Taking Probabilistic NetKAT to the Limit

ANONYMOUS AUTHOR(S)

We develop new techniques for reasoning about probabilistic network programs. The core of our approach is based on a semantic characterization of the history-free fragment of Probabilistic NetKAT in terms of finite-state, absorbing Markov chains. The key technical challenge lies in computing the semantics of the iteration operator, which we handle using an encoding in the style of a small-step operational semantics. We present a prototype implementation and develop heuristic optimizations that enable it to scale to networks of realistic size. Using examples, we show how our method can be used to establish generic properties such as program equivalence and refinement, as well as program-specific properties such as resilience to failures. We compare the scalability of our implementation against a state-of-the art tool, and we develop an extended case study involving a recently proposed design for data center networks.

1 INTRODUCTION

> *Take it to the limit, take it to the limit*
> *Take it to the limit one more time.*
> —The Eagles

Networks are some of the most complex and critical computer systems used today. As such, researchers have long sought to develop automated techniques for modeling and analyzing their behavior [49]. Over the last decade, the emergence of tools for applying ideas from programming language to problems in networking [6, 7, 35] has opened up new avenues for reasoning about networks in a rigorous and principled way [4, 14, 26, 28]. Building on these initial advances, researchers have started to target more sophisticated networks that exhibit richer phenomena.

In particular, there is renewed attention on *randomization*, both as a tool for designing network protocols and for modeling the subtle behaviors that arise in large-scale systems—e.g., uncertainty about the inputs to the network as well as device and link failures. Although programming languages for describing randomized protocols exist [13, 17], support for automated reasoning about such programs remains quite limited. The key challenges stem from the fact that even basic properties are often quantitative properties of probability distributions in disguise, and seemingly simple programs can generate highly complex distributions, especially in the presence of iteration. The (un)decidability of elementary questions, such as program equivalence and satisfiability, have been difficult to settle in the probabilistic setting, except in certain special cases [20, 25].

This paper develops new techniques for reasoning about programs in ProbNetKAT, a probabilistic language for modeling and reasoning about packet-switched networks. As its name suggests, ProbNetKAT is based on NetKAT [4, 14], which is in turn based on Kleene algebra with tests (KAT), an algebraic system combining Boolean predicates and regular expressions. ProbNetKAT extends NetKAT with a random choice operator and a semantics based on Markov kernels [13, 45]. ProbNetKAT can be used to implement randomized protocols (e.g., selecting forwarding paths to balance load [31, 46]); to describe uncertainty about traffic demands (e.g., the diurnal fluctuations commonly seen in wide-area networks [39]); and to model failures (e.g., of switches and links [19]).

Many properties of interest can be encoded using ProbNetKAT—more specifically, as quantitative properties of the distributions on output packets produced for various inputs. Hence, if we had a way to compute these distributions exactly, it would be straightforward to build tools that could verify quantitative network properties automatically. However, the semantics of ProbNetKAT is surprisingly subtle: using the iteration operator (i.e., the Kleene star from regular expressions), it is possible to write programs that generate continuous distributions over an uncountable space of packet history sets [13, Theorem 3]. Accordingly, computing the semantics of ProbNetKAT programs involves representing and manipulating infinitary objects.

Prior work [45] developed a domain-theoretic characterization of ProbNetKAT that produces a finite approximation of the semantics on any input. Unfortunately, this work did not provide guarantees on how fast the approximations converge, and so it does not lead to an algorithm for computing the semantics directly. Fortunately, as we explain below, it turns out that the full power of ProbNetKAT is not needed to solve many of the problems that arise in practice. By shifting to a more restricted model—the so-called *history-free* fragment of the language—we can develop an algorithm for exactly computing the output distributions of a program on all possible inputs. The resulting finite, closed-form representation precisely characterizes the semantics of a given ProbNetKAT program, allowing probabilities of output events to be effectively computed.

The foundation of our approach is based on a novel representation of ProbNetKAT programs as finite-state Markov chains. By carefully designing this encoding, the limiting distribution of the Markov chain can be computed efficiently and exactly in closed form, giving a concise presentation of the semantics. More specifically, we define a *big-step* semantics that models each program using a Markov chain that transitions from input to output in a single step, or equivalently, a finite stochastic matrix. While these matrices can be easily computed for simple program constructs, it is not straightforward for the iteration operator—intuitively, the finite matrix needs to somehow capture the result of an infinite stochastic process. To address this challenge, we encode programs using a refined Markov chain with a larger state space, modeling iteration in the style of a *small-step* semantics. With some care, this chain can be transformed to an absorbing Markov chain, from which we derive a closed-form solution for the limit behavior using elementary matrix calculations. We prove the soundness of this approach with respect to the denotational semantics [45].

Although the history-free fragment of ProbNetKAT is a restriction of the full language, it captures the input-output behavior of the network and so is still expressive enough to handle a wide range of practical problems. In fact, most contemporary deterministic verification tools, including Anteater [34], Header Space Analysis [26], and Veriflow [28], are also based on history-free models. To reason about properties that involve paths (e.g., waypointing, isolation, loop-freedom), one can check a series of input-output properties, one for each hop in the path, or augment the program with extra state to record the path directly. In our ProbNetKAT implementation, working with history-free programs has an important practical benefit: it reduces the space requirements by an exponential factor, making it feasible to analyze complex protocols in large topologies.

Automated reasoning for probabilistic systems is an active research area with a rich history, and there are now numerous tools based on probabilistic model checking (e.g., Dehnert et al. [9], Kwiatkowska et al. [32]) and symbolic inference (e.g., Gehr et al. [18]). Hence, it is natural to ask whether one could simply encode probabilistic networks using an existing general-purpose tool. For instance, probabilistic model checking can be used to automatically reason about probabilistic Markov chains. However, it is worth noting that to use these tools in the context of networks, one would need to somehow encode the behavior of the network as a Markov chain—a non-trivial task, given that the encoding has a direct impact on solver performance. As we show in our evaluation, there are significant benefits to focusing on a narrower programming model since it affords greater control over computationally-expensive subroutines, and gives more opportunities for optimization. In particular, because the Markov chains manipulated in our tool are of a particularly simple form, the semantics can be computed using just a few calls to a highly optimized linear algebra package.

We have built a prototype implementation of our approach in OCaml. Given a program as input, it computes a stochastic matrix that models its semantics in a finite and explicit form, using the UMFPACK linear algebra library [8] as a back-end solver to compute limiting distributions. To make the approach scale, our tool incorporates a number of optimizations and symbolic techniques to compactly represent sparse matrices. Although building a highly-optimized implementation would involve further engineering (and is not the primary focus of this paper), our prototype is already

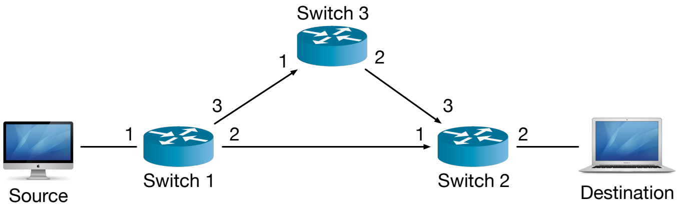


Fig. 1. Network topology for running example.

quite fast and is able to handle programs of moderate size. It also scales much better than a state-of-the-art tool [17]—by more than two orders of magnitude on a representative benchmark program. We have used our tool to carry out detailed case studies of probabilistic reasoning, analyzing the resilience of different fault-tolerant routing schemes in the context of data center networks.

Contributions and outline. The main contribution of this paper is an approach for precisely computing the full semantics of history-free ProbNetKAT programs. We develop a new, tractable semantics in terms of stochastic matrices in two stages, we establish soundness with respect to ProbNetKAT's original denotational model, we implement our method in a prototype implementation, and we evaluate it on a realistic networking case study.

In §2 and §3, we review ProbNetKAT using a simple running example.

In §4, we present a semantics based on *finite stochastic matrices* and show that it fully characterizes the behavior of ProbNetKAT programs (Theorem 4.1). In this semantics, the matrices encode Markov chains over the state space 2^{Pk}. A single step of this "big-step" chain models the entire execution of a program, going directly from the initial state corresponding to the set of input packets to the final state corresponding to the set of output packets. However, we still need a way to explicitly *compute* the matrix for p^*, which is given as a limit.

In §5, we show how to compute the big-step matrix associated with p^* in closed form. Note that this is *not* simply the calculation of the stationary distribution of a Markov chain, as the semantics of p^* is more subtle. Instead, we define a second Markov chain with a larger state space in which each "small-step" transition models one iteration of p^*. We then show how to transform this finer Markov chain into an absorbing Markov chain, which admits a closed form solution for its limiting distribution. Together, the big- and small-step semantics enable us to analytically compute a finite representation of the program semantics. This result yields an effective procedure for deciding program equivalence (Corollary 5.8)—i.e., simply compare matrix representations—and is in contrast with the original denotational semantics [13], which provides only an approximation theorem for the semantics of iteration p^* and so is not suitable for deciding equivalence.

In §6, we describe an implementation of our method including symbolic data structures and heuristic optimizations that are needed to handle the large state space efficiently and obtain good performance. We evaluate the scalability of our tool on a common data center design and compare its performance against Bayonet, a state-of-the-art probabilistic tool for analyzing networks.

In §7, we present real-world case studies that use the stochastic matrix representation to answer questions about the resilience of data center networks in the presence of link failures.

We survey related work in §8 and conclude in §9. Detailed proofs are given in the appendix.

2 OVERVIEW

This section introduces a running examples that illustrates the main features of the ProbNetKAT language as well as some quantitative network properties that arise in practice.

2.1 A Crash Course in ProbNetKAT

Consider the network shown in Figure 1, which connects a source to a destination in a topology with three switches. We will first develop a ProbNetKAT program that forwards packets from the source to the destination, and then verify that it correctly implements the desired behavior by reducing the verification problem to program equivalence. Next, we will show how to enrich our program to model the possibility of link failures, and develop a fault-tolerant forwarding scheme that automatically routes around failures. Using a quantitative version of program refinement, we will show that the fault-tolerant program is indeed more resilient than the initial program. Finally, we will show how to compute the expected resilience of each implementation analytically.

To a first approximation, a ProbNetKAT program can be thought of as a random function that maps input packets to sets of output packets. Packets are modeled as records, with fields for standard headers—such as the source (src) and destination (dst) addresses—as well as two fields switch (sw) and port (pt) encoding the current location of the packet. ProbNetKAT provides several primitives for manipulating packets. A *modification* $f{\leftarrow}n$ returns the input packet with the f field updated to n. A *test* $f{=}n$ either returns the input packet unmodified if the test succeeds, or returns the empty set if the test fails. The primitives skip and drop behave like a test that always succeeds and fails, respectively. Programs p, q can be composed in sequence ($p \mathbin{;} q$), in parallel ($p \mathbin{\&} q$), or iterated using Kleene star p^*.

Although ProbNetKAT programs can be freely constructed by composing primitive operations, a typical network model is expressed using a pair of programs: one that describes the forwarding behavior of the switches, and another that describes the network topology. The overall model of the network is obtained by composing these programs into a single program.

The forwarding policy describes how packets are transformed locally by the switches at each hop. In our running example, to route packets from the source to the destination, switches 1 and 2 can simply forward all incoming packets out on port 2 by modifying the port field (pt). This forwarding program can be encoded as a ProbNetKAT program that performs a case analysis on the location of the input packet, and then sets the port field to 2:

$$p \triangleq (\mathsf{sw}{=}1 \mathbin{;} \mathsf{pt}{\leftarrow}2) \;\;\&\;\; (\mathsf{sw}{=}2 \mathbin{;} \mathsf{pt}{\leftarrow}2) \;\;\&\;\; (sw{=}3 \mathbin{;} \mathsf{drop})$$

For the sake of completeness, we specify a policy for switch 3, even though it is unreachable.

The network topology governs how packets move between switches. To represent a simple directed link between two switches, we match on packets located at the source location of the link and update their locations to the destination end of the link. In our example network (Figure 1), a link ℓ_{ij} from switch i to switch $j \neq i$ is encoded as:

$$\ell_{ij} \triangleq \mathsf{sw}{=}i \mathbin{;} \mathsf{pt}{=}j \mathbin{;} \mathsf{sw}{\leftarrow}j \mathbin{;} \mathsf{pt}{\leftarrow}i$$

We can model the entire topology as the union of all links:

$$t \triangleq \ell_{12} \mathbin{\&} \ell_{13} \mathbin{\&} \ell_{32}$$

To build the overall network model, we combine the forwarding policy p with the topology t. A packet traversing the network is alternately processed by switches and links in the network, repeating for as many steps as necessary. In ProbNetKAT:

$$\mathrm{M}(p, t) \triangleq (p \mathbin{;} t)^* \mathbin{;} p$$

The model $\mathrm{M}(p, t)$ captures the behavior of the network on arbitrary input packets, including packets that start or end at arbitrary locations in the interior of the network. It is sometimes useful to consider such partial packet trajectories, but to restrict our attention to packets at the ingress and egress, we can wrap the program with additional predicates that identify the ingress and egress of the topology,

$$\mathit{in} \triangleq \mathsf{sw}{=}1 \,;\, \mathsf{pt}{=}1 \qquad\qquad \mathit{out} \triangleq \mathsf{sw}{=}2 \,;\, \mathsf{pt}{=}2$$

and arrive at the full network model:

$$\mathit{in} \,;\, \mathrm{M}(p,t) \,;\, \mathit{out}$$

To verify that p forwards packets to the destination, we can check the following equivalence:

$$\mathit{in} \,;\, \mathrm{M}(p,t) \,;\, \mathit{out} \;\equiv\; \mathit{in} \,;\, \mathsf{sw}{\leftarrow}2 \,;\, \mathsf{pt}{\leftarrow}2$$

The program on the left-hand side of the equality is the implementation, while the program on the right-hand side can be thought of as a specification that “teleports” each packet to its destination. Previous work [4, 14, 44] used similar reductions to equivalence in order to reason about properties such as waypointing, reachability, isolation, loop freedom.

2.2 Probabilistic Programming and Reasoning.

Real-world networks often exhibit non-deterministic behaviors. For example, networks often use randomized algorithms to balance traffic across multiple paths [31], or use fault tolerant routing schemes to handle unexpected failures [33]. To ensure that the network behaves as expected in these more complicated scenarios requires a form of probabilistic reasoning. Unfortunately, state-of-the-art network verification tools [14, 26, 28] only model deterministic behaviors.

To illustrate the need for probabilistic reasoning, suppose that we want to extend our running example to make it resilient to link failures. Most modern switches implement low-level protocols such as Bidirectional Forwarding Detection (BFD) to compute real-time healthiness information about the physical link connected to each port [5]. Formally, we can enrich our model so that each router has access to a boolean flag up_i that is true if and only if the link connected to the switch at port i is transmitting packets correctly. Then we can adjust the forwarding logic for switch 1 as follows: if link ℓ_{12} is up, use the shortest path to switch 2 as before; otherwise, take a detour via switch 3 and proceed to switch 2 from there:

$$\widehat{p_1} \triangleq (\mathsf{up}_2{=}1 \,;\, \mathsf{pt}{\leftarrow}2) \;\&\; (\mathsf{up}_2{=}0 \,;\, \mathsf{pt}{\leftarrow}3)$$

The programs for switches 2 and 3 are analogous. As before, we can encode the forwarding logic for all switches into a single program:

$$\widehat{p} \triangleq (\mathsf{sw}{=}1 \,;\, \widehat{p_1}) \;\;\&\;\; (\mathsf{sw}{=}2 \,;\, p_2) \;\;\&\;\; (\mathsf{sw}{=}3 \,;\, p_3)$$

Next, we update our encoding of the topology to take link failures into account. Links can fail for a wide variety of reasons including mistakes by human operators, fiber cuts, and hardware errors. A natural way to model these failures is with a probabilistic *failure model*—i.e., a distribution that describes how often links fail. We can encode various failure models using ProbNetKAT:

$$\begin{aligned}
f_0 &\triangleq \mathsf{up}_2{\leftarrow}1 \,;\, \mathsf{up}_3{\leftarrow}1\\
f_1 &\triangleq \bigoplus \left\{ f_0 \mathbin{@} \frac{1}{2},\; \mathsf{up}_2{\leftarrow}0 \,;\, \mathsf{up}_3{\leftarrow}1 \mathbin{@} \frac{1}{4},\; \mathsf{up}_2{\leftarrow}1 \,;\, \mathsf{up}_3{\leftarrow}0 \mathbin{@} \frac{1}{4} \right\}\\
f_2 &\triangleq (\mathsf{up}_2{\leftarrow}1 \oplus_{0.8} \mathsf{up}_2{\leftarrow}0) \,;\, (\mathsf{up}_2{\leftarrow}1 \oplus_{0.8} \mathsf{up}_2{\leftarrow}0)
\end{aligned}$$

In f_0, no links fail. Intuitively, in f_1, the links ℓ_{12} and ℓ_{13} fail with probability 25% each, but at most one fails while in f_2, the links fail independently with probability 20%. In either case, using these flags, we can model a link that only forwards packets when it is up:

$$\widehat{\ell_{ij}} \triangleq \mathsf{up}_i{=}1 \,;\, \ell_{ij}$$

Combining the policy, topology, and failure model, yields a refined model of the entire network:

$$\widehat{\mathrm{M}}(p, t, f) \triangleq \mathsf{var}\ \mathsf{up}_2{\leftarrow}1\ \mathsf{in}$$
$$\mathsf{var}\ \mathsf{up}_3{\leftarrow}1\ \mathsf{in}$$
$$\mathrm{M}((f\,; p), t)$$

This refined model $\widehat{\mathrm{M}}$ wraps our previous model M with declarations of the two local fields up_2 and up_3 and executes the failure model (f) at each hop before executing the programs for the router (p) and topology (t).

Now we can analyze our resilient routing scheme $\widehat{p}$. First, as a sanity check, we can verify that in the absence of failures, it still correctly delivers packets to the destination using the following equivalence:

$$in\,; \widehat{\mathrm{M}}(\widehat{p}, \widehat{t}, f_0)\,; out \;\equiv\; in\,; \mathsf{sw}{\leftarrow}2\,; \mathsf{pt}{\leftarrow}2$$

Next, we can verify that $\widehat{p}$ is 1-resilient—i.e., it delivers all packets provided at most one link fails. Formally, it behaves like the program that "teleports" packets under failure model f_1. This property does not hold for the original, naive routing scheme p:

$$in\,; \widehat{\mathrm{M}}(\widehat{p}, \widehat{t}, f_1)\,; out \;\equiv\; in\,; \mathsf{sw}{\leftarrow}2\,; \mathsf{pt}{\leftarrow}2 \;\not\equiv\; in\,; \widehat{\mathrm{M}}(p, \widehat{t}, f_1)\,; out$$

Under failure model f_2, two links may fail simultaneously and neither of routing schemes is 1-resilient. However, we can still show that the refined routing scheme $\widehat{p}$ performs strictly better than the naive scheme p,

$$\widehat{\mathrm{M}}(p, \widehat{t}, f_2) \;<\; \widehat{\mathrm{M}}(\widehat{p}, \widehat{t}, f_2)$$

where $p < q$ means that q delivers all packets with higher probability than p. This relation can be thought of as a quantitative version of program refinement.

We can establish a variety of properties such as reachability and other global invariants using analogous reductions to equivalence and refinement. But we can also use ProbNetKAT to go a step further and compute quantitative properties of the packet distribution generated the program program. For example, we can compute the probability that each routing scheme delivers packets to the destination under failure model f_2. The answer is 80% for the naive scheme and 96% for the resilient scheme. Such a computation could be used by an Internet Service Provider (ISP) when evaluating the design of a topology and a routing scheme to check that it meets its service-level agreements (SLA) with customers.

In §7 we will analyze a more sophisticated resilient routing scheme and see more complex examples of qualitative and quantitative reasoning with ProbNetKAT drawn from real-world data center networks. But first, we develop the theoretical machinery that underpins our approach.

3 BACKGROUND ON PROBABILISTIC NETKAT

This section reviews the syntax, semantics, and basic properties of ProbNetKAT [13, 45], focusing on the history-free fragment. A synopsis appears in Figure 2.

3.1 Syntax

A *packet* π is a record mapping a finite set of fields $f_1, f_2, \dots, f_k$ to bounded integers n. As we saw in the previous section, fields can include standard header fields such as source (src) and destination (dst) addresses, as well as logical fields for modeling the current location of the packet in the network or variables such as up_i. These logical fields are not present in a physical network packet, but they can track auxiliary information for the purposes of verification. We write $\pi.f$ to denote the value of field f of π and $\pi[f{:=}n]$ for the packet obtained from π by updating field f to hold n. We let Pk denote the set of all packets; note that this is a finite set.

ProbNetKAT can be divided into two classes: *predicates* $(t, u, \ldots)$ and *programs* $(p, q, \ldots)$. Primitive predicates include *tests* $(f{=}n)$ and the Boolean constants *false* (drop) and *true* (skip). Compound predicates are formed using the usual Boolean connectives: disjunction $(t \,\&\, u)$, conjunction $(t \,;\, u)$, and negation $(\neg t)$. Primitive programs include *predicates* (t) and *assignments* $(f{\leftarrow}n)$. The full version of the language also provides a dup primitives, which logs the current state of the packet, but we omit this operator from the history-free fragment of the language considered in this paper; we discuss technical challenges related to full ProbNetKAT in Appendix B.

Compound programs are formed using *parallel composition* $(p \,\&\, q)$, *sequential composition* $(p \,;\, q)$, *iteration* (p^*), and *probabilistic choice* $(p \oplus_r q)$. The probabilistic choice operator $p \oplus_r q$ executes p with probability r and q with probability $1 - r$, where r is rational, $0 \leq r \leq 1$. We often use an n-ary version and omit the r's as in $p_1 \oplus \cdots \oplus p_n$, which is interpreted as executing one of the p_i chosen with equal probability; this construct can be desugared into the binary version.

Inspired by Kleene algebra with tests, conjunction of predicates and sequential composition of programs use the same syntax ($t \,;\, u$ and $p \,;\, q$, respectively), as their semantics coincide. The same is true for disjunction of predicates and parallel composition of programs ($t \,\&\, u$ and $p \,\&\, q$, respectively). The negation operator $(\neg)$ may only be applied to predicates.

Figure 2 presents the core features of the language, but many other useful constructs can be derived. For instance, it is straightforward to encode conditionals and while loops:

$$\mathsf{if}\ t\ \mathsf{then}\ p\ \mathsf{else}\ q \;\triangleq\; t \,; p \,\&\, \neg t \,; q \qquad\qquad \mathsf{while}\ t\ \mathsf{do}\ p \;\triangleq\; (t \,; p)^* \,; \neg t$$

These encodings are well known from KAT [30]. Mutable local variables (*e.g.*, $\mathsf{up_i}$, used to track link healthiness in the running example from §2), can also be desugared into the core language:

$$\mathsf{var}\ f{\leftarrow}n\ \mathsf{in}\ p \;\triangleq\; f{\leftarrow}n \,; p \,; f{\leftarrow}0$$

Here f is a field that is local to p. The final assignment $f{\leftarrow}0$ sets the value of f to a canonical value, which is semantically equivalent to "erasing" it after the field goes out of scope. We often use local variables to record extra information for verification. For example, by using a local field to record whether a packet traversed a given router, one can reason about simple waypointing and isolation properties, even though the history-free fragment of ProbNetKAT does not directly model paths.

3.2 Semantics

In full ProbNetKAT, programs manipulate sets of *packet histories*—non-empty, finite sequences of packets modeling trajectories through the network [13, 45]. The resulting state space is uncountable and modeling the semantics properly requires full-blown measure theory as some programs generate continuous distributions. In the history-free fragment, programs manipulate sets of packets and the state space is finite, which makes the semantics considerably simpler.

PROPOSITION 3.1. *Let* $(\!|-|\!)$ *denote the semantics defined in [45]. Then for all* dup*-free programs* p *and inputs* $a \in 2^{\mathsf{Pk}}$, *we have* $[\![p]\!](a) = (\!|p|\!)(a)$, *where we identify packets and histories of length one.*

Throughout this paper, we can work in the discrete space 2^{Pk}, *i.e.*, the set of sets of packets. An *outcome* (denoted by lowercase variables $a, b, c, \ldots$) is a set of packets and an *event* (denoted by uppercase variables $A, B, C, \ldots$) is a set of outcomes. Given a discrete probability measure on this space, the probability of an event is the sum of the probabilities of its outcomes.

ProbNetKAT programs are interpreted as *Markov kernels* on the space 2^{Pk}. A Markov kernel is a function $2^{\mathsf{Pk}} \to \mathcal{D}(2^{\mathsf{Pk}})$ where $\mathcal{D}$ is the probability (or Giry) monad [21, 29]. Thus, a program p maps an input set of packets $a \in 2^{\mathsf{Pk}}$ to a *distribution* $[\![p]\!](a) \in \mathcal{D}(2^{\mathsf{Pk}})$ over output sets of packets. The semantics uses the following probabilistic constructions:[1]

---

[1] These can also be defined for uncountable spaces, as would be required to handle the full language.

Syntax

$$\begin{array}{rrcl}
\text{Naturals} & n & ::= & 0 \mid 1 \mid 2 \mid \cdots \\
\text{Fields} & f & ::= & f_1 \mid \cdots \mid f_k \\
\text{Packets} & \mathsf{Pk} \ni \pi & ::= & \{f_1 = n_1, \dots, f_k = n_k\} \\
\text{Probabilities} & r & \in & [0,1] \cap \mathbb{Q}
\end{array}$$

$$\begin{array}{rrcll}
\text{Predicates} & t, u & ::= & \mathsf{drop} & \textit{False} \\
 & & \mid & \mathsf{skip} & \textit{True} \\
 & & \mid & f{=}n & \textit{Test} \\
 & & \mid & t \,\&\, u & \textit{Disjunction} \\
 & & \mid & t \,;\, u & \textit{Conjunction} \\
 & & \mid & \neg t & \textit{Negation} \\
\text{Programs} & p, q & ::= & t & \textit{Filter} \\
 & & \mid & f{\leftarrow}n & \textit{Assignment} \\
 & & \mid & p \,\&\, q & \textit{Union} \\
 & & \mid & p \,;\, q & \textit{Sequence} \\
 & & \mid & p \oplus_r q & \textit{Choice} \\
 & & \mid & p^* & \textit{Iteration}
\end{array}$$

Semantics $\boxed{[\![p]\!] \in 2^{\mathsf{Pk}} \to \mathcal{D}(2^{\mathsf{Pk}})}$

$$\begin{aligned}
[\![\mathsf{drop}]\!](a) &\triangleq \delta(\varnothing) \\
[\![\mathsf{skip}]\!](a) &\triangleq \delta(a) \\
[\![f{=}n]\!](a) &\triangleq \delta(\{\pi \in a \mid \pi.f = n\}) \\
[\![f{\leftarrow}n]\!](a) &\triangleq \delta(\{\pi[f{:=}n] \mid \pi \in a\}) \\
[\![\neg t]\!](a) &\triangleq \mathcal{D}(\lambda b.a - b)([\![t]\!](a)) \\
[\![p \,\&\, q]\!](a) &\triangleq \mathcal{D}(\cup)([\![p]\!](a) \times [\![q]\!](a)) \\
[\![p \,;\, q]\!](a) &\triangleq [\![q]\!]^\dagger([\![p]\!](a)) \\
[\![p \oplus_r q]\!](a) &\triangleq r \cdot [\![p]\!](a) + (1-r) \cdot [\![q]\!](a) \\
[\![p^*]\!](a) &\triangleq \bigsqcup_{n \in \mathbb{N}} [\![p^{(n)}]\!](a)
\end{aligned}$$

$$\text{where } p^{(0)} \triangleq \mathsf{skip}, \quad p^{(n+1)} \triangleq \mathsf{skip} \,\&\, p \,;\, p^{(n)}$$

(Discrete) Probability Monad $\mathcal{D}$

$$\begin{array}{ll}
\text{Unit} & \delta : X \to \mathcal{D}(X) \qquad \delta(x) \triangleq \delta_x \\
\text{Bind} & -^\dagger : (X \to \mathcal{D}(Y)) \to \mathcal{D}(X) \to \mathcal{D}(Y) \\
 & f^\dagger(\mu)(A) \triangleq \sum_{x \in X} f(x)(A) \cdot \mu(x)
\end{array}$$

Fig. 2. ProbNetKAT core language: syntax and semantics.

- For a discrete measurable space X, $\mathcal{D}(X)$ denotes the set of probability measures over X; that is, the set of countably additive functions $\mu : 2^X \to [0,1]$ with $\mu(X) = 1$.
- For a measurable function $f : X \to Y$, $\mathcal{D}(f) : \mathcal{D}(X) \to \mathcal{D}(Y)$ denotes the *pushforward* along f; that is, the function that maps a measure μ on X to
  $$\mathcal{D}(f)(\mu) \triangleq \mu \circ f^{-1} = \lambda A \in \Sigma_Y.\ \mu(\{x \in X \mid f(x) \in A\})$$
  which is called the *pushforward measure* on Y.
- The *unit* $\delta : X \to \mathcal{D}(X)$ of the monad maps a point $x \in X$ to the point mass (or *Dirac* measure) $\delta_x \in \mathcal{D}(X)$. The Dirac measure is given by
  $$\delta_x(A) \triangleq \mathbf{1}[x \in A]$$
  That is, the Dirac measure is 1 if $x \in A$ and 0 otherwise.
- The *bind* operation of the monad,
  $$-^\dagger : (X \to \mathcal{D}(Y)) \to \mathcal{D}(X) \to \mathcal{D}(Y)$$
  lifts a function $f : X \to \mathcal{D}(Y)$ with deterministic inputs to a function $f^\dagger : \mathcal{D}(X) \to \mathcal{D}(Y)$ that takes random inputs. Intuitively, this is achieved by averaging the output of f when the inputs are randomly distributed according to μ. Formally,
  $$f^\dagger(\mu)(A) \triangleq \sum_{x \in X} f(x)(A) \cdot \mu(x).$$
- Given two measures $\mu \in \mathcal{D}(X)$ and $\nu \in \mathcal{D}(Y)$, $\mu \times \nu \in \mathcal{D}(X \times Y)$ denotes their *product measure*. This is the unique measure satisfying
  $$(\mu \times \nu)(A \times B) = \mu(A) \cdot \nu(B)$$
  Intuitively, it models distributions over pairs of independent values.

Using these primitives, we can now make our operational intuitions precise (see Figure 2 for formal definitions). A predicate t maps the set of input packets $a \in 2^{\mathsf{Pk}}$ to the subset of packets $b \subseteq a$ satisfying the predicate (with probability 1). Hence, drop drops all packets (i.e., it returns the empty set) while skip keeps all packets (i.e., it returns the input set). The test $f\!=\!n$ returns the subset of input packets whose f-field is n. Negation $\neg t$ filters out the packets returned by t.

Parallel composition $p \,\&\, q$ executes p and q independently on the input set, then returns the union of their results. Note that packet sets do *not* model nondeterminism, unlike the usual situation in Kleene algebras—rather, they model collections of packets traversing possibly different portions of the network simultaneously. In particular, the union operation is *not* idempotent: $p \,\&\, p$ need not have the same semantics as p. Probabilistic choice $p \oplus_r q$ feeds the input to both p and q and returns a convex combination of the output distributions according to r. Sequential composition $p \,;\, q$ can be thought of as a two-stage probabilistic process: it first executes p on the input set to obtain a random intermediate result, then feeds that into q to obtain the final distribution over outputs. The outcome of q is averaged over the distribution of intermediate results produced by p.

We say that two programs are *equivalent*, denoted $p \equiv q$, if they denote the same Markov kernel, *i.e.* if $[\![p]\!] = [\![q]\!]$. As usual, we expect Kleene star p^* to satisfy the characteristic fixed point equation $p^* \equiv \mathsf{skip} \,\&\, p \,;\, p^*$, which allows it to be unrolled ad infinitum. Thus we define it as the supremum of its finite unrollings $p^{(n)}$; see Figure 2. This supremum is taken in a CPO $(\mathcal{D}(2^{\mathsf{Pk}}), \sqsubseteq)$ of distributions that is described in more detail in §3.3. The partial ordering $\sqsubseteq$ on packet set distributions gives rise to a partial ordering on programs: we write $p \leq q$ iff $[\![p]\!](a) \sqsubseteq [\![q]\!](a)$ for all inputs $a \in 2^{\mathsf{Pk}}$. Intuitively, $p \leq q$ iff p produces any particular output packet π with probability at most that of q for any fixed input—q has a larger probability of delivering more output packets.

3.3 The CPO $(\mathcal{D}(2^{\mathsf{Pk}}), \sqsubseteq)$

The space 2^{Pk} with the subset order forms a CPO $(2^{\mathsf{Pk}}, \subseteq)$. Following Saheb-Djahromi [40], this CPO can be lifted to a CPO $(\mathcal{D}(2^{\mathsf{Pk}}), \sqsubseteq)$ on distributions over 2^{Pk}. Because 2^{Pk} is a finite space, the resulting ordering $\sqsubseteq$ on distributions takes a particularly easy form:

$$\mu \sqsubseteq \nu \qquad \Longleftrightarrow \qquad \mu(\{a\}\!\uparrow) \leq \nu(\{a\}\!\uparrow) \text{ for all } a \subseteq \mathsf{Pk}$$

where $\{a\}\!\uparrow \triangleq \{b \mid a \subseteq b\}$ denotes upward closure. Intuitively, ν produces more outputs then μ. As was shown in [45], ProbNetKAT satisfies various monotonicity (and continuity) properties with respect to this ordering, including

$$a \subseteq a' \implies [\![p]\!](a) \sqsubseteq [\![p]\!](a') \qquad \text{and} \qquad n \leq m \implies [\![p^{(n)}]\!](a) \sqsubseteq [\![p^{(m)}]\!](a).$$

As a result, the semantics of p^* as the supremum of its finite unrollings $p^{(n)}$ is well-defined.

While the semantics of full ProbNetKAT requires more domain theory to give a satisfactory characterization of Kleene star, a simpler characterization suffices for the history-free fragment.

Lemma 3.2 (Pointwise Convergence). *Let $A \subseteq 2^{\mathsf{Pk}}$. Then for all programs p and inputs $a \in 2^{\mathsf{Pk}}$,*

$$[\![p^*]\!](a)(A) = \lim_{n\to\infty} [\![p^{(n)}]\!](a)(A).$$

4 BIG-STEP SEMANTICS

In this section we propose an alternative program semantics in terms of finite-state Markov chains. While there are many possible translations of ProbNetKAT programs as Markov chains, we want an encoding that will enable precise computation of the semantics of ProbNetKAT programs. The design of this semantics requires some care, and we will proceed in two steps. We first present a coarse, big-step style Markov chain semantics that is conceptually simple and can be precisely computed for all program constructs with the exception of iteration. To handle iteration, we will

$$\boxed{\mathcal{B}\llbracket p\rrbracket \in \mathbb{S}(2^{\mathsf{Pk}})}$$

$$\begin{aligned}
\mathcal{B}\llbracket \mathsf{drop}\rrbracket_{ab} &\triangleq \mathbf{1}[b=\varnothing] \\
\mathcal{B}\llbracket \mathsf{skip}\rrbracket_{ab} &\triangleq \mathbf{1}[a=b] \\
\mathcal{B}\llbracket f{=}n\rrbracket_{ab} &\triangleq \mathbf{1}[b=\{\pi\in a \mid \pi.f=n\}] \\
\mathcal{B}\llbracket \neg t\rrbracket_{ab} &\triangleq \mathbf{1}[b\subseteq a]\cdot\mathcal{B}\llbracket t\rrbracket_{a,a-b} \\
\mathcal{B}\llbracket f{\leftarrow}n\rrbracket_{ab} &\triangleq \mathbf{1}[b=\{\pi[f:=n] \mid \pi\in a\}]
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}\llbracket p \,\&\, q\rrbracket_{ab} &\triangleq \sum_{c,d}\mathbf{1}[c\cup d=b]\cdot\mathcal{B}\llbracket p\rrbracket_{a,c}\cdot\mathcal{B}\llbracket q\rrbracket_{a,d} \\
\mathcal{B}\llbracket p\,;q\rrbracket &\triangleq \mathcal{B}\llbracket p\rrbracket\cdot\mathcal{B}\llbracket q\rrbracket \\
\mathcal{B}\llbracket p\oplus_r q\rrbracket &\triangleq r\cdot\mathcal{B}\llbracket p\rrbracket+(1-r)\cdot\mathcal{B}\llbracket q\rrbracket \\
\mathcal{B}\llbracket p^*\rrbracket_{ab} &\triangleq \lim_{n\to\infty}\mathcal{B}\llbracket p^{(n)}\rrbracket_{ab}
\end{aligned}$$

Fig. 3. Big-Step Semantics: $\mathcal{B}\llbracket p\rrbracket_{ab}$ denotes the probability that program p produces output b on input a.

construct a finer, small-step style Markov chain semantics in the next section that is tailored to iterative programs.

As we saw in §3, the denotational semantics of ProbNetKAT interprets programs as maps $2^{\mathsf{Pk}}\to\mathcal{D}(2^{\mathsf{Pk}})$. Since the set of packets Pk is finite, so is its powerset 2^{Pk}. Thus any distribution over packet sets is discrete and can be characterized by a *probability mass function*, *i.e.* a function

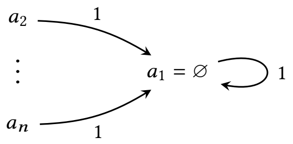
$$f: 2^{\mathsf{Pk}}\to[0,1] \qquad \text{such that} \qquad \sum_{b\subseteq\mathsf{Pk}} f(b)=1.$$

When working with Markov chains, it will be convenient to view f as a *stochastic vector*, *i.e.* a vector of non-negative entries that sums to 1. The vector is indexed by packet sets $b\subseteq\mathsf{Pk}$ with b-th component $f(b)$. A program, being a function that maps inputs a to distributions over outputs, can then be organized as a square matrix indexed by Pk in which the stochastic vector corresponding to input a appears as the a-th row.

Thus we can interpret a program p as a matrix $\mathcal{B}\llbracket p\rrbracket\in[0,1]^{2^{\mathsf{Pk}}\times 2^{\mathsf{Pk}}}$ indexed by packet sets, where the matrix entry $\mathcal{B}\llbracket p\rrbracket_{ab}$ gives the probability that p produces output $b\in 2^{\mathsf{Pk}}$ on input $a\in 2^{\mathsf{Pk}}$. The rows of $\mathcal{B}\llbracket p\rrbracket$ are stochastic vectors, each encoding the output distribution corresponding to a particular input set a; such a matrix is called *(right-)stochastic*. We denote by $\mathbb{S}(2^{\mathsf{Pk}})$ the set of right-stochastic square matrices indexed by 2^{Pk}.

The interpretation of programs as stochastic matrices is defined formally in Figure 3. Deterministic program primitives are interpreted as $(0,1)$-matrices—*e.g.*, the program primitive drop is interpreted as the stochastic matrix

$$\mathcal{B}\llbracket \mathsf{drop}\rrbracket = \begin{array}{c} \\ \varnothing \\ \vdots \\ a_n \end{array}\!\!\begin{array}{c} \begin{array}{cccc} \varnothing & b_2 & \ldots & b_n \end{array} \\ \begin{bmatrix} 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix} \end{array} \tag{1}$$



that assigns all probability mass to the $\varnothing$-column. Similarly, the primitive skip is interpreated as the identity matrix. The formal definitions in Figure 3 use Iverson brackets: $\mathbf{1}[\varphi]$ is 1 if φ is true, and 0 otherwise.

As suggested by the picture in (1), a stochastic matrix $B\in\mathbb{S}(2^{\mathsf{Pk}})$ can be viewed as a *Markov chain* (MC), a probabilistic transition system with state space 2^{Pk} that makes a random transition between states at each time step. The matrix entry B_{ab} gives the probability that the system transitions to state b starting from state a. Accordingly, sequential composition is interpreted by matrix product:

$$\mathcal{B}\llbracket p\,;q\rrbracket_{ab}=\sum_c \mathcal{B}\llbracket p\rrbracket_{ac}\cdot\mathcal{B}\llbracket q\rrbracket_{cb}=(\mathcal{B}\llbracket p\rrbracket\cdot\mathcal{B}\llbracket q\rrbracket)_{ab}.$$

This equation reflects the intuitive semantics of sequential composition: a step from a to b in $\mathcal{B}[\![p;q]\!]$ occurs via a step from a to some intermediate state c in $\mathcal{B}[\![p]\!]$, followed by a step from c to the final state b in $\mathcal{B}[\![q]\!]$.

4.1 Soundness

The main theoretical result of this section is a proof that the finite matrix $\mathcal{B}[\![p]\!]$ fully characterizes the behavior of a program p on packets.

Theorem 4.1 (Soundness). *For any program p and any sets $a, b \in 2^{\mathsf{Pk}}$, $\mathcal{B}[\![p^*]\!]$ is well-defined, $\mathcal{B}[\![p]\!]$ is a stochastic matrix, and $\mathcal{B}[\![p]\!]_{ab} = [\![p]\!](a)(\{b\})$.*

As an application, checking program equivalence for p and q reduces to checking equality of the big-step matrices $\mathcal{B}[\![p]\!]$ and $\mathcal{B}[\![q]\!]$.

Corollary 4.2. *For programs p and q, $[\![p]\!] = [\![q]\!]$ if and only if $\mathcal{B}[\![p]\!] = \mathcal{B}[\![q]\!]$.*

Because the big-step Markov chains are all finite state, the transition matrices are finite dimensional, with rationals as entries. Accordingly, program equivalence (and other quantitative properties) can be automatically verified provided we can compute the big-step matrices for given programs. This is straightforward for most program constructions, except $\mathcal{B}[\![p^*]\!]$: this matrix is defined in terms of a limit. While we can approximate this matrix, we would like to compute it exactly. The next section considers how to compute the semantics for iteration.

5 SMALL-STEP SEMANTICS

The semantics developed in the previous section can be viewed as a "big-step" semantics in which a single step of the chain models the entire execution of a program from initial state a (the set of input packets) to final state b (the set of output packets). To compute the semantics of iteration, we will build a finer, "small-step" Markov chain where each transition models one iteration of p^*.

To develop intuition, first consider simulating p^* using a transition system with states given by triples $\langle p, a, b\rangle$, consisting of a program p to be executed, a current set of (input) packets a, and an accumulator set b of packets output so far. To model the execution of p^* on input $a \subseteq \mathsf{Pk}$, we start from the initial state $\langle p^*, a, \emptyset\rangle$ and unroll p^* one iteration according to the characteristic equation $p^* \equiv \mathsf{skip} \,\&\, p\,;p^*$, yielding the following transition:

$$\langle p^*, a, \emptyset\rangle \xrightarrow{\quad 1 \quad} \langle \mathsf{skip} \,\&\, p\,;p^*, a, \emptyset\rangle$$

Then, we execute both skip and $p\,;p^*$ on the input set and take the union of their results. To execute skip, we immediately output the input set with probability 1:

$$\langle \mathsf{skip} \,\&\, p\,;p^*, a, \emptyset\rangle \xrightarrow{\quad 1 \quad} \langle p\,;p^*, a, a\rangle$$

To execute the remaining component $p\,;p^*$, we first execute p and then feed its output into p^*:

$$\forall a': \quad \langle p\,;p^*, a, a\rangle \xrightarrow{\mathcal{B}[\![p]\!]_{a,a'}} \langle p^*, a', a\rangle$$

At this point the cycle closes and we are back to executing p^*, albeit with a different input set a' and some additional accumulated output packets. The structure of the resulting Markov chain is shown in Figure 4.

As the first two steps of execution are deterministic, we can simplify the transition system by collapsing all three steps into one, as illustrated in Figure 4. Moreover, the program component can

$$\begin{tikzcd}
\langle p^*, a, b \rangle \arrow[r, dashed, "1"] \arrow[drr, "\mathcal{B}[\![p]\!]_{a,a'}"] & \langle \mathsf{skip} \,\&\, p \,;\, p^*, a, b \rangle \arrow[r, dashed, "1"] & \langle p \,;\, p^*, a, b \cup a \rangle \arrow[d, dashed, "\mathcal{B}[\![p]\!]_{a,a'}"] \\
 & & \langle p^*, a', b \cup a \rangle
\end{tikzcd}$$

Fig. 4. The small-step semantics is given by a Markov chain whose states are configurations of the form ⟨*program*, *input set*, *output accumulator*⟩. The three dashed arrows can be collapsed into the single solid arrow, rendering the program component superfluous.

also be dropped, as it remains constant across transitions. Hence, we work with a Markov chain over the state space $2^{\mathsf{Pk}} \times 2^{\mathsf{Pk}}$, defined formally as follows:

$$\mathcal{S}[\![p]\!] \in \mathbb{S}(2^{\mathsf{Pk}} \times 2^{\mathsf{Pk}})$$
$$\mathcal{S}[\![p]\!]_{(a,b),(a',b')} \triangleq \mathbf{1}[b' = b \cup a] \cdot \mathcal{B}[\![p]\!]_{a,a'}$$

As a sanity check, we can verify that the matrix $\mathcal{S}[\![p]\!]$ indeed defines a Markov chain.

Lemma 5.1. *$\mathcal{S}[\![p]\!]$ is stochastic.*

Next, we show that each step in $\mathcal{S}[\![p]\!]$ models an iteration of p^*. Formally, the $(n+1)$-step of $\mathcal{S}[\![p]\!]$ is equivalent to the big-step behavior of the n-th unrolling of p^*.

Proposition 5.2. $\mathcal{B}[\![p^{(n)}]\!]_{a,b} = \sum_{a'} \mathcal{S}[\![p]\!]^{n+1}_{(a,\emptyset),(a',b)}$

Proof. Direct induction on the number of steps $n \geq 0$ fails because the hypothesis is too weak. We first generalize from start states with empty accumulator to arbitrary start states.

Lemma 5.3. *Let p be program. Then for all $n \in \mathbb{N}$ and $a, b, b' \subseteq \mathsf{Pk}$, we have*

$$\sum_{a'} \mathbf{1}[b' = a' \cup b] \cdot \mathcal{B}[\![p^{(n)}]\!]_{a,a'} = \sum_{a'} \mathcal{S}[\![p]\!]^{n+1}_{(a,b),(a',b')}.$$

Proposition 5.2 then follows by instantiating Lemma 5.3 with $b = \emptyset$. □

Intuitively, the long-run behavior of $\mathcal{S}[\![p]\!]$ approaches the big-step behavior of p^*: letting (a_n, b_n) denote the random state of the Markov chain $\mathcal{S}[\![p]\!]$ after taking n steps starting from $(a, \emptyset)$, the distribution of b_n for $n \to \infty$ is precisely the distribution of outputs generated by p^* on input a (by Proposition 5.2 and the definition of $\mathcal{B}[\![p^*]\!]$). We show how to compute this limit next.

5.1 Closed form

The limiting behavior of finite state Markov chains has been well-studied in the literature (*e.g.*, see [27]). For so-called *absorbing* Markov chains, the limit distribution can be computed exactly. While the small-step chain $\mathcal{S}[\![p]\!]$ may not be absorbing, with a bit of work we can convert it into an absorbing Markov chain.

We will need some basic concepts from the theory of Markov chains. A state s of a Markov chain T is *absorbing* if it transitions to itself with probability 1:

$$\begin{tikzcd} \boxed{s} \arrow[loop right, "1"] \end{tikzcd} \qquad \text{(formally: } T_{s,s'} = \mathbf{1}[s = s'])$$

A Markov chain $T \in \mathbb{S}(S)$ is *absorbing* if each state can reach an absorbing state:

$$\forall s \in S.\ \exists s' \in S, n \geq 0.\ T^n_{s,s'} > 0 \text{ and } T_{s',s'} = 1$$

The non-absorbing states of an absorbing MC are called *transient*. Assume T is absorbing with n_t transient states and n_a absorbing states. After reordering the states so that absorbing states appear before transient states, T has the form

$$T = \begin{bmatrix} I & 0 \\ R & Q \end{bmatrix}$$

where I is the $n_a \times n_a$ identity matrix, R is an $n_t \times n_a$ matrix giving the probabilities of transient states transitioning to absorbing states, and Q is an $n_t \times n_t$ square matrix specifying the probabilities of transient states transitioning to transient states. Since absorbing states never transition to transient states by definition, the upper right corner contains a $n_a \times n_t$ zero matrix.

From any start state, a finite state absorbing MC always ends up in an absorbing state eventually, *i.e.* the limit $T^\infty \triangleq \lim_{n \to \infty} T^n$ exists and has the form

$$T^\infty = \begin{bmatrix} I & 0 \\ A & 0 \end{bmatrix}$$

where the $n_t \times n_a$ matrix A contains the so-called *absorption probabilities*. This matrix satisfies the following equation:

$$A = (I + Q + Q^2 + \dots)\,R$$

Intuitively, to transition from a transient state to an absorbing state, the MC can take an arbitrary number of steps between transient states before taking a single—and final—step into an absorbing state. The infinite sum $X \triangleq \sum_{n \geq 0} Q^n$ satisfies $X = I + QX$, and solving for X yields

$$X = (I - Q)^{-1} \quad \text{and} \quad A = (I - Q)^{-1} R. \tag{2}$$

(We refer the reader to [27] or Lemma A.3 in Appendix A for the proof that the inverse must exist.)

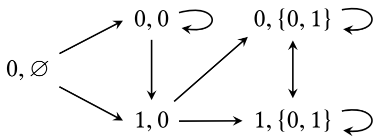
Before we apply this theory to the small-step semantics $\mathcal{S}[\![-]\!]$, it will be useful to introduce some MC-specific notation. Let T be an MC. We write $s \xrightarrow{T}_n s'$ if s can reach s' in precisely n steps, *i.e.* if $T^n_{s,s'} > 0$; and we write $s \xrightarrow{T} s'$ if s can reach s' in any number of steps, *i.e.* if $T^n_{s,s'} > 0$ for any $n \geq 0$. Two states are said to *communicate*, denoted $s \overset{T}{\longleftrightarrow} s'$, if $s \xrightarrow{T} s'$ and $s' \xrightarrow{T} s$. The relation $\overset{T}{\longleftrightarrow}$ is an equivalence relation, and its equivalence classes are called *communication classes*. A communication class is *absorbing* if it cannot reach any states outside the class. We sometimes write $\mathbf{Pr}[s \xrightarrow{T}_n s']$ to denote the probability $T^n_{s,s'}$. For the rest of the section, we fix a program p and abbreviate $\mathcal{B}[\![p]\!]$ as B and $\mathcal{S}[\![p]\!]$ as S. We also define *saturated states*, those where the accumulator has stabilized.

Definition 5.4. A state (a, b) of S is *saturated* if the accumulator b has reached its final value, *i.e.* if $(a, b) \xrightarrow{S} (a', b')$ implies $b' = b$.

Once we have reached a saturated state, the output of p^* is fully determined. The probability of ending up in a saturated state with accumulator b, starting from an initial state $(a, \varnothing)$, is

$$\lim_{n \to \infty} \sum_{a'} S^n_{(a,\varnothing),(a',b)}$$

and indeed this is the probability that p^* outputs b on input a by Proposition 5.2. Unfortunately, we cannot directly compute this limit since saturated states are not necessarily absorbing. To see this, consider the program $p^* = (f \leftarrow 0 \oplus_{1/2} f \leftarrow 1)^*$ over a single $\{0, 1\}$-valued field f. Then S has the form

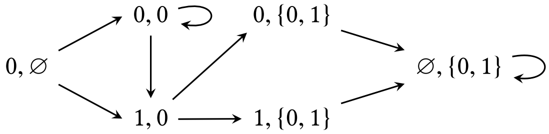


where all edges are implicitly labeled with $\frac{1}{2}$. At the nodes, 0 denotes the packet with f set to 0, and 1 denotes the packet with f set to 1; we omit states not reachable from $(0, \varnothing)$. The right-most states are saturated, but they communicate and are thus not absorbing.

To align saturated and absorbing states, we can perform a quotient of this Markov chain; roughly speaking, we will collapse the two communicating states above. We define the auxiliary matrix $U \in \mathbb{S}(2^{\mathsf{Pk}} \times 2^{\mathsf{Pk}})$ as

$$U_{(a,b),(a',b')} \triangleq \mathbf{1}[b' = b] \cdot \begin{cases} \mathbf{1}[a' = \varnothing] & \text{if } (a, b) \text{ is saturated} \\ \mathbf{1}[a' = a] & \text{else} \end{cases}$$

It sends a saturated state (a, b) to the canonical saturated state $(\varnothing, b)$—which is always absorbing—and it acts as the identity on all other states. In our example, the modified chain SU looks as follows:



Indeed, each state can reach an absorbing state and this Markov chain is absorbing as desired. To show that SU is an absorbing MC in general, we first observe:

Lemma 5.5. *S, U, and SU are* monotone *in the following sense: $(a, b) \xrightarrow{S} (a', b')$ implies $b \subseteq b'$ (and similarly for U and SU).*

Proof. The claim follows for S and U by definition, and for SU by composition. □

Now we can show that SU is indeed an absorbing MC.

Proposition 5.6. *Let $n \geq 1$.*

(1) $(SU)^n = S^n U$

(2) SU is an absorbing MC with absorbing states $\{(\varnothing, b) \mid b \subseteq \mathsf{Pk}\}$.

Arranging the states (a, b) in lexicographically ascending order according to $\subseteq$ and letting $n = |2^{\mathsf{Pk}}|$, it then follows from Proposition 5.6.2 that SU has the form

$$SU = \begin{bmatrix} I_n & 0 \\ R & Q \end{bmatrix}$$

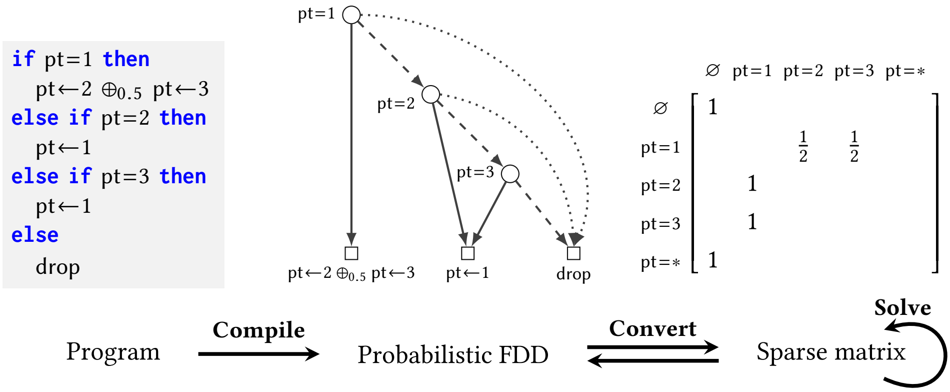
where for $a \neq \varnothing$, we have

$$(SU)_{(a,b),(a',b')} = \begin{bmatrix} R & Q \end{bmatrix}_{(a,b),(a',b')}$$

Moreover, SU converges and its limit is given by

$$(SU)^{\infty} \triangleq \begin{bmatrix} I_n & 0 \\ (I - Q)^{-1} R & 0 \end{bmatrix} = \lim_{n \to \infty} (SU)^n. \tag{3}$$

Putting together the pieces, we can use the modified Markov chain SU to compute the limit of S.



```
if pt=1 then
  pt←2 ⊕0.5 pt←3
else if pt=2 then
  pt←1
else if pt=3 then
  pt←1
else
  drop
```

$$
\begin{array}{c|ccccc}
 & \varnothing & \mathrm{pt{=}1} & \mathrm{pt{=}2} & \mathrm{pt{=}3} & \mathrm{pt{=}*} \\
\hline
\varnothing & 1 & & & & \\
\mathrm{pt{=}1} & & & \frac{1}{2} & \frac{1}{2} & \\
\mathrm{pt{=}2} & & 1 & & & \\
\mathrm{pt{=}3} & & 1 & & & \\
\mathrm{pt{=}*} & 1 & & & &
\end{array}
$$

Fig. 5. Implementation using FDDs and a sparse linear algebra solver.

THEOREM 5.7 (CLOSED FORM). *Let $a, b, b' \subseteq \mathsf{Pk}$. Then*

$$\lim_{n\to\infty} \sum_{a'} S^n_{(a,b),(a',b')} = (SU)^{\infty}_{(a,b),(\varnothing,b')} \tag{4}$$

or, using matrix notation,

$$\lim_{n\to\infty} \sum_{a'} S^n_{(-,-),(a',-)} = \begin{bmatrix} I_n \\ (I-Q)^{-1}R \end{bmatrix} \in [0,1]^{(2^{\mathsf{Pk}}\times 2^{\mathsf{Pk}})\times 2^{\mathsf{Pk}}}. \tag{5}$$

In particular, the limit in (4) *exists and can be effectively computed in closed-form.*

As an application, we can decide program equivalence.

COROLLARY 5.8. *For programs p and q, it is decidable whether $p \equiv q$.*

6 IMPLEMENTATION

We have implemented ProbNetKAT as an embedded DSL in OCaml in roughly 5000 lines of code. The frontend provides functions for defining and manipulating ProbNetKAT programs, and for translating network topologies encoded using GraphViz into ProbNetKAT. The backend is a compiler that takes ProbNetKAT ASTs as input and generates stochastic matrices as output. The resulting matrices can then be analyzed using standard linear algebra and other statistical tools.

6.1 Compilation

As the careful reader may have noticed, a direct implementation of the semantics presented in §4 and §5 would not scale as it involve constructing matrices over an intractably large state space—the powerset 2^{Pk} of the universe of possible packets! To obtain a practical analysis tool, we restrict the state space to single packets and use symbolic data structures and several optimizations.

The compilation process, which is illustrated in Figure 5, proceeds as follows. First, we translate atomic programs directly to Forwarding Decision Diagrams (FDDs), a symbolic data structure based on Binary Decision Diagrams (BDDs) that encodes sparse matrices compactly. Second, we compile composite programs by first translating the constituent programs to FDDs and then combining those into a unified FDD using standard BDD traversal algorithms. Third, we compile loops by (i) converting the FDD representing the body of the loop to a sparse matrix representation, (ii) invoking an optimized sparse linear solver to solve the system from §5.1, and (iii) converting the resulting matrix back into an FDD.

6.1.1 Restriction to Singletons. Although our semantics was developed using packet sets, our implementation sacrifices the ability to model multicast and works with singleton packets and the empty set only. Syntactically, we remove the operators for union (&) and iteration (*) from the language and expose the more restrictive `if-then-else` and `while-do` primitives instead. This ensures that no proper packet sets are ever generated, thus allowing us to work over an exponentially smaller state space. In our experience, this is rarely a limitation in practice because multicast is somewhat less common, and can be analyzed in terms of multiple unicast programs if necessary.

6.1.2 Probabilistic FDDs. Binary Decision Diagrams (BDDs) [1] and variants thereof [15] have long been used in verification and model checking to represent large state spaces compactly. We use a variant called Forwarding Decision Diagrams (FDDs) that was developed specifically for the networking domain [44] and extend it with distributions to encode probabilistic (rather than deterministic) packet-processing functions.

A probabilistic FDD is a rooted directed acyclic graph that can be understood as a control-flow graph. Interior nodes test packet fields and have outgoing true- and false- branches (which we visualize by solid lines and dashed lines, *cf.* Figure 5). Leaf nodes contain distributions over *actions*, where an action is either a set of modifications or the drop primitive. To interpret an FDD, we start at the root node with an initial packet and traverse the graph as dictated by the tests until a leaf node is reached. Then, we apply each action in the leaf node to the packet. Thus, an FDD represents a function of type $\mathsf{Pk} \to \mathcal{D}(\mathsf{Pk} + \varnothing)$, or equivalently, a stochastic matrix over the state space $\mathsf{Pk} + \varnothing$ (where the $\varnothing$-row puts all mass on $\varnothing$ by convention).

Like BDDs, FDDs respect a total order on tests and contain no isomorphic subgraphs and no redundant tests, which allows them to represent sparse matrices compactly in practice.

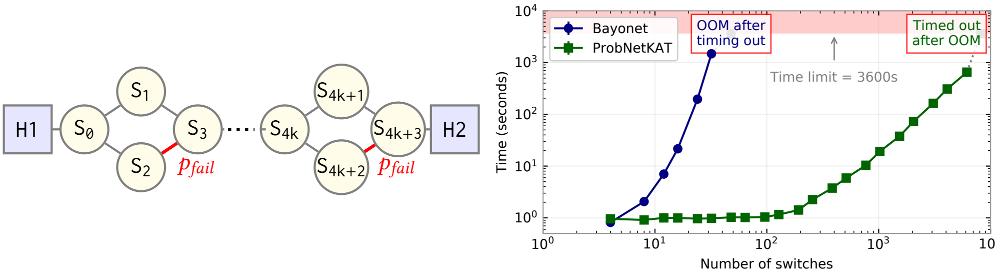
6.1.3 Dynamic Domain Reduction. As Figure 5 shows, we do not have to represent the state space $\mathsf{Pk} + \varnothing$ explicitly even when converting into sparse matrix form. In the example, the state space is represented by *symbolic packets* $\mathsf{pt} = 1$, $\mathsf{pt} = 2$, $\mathsf{pt} = 3$, and $\mathsf{pt} = *$, each representing an *equivalence class* of packets with the same behavior. For example, $\mathsf{pt} = 1$ can represent all packets π satisfying $\pi.\mathsf{pt} = 1$, because the program treats all such packets in the same way. The packet $\mathsf{pt} = *$ represent the set $\{\pi \mid \pi.\mathsf{pt} \notin \{1, 2, 3\}\}$. The symbol $*$ can be thought of as a wildcard that ranges over all values not explicitly represented by other symbolic packets.

The symbolic packets used to encode the packet domain are chosen dynamically when converting an FDD to a matrix by traversing the FDD and determining for each field which set of values appears with it, either in a test or a modification. Since FDDs never contain redundant tests or modifications, these sets are typically of manageable size.

6.2 Evaluation

We conducted several experiments to evaluate the scalability of our implementation and the effect of optimizations, using a synthetic benchmark from the literature [17] and a real-world data center with a sophisticated routing scheme. All experiments were performed on 16-core, 2.6 GHz Intel Xeon E5-2650 machines with 64 GB of memory.

6.2.1 Comparison with Bayonet. Bayonet [17] is a state of the art tool for expressing and reasoning about probabilistic networks. While ProbNetKAT is based on a custom backend tailored to the domain, Bayonet programs are translated to a general-purpose probabilistic programming language (PPL). To evaluate how these approaches compare in terms of performance, we reproduced an experiment from the Bayonet paper [17] that analyzes the reliability of a simple routing scheme in the presence of link failures.



(a) (b)

Fig. 6. Bayonet comparison: (a) topology and (b) scalability results.

The experiment considers hosts H1 and H2 connected by a family of topologies[2] indexed by k. For $k = 1$, the network consists of a quartet of switches organized as a diamond with a single link that fails with probability $p_{fail} = 1/1000$. For $k > 1$, the network consists of k diamonds linked together into a chain as shown in Figure 6(a). Within each diamond, switch S_0 uses probabilistic routing, forwarding packets with equal probability to switches S_1 and S_2, which in turn forward the packet along to switch S_3. However, S_2 drops the packet if the link to S_3 fails. We consider a packet that originates at host H1 and analyze the probability that it gets delivered correctly to host H2.

Figure 6 compares the running times of both tools when queried for the probability of packet delivery. Note that both axes are log-scaled. We see that Bayonet scales to 32 switches in about 25 minutes, before hitting the 1h time limit and 64 GB memory limit at 48 switches. ProbNetKAT answers the same query for 256 switches in about 2 seconds and scales to over 6000 switches in under 11 minutes, before running out of memory shortly thereafter.

Discussion. The experiment shows that ProbNetKAT's domain-specific backend and specialized data structures outperform an approach based on general-purpose tools by orders of magnitude. It is important to also note the drawbacks of our approach however. Because Bayonet is based on a general-purpose probabilistic programming language, it is more expressive than ProbNetKAT and can model queues and stateful functionality. It also comes with build-in support for Bayesian reasoning. Section 8 discusses the differences between the tools in detail.

6.2.2 Real-world Data Center. To evaluate how ProbNetKAT scales on more complex examples, we modeled a sophisticated resilient routing scheme from the literature called F10 [33] on a commonly used data center topology called FatTree [2] (see Figure 10). A FatTree is defined in terms of a parameter k that controls the size of the network: a k-ary FatTree connects $\frac{1}{4}k^3$ servers using $\frac{5}{4}k^2$ switches. The next section describes our model and the analyses we performed in great detail; here we discuss the performance of our tool as we increase k, and the effect of optimizations. Figures 7 and 8 report the running time of the analysis as a function of the number of switches, using log-scaled and linearly-scaled time axes, respectively. The analysis is more expensive when compared with the previous experiment, but still scales to a 10-ary FatTree with 125 switches in about 5 seconds and to a 16-ary FatTree connecting 1024 machines using 320 switches in about 25 minutes. The increase in running time is due to a denser topology and a more sophisticated routing scheme using extra fields, resulting in a larger state space.

---

[2] Note that we do not exploit the regularity of these topologies to speed up analysis in our benchmark.

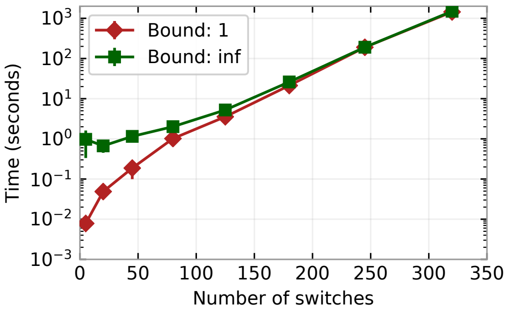


Fig. 7. Linear algebra is not the bottleneck.

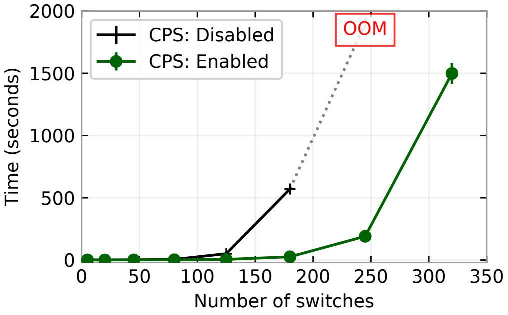


Fig. 8. Scalability gain with CPS-style compilation.

Bottlenecks. Our FatTree experiment uses a single top-level while loop that repeats routing steps until the destination is reached, as explained in §2. To evaluate the cost of compiling iteration, we replaced the loop with its body (Figure 7), modeling only a single hop. As the graph shows, the speed gap between single-hop compilation vs. full compilation quickly closes for larger topologies, meaning that the cost of computing the fixed point becomes negligible. We profit from the highly optimized UMFPack routine that takes advantage of all 16 cores on our test machines. This suggests that performance could be further improved by accelerating our FDD algorithms.

Indeed the bottleneck for this experiment lies in our representation of distributions, as it scales exponentially in k for this particular model. The problem lies in a sequence

$$(\mathsf{up}_1{\leftarrow}0 \oplus_p \mathsf{up}_1{\leftarrow}1) \,;\, \cdots \,;\, (\mathsf{up}_\mathsf{k}{\leftarrow}0 \oplus_p \mathsf{up}_\mathsf{k}{\leftarrow}1) \,;\, p$$

of independent random assignment to k binary variables modeling which switch ports are up, followed by a program p that breaks this independence, but maintains *conditional* independence. Using FDDs, we must resort to a naive exponential encoding of the joint distribution. An interesting question for future work is whether Bayesian networks could be employed to represent conditionally independent distributions efficiently.

An important property of our tool is that virtually the entire analysis time is spent on compilation: once we have synthesized an FDD, it can typically be queried in milliseconds using simple traversal algorithms. Figures 8 and 7 thus report only the compilation time. This means that ProbNetKAT performs favorably for multiple queries of the same model, as the compilation time can be amortized.

CPS-style translation. We observe empirically that a CPS-style compilation scheme can often improve scalability dramatically (Figure 8). The idea is simple: instead of compiling programs bottom-up, we compile them left-to-right. In this scheme, when we compile a subprogram p, we already have an FDD t in hand that captures the result of the program from its beginning up to p. Intuitively, this allows us to partially evaluate and thereby simplify p before converting to an FDD. If t happens to be drop, we can avoid the compilation of p altogether as drop ; $p \equiv$ drop. For FatTrees, this reduces compilation time from 10 minutes to 25 seconds for 180 switches.

7 CASE STUDY: RESILIENT ROUTING

In this section, we go beyond benchmarks and illustrate the utility of our tool for solving a real-world networking problem. Specifically, we develop an extended case study involving data center topologies and resilient routing schemes. A recent measurement study showed that failures in data centers [19] occur frequently, and have a major impact on application-level performance, motivating a line of research exploring the design of fault-tolerant data center fabrics in which

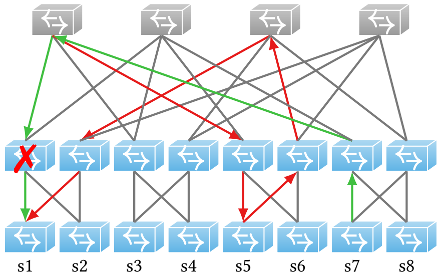


Fig. 9. A FatTree topology with 20 switches.

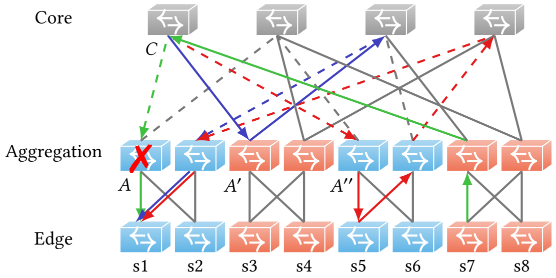


Fig. 10. An AB FatTree topology with 20 switches.

the topology and routing scheme are co-designed to simultaneously achieve high throughput, low latency, and resilience to failures.

7.1 Topology and routing

Data center topologies typically organize the network fabric into multiple levels of switches.

FatTree. A FatTree [2] is perhaps the most common example of a multi-level, multi-rooted tree topology. Figure 9 shows a 3-level FatTree topology with 20 switches. The bottom level, *edge*, consists of top-of-rack (ToR) switches; each ToR switch connects all the hosts within a rack (not shown in the figure). These switches act as ingress and egress for intra-data center traffic. The other two levels, *aggregation* and *core*, redundantly connect the switches from the edge layer.

The redundant structure of a FatTree makes it possible to implement fault-tolerant routing schemes that detect and automatically route around failed links. For instance, consider routing from a source to a destination along shortest paths—*e.g.*, the green links in the figure depict one possible path from ($s7$) to ($s1$). On the way from the ToR to the core switch, there are multiple paths that could be used to carry the traffic. Hence, if one of the links goes down, the switches can route around the failure by simply choosing a different path. Equal-cost multi-path (ECMP) routing is widely used—it automatically chooses among the available paths while avoiding longer paths that might increase latency.

However, after reaching a core switch, there is a *unique* shortest path down to the destination. Hence, ECMP no longer provides any resilience if a switch fails in the aggregation layer (*cf.* the red cross in Figure 9). A more sophisticated scheme could take a longer (5-hop) detour going all the way to another edge switch, as shown by the red lines in the figure. Unfortunately, such detours can lead to increased latency and congestion.

AB FatTree. The long detours on the downward paths in FatTrees are dictated by the symmetric wiring of aggregation and core switches. AB FatTrees [33] alleviate this by using two types of subtrees, differing in their wiring to higher levels. Figure 10 shows how to rewire a FatTree to make it an AB FatTree. The two types of subtrees are as follows:

i) Type A: switches depicted in blue and wired to core using dashed lines.
ii) Type B: switches depicted in red and wired to core using solid lines.

Type A subtrees are wired in a way similar to FatTree, but Type B subtrees differ in their connections to core switches. In our diagrams, each aggregation switch in a Type A subtree is wired to adjacent core switches, while each aggregation switch in a Type B subtree is wired to core switches in a staggered manner. (See the original paper for the general construction [33].)

```
// F10 without rerouting
f10_0 :=
  // ECMP, but don't use inport
  fwd_on_random_shortest_path

// F10 with 3-hop rerouting
f10_3 :=
  f10_0;
  if at_down_port then 3hop_rr
```

```
// F10 with 3-hop & 5-hop rerouting
f10_3_5 :=
  if at_ingress then (default <- 1);
  if default = 1 then (
    f10_3;
    if at_down_port then (5hop_rr; default <- 0)
  ) else (
    default <- 1; // back to default forwarding
    fwd_downward_uniformly_at_random
  )
```

Fig. 11. ProbNetKAT implementation of F10 in three refinement steps.

This slight change in wiring enables much shorter detours around failures in the downward direction. Consider again routing from source ($s7$) to destination ($s1$). As before, we have multiple options going upwards when following shortest paths (*e.g.*, the one depicted in green), as well as a unique downward path. But unlike FatTree, if the aggregation switch on the downward path fails, there is a short detour, as shown in blue. This path exists because the core switch, which needs to reroute traffic, is connected to aggregation switches of both types of subtrees. More generally, aggregation switches of the same type as the failed switch provide a 5-hop detour; but aggregation switches of the opposite type provide an efficient 3-hop detour.

7.2 ProbNetKAT implementation

We encode routing schemes for FatTrees in ProbNetKAT and analyze their behavior under several different failure models.

Routing. F10 [33] provides a routing algorithm that combines the three (re)routing strategies we just discussed (ECMP, 3-hop rerouting, 5-hop rerouting) into a single scheme. We implemented it in three steps, as shown in the psueocode in Figure 11. The first scheme, $\mathrm{F10}_0$, implements an ECMP-like approach:[3] it randomly selects a port along one of the shortest paths to the destination.[4]

Next, we improve the resilience of $\mathrm{F10}_0$ by augmenting it with 3-hop rerouting if the next hop switch A along the downward shortest path from a core switch C fails. To illustrate, consider the blue path in Figure 10. We find a port on C that connects to an aggregation switch A' with the opposite type of the failed aggregation switch, A, and forward the packet to A'. If there are multiple such ports which have not failed, we choose one uniformly at random. Default routing continues at A', and ECMP will know not to send the packet back to C. $\mathrm{F10}_3$ implements this refinement.

Note that if $\mathrm{F10}_3$ is still unable to find a port on C whose adjacent link is up, then all links connecting to switches of the opposite type must have failed. In this case, we attempt 5-hop rerouting via an aggregation switch A'' of the same type as A. To illustrate, consider the red path in Figure 10. We begin by forwarding the packet to A''. To let A'' know that it should not send the packet back to core layer, we unset a flag `default` to indicate that A'' should send the packet further downward. Default routing continues after A''. $\mathrm{F10}_{3,5}$ implements this final refinement.

Network and Failure Models. Our network model works much like the one from §2. However, to simplify the model, we analyze forwarding to a single ToR switch ($s1$) and elide the final hop to the host connected to this switch.

$$\mathrm{M}(p, t) \triangleq \mathit{in} \,;\, \mathrm{do}\ (p \,;\, t)\ \mathrm{while}\ (\neg \mathrm{sw}{=}1)$$

---

[3] ECMP implementations are usually based on hashing, which approximates random forwarding provided there is sufficient entropy in the header fields used to select an outgoing port.

[4] We exclude the ingress port from this set to eliminate possible forwarding loops when routing around failures.

k	$\widehat{\mathrm{M}}(\mathrm{F10}_0, t, f_k)$ $\equiv$ *teleport*	$\widehat{\mathrm{M}}(\mathrm{F10}_3, t, f_k)$ $\equiv$ *teleport*	$\widehat{\mathrm{M}}(\mathrm{F10}_{3,5}, t, f_k)$ $\equiv$ *teleport*
0	✓	✓	✓
1	✗	✓	✓
2	✗	✓	✓
3	✗	✗	✓
4	✗	✗	✗
∞	✗	✗	✗

Table 1. Evaluating k-resilience of F10.

k	compare $(\mathrm{F10}_0, \mathrm{F10}_3)$	compare $(\mathrm{F10}_3, \mathrm{F10}_{3,5})$	compare $(\mathrm{F10}_{3,5}, \textit{teleport})$
0	$\equiv$	$\equiv$	$\equiv$
1	$<$	$\equiv$	$\equiv$
2	$<$	$\equiv$	$\equiv$
3	$<$	$<$	$\equiv$
4	$<$	$<$	$<$
∞	$<$	$<$	$<$

Table 2. Comparing schemes under k failures.

The ingress predicate *in* is a disjunction of switch-and-port tests over all ingress locations. This first model is embedded into a refined model $\widehat{\mathrm{M}}(p, t, f)$ that integrates the failure model and declares all necessary local variables that track the health of individual ports:

$$\widehat{\mathrm{M}}(p, t, f) \triangleq \mathsf{var}\ \mathsf{up}_1 {\leftarrow} 1\ \mathsf{in}\ \ldots\ \mathsf{var}\ \mathsf{up}_\mathsf{d} {\leftarrow} 1\ \mathsf{in}\ (\mathrm{M}((f\,;p), t))$$

Here d denotes the maximum degree of all nodes in the FatTree and AB FatTree topologies from Figures 9 and 10, which we encode as programs *fattree* and *abfattree* much like in §2.

We define a family of failure models f_k^{pr}, where $k \in \mathbb{N} \cup \{\infty\}$ bounds the maximum number of failures that may occur, and links fail otherwise independently with probability pr. We omit pr when clear from context. To focus on the scenarios occurring on downward paths, we model failures only for links connecting the aggregation and core layer.

7.3 Checking Invariants

We can gain confidence in our implementation of F10 by verifying that it maintains certain key invariants. As an example, recall our implementation of $\mathrm{F10}_{3,5}$: 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 resets *default* back to 1. By design, a packet should never be delivered to the destination with *default* set to 0. To verify this property, we check the following equivalence:

$$\forall t, k: \ \widehat{\mathrm{M}}(\mathrm{F10}_{3,5}, t, f_k) \ \equiv \ \widehat{\mathrm{M}}(\mathrm{F10}_{3,5}, t, f_k)\ ;\textit{default}{=}1$$

We executed the check using our implementation for $k \in \{0, 1, 2, 3, 4, \infty\}$ and $t \in \{\textit{fattree}, \textit{abfattree}\}$. 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` bit to 1 at the ingress (*cf.* Figure 11, right column, line 3). We discovered this bug using our implementation.

7.4 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 not improve resilience. To verify this, we can check the following equivalence:

$$\forall k: \ \widehat{\mathrm{M}}(\mathrm{F10}_0, \textit{fattree}, f_k) \ \equiv \ \widehat{\mathrm{M}}(\mathrm{F10}_3, \textit{fattree}, f_k)$$

We have used our implementation to check that this equivalence indeed holds for $k \in \{0, 1, 2, 3, 4, \infty\}$.

7.5 Refinement

Recall that we implemented F10 in three stages: (i) we started with a basic routing scheme $\mathrm{F10}_0$ based on ECMP that provides resilience on the upward path but no rerouting capabilities on the downward path, (ii) we augmented this scheme by adding 3-hop rerouting to obtain $\mathrm{F10}_3$ which can route around certain failures in the aggregation layer, and (ii) we finally added 5-hop rerouting to address failure cases that 3-hop rerouting cannot handle, obtaining $\mathrm{F10}_{3,5}$. Hence, we would expect the probability of packet delivery to increase with each refinement of our routing scheme. Additionally, we expect all schemes to deliver packets and drop packets with some probability under the unbounded failure model. Summarizing:

$$\mathrm{drop} \;<\; \widehat{\mathrm{M}}(\mathrm{F10}_0, t, f_\infty) \;<\; \widehat{\mathrm{M}}(\mathrm{F10}_3, t, f_\infty) \;<\; \widehat{\mathrm{M}}(\mathrm{F10}_{3,5}, t, f_\infty) \;<\; \mathit{teleport}$$

where $t = \mathit{abfattree}$ and $\mathit{teleport} \triangleq \mathrm{sw}{\leftarrow}1$. To our surprise, we were not able to verify this property initially, as our implementation indicated that the ordering $\widehat{\mathrm{M}}(\mathrm{F10}_3, t, f_\infty) \;<\; \widehat{\mathrm{M}}(\mathrm{F10}_{3,5}, t, f_\infty)$ was violated. We found that $\mathrm{F10}_3$ performed better than $\mathrm{F10}_{3,5}$ for packets π with $\pi.\mathit{default} = 0$. This was due to a bug: we were missing the first line in our implementation of $\mathrm{F10}_{3,5}$ (*cf.*, Figure 11) that initializes the *default* bit to 1 at the ingress, causing packets to be dropped. After fixing the bug, we were able to confirm the expected ordering.

7.6 k-resilience

We saw that there exists a strict ordering in terms of resilience for $\mathrm{F10}_0$, $\mathrm{F10}_3$ and $\mathrm{F10}_{3,5}$ when an unbounded number of failures can happen. Another interesting way of quantifying resilience is to count the minimum number of failures at which a scheme fails to guarantee 100% delivery. Using ProbNetKAT, we can compute this metric by increasing the k parameter in f_k and checking equivalence with teleportation. Table 1 shows the results based on our decision procedure for the AB FatTree topology from Figure 10.

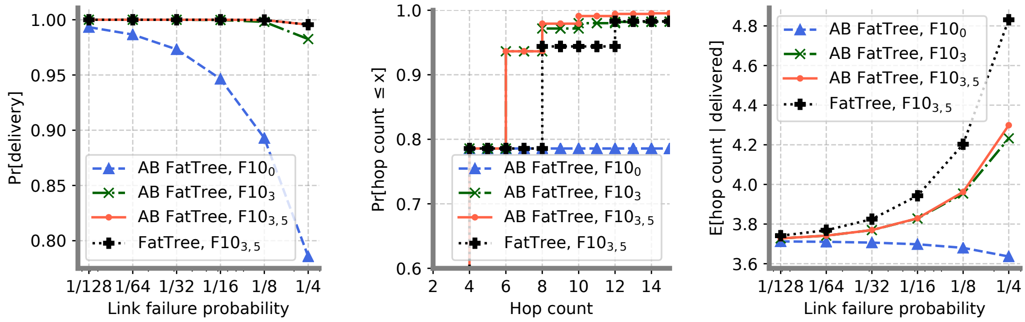
The naive scheme, $\mathrm{F10}_0$, which does not perform any rerouting, drops packets when a failure occurs on the downward path. Thus, it is 0-resilient. In the example topology, 3-hop rerouting has two possible ways to reroute for the given failure. Even if only one of the Type B subtrees is reachable, $\mathrm{F10}_3$ can still forward traffic. However, if both Type B subtrees are unreachable, then $\mathrm{F10}_3$ will not be able to reroute traffic. Thus, $\mathrm{F10}_3$ is 2-resilient. Similarly, $\mathrm{F10}_{3,5}$ can route as long as any aggregation switch is reachable from the core switch. For $\mathrm{F10}_{3,5}$ to fail the core switch would need to be disconnected from all four aggregation switches. Hence it is 3-resilient. In cases where schemes are not equivalent to *teleport*, we can characterize the relative robustness by computing the ordering, as shown in Table 2.

7.7 Resilience under increasing failure rate

We can also do more quantitative analyses, such as evaluating the effect of link failure on the packet delivery probability. Figure 12(a) shows this analysis in a failure model in which an unbounded number of failures can occur simultaneously. We find that $\mathrm{F10}_0$'s delivery probability dips significantly as the failure probability increases because $\mathrm{F10}_0$ is not resilient to failures. In contrast, both $\mathrm{F10}_3$ and $\mathrm{F10}_{3,5}$ continue to ensure high probability of delivery by rerouting around failures.

7.8 Cost of resilience

By augmenting naive routing schemes with rerouting mechanisms, we achieve a higher degree of resilience. But this benefit comes at a cost: taking detours increases latency (*i.e.*, hop count). We can quantify this increase in latency by augmenting our model with a counter that is incremented at each hop. Figure 12(b) shows the CDF of latency as the fraction of traffic delivered within a given



(a) (b) (c)

Fig. 12. Case study results ($k = \infty$): (a) Probability of delivery vs. link-failure probability; (b) Increased latency due to resilience ($pr = \frac{1}{4}$); (c) Expected hop-count conditioned on delivery.

hop count. On AB FatTree, $F10_0$ delivers ≈80% of the traffic in ≤ 4 hops, as the maximum length of a shortest path from any edge switch to $s1$ is 4 and $F10_0$ does not attempt to recover from failures. $F10_3$ and $F10_{3,5}$ deliver the same amount of traffic with hop count ≤ 4, but with 2 additional hops, they deliver significantly more traffic by using 3-hop paths to route around failures. With additional hops, the throughput of $F10_{3,5}$ increases further using 5-hop paths. $F10_3$ also delivers more traffic with 8 hops—these are the cases when $F10_3$ performs 3-hop rerouting twice for a single packet as it encountered failure twice. Similarly, we see small throughput increases for higher hop counts. On FatTree, $F10_{3,5}$ improves resilience, but the impact on latency is significantly higher as the topology does not support 3-hop rerouting.

7.9 Expected latency

Figure 12(c) shows the expected hop-count of paths taken by packets conditioned on their delivery. Both $F10_3$ and $F10_{3,5}$ deliver packets with high probability even at high failure probabilities, as we saw in Figure 12(a). However, a higher probability of link-failure implies that it becomes more likely for these schemes to invoke rerouting, which increases hop count. Hence, we see the increase in expected hop-count as failure probability increases. $F10_{3,5}$ uses 5-hop rerouting to achieve more resilience compared to $F10_3$, which performs only 3-hop rerouting, and this leads to slightly higher expected hop-count for $F10_{3,5}$. The increase is more significant for FatTree in contrast to AB FatTree, because FatTree only supports 5-hop rerouting.

As the failure probability increases, the probability of delivery for packets that are routed via the core layer decreases significantly for $F10_0$ (recall Figure 12(a)). Thus, the distribution of delivered packets shifts towards those with direct 2-hop path via an aggregation switch (such as packets from $s2$ to $s1$), and hence the expected hop-count decreases slightly.

7.10 Discussion

As this case study 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. Moreover, our tool is able to handle real-world topologies and routing schemes. These capabilities represent a significant advance over current network verification tools, which are largely based on deterministic packet-forwarding models [14, 26, 28, 34].

8 RELATED WORK

The most closely related system to ProbNetKAT is Bayonet [17]. In contrast to the domain-specific approach developed in this paper, Bayonet is based on a general-purpose probabilistic programming language and inference tool [18]. Such an approach, which reuses existing abstractions, is naturally appealing. In addition, Bayonet is more expressive than ProbNetKAT: it supports asynchronous packet scheduling, stateful transformations, and probabilistic inference, making it possible to accurately model richer phenomena, such as congestion due to packet-level interactions in queues. However, the extra generality comes at a cost. Bayonet currently requires programmers to supply an upper bound on loops as the implementation is not guaranteed to find a fixpoint. As discussed in §6, ProbNetKAT scales two orders of magnitude better than Bayonet on its own benchmark program. Finally, it is not clear how one could ensure that Bayonet faithfully models the fine-grained queuing behavior of real-world switches: writing the scheduler could be challenging, and one might also need to model host-level congestion control protocols. Current Bayonet programs use simple deterministic or uniform schedulers and model only a handful of packets at a time [16].

A key ingredient that underpins the results in this paper is the idea of representing the semantics of iteration using absorbing Markov chains, and exploiting their properties to directly compute limiting distributions on them. Of course, Markov chains have been used to represent and to analyze probabilistic programs in previous work. An early example of using Markov chains for modeling probabilistic programs is the seminal paper by Sharir, Pnueli, and Hart [42]. They present a general method for proving properties of probabilistic programs. In their work, a probabilistic program is modeled by a Markov chain and an assertion on the output distribution is extended to an invariant assertion on all intermediate distributions (providing a probabilistic generalization of Floyd's inductive assertion method). Their approach can assign semantics to infinite Markov chains for infinite processes, using stationary distributions of absorbing Markov chains in a similar way to the one used in this paper. Note however that the state space used in this and other work is not like ProbNetKAT's current and accumulator sets ($2^{Pk} \times 2^{Pk}$), but is instead is the Cartesian product of variable assignments and program location. In this sense, the absorbing states occur for program termination, rather than for accumulation as in ProbNetKAT. Although packet modification is clearly related to variable assignment, accumulation does not clearly relate to program location.

Readers familiar with prior work on probabilistic automata might wonder if we could directly apply known results on (un)decidability of probabilistic rational languages. This is not the case—probabilistic automata accept distributions over words, while ProbNetKAT programs encode distributions over languages. Similarly, probabilistic programming languages, which have gained popularity in the last decade motivated by applications in machine learning, focus largely on Bayesian inference. They typically come equipped with a primitive for probabilistic conditioning and often have a semantics based on sampling. Working with ProbNetKAT has a substantially different style, in that the focus is on on specification and verification rather than inference.

Di Pierro, Hankin, and Wiklicky have used probabilistic abstract interpretation to statically analyze probabilistic λ-calculus [10]. Their work was later extended to a language called *pWhile*, using a store plus program location state-space similar to [42]. *pWhile* is a basic imperative language augmented with random choice between program blocks with a rational probability, and limited to a finite of number of finitely-ranged variables (in our case, packet fields). In contrast to our work, they do not deal with infinite limiting behavior beyond stepwise iteration, and do not guarantee convergence. Probabilistic abstract interpretation is a new but growing field of research [47].

Olejnik, Wicklicky, and Cheraghchi provided a probabilistic compiler *pwc* for a variation of *pWhile* [37], implemented in OCaml, together with a testing framework. The *pwc* compiler has optimizations involving, for instance, the Kronecker product to help control matrix size, and a Julia backend. These optimizations could be applied to the generation of $\mathcal{S}[\![p]\!]$ from $\mathcal{B}[\![p]\!]$.

There is also significant prior work on finding explicit distributions in the context of probabilistic programming languages, see e.g. a survey on the state of the art on probabilistic inference [22]. They show how stationary distributions on Markov chains can be used for the semantics of infinite probabilistic processes, and how they converge under certain conditions. Similar to our approach, they use absorbing strongly-connected-components to represent termination.

Markov chains are used in many probabilistic model checkers, of which PRISM [32] is a prime example. PRISM supports analysis of discrete-time Markov chains, continuous-time Markov chains, and Markov decision processes. The models are checked against temporal logic specifications like PCTL and CSL. PRISM provides three model checking engines: a symbolic one with (multi-terminal) binary decision diagrams, a sparse matrix one, and a hybrid approach. The use of PRISM to analyse ProbNetKAT programs is an interesting research avenue and we intend to explore it in the future.

9 CONCLUSION

This paper describes how to compute the semantics of history-free ProbNetKAT programs in closed form, enabling automated analysis of probabilistic properties for ProbNetKAT programs. The key technical challenge is overcome by modeling the iteration operator as an absorbing Markov chain, which makes it possible to compute a closed-form solution for its semantics. Natural directions for future work include investigating full ProbNetKAT (Appendix B describes some challenges) and further optimizing the implementation, in particular by using Bayesian networks to represent joint distributions compactly. Moreover, we believe that exploring additional applications of probabilistic programing and reasoning is likely to be promising as networks increasingly incorporate various forms of randomization [31, 43]. For example, one could imagine using ProbNetKAT to verify that a multi-path routing scheme effectively spreads traffic over all available paths—in particular, detecting load *im*balance, which can arise with hash-based schemes such as ECMP [11]. In the same vein, it would be interesting to analyze the sensitivity of a routing scheme to small changes in input traffic. Another potential application is to verify the anonymity provided by “onion routing” in ToR. Here we might analyze the distribution of packets seen at relay nodes and show that an adversary is unable to use traffic analysis to infer the sender and receiver. Studying network neutrality in terms of notions of uniformity and conditional independence would also be interesting. For example, an ISP might be allowed to discriminate based on source and destination addresses, but not other header fields such as TCP ports [50]. Finally, it would be interesting to analyze the accuracy of network monitoring schemes based on sampling—in particular, the interplay between routing, monitoring, and sampling rate [41].

REFERENCES

[1] S. B. Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* 27, 6 (June 1978), 509–516. https://doi.org/10.1109/TC.1978.1675141

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 63–74.

[3] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. 2016. Regular programming for quantitative properties of data streams. In *ESOP 2016*. 15–40.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*. 113–126.

[5] Manav Bhatia, Mach Chen, Sami Boutros, Marc Binderberger, and Jeffrey Haas. 2014. Bidirectional Forwarding Detection (BFD) on Link Aggregation Group (LAG) Interfaces. RFC 7130. (Feb. 2014). https://doi.org/10.17487/RFC7130

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR* 44, 3 (July 2014), 87–95.

[7] Martin Casado, Nate Foster, and Arjun Guha. 2014. Abstractions for Software-Defined Networks. *CACM* 57, 10 (Oct. 2014), 86–95.

[8] Timothy A. Davis. 2004. Algorithm 832: UMFPACK V4.3—an Unsymmetric-pattern Multifrontal Method. *ACM Trans. Math. Softw.* 30, 2 (June 2004), 196–199. https://doi.org/10.1145/992200.992206

[9] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A storm is Coming: A Modern Probabilistic Model Checker *(LNCS)*, Vol. abs/1702.04311. arXiv:1702.04311 http://arxiv.org/abs/1702.04311

[10] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. 2005. Probabilistic λ-calculus and quantitative program analysis. *Journal of Logic and Computation* 15, 2 (2005), 159–179. https://doi.org/10.1093/logcom/exi008

[11] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. 2013. On the impact of packet spraying in data center networks. In *IEEE INFOCOM*. 2130–2138.

[12] Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of Weighted Automata*. Springer.

[13] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP*. 282–309. https://doi.org/10.1007/978-3-662-49498-1_12

[14] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *POPL*. ACM, 343–355.

[15] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. 1997. Multi-Terminal Binary Decision Diagrams: An Efficient DataStructure for Matrix Representation. *Form. Methods Syst. Des.* 10, 2-3 (April 1997), 149–169. https://doi.org/10.1023/A:1008647823331

[16] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin T. Vechev. 2018. Bayonet: Probabilistic Computer Network Analysis. (June 2018). Available at https://github.com/eth-sri/bayonet/.

[17] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin T. Vechev. 2018. Bayonet: probabilistic inference for networks. In *ACM SIGPLAN PLDI*. 586–602.

[18] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. 62–83.

[19] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *ACM SIGCOMM*. 350–361.

[20] Hugo Gimbert and Youssouf Oualhadj. 2010. Probabilistic Automata on Finite Words: Decidable and Undecidable Problems. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*. 527–538. https://doi.org/10.1007/978-3-642-14162-1_44

[21] Michele Giry. 1982. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*. Springer, 68–85. https://doi.org/10.1007/BFb0092872

[22] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*. ACM, 167–181. https://doi.org/10.1145/2593882.2593900

[23] Timothy V Griffiths. 1968. The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines. *Journal of the ACM* 15, 3 (1968), 409–413.

[24] Tero Harju and Juhani Karhumäki. 1991. The equivalence problem of multitape finite automata. *Theoretical Computer Science* 78, 2 (1991), 347–355.

[25] David M. Kahn. 2017. Undecidable Problems for Probabilistic Network Programming. In *MFCS 2017*. http://hdl.handle.net/1813/51765

[26] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI 2012*. 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[27] John G Kemeny, James Laurie Snell, et al. 1960. *Finite markov chains*. Vol. 356. van Nostrand Princeton, NJ.

[28] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and Brighten Godfrey. 2012. Veriflow: Verifying Network-Wide Invariants in Real Time. In *ACM SIGCOMM*. 467–472.

[29] Dexter Kozen. 1981. Semantics of probabilistic programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350. https://doi.org/10.1016/0022-0000(81)90036-2

[30] Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (May 1997), 427–443. https://doi.org/10.1145/256167.256195

[31] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. 2018. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *USENIX NSDI*.

[32] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (LNCS)*, G. Gopalakrishnan and S. Qadeer (Eds.), Vol. 6806. Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

[33] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E Anderson. 2013. F10: A Fault-Tolerant Engineered Network. In *USENIX NSDI*. 399–412.

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

[35] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR* 38, 2 (2008), 69–74.

[36] Mehryar Mohri. 2000. Generic ε-removal algorithm for weighted automata. In *CIAA 2000*. Springer, 230–242.

[37] Maciej Olejnik, Herbert Wiklicky, and Mahdi Cheraghchi. 2016. Probabilistic Programming and Discrete Time Markov Chains. (2016). http://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/MaciejOlejnik.pdf

[38] Michael O Rabin and Dana Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development* 3, 2 (1959), 114–125.

[39] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*. 123–137.

[40] N. Saheb-Djahromi. 1980. CPOs of measures for nondeterminism. *Theoretical Computer Science* 12 (1980), 19–37. https://doi.org/10.1016/0304-3975(80)90003-1

[41] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. 2008. CSAMP: A System for Network-wide Flow Monitoring. In *USENIX NSDI*. 233–246.

[42] Micha Sharir, Amir Pnueli, and Sergiu Hart. 1984. Verification of probabilistic programs. *SIAM J. Comput.* 13, 2 (1984), 292–314. https://doi.org/10.1137/0213021

[43] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. 2018. RADWAN: Rate Adaptive Wide Area Network. In *ACM SIGCOMM*.

[44] Steffen Smolka, Spiros Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. In *ICFP 2015*. https://doi.org/10.1145/2784731.2784761

[45] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor Meets Scott: Semantic Foundations for Probabilistic Networks. In *POPL 2017*. https://doi.org/10.1145/3009837.3009843

[46] L. Valiant. 1982. A Scheme for Fast Parallel Communication. *SIAM J. Comput.* 11, 2 (1982), 350–361.

[47] Di Wang, Jan Hoffmann, and Thomas Reps. 2018. PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs. In *POPL 2018*. https://www.cs.cmu.edu/~janh/papers/WangHR17.pdf

[48] James Worrell. 2013. Revisiting the equivalence problem for finite multitape automata. In *International Colloquium on Automata, Languages, and Programming (ICALP)*. Springer, 422–433.

[49] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. 2005. On static reachability analysis of IP networks. In *INFOCOM*.

[50] Zhiyong Zhang, Ovidiu Mara, and Katerina Argyraki. 2014. Network Neutrality Inference. In *ACM SIGCOMM*. 63–74.

A OMITTED PROOFS

Lemma A.1. *Let A be a finite boolean combination of basic open sets,* i.e. *sets of the form $B_a = \{a\}\uparrow$ for $a \in \wp_\omega(\mathsf{H})$, and let $\llparenthesis - \rrparenthesis$ denote the semantics from [45]. Then for all programs p and inputs $a \in 2^{\mathsf{H}}$,*

$$\llparenthesis p^* \rrparenthesis(a)(A) = \lim_{n \to \infty} \llparenthesis p^{(n)} \rrparenthesis(a)(A)$$

Proof. Using topological arguments, the claim follows directly from previous results: A is a Cantor-clopen set by [45] (*i.e.*, both A and $\overline{A}$ are Cantor-open), so its indicator function $\mathbf{1}_A$ is Cantor-continuous. But $\mu_n \triangleq \llparenthesis p^{(n)} \rrparenthesis(a)$ converges weakly to $\mu \triangleq \llparenthesis p^* \rrparenthesis(a)$ in the Cantor topology (Theorem 4 in [13]), so

$$\lim_{n \to \infty} \llparenthesis p^{(n)} \rrparenthesis(a)(A) = \lim_{n \to \infty} \int \mathbf{1}_A d\mu_n = \int \mathbf{1}_A d\mu = \llparenthesis p^* \rrparenthesis(a)(A)$$

(To see why A and $\overline{A}$ are open in the Cantor topology, note that they can be written in disjunctive normal form over atoms $B_{\{h\}}$.) □

Predicates in ProbNetKAT form a Boolean algebra.

Lemma A.2. *Every predicate t satisfies $\llbracket t \rrbracket(a) = \delta_{a \cap b_t}$ for a certain packet set $b_t \subseteq \mathsf{Pk}$, where*

- $b_{\mathsf{drop}} = \varnothing$,
- $b_{\mathsf{skip}} = \mathsf{Pk}$,
- $b_{f=n} = \{\pi \in \mathsf{Pk} \mid \pi.f = n\}$,
- $b_{\neg t} = \mathsf{Pk} - b_t$,
- $b_{t \& u} = b_t \cup b_u$, *and*
- $b_{t;u} = b_t \cap b_u$.

Proof. For drop, skip, and $f{=}n$, the claim holds trivially. For $\neg t$, $t \mathbin{\&} u$, and $t \mathbin{;} u$, the claim follows inductively, using that $\mathcal{D}(f)(\delta_b) = \delta_{f(b)}$, $\delta_b \times \delta_c = \delta_{(b,c)}$, and that $f^\dagger(\delta_b) = f(b)$. The first and last equations hold because $\langle \mathcal{D}, \delta, -^\dagger \rangle$ is a monad. □

Proof of Proposition 3.1. We only need to show that for dup-free programs p and history-free inputs $a \in 2^{\mathsf{Pk}}$, $\llparenthesis p \rrparenthesis(a)$ is a distribution on packets (where we identify packets and singleton histories). We proceed by structural induction on p. All cases are straightforward except perhaps the case of p^*. For this case, by the induction hypothesis, all $\llbracket p^{(n)} \rrbracket(a)$ are discrete probability distributions on packet sets, therefore vanish outside 2^{Pk}. By Lemma A.1, this is also true of the limit $\llbracket p^* \rrbracket(a)$, as its value on 2^{Pk} must be 1, therefore it is also a discrete distribution on packet sets. □

Proof of Lemma 3.2. This follows directly from Lemma A.1 and Proposition 3.1 by noticing that *any* set $A \subseteq 2^{\mathsf{Pk}}$ is a finite boolean combination of basic open sets. □

Proof of Theorem 4.1. It suffices to show the equality $\mathcal{B}\llbracket p \rrbracket_{ab} = \llbracket p \rrbracket(a)(\{b\})$; the remaining claims then follow by well-definedness of $\llbracket - \rrbracket$. The equality is shown using Lemma 3.2 and a routine induction on p:

For $p = \mathsf{drop}, \mathsf{skip}, f{=}n, f{\leftarrow}n$ we have

$$\llbracket p \rrbracket(a)(\{b\}) = \delta_c(\{b\}) = \mathbf{1}[b = c] = \mathcal{B}\llbracket p \rrbracket_{ab}$$

for $c = \varnothing, a, \{\pi \in a \mid \pi.f = n\}, \{\pi[f := n] \mid \pi \in a\}$, respectively.

For $\neg t$ we have,

$$\begin{aligned}
\mathcal{B}[\![\neg t]\!]_{ab} &= \mathbf{1}[b \subseteq a] \cdot \mathcal{B}[\![t]\!]_{a,a-b} \\
&= \mathbf{1}[b \subseteq a] \cdot [\![t]\!](a)(\{a-b\}) && \text{(IH)} \\
&= \mathbf{1}[b \subseteq a] \cdot \mathbf{1}[a-b = a \cap b_t] && \text{(Lemma A.2)} \\
&= \mathbf{1}[b \subseteq a] \cdot \mathbf{1}[a-b = a - (\mathsf{H} - b_t)] \\
&= \mathbf{1}[b = a \cap (H - b_t)] \\
&= [\![\neg t]\!](a)(b) && \text{(Lemma A.2)}
\end{aligned}$$

For $p \mathbin{\&} q$, letting $\mu = [\![p]\!](a)$ and $\nu = [\![q]\!](a)$ we have

$$\begin{aligned}
[\![p \mathbin{\&} q]\!](a)(\{b\}) &= (\mu \times \nu)(\{(b_1,b_2) \mid b_1 \cup b_2 = b\}) \\
&= \textstyle\sum_{b_1,b_2} \mathbf{1}[b_1 \cup b_2 = b] \cdot (\mu \times \nu)(\{(b_1,b_2)\}) \\
&= \textstyle\sum_{b_1,b_2} \mathbf{1}[b_1 \cup b_2 = b] \cdot \mu(\{b_1\}) \cdot \nu(\{b_2\}) \\
&= \textstyle\sum_{b_1,b_2} \mathbf{1}[b_1 \cup b_2 = b] \cdot \mathcal{B}[\![p]\!]_{ab_1} \cdot \mathcal{B}[\![q]\!]_{ab_2} && \text{(IH)} \\
&= \mathcal{B}[\![p \mathbin{\&} q]\!]_{ab}
\end{aligned}$$

where we use in the second step that $b \subseteq \mathsf{Pk}$ is finite, thus $\{(b_1,b_2) \mid b_1 \cup b_2 = b\}$ is finite.

For $p \mathbin{;} q$, let $\mu = [\![p]\!](a)$ and $\nu_c = [\![q]\!](c)$ and recall that μ is a discrete distribution on 2^{Pk}. Thus

$$\begin{aligned}
[\![p \mathbin{;} q]\!](a)(\{b\}) &= \textstyle\sum_{c \in 2^{\mathsf{Pk}}} \nu_c(\{b\}) \cdot \mu(\{c\}) \\
&= \textstyle\sum_{c \in 2^{\mathsf{Pk}}} \mathcal{B}[\![q]\!]_{c,b} \cdot \mathcal{B}[\![p]\!]_{a,c} \\
&= \mathcal{B}[\![p \mathbin{;} q]\!]_{ab}.
\end{aligned}$$

For $p \oplus_r q$, the claim follows directly from the induction hypotheses.

Finally, for p^*, we know that $\mathcal{B}[\![p^{(n)}]\!]_{ab} = [\![p^{(n)}]\!](a)(\{b\})$ by induction hypothesis. The key to proving the claim is Lemma 3.2, which allows us to take the limit on both sides and deduce

$$\mathcal{B}[\![p^*]\!]_{ab} = \lim_{n\to\infty} \mathcal{B}[\![p^{(n)}]\!]_{ab} = \lim_{n\to\infty} [\![p^{(n)}]\!](a)(\{b\}) = [\![p^*]\!](a)(\{b\}). \qquad \square$$

Proof of Lemma 5.1. For arbitrary $a, b \subseteq \mathsf{Pk}$, we have

$$\begin{aligned}
\sum_{a',b'} \mathcal{S}[\![p]\!]_{(a,b),(a',b')} &= \sum_{a',b'} \mathbf{1}[b' = a \cup b] \cdot \mathcal{B}[\![p]\!]_{a,a'} \\
&= \sum_{a'} \Big(\sum_{b'} \mathbf{1}[b' = a \cup b] \Big) \cdot \mathcal{B}[\![p]\!]_{a,a'} \\
&= \sum_{a'} \mathcal{B}[\![p]\!]_{a,a'} = 1
\end{aligned}$$

where in the last step, we use that $\mathcal{B}[\![p]\!]$ is stochastic (Theorem 4.1). $\square$

Proof of Lemma 5.3. By induction on $n \geq 0$. For $n = 0$, we have

$$\begin{aligned}\sum_{a'} \mathbf{1}[b' = a' \cup b] \cdot \mathcal{B}\llbracket p^{(n)} \rrbracket_{a,a'} &= \sum_{a'} \mathbf{1}[b' = a' \cup b] \cdot \mathcal{B}\llbracket \mathsf{skip} \rrbracket_{a,a'} \\ &= \sum_{a'} \mathbf{1}[b' = a' \cup b] \cdot \mathbf{1}[a = a'] \\ &= \mathbf{1}[b' = a \cup b] \\ &= \mathbf{1}[b' = a \cup b] \cdot \sum_{a'} \mathcal{B}\llbracket p \rrbracket_{a,a'} \\ &= \sum_{a'} \mathcal{S}\llbracket p \rrbracket_{(a,b),(a',b')}\end{aligned}$$

In the induction step ($n > 0$),

$$\begin{aligned}&\sum_{a'} \mathbf{1}[b' = a' \cup b] \cdot \mathcal{B}\llbracket p^{(n)} \rrbracket_{a,a'} \\ &= \sum_{a'} \mathbf{1}[b' = a' \cup b] \cdot \mathcal{B}\llbracket \mathsf{skip} \,\&\, p \,;\, p^{(n-1)} \rrbracket_{a,a'} \\ &= \sum_{a'} \mathbf{1}[b' = a' \cup b] \cdot \sum_{c} \mathbf{1}[a' = a \cup c] \cdot \mathcal{B}\llbracket p \,;\, p^{(n-1)} \rrbracket_{a,c} \\ &= \sum_{c} \left(\sum_{a'} \mathbf{1}[b' = a' \cup b] \cdot \mathbf{1}[a' = a \cup c] \right) \cdot \sum_{k} \mathcal{B}\llbracket p \rrbracket_{a,k} \cdot \mathcal{B}\llbracket p^{(n-1)} \rrbracket_{k,c} \\ &= \sum_{c,k} \mathbf{1}[b' = a \cup c \cup b] \cdot \mathcal{B}\llbracket p \rrbracket_{a,k} \cdot \mathcal{B}\llbracket p^{(n-1)} \rrbracket_{k,c} \\ &= \sum_{k} \mathcal{B}\llbracket p \rrbracket_{a,k} \cdot \sum_{a'} \mathbf{1}[b' = a' \cup (a \cup b)] \cdot \mathcal{B}\llbracket p^{(n-1)} \rrbracket_{k,a'} \\ &= \sum_{k} \mathcal{B}\llbracket p \rrbracket_{a,k} \cdot \sum_{a'} \mathcal{S}\llbracket p \rrbracket^{n}_{(k, a \cup b),(a',b')} \\ &= \sum_{a'} \sum_{k_1,k_2} \mathbf{1}[k_2 = a \cup b] \cdot \mathcal{B}\llbracket p \rrbracket_{a,k_1} \cdot \mathcal{S}\llbracket p \rrbracket^{n}_{(k_1,k_2),(a',b')} \\ &= \sum_{a'} \sum_{k_1,k_2} \mathcal{S}\llbracket p \rrbracket_{(a,b)(k_1,k_2)} \cdot \mathcal{S}\llbracket p \rrbracket^{n}_{(k_1,k_2),(a',b')} \\ &= \sum_{a'} \mathcal{S}\llbracket p \rrbracket^{n+1}_{(a,b),(a',b')}\end{aligned}$$

$\square$

Lemma A.3. *The matrix $X = I - Q$ in Equation (2) of §5.1 is invertible.*

Proof. Let S be a finite set of states, $|S| = n$, M an $S \times S$ substochastic matrix ($M_{st} \geq 0$, $M\mathbf{1} \leq \mathbf{1}$). A state s is *defective* if $(M\mathbf{1})_s < 1$. We say M is *stochastic* if $M\mathbf{1} = \mathbf{1}$, *irreducible* if $(\sum_{i=0}^{n-1} M^i)_{st} > 0$ (that is, the support graph of M is strongly connected), and *aperiodic* if all entries of some power of M are strictly positive.

We show that if M is substochastic such that every state can reach a defective state via a path in the support graph, then the spectral radius of M is strictly less than 1. Intuitively, all weight in the system eventually drains out at the defective states.

Let e_s, $s \in S$, be the standard basis vectors. As a distribution, e_s^T is the unit point mass on s. For $A \subseteq S$, let $e_A = \sum_{s \in A} e_s$. The L_1-norm of a substochastic vector is its total weight as a distribution.

Multiplying on the right by M never increases total weight, but will strictly decrease it if there is nonzero weight on a defective state. Since every state can reach a defective state, this must happen after n steps, thus $\|e_s^T M^n\|_1 < 1$. Let $c = \max_s \|e_s^T M^n\|_1 < 1$. For any $y = \sum_s a_s e_s$,

$$\begin{aligned} \|y^T M^n\|_1 &= \|(\sum_s a_s e_s)^T M^n\|_1 \\ &\leq \sum_s |a_s| \cdot \|e_s^T M^n\|_1 \leq \sum_s |a_s| \cdot c = c \cdot \|y^T\|_1. \end{aligned}$$

Then M^n is contractive in the L_1 norm, so $|\lambda| < 1$ for all eigenvalues λ. Thus $I - M$ is invertible because 1 is not an eigenvalue of M. □

Proof of Proposition 5.6.

(1) It suffices to show that $USU = SU$. Suppose that

$$\mathbf{Pr}[(a,b) \xrightarrow{USU}_1 (a',b')] = p > 0.$$

It suffices to show that this implies

$$\mathbf{Pr}[(a,b) \xrightarrow{SU}_1 (a',b')] = p.$$

If (a,b) is saturated, then we must have $(a',b') = (\emptyset, b)$ and

$$\mathbf{Pr}[(a,b) \xrightarrow{USU}_1 (\emptyset,b)] = 1 = \mathbf{Pr}[(a,b) \xrightarrow{SU}_1 (\emptyset,b)]$$

If (a,b) is not saturated, then $(a,b) \xrightarrow{U}_1 (a,b)$ with probability 1 and therefore

$$\mathbf{Pr}[(a,b) \xrightarrow{USU}_1 (a',b')] = \mathbf{Pr}[(a,b) \xrightarrow{SU}_1 (a',b')]$$

(2) Since S and U are stochastic, clearly SU is a MC. Since SU is finite state, any state can reach an absorbing communication class. (To see this, note that the reachability relation $\xrightarrow{SU}$ induces a partial order on the communication classes of SU. Its maximal elements are necessarily absorbing, and they must exist because the state space is finite.) It thus suffices to show that a state set $C \subseteq 2^{\mathsf{Pk}} \times 2^{\mathsf{Pk}}$ in SU is an absorbing communication class iff $C = \{(\emptyset, b)\}$ for some $b \subseteq \mathsf{Pk}$.

"⇐": Observe that $\emptyset \xrightarrow{B}_1 a'$ iff $a' = \emptyset$. Thus $(\emptyset, b) \xrightarrow{S}_1 (a',b')$ iff $a' = \emptyset$ and $b' = b$, and likewise $(\emptyset, b) \xrightarrow{U}_1 (a',b')$ iff $a' = \emptyset$ and $b' = b$. Thus $(\emptyset, b)$ is an absorbing state in SU as required.

"⇒": First observe that by monotonicity of SU (Lemma 5.5), we have $b = b'$ whenever $(a,b) \xleftrightarrow{SU} (a',b')$; thus there exists a fixed b_C such that $(a,b) \in C$ implies $b = b_C$.

Now pick an arbitrary state $(a, b_C) \in C$. It suffices to show that $(a, b_C) \xrightarrow{SU} (\emptyset, b_C)$, because that implies $(a, b_C) \xleftrightarrow{SU} (\emptyset, b_C)$, which in turn implies $a = \emptyset$. But the choice of $(a, b_C) \in C$ was arbitrary, so that would mean $C = \{(\emptyset, b_C)\}$ as claimed.

To show that $(a, b_C) \xrightarrow{SU} (\emptyset, b_C)$, pick arbitrary states such that

$$(a, b_C) \xrightarrow{S} (a',b') \xrightarrow{U}_1 (a'',b'')$$

and recall that this implies $(a, b_C) \xrightarrow{SU} (a'', b'')$ by claim (1). Then $(a'', b'') \xrightarrow{SU} (a, b_C)$ because C is absorbing, and thus $b_C = b' = b''$ by monotonicity of S, U, and SU. But (a', b') was chosen as an arbitrary state S-reachable from (a, b_C), so (a, b) and by transitivity (a', b') must be saturated. Thus $a'' = \varnothing$ by the definition of U. □

Proof of Theorem 5.7. Using Proposition 5.6.1 in the second step and equation (3) in the last step,

$$\begin{aligned} \lim_{n\to\infty} \sum_{a'} S^n_{(a,b),(a',b')} &= \lim_{n\to\infty} \sum_{a'} (S^n U)_{(a,b),(a',b')} \\ &= \lim_{n\to\infty} \sum_{a'} (SU)^n_{(a,b),(a',b')} \\ &= \sum_{a'} (SU)^\infty_{(a,b),(a',b')} = (SU)^\infty_{(a,b),(\varnothing,b')} \end{aligned}$$

$(SU)^\infty$ is computable because S and U are matrices over $\mathbb{Q}$ and hence so is $(I - Q)^{-1}R$. □

Proof of Corollary 5.8. Recall from Corollary 4.2 that it suffices to compute the finite rational matrices $\mathcal{B}[\![p]\!]$ and $\mathcal{B}[\![q]\!]$ and check them for equality. But Theorem 5.7 together with Proposition 5.2 gives us an effective mechanism to compute $\mathcal{B}[\![-]\!]$ in the case of Kleene star, and $\mathcal{B}[\![-]\!]$ is straightforward to compute in all other cases. Summarizing the full chain of equalities, we have:

$$[\![p^*]\!](a)(\{b\}) = \mathcal{B}[\![p^*]\!]_{a,b} = \lim_{n\to\infty} \mathcal{B}[\![p^{(n)}]\!]_{a,b} = \lim_{n\to\infty} \sum_{a'} \mathcal{S}[\![p]\!]^n_{(a,\varnothing),(a',b)} = (SU)^\infty_{(a,\varnothing),(\varnothing,b)}$$

following from Theorem 4.1, Definition of $\mathcal{B}[\![-]\!]$, Proposition 5.2, and finally Theorem 5.7. □

B HANDLING FULL PROBNETKAT: OBSTACLES AND CHALLENGES

History-free ProbNetKAT can describe sophisticated network routing schemes under various failure models, and the program semantics can be computed exactly. Performing quantitative reasoning in full ProbNetKAT appears significantly more challenging. We illustrate some of the difficulties in deciding program equivalence; recall that this is decidable for the history-free fragment (Corollary 5.8).

The main difference in the original ProbNetKAT language is an additional primitive dup. Intuitively, this command duplicates a packet $\pi \in \mathsf{Pk}$ and outputs the word $\pi\pi \in \mathsf{H}$, where $\mathsf{H} = \mathsf{Pk}^*$ is the set of non-empty, finite sequences of packets. An element of H is called a *packet history*, representing a log of previous packet states. ProbNetKAT policies may only modify the first (*head*) packet of each history; dup fixes the current head packet into the log by copying it. In this way, ProbNetKAT policies can compute distributions over the paths used to forward packets, instead of just over the final output packets.

However, with dup, the semantics of ProbNetKAT becomes significantly more complex. Policies p now transform sets of packet histories $a \in 2^{\mathsf{H}}$ to distributions $[\![p]\!](a) \in \mathcal{D}(2^{\mathsf{H}})$. Since 2^{H} is uncountable, these distributions are no longer guaranteed to be discrete, and formalizing the semantics requires full-blown measure theory (see prior work for details [45]).

Without dup, policies operate on sets of packets 2^{Pk}; crucially, this is a *finite* set and we can represent each set with a single state in a finite Markov chain. With dup, policies operate on sets of packet histories 2^{H}. Since this set is not finite—in fact, it is not even countable—encoding each packet history as a state would give a Markov chain with infinitely many states. Procedures for deciding equivalence are not known for such systems in general.

While in principle there could be a more compact representation of general ProbNetKAT policies as finite Markov chains or other models where equivalence is decidable, (*e.g.*, weighted or probabilistic automata [12] or quantitative variants of regular expressions [3]), we suspect that deciding equivalence in the presence of dup may be intractable. As circumstantial evidence, ProbNetKAT policies can simulate a probabilistic variant of multitape automaton originally introduced by Rabin and Scott [38]. We specialize the definition here to two tapes, for simplicity, but ProbNetKAT programs can encode any multitape automata with any fixed number of tapes.

Definition B.1. Let A be a finite alphabet. A *probabilistic multitape automaton* is defined by a tuple (S, s_0, ρ, τ) where S is a finite set of states; $s_0 \in S$ is the initial state; $\rho : S \to (A \cup \{\_\})^2$ maps each state to a pair of letters (u, v), where either u or v may be a special blank character _; and the transition function $\tau : S \to \mathcal{D}(S)$ gives the probability of transitioning from one state to another.

The semantics of an automaton can be defined as a probability measure on the space $A^\infty \times A^\infty$, where A^∞ is the set of finite and (countably) infinite words over the alphabet A. Roughly, these measures are fully determined by the probabilities of producing any two finite prefixes of words $(w, w') \in A^* \times A^*$.

Presenting the formal semantics would require more concepts from measure theory and take us far afield, but the basic idea is simple to describe. An infinite trace of a probabilistic multitape automaton over states $s_0, s_1, s_2, \ldots$ gives a sequence of pairs of (possibly blank) letters:

$$\rho(s_0), \rho(s_1), \rho(s_2) \ldots$$

By concatenating these pairs together and dropping all blank characters, a trace induces two (finite or infinite) words over the alphabet A. For example, the sequence,

$$(a_0, \_), (a_1, \_), (\_, a_2), \ldots$$

gives the words $a_0 a_1 \ldots$ and $a_2 \ldots$. Since the traces are generated by the probabilistic transition function τ, each automaton gives rise to a probability measure over pairs of infinite words.

Probabilistic multitape automata can be encoded as ProbNetKAT policies with dup. We sketch the idea here, deferring further details to Appendix C. Suppose we are given an automaton (S, s_0, ρ, τ). We build a ProbNetKAT policy over packets with two fields, st and id. The first field st ranges over the states S and the alphabet A, while the second field id is either 1 or 2; we suppose the input set has exactly two packets labeled with id = 1 and id = 2. In a set of packet history, the two active packets have the same value for st $\in S$—this represents the current state in the automaton. Past packets in the history have st $\in A$, representing the words produced so far; the first and second components of the output are tracked by the histories with id = 1 and id = 2. We can encode the transition function τ as a probabilistic choice in ProbNetKAT, updating the current state st of all packets, and recording non-blank letters produced by ρ in the two components by applying dup on packets with the corresponding value of id.

Intuitively, a set of packet histories generated by the resulting ProbNetKAT term describes a pair of words generated by the original automaton. With a bit more bookkeeping (see Appendix C), we can show that two probabilistic multitape automata are equivalent if and only if their encoded ProbNetKAT policies are equivalent. Thus, deciding equivalence for ProbNetKAT with dup is harder than deciding equivalence for probabilistic multitape automata; similar reductions have been considered before for showing undecidability of related problems about KAT [30] and probabilistic NetKAT [25].

Deciding equivalence between probabilistic multitape automata is a challenging open problem. In the special case where only one word is generated (say, when the second component produced is always blank), these automata are equivalent to standard automata with ε-transitions (*e.g.*, see [36]).

In this setting, non-productive steps can be eliminated and the automata can be modeled as finite state Markov chains, where equivalence is decidable. In our setting, however, steps producing blank letters in one component may produce non-blank letters in the other. As a result, it is not clear how to eliminate these steps and encode our automata as Markov chains. Removing probabilities, it is known that equivalence between non-deterministic multitape automata is undecidable [23]. Deciding equivalence of deterministic multitape automata remained a challenging open question for many years, until Harju and Karhumäki [24] surprisingly settled the question positively; Worrell [48] later gave an alternative proof. If equivalence of probabilistic multitape automata is undecidable, then equivalence is undecidable for ProbNetKAT programs as well. However if equivalence turns out to be decidable, the proof technique may shed light on how to decide equivalence for the full ProbNetKAT language.

C ENCODING 2-GENERATIVE AUTOMATA IN FULL PROBNETKAT

To keep notation light, we describe our encoding in the special case where the alphabet $A = \{x, y\}$, there are four states $S = \{s_1, s_2, s_3, s_4\}$, the initial state is s_1, and the output function ρ is

$$\rho(s_1) = (x, \_) \qquad \rho(s_2) = (y, \_) \qquad \rho(s_3) = (\_, x) \qquad \rho(s_4) = (\_, y).$$

Encoding general automata is not much more complicated. Let $\tau : S \to \mathcal{D}(S)$ be a given transition function; we write $p_{i,j}$ for $\tau(s_i)(s_j)$. We will build a ProbNetKAT policy simulating this automaton. Packets have two fields, st and id, where st ranges over $S \cup A \cup \{\bullet\}$ and id ranges over $\{1, 2\}$. Define:

$$p \triangleq \mathsf{st}{=}s_1 \mathbin{;} \mathsf{loop}^* \mathbin{;} \mathsf{st}{\leftarrow}\bullet$$

The initialization keeps packets that start in the initial state, while the final command marks histories that have exited the loop by setting st to be the special letter •.

The main program loop first branches on the current state st:

$$\mathsf{loop} \triangleq \mathsf{case} \begin{cases} \mathsf{st}{=}s_1 : \mathsf{state1} \\ \mathsf{st}{=}s_2 : \mathsf{state2} \\ \mathsf{st}{=}s_3 : \mathsf{state3} \\ \mathsf{st}{=}s_4 : \mathsf{state4} \end{cases}$$

Then, the policy simulates the behavior from each state. For instance:

$$\mathsf{state1} \triangleq \bigoplus \begin{cases} (\mathsf{if\ id{=}1\ then\ st}{\leftarrow}x \mathbin{;} \mathsf{dup\ else\ skip}) \mathbin{;} \mathsf{st}{\leftarrow}s_1 \ @\ p_{1,1}, \\ (\mathsf{if\ id{=}1\ then\ st}{\leftarrow}y \mathbin{;} \mathsf{dup\ else\ skip}) \mathbin{;} \mathsf{st}{\leftarrow}s_2 \ @\ p_{1,2}, \\ (\mathsf{if\ id{=}2\ then\ st}{\leftarrow}x \mathbin{;} \mathsf{dup\ else\ skip}) \mathbin{;} \mathsf{st}{\leftarrow}s_3 \ @\ p_{1,3}, \\ (\mathsf{if\ id{=}2\ then\ st}{\leftarrow}y \mathbin{;} \mathsf{dup\ else\ skip}) \mathbin{;} \mathsf{st}{\leftarrow}s_4 \ @\ p_{1,4} \end{cases}$$

The policies state2, state3, state4 are defined similarly.

Now, suppose we are given two probabilistic multitape automata W, W' that differ only in their transition functions. For simplicity, we will further assume that both systems have strictly positive probability of generating a letter in either component in finitely many steps from any state. Suppose they generate distributions μ, μ' respectively over pairs of infinite words $A^\omega \times A^\omega$. Now, consider the encoded ProbNetKAT policies p, p'. We argue that $[\![p]\!] = [\![q]\!]$ if and only if $\mu = \mu'$.[5]

First, it can be shown that $[\![p]\!] = [\![p']\!]$ if and only if $[\![p]\!](e) = [\![p']\!](e)$, where

$$e \triangleq \{\pi\pi \mid \pi \in \mathsf{Pk}\}.$$

---

[5] We will not present the semantics of ProbNetKAT programs with dup here; instead, the reader should consult earlier papers [13, 45] for the full development.

Let $\nu = [\![p]\!](e)$ and $\nu' = [\![p']\!](e)$. The key connection between the automata and the encoded policies is the following equality:

$$\mu(S_{u,v}) = \nu(T_{u,v}) \tag{6}$$

for every pair of finite prefixes $u, v \in A^*$. In the automata distribution on the left, $S_{u,v} \subseteq A^\omega \times A^\omega$ consists of all pairs of infinite strings where u is a prefix of the first component and v is a prefix of the second component. In the ProbNetKAT distribution on the right, we first encode u and v as packet histories. For $i \in \{1, 2\}$ representing the component and $w \in A^*$ a finite word, define the history

$$h_i(w) \in \mathsf{H} \triangleq (\mathrm{st} = \bullet, \mathrm{id} = i), (\mathrm{st} = w[|w|], \mathrm{id} = i), \ldots, (\mathrm{st} = w[1], \mathrm{id} = i), (\mathrm{st} = s_1, \mathrm{id} = i).$$

The letters of the word w are encoded in reverse order because by convention, the head/newest packet is written towards the left-most end of a packet history, while the oldest packet is written towards the right-most end. For instance, the final letter $w[|w|]$ is the most recent (*i.e.*, the latest) letter produced by the policy. Then, $T_{u,v}$ is the set of all history sets including $h_1(u)$ and $h_2(v)$:

$$T_{u,v} \triangleq \{a \in 2^{\mathsf{H}} \mid h_1(u) \in a, h_2(v) \in a\}.$$

Now $[\![p]\!] = [\![p']\!]$ implies $\mu = \mu'$, since (6) gives

$$\mu(S_{u,v}) = \mu'(S_{u,v}).$$

The reverse implication is a bit more delicate. Again by (6), we have

$$\nu(T_{u,v}) = \nu'(T_{u,v}).$$

We need to extend this equality to all cones, defined by packet histories h:

$$B_h \triangleq \{a \in 2^{\mathsf{H}} \mid h \in a\}.$$

This follows by expressing B_h as boolean combinations of $T_{u,v}$, and observing that the encoded policy produces only sets of encoded histories, *i.e.*, where the most recent state st is set to $\bullet$ and the initial state st is set to s_1.