Probabilistic Network Verification

Steffen Smolka
Cornell University
Overview:
Network Verification
(In particular, data plane verification)
Network verification has taken off!

Several start-ups

Deployed at big cloud providers

Lots of research
Example: Network Verification
This paper applies this theory to a new domain: networks. Moreover, equivalence in successfully in a number of application areas, including compiler, de-
Example: Network Verification

"Are packets routed between hosts?"
"Are ssh packets dropped?"

Verification Tool

Inputs: Network config & topology + question

Outputs: "Yes" / "No" + counterexample
This paper applies this theory to a new domain: networks.

Addressing the challenges of programming and verification of network equipment has been a long-standing goal for researchers, aiming to ensure the performance, security, and correctness of networked systems. 

The approach taken in this paper leverages a foundational model for network programming based on Kleene Algebra with Tests (KAT). KAT provides a powerful framework for reasoning about network behavior, with a well-developed metatheory that includes a sound and complete equational theory.

KAT's expressive power is key, allowing it to capture both the global behavior of the network and the local properties of individual switches. This dual perspective is essential for addressing modern challenges in network programming, which include:

- **Static Policy Languages**: Essential for configuring network devices.
- **Flow-Based Policing**: Necessary to enforce traffic policies.
- **Dynamic Policy Composition**: Required for creating complex policies from simpler ones.
- **Policy Management**: Needs efficient mechanisms for configuring and monitoring policies.
- **Policy Evolution**: Requires frameworks to support the extension and evolution of policies.

KAT and its dynamic semantics inspired by NetCore [23] provide a solid foundation for network programming and verification. These semantics can be extended in key ways to make it sound for practical applications.

**Key Contributions**

- **Foundational Model**: A KAT-based model for reasoning about formal properties.
- **Syntax and Semantics**: Formal definitions of the language, its syntax, and its denotational semantics.
- **Completeness Proof**: A proof showing the completeness of the system with respect to the equational axioms.
- **Soundness**: A proof showing the soundness of the system.
- **Practical Applications**: Demonstrations of the system's capabilities on a diverse collection of applications.

**Example: Network Verification**

Consider a simple network consisting of two hosts, Host 1 and Host 2, connected by Switch A and Switch B. The question: "Can a packet sent from Host 1 be received by Host 2?"

The inputs to the verification tool are:

- Network configuration and topology
- A question or query about network behavior

The output includes:

- A "Yes"/"No" answer to the question
- A counterexample (if the answer is "No")

This framework enables a variety of interesting reachability queries about the network structure and behavior. It also allows us to reason about network structure and end-to-end functionality.

**What's the big deal?**

The significance of this approach lies in its ability to bridge the gap between theoretical foundations and practical applications. It provides a way to systematically reason about network behavior and to verify policies and configurations. This is crucial for ensuring the correctness of modern network systems, which are increasingly complex and interconnected.

In conclusion, the foundational model for network programming and verification based on KAT offers a powerful and flexible framework for addressing the challenges in networked systems. It sets the stage for the development of tools and techniques that can support the design, verification, and evolution of networked systems in the future.
The Power of Verification

Guesswork that network will behave correctly
The Power of Verification

Guesswork that network will behave correctly

Mathematical proof of policy compliance
The Power of Verification

Guesswork that network will behave correctly

Mathematical proof of policy compliance

Consequences:
The Power of Verification

Guesswork that network will behave correctly

Mathematical proof of policy compliance

Consequences:

✦ Bugs can be found before they ever manifest
The Power of Verification

Guesswork that network will behave correctly

Mathematical proof of policy compliance

Consequences:

- Bugs can be found before they ever manifest
- Can change network config with confidence
The Power of Verification

Guesswork that network will behave correctly

Mathematical proof of policy compliance

Consequences:

✦ Bugs can be found before they ever manifest
✦ Can change network config with confidence
✦ More robust & more efficient network
The Power of Verification

Guesswork that network will behave correctly

Mathematical proof of policy compliance

Consequences:
- Bugs can be found before they ever manifest
- Can change network config with confidence
- More robust & more efficient network
- Your network operators can sleep better...
Example Network Properties

State of the art tools verify reachability properties:
Example Network Properties

State of the art tools verify reachability properties:

Waypointing: every packet traverses a firewall
Example Network Properties

State of the art tools verify **reachability properties:**

**Waypointing:** *every packet traverses a firewall*

**Isolation:** *packets from VLAN 1 cannot enter VLAN 2*
Example Network Properties

State of the art tools verify reachability properties:

**Waypointing:** every packet traverses a firewall

**Isolation:** packets from VLAN 1 cannot enter VLAN2

**Connectivity:** all hosts in the network can communicate
Example Network Properties

State of the art tools verify reachability properties:

**Waypointing:** every packet traverses a firewall

**Isolation:** packets from VLAN 1 cannot enter VLAN2

**Connectivity:** all hosts in the network can communicate

**Loop Freedom:** there exist no forwarding loops
Example Network Properties

State of the art tools verify **reachability properties**:

**Waypointing**: every packet traverses a firewall

**Isolation**: packets from VLAN 1 cannot enter VLAN2

**Connectivity**: all hosts in the network can communicate

**Loop Freedom**: there exist no forwarding loops

**Access Control**: internet packets cannot enter the VLAN
Example Network Properties

State of the art tools verify reachability properties:

Waypointing: every packet traverses a firewall

Isolation: packets from VLAN 1 cannot enter VLAN2

Connectivity: all hosts in the network can communicate

Loop Freedom: there exist no forwarding loops

Access Control: internet packets cannot enter the VLAN

Key Assumption: network behavior is deterministic
Example Network Properties

State of the art tools verify reachability properties:

**Waypointing:** every packet traverses a firewall

**Isolation:** packets from VLAN 1 cannot enter VLAN 2

**Connectivity:** all hosts in the network can communicate

**Loop Freedom:** there exist no forwarding loops

**Access Control:** internet packets cannot enter the VLAN

**Key Assumption:** network behavior is deterministic
Probabilistic Network Behavior

Most of the time, networks behave deterministically.
Probabilistic Network Behavior

Most of the time, networks behave deterministically.

But what if...
Probabilistic Network Behavior

Most of the time, networks behave deterministically.

But what if...

✦ ... a link or switch fails?
Probabilistic Network Behavior

Most of the time, networks behave deterministically.

But what if...

✦ ... a link or switch fails?
✦ ... the network employs resilient routing?
Probabilistic Network Behavior

Most of the time, networks behave deterministically.

But what if...

✦ ... a link or switch fails?
✦ ... the network employs resilient routing?
    ✦ "what's the probability of packet delivery?"
Probabilistic Network Behavior

Most of the time, networks behave deterministically.

But what if...

✧ ... a link or switch fails?

✧ ... the network employs resilient routing?
  ✧ "what's the probability of packet delivery?"
  ✧ "what's the expected path length?"
Probabilistic Network Behavior

Most of the time, networks behave deterministically.

But what if...

✦ ... a link or switch fails?

✦ ... the network employs resilient routing?
  ✦ "what's the probability of packet delivery?"
  ✦ "what's the expected path length?"

✦ ... the network employs traffic engineering?
Probabilistic Network Behavior

Most of the time, networks behave deterministically.

But what if...

✦ ... a link or switch fails?
✦ ... the network employs resilient routing?
  ✦ "what's the probability of packet delivery?"
  ✦ "what's the expected path length?"
✦ ... the network employs traffic engineering?
  ✦ "what's the expected congestion of this link?"
Probabilistic Network Verification
Probabilistic NetKAT
Probabilistic NetKAT

ADS for programming, modeling & reasoning about probabilistic networks

Probabilistic NetKAT

Nate Foster1,2,*, Dexter Kozen3, Konstantinos Mancassou2, Mark Reiß4, and Alexandra Silva2

1 Cornell University, New York, USA
2 University of Pennsylvania, Philadelphia, USA
3 University College London, London, UK

Abstract. This paper presents a new language for network programming based on a probabilistic semantics. We extend the NetKAT language with new primitives for expressing probabilistic behaviors and studying the semantics from a first-order or deterministic function style based on measurable functions on sets of packet histories. We establish fundamental properties of the semantics, prove that it is a conservative extension of the deterministic semantics, show that it enables a number of natural extensions, and develop a notion of approximation. We present case studies that show how the language can be used to model a diverse collection of scenarios drawn from real-world networks.

1 Introduction

Formal specification and verification of networks has become a reality in recent years with the emergence of network-specific programming languages and property-checking tools. Programming languages like Fernet [11], Pyretic [23], HPL [21], and others are enabling programmers to specify the intended behavior of a network in terms of high-level constructs such as packet redirection and connection-oriented services. However, despite many notable advances, these frameworks allow a fundamental limitation: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network functionality is simple, or when the proportion of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory solutions to many more complex situations that arise in practice:

- Congestion: the network operator wishes to control the expected degree of congestion on each link given a model of the demands for traffic.
- Failure: the network operator wishes to calculate the probability that packets will be delivered to their destination, given that devices and links fail with a certain probability.

Probabilistic Program Equivalence for NetKAT

STEFANIE BÖNSCH, Cornell University, USA
NATE FOSTER, Cornell University, USA
ALEXANDRA SILVA, University College London, UK

We tackle the problem of deciding whether two probabilistic programs are equivalent in Probabilistic NetKAT (PNKAT), a formal language for modeling and reasoning about the behavior of packet-switched networks. As a result, PNKAT is a formalism for network programming, with a formal semantics, and a formal logic for proving program equivalence. We use a probabilistic semantics based on the Markov kernel formalism for probabilistic programs [17, 18], and a first-order semantics to express probabilistic behavior of packet networks. We present two main results:

1. [ESOP '16] Cantor Meets Scott: Semantic Foundations for Probabilistic Networks

Stefan Smolka
Cornell University, USA

Pawan Kumar
Cornell University, USA

Nate Foster
Cornell University, USA

Alexandra Silva
University College London, UK

Abstract

Probabilistic NetKAT (PNKAT) is a language for programming packet-switched networks, with a formal semantics based on Markov kernels. We present two main results:

- We prove the semantics is sound with respect to the denotational semantics of PNKAT programs, as defined in [17, 18], and the first-order semantics of PNKAT programs.
- We prove the semantics is complete with respect to the denotational semantics of PNKAT programs, as defined in [17, 18], and the first-order semantics of PNKAT programs.

This work is supported by the National Science Foundation (NSF) under grant CCF-1529218.

Keywords: Probabilistic programming, Probabilistic semantics, Kleene algebra with tests, Denotational semantics, PNKAT.

1. Introduction

The recent emergence of software-defined networking (SDN) has led to the development of new network programming languages such as Fernet [11], Pyretic [23], and others. These languages are designed to allow network operators to specify the intended behavior of a network in terms of high-level constructs such as packet redirection and connection-oriented services. However, despite many notable advances, these frameworks allow a fundamental limitation: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network functionality is simple, or when the proportion of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory solutions to many more complex situations that arise in practice:

- Congestion: the network operator wishes to control the expected degree of congestion on each link given a model of the demands for traffic.
- Failure: the network operator wishes to calculate the probability that packets will be delivered to their destination, given that devices and links fail with a certain probability.

[ESOP '16] Cantor Meets Scott: Semantic Foundations for Probabilistic Networks

Stefan Smolka
Cornell University, USA

Pawan Kumar
Cornell University, USA

Nate Foster
Cornell University, USA

Alexandra Silva
University College London, UK

Abstract

Probabilistic NetKAT (PNKAT) is a language for programming packet-switched networks, with a formal semantics based on Markov kernels. We present two main results:

- We prove the semantics is sound with respect to the denotational semantics of PNKAT programs, as defined in [17, 18], and the first-order semantics of PNKAT programs.
- We prove the semantics is complete with respect to the denotational semantics of PNKAT programs, as defined in [17, 18], and the first-order semantics of PNKAT programs.

This work is supported by the National Science Foundation (NSF) under grant CCF-1529218.

Keywords: Probabilistic programming, Probabilistic semantics, Kleene algebra with tests, Denotational semantics, PNKAT.

1. Introduction

The recent emergence of software-defined networking (SDN) has led to the development of new network programming languages such as Fernet [11], Pyretic [23], and others. These languages are designed to allow network operators to specify the intended behavior of a network in terms of high-level constructs such as packet redirection and connection-oriented services. However, despite many notable advances, these frameworks allow a fundamental limitation: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network functionality is simple, or when the proportion of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory solutions to many more complex situations that arise in practice:

- Congestion: the network operator wishes to control the expected degree of congestion on each link given a model of the demands for traffic.
- Failure: the network operator wishes to calculate the probability that packets will be delivered to their destination, given that devices and links fail with a certain probability.
What ProbNetKAT can do
What ProbNetKAT can do

Verify reachability properties
What ProbNetKAT can do

Verify reachability properties

✦ but for probabilistic networks
What ProbNetKAT can do

Verify reachability properties
✦ but for probabilistic networks

Verify fault tolerance
What ProbNetKAT can do

Verify reachability properties
✦ but for probabilistic networks

Verify fault tolerance
✦ k-resilience
What ProbNetKAT can do

Verify reachability properties
✦ but for probabilistic networks

Verify fault tolerance
✦ k-resilience
✦ "is scheme A is more resilient than scheme B?"
What ProbNetKAT can do

Verify reachability properties
✦ but for probabilistic networks

Verify fault tolerance
✦ k-resilience
✦ "is scheme A is more resilient than scheme B?"
✦ probability of packet delivery
What ProbNetKAT can do

Verify reachability properties
✦ but for probabilistic networks

Verify fault tolerance
✦ k-resilience
✦ "is scheme A is more resilient than scheme B?"
✦ probability of packet delivery

Compute quantitative network metrics
What ProbNetKAT can do

Verify reachability properties
✦ but for probabilistic networks

Verify fault tolerance
✦ k-resilience
✦ "is scheme A is more resilient than scheme B?"
✦ probability of packet delivery

Compute quantitative network metrics
✦ "expected number of hops?"
What ProbNetKAT can do

Verify reachability properties
✦ but for probabilistic networks

Verify fault tolerance
✦ k-resilience
✦ "is scheme A more resilient than scheme B?"
✦ probability of packet delivery

Compute quantitative network metrics
✦ "expected number of hops?"
✦ "expected link congestion?"
What ProbNetKAT can do

Verify reachability properties
✦ but for probabilistic networks

Verify fault tolerance
✦ k-resilience
✦ "is scheme A is more resilient than scheme B?"
✦ probability of packet delivery

Compute quantitative network metrics
✦ "expected number of hops?"
✦ "expected link congestion?"
✦ computes analytical solution, not approximation
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington
Case Study

F10: A Fault-Tolerant Engineered Network

Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson

University of Washington

Motivation

 대하여 short-term failures in data centers are common
Motivation

- short-term failures in data centers are common
- application performance suffers
**Motivation**

- short-term failures in data centers are common
- application performance suffers
- despite 1:1 redundancy!
Motivation

✧ short-term failures in data centers are common
✧ application performance suffers
✧ despite 1:1 redundancy!
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington

Motivation
✦ short-term failures in data centers are common
✦ application performance suffers
✦ despite 1:1 redundancy!

Solution
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington

Motivation

✧ short-term failures in data centers are common
✧ application performance suffers
✧ despite 1:1 redundancy!

Solution

✧ detect failures of neighboring links & switches...
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington

Motivation

✧ short-term failures in data centers are common
✧ application performance suffers
✧ despite 1:1 redundancy!

Solution

✧ detect failures of neighboring links & switches...
✧ ...and route around them
Case Study: Topology

An ABFatTree is much like a regular FatTree
Case Study: Topology

An ABFatTree is much like a regular FatTree
Case Study: Topology

An ABFatTree is much like a regular FatTree
An ABFatTree is much like a regular FatTree
Case Study: Topology

An ABFatTree is much like a regular FatTree

But it provides shorter detours around failures
Case Study: Topology

An ABFatTree is much like a regular FatTree

But it provides shorter detours around failures
Case Study: Topology

An ABFatTree is much like a regular FatTree

But it provides shorter detours around failures
Case Study: Routing Schemes

We implemented F10 as a series of 3 refinements
Case Study: Routing Schemes

We implemented F10 as a series of 3 refinements
Case Study: Routing Schemes

We implemented F10 as a series of 3 refinements:

- **F10₀**: Shortest path routing
- **F10₃**: [Additional details not provided]
- **F10₃,₅**: [Additional details not provided]
Case Study: Routing Schemes

We implemented F10 as a series of 3 refinements:

- **F10\textsubscript{0}**: shortest path routing
- **F10\textsubscript{3}**: $F10\textsubscript{0}$ + 3-hop rerouting
- **F10\textsubscript{3,5}**
Case Study: Routing Schemes

We implemented F10 as a series of 3 refinements

\[
\begin{align*}
F_{10_0} & : \text{shortest path routing} \\
F_{10_3} & : F_{10_0} + 3\text{-hop rerouting} \\
F_{10_{3,5}} & : F_{10_3} + 5\text{-hop rerouting}
\end{align*}
\]
Case Study: Routing Schemes

We implemented F10 as a series of 3 refinements:

- **F10₀**: Shortest path routing
- **F10₃**: F10₀ + 3-hop rerouting
- **F10₃,5**: F10₃ + 5-hop rerouting

Requires State
Case Study: k-resilience

We verified k-resilience using ProbNetKAT
Case Study: k-resilience

We verified k-resilience using ProbNetKAT

<table>
<thead>
<tr>
<th>$k$</th>
<th>F10$_0$</th>
<th>F10$_3$</th>
<th>F10$_{3,5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>$\infty$</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

$k = \text{number of failures} \quad \checkmark = 100\% \text{ packet delivery}$

Table 1. Evaluating $k$-resilience of F10.

Table 2. Comparing schemes under $k$ failures.
Case Study: $k$-resilience

We verified $k$-resilience using ProbNetKAT

<table>
<thead>
<tr>
<th>$k$</th>
<th>$F_{10_0}$</th>
<th>$F_{10_3}$</th>
<th>$F_{10_{3,5}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>$\infty$</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

$k$ = number of failures

✓ = 100% packet delivery
Case Study: k-resilience

We verified k-resilience using ProbNetKAT

<table>
<thead>
<tr>
<th>k</th>
<th>F10₀</th>
<th>F10₃</th>
<th>F10₃,₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>×</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

An uninitialized flag caused all packets to be dropped

k = number of failures ✓ = 100% packet delivery
Case Study: \( k \)-resilience

After fixing the bug...

<table>
<thead>
<tr>
<th>( k )</th>
<th>( F10_0 )</th>
<th>( F10_3 )</th>
<th>( F10_{3,5} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>1</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>3</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>∞</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

\( k \) = number of failures, ✔️ = 100% packet delivery

Table 1. Evaluating \( k \)-resilience of \( F10 \).

Comparing schemes under \( k \) failures.

Here \( d \) denotes the maximum degree of all nodes in the FatTree and AB FatTree topologies from Figures 5 and 6, which we encode as programs \( fa\text{\_tree} \) and \( abfa\text{\_tree} \).

6.3 Checking invariants

We can gain confidence in the correctness of our implementation of \( F10 \) by verifying that it maintains certain key invariants. As an example, recall our implementation of \( F10_{3,5} \): when we perform 5-hop rerouting, we use an extra bit (default) to notify the next hop aggregation switch to forward the packet downwards instead of performing default forwarding. The next hop follows this instruction and also sets default back to 1.

To verify this property, we check the following equivalence:

\[
\begin{align*}
&8 t, k: b_M \leftarrow F10_{3,5}, t, f_k \Rightarrow \checkmark \\
&\text{\( b_M \leftarrow F10_{3,5}, t, f_k \Rightarrow \checkmark \); default} = 1
\end{align*}
\]

We executed the check using our implementation for \( k \in \{0, 1, 2, 3, 4, 1\} \) and \( t \in \{fa\text{\_tree}, abfa\text{\_tree}\} \).

As discussed below, we actually failed to implement this feature correctly on our first attempt due to a subtle bug—we neglected to initialize the default to 1 at the ingress.

6.4 \( F10 \) routing with FatTree

We previously saw that the structure of FatTree doesn’t allow 3-hop rerouting on failures because all subtrees are of the same type. This would mean that augmenting ECMP with 3-hop rerouting should have no effect, i.e., 3-hop rerouting should never kick in and act as a no-op. To verify this, we can check the following equivalence:

\[
\begin{align*}
&8 k: b_M \leftarrow F10_0, fa\text{\_tree}, f_k \Rightarrow \checkmark \\
&\text{\( b_M \leftarrow F10_3, fa\text{\_tree}, f_k \Rightarrow \checkmark \)}
\end{align*}
\]

We have used our implementation to check that this equivalence indeed holds for \( k \in \{0, 1, 2, 3, 4, 1\} \).
Case Study: probability of delivery

We evaluated packet loss when link failures increase

Dramatic improvement when using rerouting
Case Study: probability of delivery

We evaluated packet loss when link failures increase.

Dramatic improvement when using rerouting.
Case Study: expected hop count

The price of resilience: increased paths lengths

![Graph showing expected hop count vs link failure probability for different network configurations]

AB FatTree outperforms regular FatTree
Case Study: expected hop count

The price of resilience: increased paths lengths

![Graph showing expected hop count vs link failure probability for different routing strategies.

Legend:
- AB FatTree, F10 no rerouting
- AB FatTree, F10 3-hop rerouting
- AB FatTree, F10 3+5-hop rerouting
- FatTree, F10 3+5-hop rerouting

Graph notes:
- Regular FatTree only has long backup paths
- AB FatTree outperforms regular FatTree

6.10 Discussion

As this case study of resilient routing in datacenters shows, the stochastic matrix representation of ProbNetKAT programs and accompanying decision procedure enable us to answer a wide variety of questions about probabilistic networks completely automatically. These new capabilities represent a significant advance over current network verification tools, which are based on deterministic packet-forwarding models [9, 15, 17, 22].

7 DECIDING FULL PROBNETKAT: OBSTACLES AND CHALLENGES

As we have just seen, history-free ProbNetKAT can describe sophisticated network routing schemes under various failure models, and program equivalence for the language is decidable. However, it is less expressive than the original ProbNetKAT language, which includes an additional primitive \( \text{dup} \). Intuitively, this command duplicates a packet \( P_k \) and outputs the word \( P_k^\ast \), where \( H = P_k^\ast \) is the set of non-empty, finite sequences of packets. An element of \( H \) is called a packet.
Wrapping Up
Conclusion

ProbNetKAT is the first probabilistic network verification tool
Conclusion

ProbNetKAT is the first **probabilistic** network verification tool.

Can verify reachability properties even if network behavior is not deterministic.
Conclusion

ProbNetKAT is the first **probabilistic** network verification tool

Can verify reachability properties
even if network behavior is not deterministic

Can reason about resilience
e.g., k-resilience, probability of delivery
Conclusion

ProbNetKAT is the first **probabilistic** network verification tool

Can verify reachability properties even if network behavior is not deterministic

Can reason about resilience e.g., k-resilience, probability of delivery

Can reason about quantitative properties e.g., expected path length under failure model
Future Work
Future Work

Scalable implementation
Current prototype does not scale beyond 100 switches
Future Work

Scalable implementation
Current prototype does not scale beyond 100 switches

Probabilistic Inference
Given observation of packet loss, what link failure has most likely occurred?
Future Work

Scalable implementation
Current prototype does not scale beyond 100 switches

Probabilistic Inference
Given observation of packet loss, what link failure has most likely occurred?

More expressive language
ProbNetKAT has no notion of queuing or time