessed: 2025-07-02.

[36] Mohammad A Noureddine, Amanda Hsu, Matthew Caesar, Fadi A Zaraket, and William H Sanders. 2019. P4aig: Circuit-level verification of p4 programs. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Supplemental Volume (DSN-S)*. IEEE, 21–22.

[37] NVIDIA. 2021. Proving Superior Cloud, AI, and Storage Performance with NVIDIA Spectrum-3 Switches. https://developer.nvidia.com/blog/proving-superior-cloud-ai-and-storage-performance-with-spectrum-3-switches Accessed: 2026.

[38] Edgecore Networks on behalf of Broadcom. 2025. Broadcom Tomahawk 6: Powering the Latest Generation of AI and Hyperscale Networks. https://www.edge-core.com/blogs/broadcom-tomahawk-6-powering-the-latest-generation-of-ai-and-hyperscale-networks Accessed: 2026.

[39] Open Compute Project. [n. d.]. *Switch Abstraction Interface (SAI)*. https://github.com/opencomputeproject/SAI Accessed: 2026-03.

[40] Open Networking Foundation. [n. d.]. fabric.p4: ONOS SD-Fabric Data Plane Program. https://github.com/opennetworkinglab/onos/blob/master/pipelines/fabric/impl/src/main/resources/fabric.p4. Accessed: 2026-03.

[41] P4-BAR team or anonymous. 2024. P4-Based Automated Reasoning (P4-BAR) for the (Networking) Masses! Presentation at the P4 Workshop 2024. https://p4.org/wp-content/uploads/sites/53/2024/10/17-2024-P4-Workshop-P4-BasedAutomated-Reasoning-for-the-Networking-Masses.pdf Slides from P4

Workshop 2024; part of the P4-BAR session.

[42] P4-BAR team or anonymous. 2024. Scaling P4-Based Automated Reasoning (Performance and Coverage). Presentation at the P4 Workshop 2024. https://p4.org/wp-content/uploads/sites/53/2024/10/18-2024-P4-Workshop-Scaling-P4Based-Automated-Reasoning-Performance-and-Coverage.pdf Part of P4-Based Automated Reasoning track; slides from P4 Workshop 2024.

[43] P4.org Consortium. 2024. *The P4 Language Specification.* Technical Report. P4.org. https://p4.org/p4-spec/ Accessed: 2026.

[44] Peter Perešíni, Maciej Kuźniar, and Dejan Kostić. 2015. Monocle: Dynamic, fine-grained data plane monitoring. In *Proceedings of the 11th ACM conference on emerging networking experiments and technologies*. 1–13.

[45] Justin Pettit, Jesse Gross, Ben Pfaff, Martin Casado, and Simon Crosby. 2010. Virtual switching in an era of advanced edges. (2010).

[46] Chunming Qiao and Myungsik Yoo. 1999. Optical burst switching (OBS)–a new paradigm for an optical Internet. *Journal of high speed networks* 8, 1 (1999), 69–84.

[47] Yiming Qiu, Ryan Beckett, and Ang Chen. 2023. Synthesizing runtime programmable switch updates. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 613–628.

[48] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. Association for Computing Machinery, 136–151. doi:10.1145/3603269.3604834

[49] Puneet Sharma et al. 2024. Modeling Hardware Blocks of Network ASICs using P4. Presentation at the P4 Workshop 2024. https://p4.org/wp-content/uploads/sites/53/2024/10/13-2024-P4-Workshop-Modelinghardware-blocks-of-network-ASICs-using-P4.pdf Slides available via P4.org (link may require archive.org or direct request if 404 persists).

[50] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the data center network. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[51] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 518–532.

[52] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. 2021. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 17–32.

[53] Arthur Tran Van, Olivier Levillain, and Herve Debar. 2024. Mealy verifier: An automated, exhaustive, and explainable methodology for analyzing state machines in protocol implementations. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*. 1–10.

[54] Frits Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95.

[55] Davide Villa, Miead Tehrani-Moayyed, Clifton Paul Robinson, Leonardo Bonati, Pedram Johari, Michele Polese, and Tommaso Melodia. 2024. Colosseum as a digital twin: Bridging real-world experimentation and wireless network emulation. *IEEE Transactions on Mobile Computing* 23, 10 (2024), 9150–9166.

[56] Tianyu Wang, Shaowei Wang, and Zhi-Hua Zhou. 2019. Machine learning for 5G and beyond: From model-based to data-driven mobile wireless networks. *China Communications* 16, 1 (2019), 165–175.

[57] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: Automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 419–430.

[58] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 731–743.

[59] SJ Ben Yoo. 2010. Energy efficiency in the future internet: The role of optical packet switching and optical-label switching. *IEEE Journal of Selected Topics in Quantum Electronics* 17, 2 (2010), 406–418.

[60] SJ Ben Yoo, Fei Xue, Yash Bansal, Julie Taylor, Zhong Pan, Jing Cao, Minyong Jeon, Tony Nady, Gary Goncher, Kirk Boyer, et al. 2003. High-performance optical-label switching packet routers and smart edge routers for the next-generation Internet. *IEEE Journal on Selected Areas in Communications* 21, 7 (2003), 1041–1051.

[61] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. 2014. NetEgg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. 1–7.

[62] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 241–252.

[63] Peng Zhang, Cheng Zhang, and Chengchen Hu. 2018. Fast data plane testing for software-defined networks with rulechecker. *IEEE/ACM transactions on networking* 27, 1 (2018), 173–186.

[64] Shenglin Zhang, Ying Liu, Weibin Meng, Zhiling Luo, Jiahao Bu, Sen Yang, Peixian Liang, Dan Pei, Jun Xu, Yuzhi Zhang, et al. 2018. Prefix: Switch failure prediction in datacenter networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (2018), 1–29.

[65] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. 2022. Meissa: scalable network testing for programmable data planes. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 350–364.

[66] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. 2019. Hypertester: high-performance network testing driven by programmable switches. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 30–43.