# ProbNV: Probabilistic Verification of Network Control Planes

NICK GIANNARAKIS, University of Wisconsin-Madison, USA
ALEXANDRA SILVA, University College London, UK and Cornell University, USA
DAVID WALKER, Princeton University, USA

ProbNV is a new framework for probabilistic network control plane verification that strikes a balance between generality and scalability. ProbNV is general enough to encode a wide range of features from the most common protocols (eBGP and OSPF) and yet scalable enough to handle challenging properties, such as probabilistic all-failures analysis of medium-sized networks with 100-200 devices. When there are a small, bounded number of failures, networks with up to 500 devices may be verified in seconds. ProbNV operates by translating raw CISCO configurations into a probabilistic and functional programming language designed for network verification. This language comes equipped with a novel type system that characterizes the sort of representation to be used for each data structure: *concrete* for the usual representation of values; *symbolic* for a BDD-based representation of sets of values; and *multi-value* for an MTBDD-based representation of values that depend upon symbolics. Careful use of these varying representations speeds execution of symbolic simulation of network models. The MTBDD-based representations are also used to calculate probabilistic properties of network models once symbolic simulation is complete. We implement the language and evaluate its performance on benchmarks constructed from real network topologies and synthesized routing policies.

CCS Concepts: • **Networks** → **Network reliability**; **Protocol testing and verification**; • **Theory of computation** → *Automated reasoning*; • **Mathematics of computing** → **Probabilistic inference problems**.

Additional Key Words and Phrases: Network Verification, Network Simulation, Probabilistic verification, Probabilistic Network Analysis, Control Plane Analysis, Router Configuration Analysis

**ACM Reference Format:**
Nick Giannarakis, Alexandra Silva, and David Walker. 2021. ProbNV: Probabilistic Verification of Network Control Planes. *Proc. ACM Program. Lang.* 5, ICFP, Article 90 (August 2021), 30 pages. https://doi.org/10.1145/3473595

## 1 INTRODUCTION

Computer networks come in all shapes and sizes, from single-router home networks to modest-sized office or university networks to the large industrial data center or wide area networks of a select few tech companies. Unfortunately, once a network grows beyond a few devices, it becomes difficult to manage and maintain as configurations for each device often involve hundreds or thousands of lines of domain-specific, assembly-like code that controls the routing process. Changes to such configurations can cause trickle-down effects and bring down services or even cut off portions of the internet [Godfrey 2016; Graham-Cumming 2020; McCarthy 2019; Sharwood 2016; Sverdlik 2012]. Unfortunately, many such problems are difficult to anticipate because doing so depends

Authors' addresses: Nick Giannarakis, nick.giannarakis@gmail.com, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI, USA, 53706; Alexandra Silva, alexandra.silva@ucl.ac.uk, University College London, Computer Science Department, Gower Street, London, UK, WC1E 6BT and Cornell University, 402 Gates Hall, Ithaca, USA, 14853; David Walker, dpw@cs.princeton.edu, Princeton University, 35 Olden Street, Princeton, NJ, USA, 08542.

2475-1421/2021/8-ART90
https://doi.org/10.1145/3473595

upon reasoning about unknowns or probabilistic phenomena. The development of new techniques for probabilistic analysis of networks should play a significant role in improving the reliability of this critical infrastructure.

In this paper, we consider the problem of probabilistic reasoning about network *control planes*—that is, the protocols used by traditional routers to exchange information about the available *routes* for sending user traffic from one place to another.[1] Recent research efforts in this area have made different choices about how to navigate the tradeoff between generality (the ability to handle a wide range of different kinds of network configurations) and performance. For instance, Bayonet [Gehr et al. 2018] can model a wide range of network protocols (not just control plane protocols) and probabilistic features, but it only scales to very small networks: Bayonet verifies data plane properties of 30-node networks in 5-10 minutes. On the other end of the spectrum, NetDice [Steffen et al. 2020] is able to analyze some networks with up to 750 devices, but imposes restrictions on the set of routing features that may be analyzed (*e.g.*, it does not support widely used features, such as BGP communities or BGP local-preference). It exploits these restrictions to implement clever pruning optimizations that allow it to ignore parts of the search space in common cases.

In this paper, we introduce ProbNV, a probabilistic control plane analysis system that strikes a balance between these two extremes. ProbNV is quite general when it comes to control plane analysis, as it is capable of processing a wide array of features of the most common control plane protocols such as eBGP (with local preference, path lengths, MEDs, communities, and administrative distances), OSPF (with link weights, areas and administrative distances), static routes, connected routes, and combinations thereof, and yet can scale to medium-sized networks. For instance, it can analyze networks with roughly 500 or more nodes for properties that involve a small, bounded number failures in a few seconds and networks with roughly 100-200 nodes when considering any number of (probabilistic) failures in a few minutes.

ProbNV's implementation consists of two main components: (1) a *configuration compiler* that translates CISCO configurations for BGP and OSPF protocols into a functional and probabilistic network modeling language, and (2) a symbolic simulation engine that executes the resulting functional models efficiently and is optimized for network control plane models. The configuration compiler is not novel—we reuse most of an existing compiler [Giannarakis et al. 2020]. Our main technical achievement comes in the design and implementation of the symbolic simulator.

Efficient simulation of network protocols must itself navigate a difficult tradeoff. On the one hand, many networks display enormous amounts of symmetry. For instance, modern industrial datacenters arrange their routers in a "fat tree" topology—a multi-level tree with extra connections for fault tolerance. At any level in the tree, every node is wired and configured similarly. Simulators that exploit such symmetries using efficient data structures, like Binary Decision Diagrams [Bryant 1986] (BDDs), to represent large sets of related values can produce great savings in time and space [Beckett et al. 2019a]. On the other hand, representing basic data like integers as BDDs induces significant overheads—implementing addition, for instance, over a BDD representation of integers amounts to implementing a standard logical adder in software as a series of bitwise manipulations. Simulation of network protocols often requires adding, incrementing, and comparing various integer values; representing them all as BDDs is inefficient.

To overcome this tension between representations, our symbolic simulator executes different parts of the same program in different *modes*. In *concrete mode* (C), the simulator executes individual

[1]In contrast, the *data plane* of a network is the part of the network responsible for forwarding user traffic. A single control plane will generate many different data planes, depending upon the environment in which it operates, making the control plane analysis problem strictly more challenging than the data plane analysis problem. Systems such as Header Space Analysis [Kazemian et al. 2012], Veriflow [Khurshid et al. 2013], Atomic Predicates [Yang and Lam 2016], NetKAT [Anderson et al. 2014], and McNetKAT [Smolka et al. 2019] analyze network data planes and cannot analyze network control planes.

lambda terms in the usual way, applying fast, machine-level operations to concrete values. In *symbolic mode* (S), the simulator represents sets of values compactly as BDDs and computes over BDDs. In *multi-value mode* (M), the simulator represents families of concrete values as Multi-Terminal BDDs (MTBDDs, also known as Algebraic Decision Diagrams) [Bahar et al. 1993; Clarke et al. 1996]. MTBDDs act as a connection point between the concrete and symbolic worlds; they allow us to compactly represent sets of concrete values that depend on symbolic values. Not all programs conform to these execution modes; we define a type system to identify routing models that are amenable to efficient symbolic simulation. Further, we define the operational semantics of this typed lambda calculus via a translation to a lower-level, (mostly) conventional typed lambda calculus. We prove the translation is type-preserving.

A second factor that impacts control plane simulation performance is the order in which nodes of the distributed network system are chosen to execute their routing functions. Using a smart order can cut down the number of different transitions of the system, allowing it to converge more quickly. We develop a new simulation algorithm that dynamically selects nodes to avoid duplicated work and show that this new algorithm can often improve performance by up to a factor of two.

For probabilistic reasoning, top-level symbolic values defined in our lambda calculus come equipped with distributions. For instance, such distributions may specify the probability of a link or router failure in a network. ProbNV's design is such that probabilistic reasoning is not much more expensive than deterministic reasoning about symbolic values. As in any control plane simulator, ProbNV computes the routes each device converges to; unlike conventional simulators, each route is a multi-value that explains how concrete routes depend upon symbolic inputs. Given a (discrete) distribution for each symbolic, we can then compute the probability that a property holds.

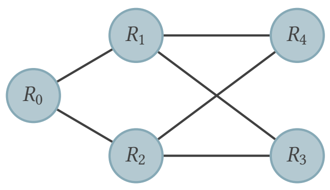
We evaluate the effectiveness of our framework on a set of real network topologies running protocols with synthesized policies. We find that ProbNV enables fast probabilistic verification for mid-sized networks when considering any number of node failures, and fast verification of bounded link failure scenarios (*i.e.*, "up to K failures") even for networks with thousands of links.

To summarize, this paper presents a new system for general, probabilistic analysis of control plane protocols. We make the following technical contributions:

- A typed lambda calculus that distinguishes between concrete, symbolic or multi-value execution modes (section 3).
- A high-performance implementation of this lambda-calculus, using BDDs to implement the symbolic parts of a program and MTBDDs to implement its multi-value parts (section 4).
- A simple approach to checking probabilistic properties by leveraging deterministic simulations (section 4.3).
- A new control plane simulation algorithm that optimizes the order in which nodes are processed (section 5).
- An evaluation of ProbNV on real network topologies running synthesized policies (section 6)

## 2 KEY IDEAS

Network analysis tools such as Batfish [Fogel et al. 2015] and NV [Giannarakis et al. 2020] operate by translating the domain-specific languages for configuring routers, such as those produced by CISCO and Juniper, into intermediate *modeling languages* for analysis. Our tool, ProbNV, uses the translation infrastructure developed for NV, and most of its functional intermediate language for modeling network control planes. It extends that functional core with constructs for defining probabilistic environments in which the protocols will operate. Whereas NV was limited to concrete simulation of networks, our new backend enables efficient *symbolic* simulation and evaluation



(a) The network topology.

```
let nodes = 5
let edges = {0=1; 0=2; 1=3; 1=4; 2=3; 2=4}

type route = option[int]

let init u = if u = 0n then Some 0 else None
let trans (e : tedge) (x : route) =
  match x with
  | None → None
  | Some len → Some (len+1)
let merge (u : tnode) (x y : route) =
  match (x,y) with
  | _, None → x
  | None, _ → y
  | Some x, Some y →
    Some (if x <= y then x else y)
```

(b) A simple protocol model.

```
distribution d : bool =
| false → 0.9
| true → 0.1

symbolic f1: bool = d
symbolic f2: bool = d
symbolic f3: bool = d

let initF u = if fNode u then None else init u
let transF e x = if fNode e.1 || fNode e.2 then None else trans e x
let mergeF u x y = merge u x y
```

```
let fNode u =
 match u with
 | 0n → false
 | 1n → f1
 | 2n → f2
 | 3n → f3
 | 4n → false
```

(c) A probabilistic failure model.

Fig. 1. Routing models in ProbNV.

of probabilistic properties via a new multi-mode execution engine. A novel type system for our intermediate language separates the different execution modes from one another.

## 2.1 Crafting Probabilistic Control Plane Models

Traditional routers use *distributed routing protocols* to exchange *messages*, often referred to as *routes*, with one another. These routes contain information about the paths that may be used to forward user traffic to a particular *destination*. Destinations are often designated by a *destination IP address* (*e.g.*, 10.126.0.0) or *destination prefix* (*e.g.*, 10.126.0.0/16), which refers to a set of related IP addresses. Different protocols disseminate different kinds of information and follow different rules about prioritizing paths to a destination. BGP (Border Gateway Protocol) messages include the path used, an integer called *local preference* that is used to increase (or, decrease) a path's preference, a set of optional tags called *community values*, as well as several other pieces of information. Simpler protocols such as RIP and OSPF carry information about (weighted) path costs.

Routers maintain a table associating destinations to a route (known as *Routing Information Base (RIB)*), and the goal of a routing protocol is to populate this table using the information exchanged with other routers. When a router receives a new route from a neighbor, it modifies it according to an import policy, and then compares it with any existing route towards the same destination. If

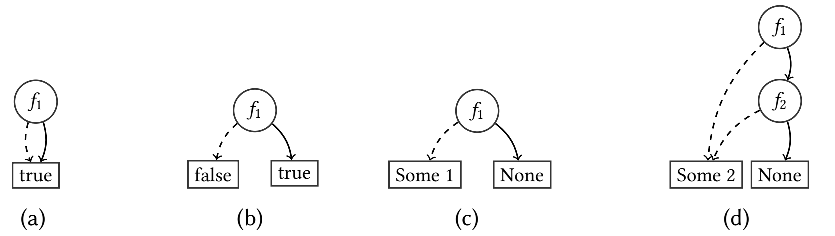


Fig. 2. Circles represent decision variables of BDDs/MTBDDs. If a decision variable is true, follow the solid arrow. If a decision variable is false, follow the dashed arrow. Rectangles are terminal nodes. (a) Symbolic value `f1` as an (unreduced) BDD. When viewed as a set, the BDD states that $f1$ may be true or false. (b) shows the BDD that results from the symbolic operation `fNode e.1 || fNode e.2` on edge $(0, 1)$. (c) illustrates the MTBDD that represents the route of $R_1$ ($R_1$ receives a route unless it fails). (d) illustrates the MTBDD that represents the route of $R_4$ ($R_4$ receives a route unless both $R_1$ and $R_2$ fail).

the new route is "better", it will update its RIB. The notion of a "better" route is protocol-specific. BGP ranks routes according to multiple metrics, among others, local-preference, the path length, and where the path originated from (an internal or an external network). Other protocols, such as RIP, only compare the length of the routes. In any case, when a router selects a new route, it needs to propagate it to its neighbors. Before propagating the new route, it modifies it according to a per-neighbor policy (*e.g.*, increasing the path length and attaching a tag to a BGP route). If all nodes in a network repeat this process long enough, the network should converge to a *stable state*, where no router may improve its routes by continuing to exchange messages with its neighbors.[2] While bad behavior can occur during the transient states prior to convergence, stable states persist and so do any errors in them. Consequently, network operators we have spoken with are most interested in verifying properties of stable states, which is the focus of this paper.

Figure 1 presents a simple network model. Part (a) presents the topology of the network pictorially. Part (b) presents a ProbNV model. This model defines the topology of the network (lines 1-2), the type of messages/routes circulated (line 3) by the protocol, and three functions, `init`, `trans`, and `merge` (lines 4-14) that define the semantics of the protocol and its configuration. In this case, the routing messages are simpler integer options—`None` represents "no route" to a destination and `Some len` represents a route of length `len` to a destination. The function `init` defines the initial state of the routing system at each node `u`, the function `trans` defines the transformation of a routing message `x` across an edge `e` (encompassing both the export policy of the sending router and the import policy of the receiving router), and the function `merge` defines the computation at router `u` that selects the preferred route (given options `x` and `y` from a pair of neighbors) to a destination.

While the details of the functions used to compute routes are relatively unimportant, two features of the language are critical: (1) `init`, `trans` and `merge` functions are non-recursive, and (2) all data types have finite cardinality.[3] Because data types are finitary and there is no recursion, ProbNV does not need to resort to bounded simulation techniques. Simulation does involve a top-level loop, which, if presented with an adversarial example, may not converge. However, as mentioned earlier, this appears rare in practice and we have yet to encounter a non-convergent network.

Part (c) of Figure 1 extends the protocol model with a probabilistic *environment model.* The environment model can capture unknowns a network operator may wish to reason about. In this particular example, we have developed a failure model for the network. Here, we define a

[2]Route propagation does not always terminate. However, in practice, within a network, non-termination appears rare. In this paper, we focus exclusively on networks that converge to a stable state.
[3]The type `int` is a 32-bit integer; types for integers of any bitwidth are available to users. Precise control bitwidth enables space-efficient representations of network data.

distribution `d` over boolean values and we declare `f1`-`f3` to be random variables drawn from `d`. Each variable represents the independent probability a router will fail. Next, we create refined implementations of the protocol (`initF`, `transF` and `mergeF`) to represent the semantics of the protocol operating on a faulty network. The interesting case is the transfer function `transF`. To propagate a message along a link from `u` to `v`, `transF` checks to see whether the corresponding failure bit has been turned on (via the function `fNode`). If it has, no route (`None`) is transmitted; if it has not, the protocol acts as it did originally. Other kinds of failures, such as link failures or shared-risk link groups can be programmed using such mechanisms.

## 2.2 Efficient Symbolic Simulation

Past control plane simulators, such as Batfish [Fogel et al. 2015], NV [Giannarakis et al. 2020] and FastPlane [Lopes and Rybalchenko 2019] have operated over concrete networks.[4] ProbNV differs by making it possible to simulate models that include symbolic and probabilistic hypotheses. While other functional languages, such as Rosette [Torlak and Bodik 2013], also include symbolic values and facilities for symbolic execution, network models have unusual properties that make domain-specific simulation techniques more effective than the usual symbolic execution techniques. In particular, network models have:

(1) high branching factors (*e.g.*, conditionals with many branches because there are many different possible destinations) [Beckett et al. 2019a; Giannarakis et al. 2020], and
(2) many symmetries (*e.g.*, topologies used in datacenters are highly symmetric in part to tolerate failures; and many destinations may share common processing components and generate similar routes) [Beckett et al. 2018; Giannarakis et al. 2019, 2020; Plotkin et al. 2016]
(3) a large number of simple, repeated computations.

An efficient network simulator must be able to process code with a high branching factor, represent symmetric data compactly, and execute the many simple computations rapidly. To achieve these goals, ProbNV must carefully choose how it represents each piece of data. Indeed, it shifts representation between different bits to optimize simulation performance.

Like many model checkers in the past, ProbNV uses BDDs [Bryant 1986]—a compact representations of finite sets (hence the restrictions to finite data in ProbNV)—to represent symbolic values, including, for instance, the symbolic booleans `f1`-`f3` that appear in fig. 1; we visualize their representation as BDDs in fig. 2a. Similarly, BDDs are used to model more complex symbolic values such as nodes or edges, that might appear in more sophisticated models.

The routes generated by our protocols could also be represented as BDDs, if we chose to. However, routes are often involved in some not-completely-trivial computations. Even in the very simple model of fig. 1, we must increment the length of the route across each edge in the network. It is possible to implement operations such as addition over BDD representations of (optional) integers —one must simply code up a bitwise adder over BDDs. However, such additions occur thousands of times throughout network simulation. Using bitwise operations implemented over BDDs in software rather than single-cycle hardware additions is a substantial drag on performance. We would like to leave such values, in their natural, concrete representation so as to execute hardware operations over them. Indeed, a *concrete* representation may be used for several values in our simulator (though, as we will see in a moment, it is *not* used for route computation!).

[4]NV models can contain symbolic values, but it was not possible to simulate those models. They could only be analyzed via an SMT back-end to NV that was orders of magnitude slower than the simulator.

Examination of the `transF` function (fig. 1c), illustrates the problem with using a concrete representation of routes. In `transF`, the computation of a route *depends upon* the symbolic computation `fNode e.1 || fNode e.2`. Here, `transF` over the edge `(0,1)` corresponds to the symbolic computation `false || f1`, which is true iff `f1` is true, a fact represented as a BDD (see fig. 2b).

If that expression evaluates to true then the route `None` is produced and otherwise the route `trans e x` is produced. To represent such dependent computations, we use multi-terminal BDDs (MTBDDs). Figure 2c shows the MTBDD that captures the computation of the transfer func tion described above over the edge `(0,1)`. Computations over the leaves (such as addition) will use efficient hardware rather than software encodings of such operations. If, thanks to symmetry in network design, several dependencies lead to a small number of possible route values, MTBDDs will contain one terminal node for each possible route value. For example, consider the route that node `R4` will converge to based on the messages it receives from `R1` and `R2`. Since in our model `R4` cannot fail, it will receive a route of length 2 if `f1` is false (similar to the route shown in fig. 2c), and another route of length 2 if `f2` is false. When merging these two messages, operating on top of the MTBDDs that represent them, `R4` considers all possibilities; if `f1` is false and `f2` is true, the best route is the route of length 2. Likewise when both `f1` and `f2` are false, or if `f1` is true and `f2` is false. MTBDDs compactly represent the routes of `R4` in all of these failure scenarios, as shown in fig. 2d.

As our running example illustrates, ProbNV uses three different representations of data. Moreover, the choice to represent one bit of data one way (e.g. as a symbolic value) has a trickle-down effect on the way other data that depends on it is also represented. To keep these varying representations and dependencies straight, we use a type system internally to classify data structures and computations as concrete, symbolic/BDD or multi-value/MTBDD. This type system ensures that computations expecting to process symbolics do not wind up processing concrete values instead (or vice versa).

In addition to defining a type system for our intermediate language, we also define an elaboration procedure, which transforms well-typed terms of our intermediate language into well-typed terms of a conventional lambda calculus. Though it only applies to well-typed programs, it suffices to encode the network models we have developed, which include a variety of benchmarks generated automatically from raw industrial CISCO configurations.

## 2.3 Processing Probabilistic Models

ProbNV is carefully designed to make processing probabilistic models just as easy and flexible for users, and almost as efficient, as processing deterministic models. For instance, given the model in fig. 1, a user may want to know the probability with which `R4` will be disconnected from `R0`.

To answer that question, ProbNV will symbolically simulate the network model, ignoring probabilistic information. Once a set of MTBDDs representing the dependence between symbolic values and final routes has been computed, those MTBDDs may be analyzed with the distributions assigned to each symbolic value in hand. Doing so involves a traversal of the MTBDD structure to count paths through the MTBDD and assign probabilities to the terminal values. Separating probabilistic reasoning from route computation allows ProbNV to scale to larger and more complex networks, unlike (more expressive) systems such as Bayonet [Gehr et al. 2018], which mix probabilistic inference with network-related computations.

## 3 THE PROBNV LANGUAGE

The surface syntax of ProbNV (fig. 3) is based on NV [Giannarakis et al. 2020], a functional language designed to build models of routing protocols using conventional constructs such as fixed-width integers, booleans, tuples, dictionaries, and (non-recursive) functions. The full language includes features to make it more user friendly, such as records, and optional values, but we omit them from our formalization for brevity. Values in ProbNV's syntax include node values (we write `vn` for

$$
\begin{array}{lll}
\text{(Values)} & \mathsf{v} & ::= \mathbb{N}u\mathbb{N} \mid \mathsf{true} \mid \mathsf{false} \mid (\mathsf{v}_1,\ldots,\mathsf{v}_n) \mid \mathbb{N}n \mid \mathbb{N}{\sim}\mathbb{N} \\
\text{(Patterns)} & \mathsf{g} & ::= \mathbb{N}u\mathbb{N} \mid \mathsf{true} \mid \mathsf{false} \mid \mathbb{N}n \mid \mathbb{N}{\sim}\mathbb{N} \mid (\mathsf{g}_1,\ldots,\mathsf{g}_n) \mid \_ \\
\text{(Expressions)} & \mathsf{e} & ::= \mathsf{v} \mid \mathsf{x} \mid \mathsf{let}\ \mathsf{x} : \mathsf{ty} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 \mid \mathsf{fun}\ (\mathsf{x} : \tau) \rightarrow \mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid (\mathsf{e}_1,\ldots,\mathsf{e}_n) \\
 & & \quad \mid \mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ \mathsf{e}_2\ \mathsf{else}\ \mathsf{e}_3 \mid e_1\ \mathsf{op}\ e_2 \mid \mathsf{not}\ \mathsf{e}_1 \mid \pi_{\mathsf{i}}\ \mathsf{e}_1 \mid [\_ := \mathsf{e}_1] \\
 & & \quad \mid \mathsf{e}_1.[\mathsf{e}_2] \mid \mathsf{e}_1.[\mathsf{e}_2 := \mathsf{e}_3] \mid (\mathsf{match}\ \mathsf{e}_0\ \mathsf{with} \mid \mathsf{g}_1 \rightarrow \mathsf{e}_1 \ldots \mid \mathsf{g}_n \rightarrow \mathsf{e}_n) \\
\text{(Operations)} & \mathsf{op} & ::= + \mid = \mid < \mid = \mid \&\& \mid || \\
\text{(Probabilities)} & \mathsf{p} & ::= \mathbb{R}^{\leq 1}_{\geq 0} \\
\text{(Cases)} & \mathsf{c} & ::= \mathsf{true} \mid \mathsf{false} \mid [\mathbb{N},\mathbb{N}] \mid \mathbb{N}n \mid \mathbb{N}{\sim}\mathbb{N} \mid \_ \\
\text{(Distributions)} & \mathcal{D} & ::= (\mid \mathsf{c}_1 \rightarrow \mathsf{p}_1 \ldots \mid \mathsf{c}_n \rightarrow \mathsf{p}_n) \\
\text{(Declarations)} & \mathsf{d} & ::= \mathsf{symbolic}\ \mathsf{x} : \tau \mid \mathsf{symbolic}\ \mathsf{x} : \tau = \mathcal{D} \mid \mathsf{let}\ \mathsf{x} : \tau = \mathsf{e} \\
 & & \quad \mid \mathsf{let}\ \mathsf{x} : \tau = \mathsf{solve}(\mathsf{e}_1,\mathsf{e}_2,\mathsf{e}_3) \mid \mathsf{assert}(\mathsf{e}_1 \mid \mathsf{e}_2) \\
 & & \quad \mid \mathsf{let\ nodes} = \mathbb{N}^+ \mid \mathsf{let\ edges} = \{(\mathbb{N}^+ - \mathbb{N}^+)\} \\
\text{(Modes)} & \mathsf{m} & ::= \mathrm{C} \mid \mathrm{S} \mid \mathrm{M} \\
\text{(Base Types)} & \mathsf{b} & ::= \mathsf{int}\mathbb{N}^+ \mid \mathsf{bool} \mid \mathsf{tnode} \mid \mathsf{tedge} \mid \tau_1 \rightarrow \tau_2 \mid (\tau_1,\ldots,\tau_n) \mid \mathsf{dict}(\tau_1,\tau_2) \\
\text{(Types)} & \tau & ::= [\mathsf{m}]\mathsf{b}
\end{array}
$$

Fig. 3. ProbNV core syntax of expressions, declarations and types.

node v), edge values (we write $\mathsf{u} \sim \mathsf{v}$ for a directed edge from node u to node v), booleans, finite integers (we write kuN for an integer k consisting of N bits, and k for a 32-bit integer) and tuples. The dictionary datastructure supports operations such as creation of a new dictionary where all keys map to a value v ($[\_ := \mathsf{v}]$), retrieval of the value in map $\mathsf{e}_1$ at key $\mathsf{e}_2$ ($\mathsf{e}_1.[\mathsf{e}_2]$) and creation of a new map from $\mathsf{e}_1$ updated so that $\mathsf{e}_2$ maps to $\mathsf{e}_3$ ($\mathsf{e}_1.[\mathsf{e}_2 := \mathsf{e}_3]$). Besides using dictionaries to model routing features, we also use them to represent the solutions of routing algebras (*i.e.*, mapping nodes to routes).

A ProbNV program may define multiple routing algebras, binding their solutions to a variable using a solve declaration: `let x = solve(init,trans,merge)`, where `init`, `trans`, `merge` are the functions defining the routing algebra. This allows us to model more complex systems where a routing algebra depends on another routing algebra. For instance, one could define a routing algebra to describe traffic flowing using the routes computed by another routing algebra, or to model protocols such as iBGP, where route computation depends on routes to other destinations.

ProbNV also includes syntactic constructs to support probabilistic reasoning. When extending the language with a probabilistic fragment, we opted to retain compatibility with the existing deterministic models and analysis techniques. To achieve this, we avoid probabilistic operators, such as probabilistic choice or test operations that are probabilistic.

In our probabilistic extension, the expression language remains unchanged; however, declarations of symbolics include a probability distribution, by default, a uniform distribution over the values admitted by the type of the symbolic. Users can skew the distribution by assigning a probability to each possible value that the symbolic may represent; naturally, the sum of all cases should be 1.0. Defining each point in the distribution is cumbersome for large state spaces (*e.g.*, integers); thus, the language includes some constructs to simplify this process. For symbolic integers, users can define the probability of intervals; this probability is distributed uniformly within the interval. We require that the intervals are non-overlapping. Users may also utilize a "catch-all" case that matches any value not captured by a previous case. The following example illustrates these points:

```
symbolic x : int =
  | [10,100] → 0.5
```

```
                  | [150,200] → 0
                  | _ → 0.5
```

The probability that $10 \leq x \leq 100$ is 0.5, hence the probability that $x = n$ for any $n$ in this interval is $\frac{1}{100-10+1} \times 0.5 = \frac{1}{91\times 2} \approx 0.005$. The second case denotes that the probability of every integer in the interval [150, 200] is zero. Finally, the "catch-all" case distributes the remaining 0.5, among all other integers, *i.e.*, the probability that $\mathtt{x} = n$ for $n \notin [10, 100] \cup [150, 200]$ is $\frac{1}{2^{32}-(91+51)} \times 0.5 \approx 0.23 \times 10^{-9}$.

Assertions in ProbNV are probabilistic. An assertion `assert(b, c)` computes the probability that `b` is true given that the boolean expression `c` is true, that is $P(b \mid c)$. If the asserted expression depends on symbolic values, their distribution impacts the probability that the assertion holds.

The language of distributions and probabilistic assertions is relatively simple and while one can easily imagine extensions the current system fits our existing use cases, which are oriented around proving fault tolerance properties (key properties of interest to network operators).

## 3.1 ProbNV Type System

We extend a traditional type system for the simply-typed lambda calculus to distinguish between different representations of values and different modes of computation. We write $[\mathrm{C}]\tau$ to denote the *concrete* representation of type $\tau$, where values represented as usual; $[\mathrm{S}]\tau$ to denote the *symbolic* representation of type $\tau$, where value are represented as BDDs; and $[\mathrm{M}]\tau$ to denote the *multi-value* representation of type $\tau$ where the set of concrete values of type $\tau$ depend on symbolic values. For instance, a typing judgment of the form $\Gamma \vdash \mathtt{e} : [\mathrm{S}]\tau$ means that the expression e will compute a symbolic value of type $\tau$. Note that some types do not admit a symbolic or multi-value mode. For symbolics, this restriction stems from the fact that values need to be encoded as a series of binary decisions. A symbolic integer `symbolic x : [S]int8` requires 8 binary decisions, one for each bit and is easy to encode. In principle, one can also encode symbolic functions with type `[S]([C]int8 → [C]int8)`, but in practice we have not needed to and doing so would be extremely costly so ProbNV does not support symbolic functions or symbolic dictionaries. ProbNV's type system allows multi-value functions to be constructed only through a partial application of a concrete function on a multi-value argument (rule App-M of fig. 5). Despite this, our implementation avoids constructing multi-value functions. This is because values stored in the leaves of MTBDDs must admit a computable equality, and functional values in our system (which at the implementation level are represented as OCaml functions) cannot be tested for equality. We overcome this limitation by inlining all function applications that involve multi-value functions.

Symbolic base types (integer, booleans, nodes, edges) as well as symbolic tuples are supported. Note that ProbNV always represents a tuple and all of its components uniformly. If a tuple is symbolic, then both of its components are symbolic, for instance. Moreover, values with type `[M]([M]b, [M]b)` and `[M]([C]b, [M]b)` are both represented the same way—as MTBDDs with tuples of $b$ at the leaves (they are not represented as nested MTBDDs).

Predicates `canLiftS` and `canLiftM` specify the set of types that can be represented symbolically and as multi-values respectively (see the appendix [Giannarakis et al. 2021] for definitions).

The goal of the type system is to separate the different computation modes, but it would be very restrictive if we could not eventually mix values of different modes. Intuitively, though there is a cost, we can always *lift* concrete computation/values to use them in a symbolic or a multi-value computation. Formally we define a join-relation on modes (and subsequently, on types):

$$\mathtt{m}_1 \sqcup \mathtt{m}_2 = \begin{cases} \mathrm{C} & \text{if } \mathtt{m}_1 = \mathtt{m}_2 = \mathrm{C} \\ \mathrm{S} & \text{if } (\mathtt{m}_1 = \mathrm{S} \vee \mathtt{m}_2 = \mathrm{S}) \wedge \mathtt{m}_1, \mathtt{m}_2 \in \{\mathrm{C}, \mathrm{S}\} \\ \mathrm{M} & \text{if } (\mathtt{m}_1 = \mathrm{M} \vee \mathtt{m}_2 = \mathrm{M}) \wedge \mathtt{m}_1, \mathtt{m}_2 \in \{\mathrm{C}, \mathrm{M}\} \end{cases}$$

$$\frac{\mathtt{ty} \in \{\mathtt{int}, \mathtt{bool}, \mathtt{tnode}, \mathtt{tedge}\}}{[\mathtt{m}_1]\mathtt{ty} \sqcup [\mathtt{m}_2]\mathtt{ty} = [\mathtt{m}_1 \sqcup \mathtt{m}_2]\mathtt{ty}} \textsc{Join-Base}$$

$$\frac{[\mathtt{m}_{1i}]\mathtt{ty}_{1i} \sqcup [\mathtt{m}_{2i}]\mathtt{ty}_{2i} = [\mathtt{m}_{3i}]\mathtt{ty}_{3i}}{[\mathtt{m}_1]([\mathtt{m}_{11}]\mathtt{ty}_{11}, \ldots, [\mathtt{m}_{1n}]\mathtt{ty}_{1n}) \sqcup [\mathtt{m}_2]([\mathtt{m}_{21}]\mathtt{ty}_{21}, \ldots, [\mathtt{m}_{2n}]\mathtt{ty}_{2n}) = [\mathtt{m}_1 \sqcup \mathtt{m}_2]([\mathtt{m}_{31}]\mathtt{ty}_{31}, \ldots, [\mathtt{m}_{3n}]\mathtt{ty}_{3n})} \textsc{Join-Tup}$$

$$\frac{[\mathtt{m}_1]([\mathtt{m}_3]\mathtt{ty}_3 \to [\mathtt{m}_4]\mathtt{ty}_4) = [\mathtt{m}_2]([\mathtt{m}_5]\mathtt{ty}_5 \to [\mathtt{m}_6]\mathtt{ty}_6)}{[\mathtt{m}_1]([\mathtt{m}_3]\mathtt{ty}_3 \to [\mathtt{m}_4]\mathtt{ty}_4) \sqcup [\mathtt{m}_2]([\mathtt{m}_5]\mathtt{ty}_5 \to [\mathtt{m}_6]\mathtt{ty}_6) = [\mathtt{m}_1]([\mathtt{m}_3]\mathtt{ty}_3 \to [\mathtt{m}_4]\mathtt{ty}_4)} \textsc{Join-Arrow}$$

$$\frac{\mathtt{dict}([\mathtt{m}_3]\mathtt{ty}_3, [\mathtt{m}_4]\mathtt{ty}_4) = \mathtt{dict}([\mathtt{m}_5]\mathtt{ty}_5, [\mathtt{m}_6]\mathtt{ty}_6)}{[\mathtt{m}_1](\mathtt{dict}([\mathtt{m}_3]\mathtt{ty}_3, [\mathtt{m}_4]\mathtt{ty}_4)) \sqcup [\mathtt{m}_2](\mathtt{dict}([\mathtt{m}_5]\mathtt{ty}_5, [\mathtt{m}_6]\mathtt{ty}_6)) = [\mathtt{m}_1 \sqcup \mathtt{m}_2](\mathtt{dict}([\mathtt{m}_3]\mathtt{ty}_3, [\mathtt{m}_4]\mathtt{ty}_4))} \textsc{Join-Map}$$

Fig. 4. Lifting the join relation over types.

Figures 5 and 6 show some of the typing rules[5] for the surface language. In what follows, we comment on the ones that illustrate key design choices.

- The rule for tuples (Tuple) dictates that the computation mode of a tuple is equal to the join of the modes of its components, *e.g.*, if a component of the tuple has a multi-value type, then the whole computation is lifted to a multi-value computation (as mentioned earlier, the tuple and its components are treated uniformly). Another implication of this decision is that if one component is symbolic (or multi-value), all components need to be of a type compatible with these modes (according to `canLiftS` and `canLiftM`); a tuple containing a symbolic boolean and a function is not well-typed as symbolic functions are not supported.
- Rule Ite-C applies when the guard of the conditional is a concrete value. In this case the join of the branches dictates the representation of the value returned. For example, if one of the branches is concrete and the other symbolic, then the if-then-else expression returns a symbolic. At first glance, it might seem peculiar that the types of the branches ($\mathtt{ty}_2$ and $\mathtt{ty}_3$) are not equal. The typing of tuples motivates why syntactic equality is too weak to be practical; the types [S]([C]int, [S]int) and [S]([S]int, [C]int) are not syntactically equal, yet they both represent a symbolic tuple of two integers. The join-relation on types (fig. 4) ensures that the two branches are equal modulo such inconsequential differences in modes.
- Ite-M introduces multi-value computations to a program. In this case, the guard is either a symbolic or a multi-value, and as such, the result of the expression depends on a symbolic condition, either directly or indirectly. If the branches are concrete or multi-values, the result is a collection of concrete values guarded by symbolics, *i.e.*, another multi-value. The `liftTy` function ensures that the result type is well-formed after lifting it to multi-value mode: if the type of the branches is [C]([C]int, [C]int), then simply returning [M]ty would create a type of the form [M]([C]int, [C]int) which, in our formalization, is not a valid tuple type; instead `liftTy`([M]([C]int, [C]int)) produces [M]([M]int, [M]int).
- The last rule for if-then-else expressions, Ite-S, applies when the if-then-else expression represents a symbolic computation; the guard is symbolic, and the branches are either concrete or symbolic. Note that if both branches are concrete, both Ite-S and Ite-M may apply. One needs to infer the correct rule from the context; our type inference implementation currently defaults to applying Ite-M and requires the user to add an annotation otherwise.
- The application rule App-C applies to functions that expect a concrete argument. The applied argument can be of either concrete or multi-value type. This might seem counterintuitive, but a multi-value is a set of guarded concrete values; a computation over a concrete argument

[5]See the appendix [Giannarakis et al. 2021] for a complete presentation.

$$\dfrac{(\mathtt{x} : [\mathtt{m}_1]\mathtt{ty}) \in \Gamma}{\Gamma \vdash \mathtt{x} : [\mathtt{m}_1]\mathtt{ty}}\ \textsc{Var} \qquad \dfrac{\mathtt{b} \in \{\mathtt{true}, \mathtt{false}\}}{\Gamma \vdash \mathtt{b} : [\mathrm{C}]\mathrm{bool}}\ \textsc{Bool} \qquad \dfrac{\mathtt{n} \in \{0, \ldots\}}{\Gamma \vdash \mathtt{n} : [\mathrm{C}]\mathrm{int}}\ \textsc{Int}$$

$$\dfrac{\Gamma \vdash \mathtt{e}_1 : [\mathrm{C}]\mathrm{bool} \qquad \Gamma \vdash \mathtt{e}_2 : [\mathtt{m}_2]\mathtt{ty}_2 \qquad \Gamma \vdash \mathtt{e}_3 : [\mathtt{m}_3]\mathtt{ty}_3 \qquad [\mathtt{m}]\mathtt{ty} = [\mathtt{m}_2]\mathtt{ty}_2 \sqcup [\mathtt{m}_3]\mathtt{ty}_3}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : [\mathtt{m}]\mathtt{ty}}\ \textsc{Ite-C}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : [\mathtt{m}_1]\mathtt{bool} \qquad \Gamma \vdash \mathtt{e}_2 : [\mathtt{m}_2]\mathtt{ty}_2 \qquad \Gamma \vdash \mathtt{e}_3 : [\mathtt{m}_3]\mathtt{ty}_3 \\ [\mathtt{m}]\mathtt{ty} = [\mathtt{m}_2]\mathtt{ty}_2 \sqcup [\mathtt{m}_3]\mathtt{ty}_3 \qquad \mathtt{m}_1 \in \{\mathrm{S}, \mathrm{M}\} \qquad \mathtt{m} \in \{\mathrm{C}, \mathrm{M}\} \qquad \mathtt{canLiftM}\ \mathtt{ty}\end{array}}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{liftTy}([\mathrm{M}]\mathtt{ty})}\ \textsc{Ite-M}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : [\mathrm{S}]\mathrm{bool} \qquad \Gamma \vdash \mathtt{e}_2 : [\mathtt{m}_2]\mathtt{ty}_2 \qquad \Gamma \vdash \mathtt{e}_3 : [\mathtt{m}_3]\mathtt{ty}_3 \\ [\mathtt{m}]\mathtt{ty} = [\mathtt{m}_2]\mathtt{ty}_2 \sqcup [\mathtt{m}_3]\mathtt{ty}_3 \qquad \mathtt{m} \in \{\mathrm{S}, \mathrm{C}\} \qquad \mathtt{canLiftS}\ \mathtt{ty}\end{array}}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{liftTy}([\mathrm{S}]\mathtt{ty})}\ \textsc{Ite-S}$$

$$\dfrac{\Gamma, \mathtt{x} : [\mathtt{m}_1]\mathtt{ty}_1 \vdash \mathtt{e} : [\mathtt{m}_2]\mathtt{ty}_2}{\Gamma \vdash \mathtt{fun}\ \mathtt{x} \rightarrow \mathtt{e} : [\mathrm{C}]([\mathtt{m}_1]\mathtt{ty}_1 \rightarrow [\mathtt{m}_2]\mathtt{ty}_2)}\ \textsc{Lambda-C}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : [\mathrm{C}]([\mathrm{C}]\mathtt{ty}_1 \rightarrow [\mathtt{m}_2]\mathtt{ty}_2) \qquad \Gamma \vdash \mathtt{e}_2 : [\mathtt{m}_3]\mathtt{ty}_3 \qquad [\mathrm{C}]\mathtt{ty}_1 \sqcup [\mathtt{m}_3]\mathtt{ty}_3 \\ \mathtt{m}_3 \in \{\mathrm{C}, \mathrm{M}\} \qquad \mathtt{m} = \mathtt{m}_2 \sqcup \mathtt{m}_3\end{array}}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{liftTy}([\mathtt{m}]\mathtt{ty}_2)}\ \textsc{App-C}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : [\mathrm{M}]([\mathtt{m}_1]\mathtt{ty}_1 \rightarrow [\mathtt{m}_2]\mathtt{ty}_2) \\ \Gamma \vdash \mathtt{e}_2 : [\mathtt{m}_3]\mathtt{ty}_3 \qquad [\mathtt{m}_1]\mathtt{ty}_1 \sqcup [\mathtt{m}_3]\mathtt{ty}_3 \qquad \mathtt{m}_2 \in \{\mathrm{C}, \mathrm{M}\} \qquad \mathtt{m}_1 = \mathrm{C} \Rightarrow \mathtt{m}_3 \in \{\mathrm{C}, \mathrm{M}\}\end{array}}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{liftTy}([\mathrm{M}]\mathtt{ty}_2)}\ \textsc{App-M}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : [\mathtt{m}_1]\mathtt{ty}_1 \qquad \ldots \qquad \Gamma \vdash \mathtt{e}_n : [\mathtt{m}_n]\mathtt{ty}_n \qquad \mathtt{m} = \mathtt{m}_1 \sqcup \ldots \sqcup \mathtt{m}_n \\ \mathtt{m} = \mathrm{M} \Rightarrow \forall i \in \{1..n\}.\ \mathtt{canLiftM}\ [\mathtt{m}_i]\mathtt{ty}_i \qquad \mathtt{m} = \mathrm{S} \Rightarrow \forall i \in \{1..n\}.\ \mathtt{canLiftS}\ [\mathtt{m}_i]\mathtt{ty}_i\end{array}}{\Gamma \vdash (\mathtt{e}_1, \ldots, \mathtt{e}_n) : [\mathtt{m}]([\mathtt{m}_1]\mathtt{ty}_1, \ldots, [\mathtt{m}_n]\mathtt{ty}_n)}\ \textsc{Tuple}$$

$$\dfrac{\Gamma \vdash \mathtt{e}_1 : [\mathtt{m}_1]\mathtt{ty}_2 \qquad \mathtt{m}_1 \in \{\mathrm{C}, \mathrm{M}\} \qquad \mathtt{canLiftM}\ \mathtt{ty}_1 \qquad \mathtt{canLiftM}\ \mathtt{ty}_2}{\Gamma \vdash [\_ := \mathtt{e}_1] : [\mathtt{m}_1](\mathtt{dict}([\mathrm{C}]\mathtt{ty}_1, [\mathrm{C}]\mathtt{ty}_2))}\ \textsc{Map-Create}$$

$$\dfrac{\Gamma \vdash \mathtt{e}_1 : [\mathrm{C}](\mathtt{dict}([\mathrm{C}]\mathtt{ty}_1, [\mathrm{M}]\mathtt{ty}_2)) \qquad \Gamma \vdash \mathtt{e}_2 : [\mathrm{C}]\mathtt{ty}_1}{\Gamma \vdash \mathtt{e}_1.[\mathtt{e}_2] : [\mathrm{M}]\mathtt{ty}_2}\ \textsc{Map-Get-M}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : [\mathtt{m}_1](\mathtt{dict}([\mathrm{C}]\mathtt{ty}_1, [\mathrm{C}]\mathtt{ty}_2)) \qquad \Gamma \vdash \mathtt{e}_2 : [\mathtt{m}_3]\mathtt{ty}_3 \\ [\mathtt{m}]\mathtt{ty} = [\mathtt{m}_1]\mathtt{ty}_1 \sqcup [\mathtt{m}_3]\mathtt{ty}_3 \qquad \mathtt{m} \in \{\mathrm{C}, \mathrm{M}\}\end{array}}{\Gamma \vdash \mathtt{e}_1.[\mathtt{e}_2] : \mathtt{liftTy}([\mathtt{m}]\mathtt{ty}_2)}\ \textsc{Map-Get-C}$$

Fig. 5. Typing rules for ProbNV expressions.

can be lifted over that set, resulting in a new multi-value. Of course, this is only possible if the argument's mode and the mode of the return type are ⊔-compatible, *i.e.*, if the function computes a symbolic value, it would not be possible to use a multi-value as an argument.

- Rule App-M applies to functions typed as multi-values. As we previously mentioned, such functions stem through applications of the rule App-C. Consider the following example:

```
let f : [C]([C]int → [C]([C]int → [C]int)) = fun x y → x + y
let g : [M]([C]int → [C]int) = f (v : [M]int)
let u : [M]int = g 0
```

$$\frac{\Gamma, \mathsf{x} : [\mathsf{S}]\mathtt{ty}_1 \vdash \mathsf{p} : [\mathsf{m}]\mathtt{ty}_2 \qquad \mathtt{canLiftS}\ [\mathsf{S}]\mathtt{ty}_1}{\Gamma \vdash \mathsf{symbolic}\ (\mathsf{x} : [\mathsf{S}]\mathtt{ty}_1)\ ;;\ \mathsf{p} : [\mathsf{m}]\mathtt{ty}_2}\ \textsc{Symbolic}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{fun}\ \mathsf{u} \to \mathsf{e}_{\mathtt{init}} : [\mathsf{C}]\mathsf{tnode} \to [\mathsf{m}_1]\mathtt{ty}_1 \\ \Gamma \vdash \mathsf{fun}\ \mathsf{e}\ \mathsf{x} \to \mathsf{e}_{\mathtt{trans}} : [\mathsf{C}]\mathsf{tedge} \to [\mathsf{m}]\mathtt{ty} \to [\mathsf{m}_2]\mathtt{ty}_2 \\ \Gamma \vdash \mathsf{fun}\ \mathsf{u}\ \mathsf{x}\ \mathsf{y} \to \mathsf{e}_{\mathtt{merge}} : [\mathsf{C}]\mathsf{tnode} \to [\mathsf{m}]\mathtt{ty} \to [\mathsf{m}]\mathtt{ty} \to [\mathsf{m}_3]\mathtt{ty}_3 \qquad \mathtt{canLiftM}\ [\mathsf{m}]\mathtt{ty} \\ \Gamma, \mathsf{x} : [\mathsf{C}](\mathsf{dict}([\mathsf{C}]\mathsf{tnode}, [\mathsf{m}]\mathtt{ty})) \vdash \mathsf{p} : [\mathsf{m}_4]\mathtt{ty}_4 \qquad [\mathsf{m}]\mathtt{ty} = [\mathsf{m}_1]\mathtt{ty}_1 \sqcup [\mathsf{m}_2]\mathtt{ty}_2 \sqcup [\mathsf{m}_3]\mathtt{ty}_3\end{array}}{\Gamma \vdash \mathsf{let}\ \mathsf{x} = \mathsf{solution}(\mathsf{fun}\ \mathsf{u} \to \mathsf{e}_{\mathtt{init}}, \mathsf{fun}\ \mathsf{e}\ \mathsf{x} \to \mathsf{e}_{\mathtt{trans}}, \mathsf{fun}\ \mathsf{u}\ \mathsf{x}\ \mathsf{y} \to \mathsf{e}_{\mathtt{merge}})\ ;;\ \mathsf{p} : [\mathsf{m}_4]\mathtt{ty}_4}\ \textsc{Solution}$$

Fig. 6. Typing rules for ProbNV declarations.

$$\frac{\Gamma \vdash \mathsf{e}_1 : [\mathsf{S}]\mathsf{bool} \qquad \Gamma \vdash \mathsf{e}_2 : [\mathsf{S}]\mathtt{ty} \qquad \Gamma \vdash \mathsf{e}_3 : [\mathsf{S}]\mathtt{ty}}{\Gamma \vdash \mathtt{bddIf}\ \mathsf{e}_1\ \mathtt{then}\ \mathsf{e}_2\ \mathtt{else}\ \mathsf{e}_3 : [\mathsf{S}]\mathtt{ty}}\ \textsc{Ite-S} \qquad \frac{\Gamma \vdash \mathsf{e}_1 : [\mathsf{S}]\mathtt{ty} \qquad \Gamma \vdash \mathsf{e}_2 : [\mathsf{S}]\mathtt{ty}}{\Gamma \vdash \mathsf{e}_1 \diamond_b \mathsf{e}_2 : [\mathsf{S}]\mathtt{ty}}\ \textsc{Bop-S}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : [\mathsf{C}]\mathtt{ty} \qquad \mathtt{canLiftS}\ [\mathsf{C}]\mathtt{ty}}{\Gamma \vdash \mathsf{toBdd}\ \mathsf{e}_1 : \mathtt{liftTy}([\mathsf{S}]\mathtt{ty})}\ \textsc{ToBdd} \qquad \frac{\Gamma \vdash \mathsf{e}_1 : [\mathsf{C}]\mathtt{ty} \qquad \mathtt{canLiftM}\ [\mathsf{C}]\mathtt{ty}}{\Gamma \vdash \mathsf{toMulti}\ \mathsf{e}_1 : \mathtt{liftTy}([\mathsf{M}]\mathtt{ty})}\ \textsc{ToMulti}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : [\mathsf{m}_1]\mathtt{ty}_1 \ \dots\ \Gamma \vdash \mathsf{e}_n : [\mathsf{m}_n]\mathtt{ty}_n \qquad m_i \in \{\mathsf{M}, \mathsf{S}\} \\ \Gamma \vdash \mathsf{fun}\ \mathsf{x}_1 \dots \mathsf{x}_n \to \mathsf{e} : \mathtt{liftTy}([\mathsf{C}]\mathtt{ty}_1) \to \dots \mathtt{liftTy}([\mathsf{C}]\mathtt{ty}_n) \to [\mathsf{C}]\mathtt{ty}\end{array}}{\Gamma \vdash \mathsf{apply}((\mathsf{fun}\ \mathsf{x}_1 \dots \mathsf{x}_n \to \mathsf{e}), \mathsf{e}_1, \dots, \mathsf{e}_n) : \mathtt{liftTy}([\mathsf{M}]\mathtt{ty})}\ \textsc{Apply}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : [\mathsf{m}_1]\mathtt{ty}_1 \to [\mathsf{m}_2]\mathtt{ty}_2 \qquad \Gamma \vdash \mathsf{e}_2 : [\mathsf{m}_1]\mathtt{ty}_1}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 : [\mathsf{m}_2]\mathtt{ty}_2}\ \textsc{App} \qquad \frac{\Gamma \vdash \mathsf{e}_1 : [\mathsf{m}]\mathtt{ty}_1 \quad \dots \quad \Gamma \vdash \mathsf{e}_n : [\mathsf{m}]\mathtt{ty}_n}{\Gamma \vdash (\mathsf{e}_1, \dots, \mathsf{e}_n) : [\mathsf{m}]([\mathsf{m}]\mathtt{ty}_1, \dots, [\mathsf{m}]\mathtt{ty}_n)}\ \textsc{Tuple}$$

Fig. 7. Typing rules for DDNV expressions. The rest of the type system follows conventional STLC rules.

According to rule App-C the type of g is $[\mathsf{M}]([\mathsf{C}]\mathtt{int} \to [\mathsf{C}]\mathtt{int})$. But, using App-C to also typecheck u would be problematic; we need to capture the fact that g has been partially applied to a multi-value argument and hence the resulting computation will also be a multi-value computation. Rule App-M captures precisely this fact.

- As Map-Create shows when creating a dictionary using a multi-value (or, when setting a multi-value key/value), we treat the dictionary as a multi-value that stores concrete entries. The only exception to this is the dictionaries that the `solve` primitive generates, as they may map concrete keys (nodes) to multi-values (routes). For the latter case, the user may only retrieve keys from the dictionary and not store new ones (see rule Map-Get-M).

## 4 EFFICIENT SYMBOLIC EXECUTION

In this section, we implement the different notions of computation of ProbNV. We do so in two steps. First, we define an extension to ProbNV's language, DDNV (for *Decision-Diagram NV*), adding BDD and MTBDD values, and operations over them. Subsequently, we define a translation from well-typed ProbNV programs to DDNV. We detail these steps in the next paragraphs.

### 4.1 Interpreting Symbolic Expressions

Binary Decision Diagrams (BDDs) [Bryant 1986] have enjoyed widespread adoption in the verification community thanks to their succinct representation of large sets of values and the (relatively) efficient implementation of operations between such sets. Network verification is no exception to this as prior work has demonstrated [Beckett et al. 2019a; Beckett and Mahajan 2020; Giannarakis et al. 2020; Smolka et al. 2019]. In section 2.2, we showed how we use BDDs to implement boolean symbolic values and operations over them. More complex types require more decision nodes to represent them; for n-bit integers we use n decision nodes, for options we use one bit to distinguish between the None and Some constructor plus as many bits required by the underlying type, and for a tuple, we use the decision nodes of its components. Nodes and edges are treated as finite integers.

DDNV extends ProbNV's language with explicit BDD operations:

$$\begin{array}{lll} \texttt{bop} & ::= & \mid +_b \mid =_b \mid <_b \mid \&\&_b \mid \,||_b \\ \texttt{e} & ::= & \mid \texttt{not}_b\ \texttt{e}_1 \mid \texttt{e}_1\ \texttt{bop}\ \texttt{e}_2 \mid \texttt{bddIf}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 \mid \texttt{toBdd}\ \texttt{e}_1 \end{array}$$

These include BDD variants of binary and unary operations, an if-then-else operation over BDDs, and a cast operation that converts a concrete value to a BDD. Figure 7 shows the typing rules for the new expressions found in DDNV. According to rule ToBdd, `toBdd` takes a concrete value as argument and produces a symbolic value. The only requirement is that values of the given type can be lifted to symbolic mode. Rules for `bddIf` and binary operations over BDDs are standard, with the only added requirement being that they operate over symbolic values.

To interpret basic boolean operations, such as conjunction and disjunction, over BDDs we rely on primitives implemented in CUDD [Somenzi 1997], a popular package for Binary Decision Diagrams. We build on these primitives to implement more complex operations; for instance, we implement addition over symbolic integers as a full adder circuit.

*Interpreting Multi-Value Expressions.* Multi-Terminal Binary Decision Diagrams (MTBDDs) are similar to BDDs, but terminal nodes can hold any value, rather than just booleans. The duo of BDDs and MTBDDs matches well our notion of symbolic values and multi-values; we represent symbolics as BDDs and multi-values as MTBDDs with concrete values as terminal nodes. To support MTBDDs we extend DDNV with two more constructs:

$$\texttt{e} \quad ::= \quad \mid \texttt{apply}((\texttt{fun}\ \texttt{x}_1 \ldots \texttt{x}_n \rightarrow \texttt{e}_\emptyset), \texttt{e}_1, \ldots, \texttt{e}_n) \mid \texttt{toMulti}\ \texttt{e}_1$$

`toMulti` v lifts the concrete value v to a multi-value. This is a straightforward operation; we create an MTBDD with one terminal node containing v. `apply` is the most fundamental operation over decision diagrams. Given (MT)BDDs $\texttt{e}_1 \ldots \texttt{e}_n$, the expression $\texttt{apply}((\texttt{fun}\ \texttt{x}_1 \ldots \texttt{x}_n \rightarrow \texttt{e}_\emptyset), \texttt{e}_1, \ldots, \texttt{e}_n)$ recursively descends its arguments to generate a new MTBDD by applying the expression $\texttt{e}_\emptyset$ over their terminal nodes (denoted by the arguments $\texttt{x}_1 \ldots \texttt{x}_n$). Importantly, the computation over terminal nodes is concrete. Intuitively, the `apply` operation is similar to a standard list map operation over N lists. The typing rule Apply shows that an `apply` operation takes a variable number of symbolic and multi-values arguments, and a function parameterized with the same number of concrete arguments (*i.e.*, denoting the value of the terminal nodes). We only require that the type of each argument to the function matches with the type of the corresponding inputs to `apply` and that the return value of that function is concrete.

The rest of DDNV uses a conventional STLC-based type system. As an example, fig. 7 shows the rules for function application and tuples; the typing rules require that modes match instead of implicitly lifting them. In essence, DDNV is a basic functional programming language that interfaces to an (MT)BDD library. We interpret DDNV programs via translation to OCaml.

## 4.2 Translating ProbNV to DDNV

Next, we define a translation from well-typed ProbNV programs to well-typed DDNV programs. We explain the key ideas of the translation, formally define it, and prove a correctness theorem.

*Translation of ProbNV Expressions.* Before we dive deeper into the formalization of the translation, we illustrate the main ideas behind it. Consider the transfer function in fig. 10a; it either returns `None` or the variable `x` (which is of a multi-value type), depending on the value of the symbolic guard. The key idea of the translation is to construct an open expression of concrete type, which

$$\frac{\begin{array}{c} m \in \{C, M\} \qquad \Gamma \vdash eh_1 \;:\; [m_1]\mathtt{ty}_1 \rightsquigarrow \langle \mathtt{el}_1 \;:\; \psi([m_1]\mathtt{ty}_1), \rho_1 \rangle \\ \Gamma \vdash eh_2 \;:\; [m_2]\mathtt{ty}_2 \rightsquigarrow \langle \mathtt{el}_2 \;:\; \psi([m_2]\mathtt{ty}_2), \rho_2 \rangle \qquad [m]\mathtt{ty} = [m_1]\mathtt{ty}_1 \sqcup [m_2]\mathtt{ty}_2 \end{array}}{\Gamma \vdash eh_1 \diamond eh_2 \;:\; [m]\mathtt{int} \rightsquigarrow \langle \mathtt{el}_1 \diamond \mathtt{el}_2 \;:\; \psi([m]\mathtt{ty}), \rho_1 \uplus \rho_2 \rangle}\ \textsc{C-Bop-Int}$$

$$\frac{\Gamma \vdash eh_1 \;:\; [m_1]\mathtt{int} \rightsquigarrow \langle \mathtt{el}_1 \;:\; \psi([m_1]\mathtt{int}), \emptyset \rangle \qquad \Gamma \vdash eh_2 \;:\; [m_2]\mathtt{int} \rightsquigarrow \langle \mathtt{el}_2 \;:\; \psi([m_2]\mathtt{int}), \emptyset \rangle}{\Gamma \vdash eh_1 \diamond eh_2 \;:\; [S]\mathtt{int} \rightsquigarrow \langle \mathtt{liftBdd}(\mathtt{el}_1) \diamond_b \mathtt{liftBdd}(\mathtt{el}_2) \;:\; [S]\mathtt{int}, \emptyset \rangle}\ \textsc{C-Bop-Int-S}$$

$$\frac{\begin{array}{c} \Gamma \vdash eh_1 \;:\; [S]\mathtt{bool} \rightsquigarrow \langle \mathtt{el}_1 \;:\; [S]\mathtt{bool}, \emptyset \rangle \\ \Gamma \vdash eh_2 \;:\; [m_2]\mathtt{ty} \rightsquigarrow \langle \mathtt{el}_2 \;:\; \psi([m_2]\mathtt{ty}), \rho_2 \rangle \qquad \Gamma \vdash eh_3 \;:\; [m_3]\mathtt{ty} \rightsquigarrow \langle \mathtt{el}_3 \;:\; \psi([m_3]\mathtt{ty}), \rho_3 \rangle \\ b = \mathrm{fresh}() \qquad [m]\mathtt{ty} = [m_2]\mathtt{ty}_2 \sqcup [m_3]\mathtt{ty}_3 \qquad m \in \{C, M\} \end{array}}{\Gamma \vdash \mathrm{if}\ eh_1\ \mathrm{then}\ eh_2\ \mathrm{else}\ eh_3 \;:\; [M]\mathtt{ty} \rightsquigarrow \langle \mathrm{if}\ b\ \mathrm{then}\ \mathtt{el}_2\ \mathrm{else}\ \mathtt{el}_3 \;:\; \psi([M]\mathtt{ty}), \{b \mapsto \mathtt{el}_1\} \uplus \rho_2 \uplus \rho_3 \rangle}\ \textsc{C-Ite-M2}$$

$$\frac{\begin{array}{c} \Gamma \vdash eh_1 \;:\; [m_1](\mathtt{dict}([C]\mathtt{ty}_1, [C]\mathtt{ty}_2)) \rightsquigarrow \langle \mathtt{el}_1 \;:\; \psi([m_1](\mathtt{dict}([C]\mathtt{ty}_1, [C]\mathtt{ty}_2))), \rho_1 \rangle \\ \Gamma \vdash eh_2 \;:\; [m_3]\mathtt{ty}_3 \rightsquigarrow \langle \mathtt{el}_2 \;:\; \psi([m_3]\mathtt{ty}_3), \rho_2 \rangle \qquad [m]\mathtt{ty} = [m_1]\mathtt{ty}_1 \sqcup [m_3]\mathtt{ty}_3 \end{array}}{\Gamma \vdash eh_1.[eh_2] \;:\; [m]\mathtt{ty}_2 \rightsquigarrow \langle \mathtt{el}_1.[\mathtt{el}_2] \;:\; \psi([m]\mathtt{ty}_2), \rho_1 \uplus \rho_2 \rangle}\ \textsc{C-Map-Get-C}$$

$$\frac{\begin{array}{c} \Gamma \vdash eh_1 \;:\; [C](\mathtt{dict}([C]\mathtt{ty}_1, [M]\mathtt{ty}_2)) \rightsquigarrow \langle \mathtt{el}_1 \;:\; [C](\mathtt{dict}([C]\mathtt{ty}_1, [M]\mathtt{ty}_2)), \emptyset \rangle \\ \Gamma \vdash eh_2 \;:\; [C]\mathtt{ty}_1 \rightsquigarrow \langle \mathtt{el}_2 \;:\; \psi([C]\mathtt{ty}_1), \emptyset \rangle \qquad b = \mathrm{fresh}() \end{array}}{\Gamma \vdash eh_1.[eh_2] \;:\; [M]\mathtt{ty}_2 \rightsquigarrow \langle b \;:\; \psi([M]\mathtt{ty}_2), \{b \mapsto eh_1.[eh_2]\} \rangle}\ \textsc{C-Map-Get-M}$$

Fig. 8. Translation judgement for ProbNV expressions.

$$\frac{\Gamma \vdash eh_1 \;:\; [m_1]\mathtt{ty}_1 \rightsquigarrow \langle \mathtt{el}_1 \;:\; \psi([m_1]\mathtt{ty}_1), \emptyset \rangle \qquad \Gamma, x \;:\; [m_1]\mathtt{ty}_1 \vdash d_h \;:\; [m_2]\mathtt{ty}_2 \hookrightarrow d_l \;:\; \mathtt{liftTy}([m_2]\mathtt{ty}_2)}{\Gamma \vdash \mathrm{let}\ x\ =\ eh_1\ ;;\ d_h \;:\; [m_2]\mathtt{ty}_2 \hookrightarrow \mathrm{let}\ x\ =\ \mathtt{el}_1\ ;;\ d_l \;:\; \mathtt{liftTy}([m_2]\mathtt{ty}_2)}\ \textsc{C-Let}$$

$$\frac{\begin{array}{c} \Gamma \vdash eh_1 \;:\; [M]\mathtt{bool} \rightsquigarrow \langle \mathtt{el}_1 \;:\; \psi([M]\mathtt{bool}), \rho_1 \rangle \\ \Gamma \vdash eh_2 \;:\; [m]\mathtt{bool} \rightsquigarrow \langle \mathtt{el}_2 \;:\; \psi([m]\mathtt{bool}), \rho_2 \rangle \qquad m \in \{M, S\} \end{array}}{\Gamma \vdash \mathtt{assert}(eh_1 \mid eh_2) \;:\; [M]\mathtt{bool} \hookrightarrow \mathtt{assert}(\mathtt{liftM}(\mathtt{el}_1, \rho_1 \uplus \rho_2) \mid \mathtt{el}_2) \;:\; [M]\mathtt{bool}}\ \textsc{C-Assert-M}$$

$$\frac{\begin{array}{c} \Gamma, u \;:\; [C]\mathrm{tnode} \vdash eh_i \;:\; [m_i]\mathtt{ty}_i \rightsquigarrow \langle \mathtt{el}_i \;:\; \psi([m_i]\mathtt{ty}_i), \rho_i \rangle \\ \Gamma, e \;:\; [C]\mathrm{tedge}, x \;:\; [M]\mathrm{ty} \vdash eh_t \;:\; [m_t]\mathtt{ty}_t \rightsquigarrow \langle \mathtt{el}_t \;:\; \psi([m_t]\mathtt{ty}_t), \rho_t \rangle \\ \Gamma, u \;:\; [C]\mathrm{tnode}, x \;:\; [M]\mathrm{ty}, y \;:\; [M]\mathrm{ty} \vdash eh_m \;:\; [m_m]\mathtt{ty}_m \rightsquigarrow \langle \mathtt{el}_m \;:\; \psi([m_m]\mathtt{ty}_m), \rho_m \rangle \\ \mathtt{el}'_i = \mathrm{fun}\ u \rightarrow \mathtt{liftM}(\mathtt{el}_i, \rho_i) \qquad \mathtt{el}'_t = \mathrm{fun}\ e \rightarrow \mathrm{fun}\ x_m \rightarrow \mathtt{liftM}(\mathtt{el}_t, \{x \mapsto x_m\} \uplus \rho_t) \\ \mathtt{el}'_m = \mathrm{fun}\ u \rightarrow \mathrm{fun}\ x_m \rightarrow \mathrm{fun}\ y_m \rightarrow \mathtt{liftM}(\mathtt{el}_m, \{x \mapsto x_m, y \mapsto y_m\} \uplus \rho_m) \\ \Gamma, x \;:\; [C](\mathtt{dict}([C]\mathrm{tnode}, [M]\mathrm{ty})) \vdash d_h \;:\; [m_2]\mathtt{ty}_2 \hookrightarrow d_l \;:\; \mathtt{liftTy}([m_2]\mathtt{ty}_2) \end{array}}{\begin{array}{c} \Gamma \vdash \mathrm{let}\ x\ =\ \mathrm{solution}(\mathrm{fun}\ u \rightarrow eh_i, \mathrm{fun}\ e\ x \rightarrow eh_t, \mathrm{fun}\ u\ x\ y \rightarrow eh_m)\ ;;\ d_h \;:\; [m_2]\mathtt{ty}_2 \\ \hookrightarrow \mathrm{let}\ x\ =\ \mathrm{solution}(\mathtt{el}'_i, \mathtt{el}'_t, \mathtt{el}'_m)\ ;;\ d_l \;:\; \mathtt{liftTy}([m_2]\mathtt{ty}_2) \end{array}}\ \textsc{C-Sol-M}$$

Fig. 9. Translation judgement for ProbNV declarations.

we can later close with symbolic and multi-value values. To do this, we traverse the body of the function finding symbolic/multi-value expressions over which we want to parameterize the computation. We then replace them with a fresh variable and retain a mapping between the two. In this example, we replace the symbolic guard with the variable `b`, and the variable `x` with the variable `xc`. Partioning the program in this way allows us to use the `apply` operation to compute the possible outcomes, based on the symbolic/multi-value inputs. Of course, this transformation can be unsound. For instance, consider the slightly more complicated transfer function of fig. 10c. In this case, if we move the guard `f = a || f = b` outside the match expression it will result in an ill-formed program as the variables `a` and `b` will not be bound. Moreover, we cannot place the `apply` operation inside this match expression, as the computation is also parameterized by the variable `x` and any `apply` operation must include it. To overcome this problem, we can inline all variable bindings that are used in symbolic expressions. For match expressions, we propagate the

```
symbolic f: tnode

let trans e (x : [M](option[int])) =
  if f = e.1 || f = e.2 then
    None
  else x
```

(a)

```
symbolic f: tnode

let trans e (x : [M](option[int])) =
  apply(fun b xc ->
    if b then
      None
    else xc, f =_b e.1 ||_b f =_b e.2, x)
```

(b)

```
symbolic f: tnode

let trans e x =
  match x with
  | None -> None
  | Some y -> (match e with
               | (a,b) ->
                 if f = a || f = b then None else y)
```

(c)

Fig. 10. Translating a ProbNV function (a) to DDNV (b).

scrutinee on each branch of the match expression. To handle tuples we can expand them using their projections, *e.g.*, e becomes ($\pi_1$ e, $\pi_2$ e).

In the previous examples we intentionally focused on one of the functions provided as arguments to the solution command because we inline all other functions that return a multi-value result; the absence of recursive functions makes this possible. The only functions we cannot inline are the arguments to solution declarations. There are multiple technical reasons to inline functions that return a multi-value result; consider the following code fragment.

```
symbolic x: bool
let f (v: [C]int) = if x then v else v+1
let a = if x then f 0 else 1
```

Suppose we translate function f as before and then we naively translate the definition of a as well. Since the expression f 0 also returns a multi-value, the following is a reasonable translation.

```
symbolic x: bool
let f (v: [C]int) : [M]int = apply (fun x -> if x then v else v+1, x)
let a = apply (fun x v -> if x then v else 1, x, f 0)
```

However, the program now executes two separate apply operations, resulting in two separate traversals of the BDD when a single apply over x suffices. Consequently, we inline such functions, as well as top-level let declarations of multi-value type. We do not inline functions returning a concrete value, even though such functions can be applied to a multi-value argument (see the typing rule App-C for applications (fig. 5)).

Formally the translation is structured as a type-driven translation with two parts: one describing the translation of expressions and one describing the translation of declarations.

*Formal Translation Judgement for Expressions.* The formal judgement for expressions is of the form $\Gamma \vdash \mathtt{e_h} : \mathtt{ty} \rightsquigarrow \langle \mathtt{e_1}, \rho \rangle$, where $\mathtt{e_h}$ is a well-typed (in the context $\Gamma$) ProbNV expression, $\mathtt{e_1}$ is a DDNV expression, and $\rho$ is a mapping from variables to expressions. The map $\rho$ contains the bindings from fresh variables to their corresponding symbolic/multi-value expressions, we previously described. Figure 8 shows a subset of the translation rules for ProbNV expressions. The translation of a well-typed expression $\mathsf{e_h} : \mathtt{ty}$ produces an expression $\mathsf{e_l}$ and a type $\psi(\mathtt{ty})$. At a high-level, the function $\psi$ changes a multi-value type to a concrete type. Additionally, it makes mode switches in types explicit. For instance, consider the tuple type [S]([C]int, [S]int); the type system of ProbNV treats computations performed over the components of this tuple as symbolic, even if they are only concerned with the first component. The type system of DDNV does not allow such implicit mode changes; hence our translation casts (using `toBdd`) the first component of the tuple (and in extension, its type) to a symbolic value.

*Definition 4.1 (Type Translation).*

$$\psi(\mathtt{[m]ty}) = \begin{cases} \mathtt{[C]ty} & \text{if } \mathsf{m} \in \{\mathrm{M}, \mathrm{C}\} \\ \mathtt{[S]ty} & \text{if } \mathsf{m} = \mathrm{S} \end{cases} \qquad \text{when } \mathtt{ty} \in \{\mathtt{int}, \mathtt{bool}, \mathtt{tedge}, \mathtt{tnode}\}$$

$$\psi(\mathtt{[m]([m_1]ty_1, \ldots, [m_n]ty_n)}) = \begin{cases} \mathtt{[C]}(\psi(\mathtt{[C]ty_1}), \ldots, \psi(\mathtt{[C]ty_n})) & \text{if } \mathsf{m} \in \{\mathrm{M}, \mathrm{C}\} \\ \mathtt{[S]}(\psi(\mathtt{[S]ty_1}), \ldots, \psi(\mathtt{[S]ty_n})) & \text{otherwise} \end{cases}$$

$$\psi(\mathtt{[m](ty_1 \rightarrow ty_2)}) = \begin{cases} \mathtt{[C]}(\psi(\mathtt{ty_1}) \rightarrow \psi(\mathtt{ty_2})) & \text{if } \mathsf{m} \in \{\mathrm{M}, \mathrm{C}\} \\ \mathtt{[S]}(\psi(\mathtt{ty_1}) \rightarrow \psi(\mathtt{ty_2})) & \text{otherwise} \end{cases}$$

$$\psi(\mathtt{[m](dict(ty_1, ty_2))}) = \mathtt{[C]dict(ty_1, liftTy(ty_2))}$$

Rules such as C-Bop-Int are representative of how the translation works for computations involving multi-value and/or concrete expressions. Arguments to the binary operation are recursively translated, producing a binary operation over concrete arguments, and at the same time, accumulating any symbolic and multi-value expressions over which we parameterize this operation. C-Bop-Int-S is the corresponding rule for translating symbolic operations. In this case, the translation introduces a BDD operation (denoted by $\diamond_b$ in the rule). Moreover, symbolic expressions are not parameterized by other symbolics or multi-values, hence an empty set of bindings is produced. Assuming they are not already symbolic, arguments to the binary operation need to be explicitly converted to BDD values. In the translation we use the function `liftBdd` for brevity; it casts a concrete value to a symbolic one, or acts as the identity function on symbolic inputs:

$$\mathtt{liftBdd(e : [m]ty)} = \begin{cases} \mathtt{toBdd\ (e)} & \text{if } \mathsf{m} = \mathrm{C} \\ \mathtt{e} & \text{if } \mathsf{m} = \mathrm{S} \end{cases}$$

When translating conditional expressions, such as if-then-else, we accumulate the symbolic/multi-value expressions over which we parameterize the computation. For instance, in rule C-Ite-M2, since the guard is symbolic, we replace the translated symbolic expression with a fresh variable and add a new binding to the map of accumulated expressions. The resulting expression is concrete, as shown by the application of $\psi$ on its type. Similarly, rule C-Map-Get-M shows that we also accumulate dictionary-get expressions when the dictionary stores multi-values (*i.e.*, for dictionaries generated by solution declarations). Since we inline multi-value functions and top-level let-declarations, the only remaining multi-value expressions to extract out of an expression are the ones passed via function arguments; these are arguments to functions in solution primitives that cannot be inlined. We handle them as special cases during translation of solution declarations. The rest of the rules follow these principles; we refer the reader to the appendix for the full translation rules.

*Formal Translation Judgement for Declarations.* The formal judgement for declarations is of the form $\Gamma \vdash \mathsf{d_h} : \mathtt{ty} \hookrightarrow \mathsf{d_l}$, where $\mathsf{d_h}$ is a well-typed program, *i.e.*, a sequence of well-typed ProbNV declarations[6], and $\mathsf{d_l}$ is a DDNV program. Figure 9 shows some of the translation rules concerning declarations. Let declarations are straightforward, since they are either concrete or symbolic we only emit the translated expression. For assertions over multi-value expressions (C-Assert-M),

[6]In our formalization we assume programs end with an assertion for our convenience; this restriction can easily be lifted.

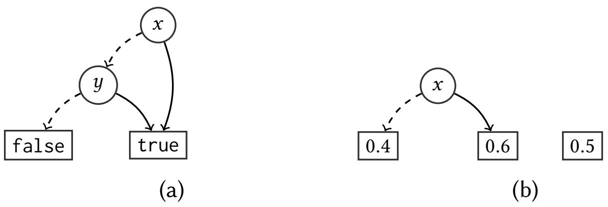


Fig. 11. (a) example representation of an assertion that depends on boolean symbolics x and y. (b) shows the MTBDD representation of the distribution of x and y.

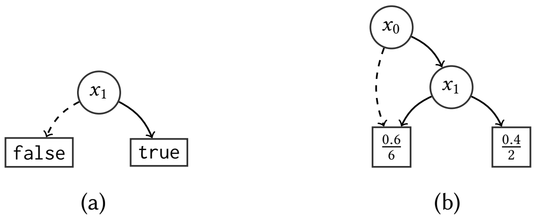


Fig. 12. (a) shows an example representation of an assertion that depends on a 3-bit integer symbolic. (b) shows the MTBDD representation of its distributions.

we "close" the translated expression, by constructing an apply operation. Function `liftM` does this using the bindings $\rho$ computed during the translation of the expression:

$$\texttt{liftM}(e,\rho) = \begin{cases} \texttt{apply}((\texttt{fun}\ \mathtt{x}_1 \ldots \mathtt{x}_n \to \mathtt{e}), \mathtt{e}_1, \ldots, \mathtt{e}_n) & \text{if } \rho = \{\mathtt{x}_1 \mapsto \mathtt{e}_1, \ldots \mathtt{x}_n \mapsto \mathtt{e}_n\} \\ \texttt{toMulti e} & \text{if } \rho = \{\} \end{cases}$$

When translating solution declarations we do not directly translate the expressions of the init/transfer/merge functions. The expression translation judgement is not designed to handle functions with multi-value arguments (this is not a problem in all other cases because, as previously explained, we inline them). Instead, we translate the body of these functions directly. We then extend the set of bindings $\rho$ to associate the variables used in the body of these functions with the multi-value arguments. Finally, we prove that the translation produces well-typed DDNV programs, *i.e.*, programs that will not get stuck during execution.

Theorem 4.2. *Given a program* dh *in ProbNV, such that* $\Gamma \vdash \mathtt{dh} : [\mathtt{m}]\mathtt{ty}$, *assuming multi-values and local bindings that appear in symbolic expressions have been inlined, we have that:*

$$\exists\, \mathtt{dl}.\, \Gamma \vdash \mathtt{dh} : [\mathtt{m}]\mathtt{ty} \rightsquigarrow \mathtt{dl} : \mathtt{liftTy}([\mathtt{m}]\mathtt{ty}) \wedge \mathtt{liftTy}(\Gamma) \vdash_{\mathrm{DD}} \mathtt{dl} : \mathtt{liftTy}([\mathtt{m}]\mathtt{ty})$$

$\mathtt{liftTy}(\Gamma)$ *means that* `liftTy` *is applied to the type of every binding in* $\Gamma$.

## 4.3 Interpreting the Probabilistic Fragment of ProbNV

Next, we discuss the implementation of the probabilistic fragment of the language. The first step is to choose a representation for probability distributions of symbolic values. Since we already use BDDs to implement symbolic values, a natural choice is to use MTBDDs to represent probability distributions. A path in the MTBDD describes a class (set) of equiprobable values; the terminal leaf stores the probability. For example, fig. 11b shows the MTBDD representation of a symbolic boolean x s.t., $\mathcal{D}_x = \{\mathtt{true} \mapsto 0.4, \mathtt{false} \mapsto 0.6\}$. For a more complex example, consider a symbolic 3-bit integer x s.t., $\mathcal{D}_x = \{x = 3 \vee x = 7 \mapsto 0.4, \neg(x = 3 \vee x = 7) \mapsto 0.6\}$. Its representation using MTBDDs is shown in fig. 12b. First, notice that the values of the leaves depend on the size of the space they cover. In each case, we distribute the given probability uniformly, *e.g.*, if the space $x = 3 \vee x = 7$ has probability 0.4, every integer satisfying this predicate has probability $\frac{0.4}{2}$. Moreover, notice that not all decision nodes appear in all paths; like BDDs, MTBDDs contain no redundant decision nodes. This keeps the representation of the distributions compact but at the same time complicates the probability computation procedure.

*Computing Probabilities.* Given an assertion of a multi-value expression—which reduces to a boolean multi-value—and the distributions of the symbolics it depends upon, our goal is to compute the probability that the assertion is true. In other words, we wish to compute the combined probability of all paths in the assertion's BDD that lead to the true leaf.

For starters, consider the case where all symbolics are of boolean type, *i.e.*, encoding each symbolic requires (at most) one decision node. In this case, the computation of the probability is a straightforward recursion where we traverse the assertion's BDD accumulating the probability of the paths that lead to the true leaf. A node in the assertion's BDD implies that the result depends on this decision node. Naturally, at each decision point, we can divide the computation into two sub-computations: one for the paths resulting by following the true case of the decision node and one for the paths following the false case. The sub-computations represent independent cases; the total probability is the sum of each subtree's probability. We note that decision nodes in (MT)BDDs appear in a fixed order, and a decision node cannot appear more than once in every path. Since decisions across different symbolics are independent, a path's probability is given by the product of the probability of each symbolic. Based on the above, we define a function $\mathsf{P}_{\mathcal{D}}(\mathsf{b})$ that is parameterized over a mapping $\mathcal{D}$ from symbolics to distributions. Given a BDD b, $\mathsf{P}_{\mathcal{D}}(\mathsf{b})$ computes the probability $\mathsf{p}(\mathsf{b} = \mathtt{true})$. We write `x.t` (resp. `x.f`) for the true (resp. false) sub-tree of `x`, and `Leaf v` for a leaf with value `v`.

$$\mathsf{P}_{\mathcal{D}}(\mathsf{b}) = \begin{cases} 0.0 & \text{if } \mathsf{b} = \mathtt{Leaf\ false} \\ 1.0 & \text{if } \mathsf{b} = \mathtt{Leaf\ true} \\ p \cdot (\mathsf{P}_{\mathcal{D}}(\mathsf{x.t}) + \mathsf{P}_{\mathcal{D}}(\mathsf{x.f})) & \text{if } \mathsf{b} = \mathsf{x} \text{ and } \mathcal{D}(\mathsf{x}) = \mathtt{Leaf\ p} \\ \mathcal{D}(\mathsf{x.t}) \cdot \mathsf{P}_{\mathcal{D}}(\mathsf{x.t}) + \\ \quad \mathcal{D}(\mathsf{x.f}) \cdot \mathsf{P}_{\mathcal{D}}(\mathsf{x.f}) & \text{otherwise} \end{cases}$$

For instance, consider an assertion a equivalent to `if x || y then true else false`. Its BDD form is shown in fig. 11a. Assuming distributions for `x` and `y` as shown in fig. 11b, the probability $\mathsf{p}(\mathsf{a} = \mathtt{true})$ is:

$$\begin{aligned} \mathsf{P}_{\mathcal{D}}(\mathsf{a}) &= 0.6 \cdot \mathsf{P}_{\mathcal{D}}(\mathsf{x.t}) + 0.4 \cdot \mathsf{P}_{\mathcal{D}}(\mathsf{x.f}) = 0.6 \cdot 1.0 + 0.4 \cdot \mathsf{P}_{\mathcal{D}}(\mathsf{y}) \\ &= 0.6 + 0.4 \cdot (0.5 \cdot (\mathsf{P}_{\mathcal{D}}(\mathsf{y.t}) + \mathsf{P}_{\mathcal{D}}(\mathsf{y.f}))) = 0.6 + 0.4 \cdot (0.5 \cdot (1.0 + 0)) = 0.8 \end{aligned}$$

*Extending to Integer Symbolics.* Compared to the boolean-only case, when working with integer symbolics (and in general, with types that are represented via multiple decision nodes), we have to deal with two additional complexities. First, unlike the boolean case, distributions may involve multiple decision nodes and arbitrarily complex MTBDDs; hence, the recursion needs to be over both the assertion and the distributions. Second, BDDs and MTBDDs do not encode information about the size of the state space represented (*i.e.*, the ProbNV type represented). We have to keep track of these separately and use them when calculating probabilities.

In what follows, we assume that decision nodes are indexed by an integer, and that nodes with smaller index are closer to the root of the decision diagram. For simplicity of presentation, we assume the assertion depends on a single symbolic `x`; extending to multiple symbolics requires some extra (and tedious) index arithmetic and bookkeeping to relate decision nodes to ProbNV symbolics. The recursive function $\mathsf{P}(\mathsf{b}, \mathcal{D}_\mathsf{x}, \text{start}, \text{end})$ traverses the assertion (b) and the distribution ($\mathcal{D}_x$) simultaneously, and computes the probability that the assertion is true:

$$
\mathsf{P}(\mathsf{b},\mathcal{D},\mathsf{start},\mathsf{end}) =
\begin{cases}
0.0 & \text{if } \mathsf{b} = \mathsf{Leaf\ false}\\
2^{\mathsf{end-start+1}} \cdot \mathsf{p} & \text{if } \mathsf{b} = \mathsf{Leaf\ true}, \mathcal{D}_\mathsf{x} = \mathsf{Leaf\ p}\\
2^{j-\mathsf{start}} \cdot [\mathsf{P}(\mathsf{b}, \mathcal{D}_\mathsf{x}.\mathsf{t}, \mathsf{j}+1, \mathsf{end}) + \mathsf{P}(\mathsf{b}, \mathcal{D}_\mathsf{x}.\mathsf{f}, \mathsf{j}+1, \mathsf{end})] & \text{if } \mathsf{b} = \mathsf{Leaf\ true}, \mathcal{D}_\mathsf{x} = \mathsf{x_j}\\
2^{i-\mathsf{start}} \cdot [\mathsf{P}(\mathsf{b}.\mathsf{t}, \mathcal{D}_\mathsf{x}, \mathsf{i}+1, \mathsf{end}) + \mathsf{P}(\mathsf{b}, \mathcal{D}_\mathsf{x}, \mathsf{i}+1, \mathsf{end})] & \text{if } \mathsf{b} = \mathsf{x_i}, \mathcal{D}_\mathsf{x} = \mathsf{Leaf\ p}\\
2^{i-\mathsf{start}} \cdot [\mathsf{P}(\mathsf{b}.\mathsf{t}, \mathcal{D}_\mathsf{x}.\mathsf{t}, \mathsf{i}+1, \mathsf{end}) + \mathsf{P}(\mathsf{b}.\mathsf{f}, \mathcal{D}_\mathsf{x}.\mathsf{f}, \mathsf{i}+1, \mathsf{end})] & \text{if } \mathsf{b} = \mathsf{x_i}, \mathcal{D}_\mathsf{x} = \mathsf{x_j}, \mathsf{i} = \mathsf{j}\\
2^{i-\mathsf{start}} \cdot [\mathsf{P}(\mathsf{b}.\mathsf{t}, \mathcal{D}_\mathsf{x}, \mathsf{i}+1, \mathsf{end}) + \mathsf{P}(\mathsf{b}.\mathsf{f}, \mathcal{D}_\mathsf{x}.\mathsf{f}, \mathsf{i}+1, \mathsf{end})] & \text{if } \mathsf{b} = \mathsf{x_i}, \mathcal{D}_\mathsf{x} = \mathsf{x_j}, \mathsf{i} < \mathsf{j}\\
2^{j-\mathsf{start}} \cdot [\mathsf{P}(\mathsf{x}, \mathcal{D}_\mathsf{x}.\mathsf{t}, \mathsf{j}+1, \mathsf{end}) + \mathsf{P}(\mathsf{x}, \mathcal{D}_\mathsf{x}.\mathsf{f}, \mathsf{j}+1, \mathsf{end})] & \text{if } \mathsf{b} = \mathsf{x_i}, \mathcal{D}_\mathsf{x} = \mathsf{x_j}, \mathsf{j} < \mathsf{i}
\end{cases}
$$

The third and fourth arguments are used to define the size of the space modeled by the symbolic under consideration. Initially, these are the first and last BDD nodes allocated for the symbolic, *e.g.*, for a 3-bit symbolic integer these variables will be $i$ and $i + 2$ respectively —if this is the first symbolic declared $i$ will be 0. The idea is to traverse the assertion and the distribution in tandem; if the assertion is at a variable on index $i$ and the distribution is at index $j$, then if $i < j$ we recurse on the assertion to catch-up to the distribution, and vice-versa. If they are at the same index, we recurse on both. This process repeats until we reach the leaves. We explain the functionality of P in more detail through an example. Consider the BDD of fig. 12a as the asserted value, we compute $\mathsf{P}(\mathsf{b}, \mathcal{D}_\mathsf{x}, 0, 2)$. While the assertion does not depend on the decision node $\mathsf{x_0}$, it does affect the value of the distribution. Similar to when we encounter a decision point in the assertion, we may split the computation into two sub-computations (*i.e.*, the last case of P applies):

$$2^{0-0} \cdot (\mathsf{P}(\mathsf{x_1}, \mathsf{x_1}, 1, 2) + \mathsf{P}(\mathsf{x_1}, \mathsf{Leaf\ 0.1}, 1, 2))$$

The term $2^{0-0}$ accounts for omitted nodes in the decision diagrams, capturing the number of integers with probability equal to the second term of the product. So far we have not omitted any nodes; the starting index is $0$ and the sum $(\mathsf{P}(\mathsf{x_1}, \mathsf{x_1}, 1, 2) + \mathsf{P}(\mathsf{x_1}, \mathsf{Leaf\ 0.1}, 1, 2))$ captures the probability for both values of $\mathsf{x_0}$. The distribution has now advanced and the minimum variable across the distribution and the assertion is $\mathsf{x_1}$. We proceed by applying the fourth case of P:

$$
\begin{gathered}
[\mathsf{P}(\mathsf{Leaf\ true}, \mathsf{Leaf\ 0.2}, 2, 2) + \mathsf{P}(\mathsf{Leaf\ false}, \mathsf{Leaf\ 0.1}, 2, 2)] +\\
[\mathsf{P}(\mathsf{Leaf\ true}, \mathsf{Leaf\ 0.1}, 2, 2) + \mathsf{P}(\mathsf{Leaf\ false}, \mathsf{Leaf\ 0.1}, 2, 2)]
\end{gathered}
$$

We have reached the leaves of the assertion and the distribution; however, simply returning the probabilities at the leaves of the distribution would be wrong. To compute the correct probabilities, we need to consider the number of integers corresponding to each true leaf of the assertion. In our so far computations, we have determined the result according to the different values of $\mathsf{x_0}$, $\mathsf{x_1}$, but since $\mathsf{x_2}$ appears in neither the assertion nor the distribution, its value remains unconstrained. Hence, we need to double the probability we have computed to account for both values of $\mathsf{x_2}$; the term $2^{\mathsf{end-start+1}}$ in the second case of P accounts for such unconstrained variables:

$$\mathsf{p}(\mathsf{a} = \mathsf{true}) = 2 \cdot 0.2 + 0 + 2 \cdot 0.1 + 0 = 0.6$$

*Conditional Assertions.* Computing the probability of a conditional assertion is a straightforward application of Bayes' theorem:

$$P(b \mid c) = \frac{P(b \wedge c)}{P(c)}$$

We use the techniques explained previously to compute the probabilities $P(b \wedge c)$ and $P(c)$.

**Algorithm 1** Dependency-Based Network Simulator

```
procedure UpdateSchedule($u$, E, sched)
    for $(u, v) \in E$ do
        if $v \in$ sched then sched$(v) \leftarrow$ sched$(v) \cup \{u\}$ else sched$(v) \leftarrow \{u\}$

function PopNode($u, vs$, sched)
    for $v \in vs$ do
        if $v \in$ sched then return PopNode($v$, sched($v$), sched)        ▷ Work through dependencies first
    return $(u, vs, \text{sched} \setminus u)$
function Pop(sched)
    $(u, vs) \leftarrow$ head(sched)                                         ▷ Start from top of schedule
    return PopNode($u, vs$, sched)

procedure simulate(V, E, init, trans, merge)
    sched $\leftarrow$ []
    for $u \in V$ do
        sched $\leftarrow (u, u)$ :: sched                                    ▷ Initially every node has a route from itself
    while sched $\neq$ [] do
        $u, vs$, sched $\leftarrow$ Pop(sched)                                 ▷ Pick the next node to process
        for $v \in vs$ do
            new $\leftarrow u = v$ ? $[\![\text{init}(u)]\!]$ : $[\![\text{trans}((v,u), \mathcal{L}(v))]\!]$     ▷ Compute the incoming route
            received$(u)(v) \leftarrow$ new
        oldLabel $\leftarrow \mathcal{L}(u)$
        $\mathcal{L}(u) \leftarrow \bigcup_{[\![\text{merge}]\!]}$ received$(u)$             ▷ Update the label
        if oldLabel $\neq \mathcal{L}(u)$ then UpdateSchedule($u, E$, sched)     ▷ Update schedule if necessary
```

## 5 SYMBOLIC NETWORK SIMULATION

The goal of a network simulator is to compute the stable state of the network. Typically, simulation algorithms follow a simple operation model: *1.* pick the next node to process from a list of unprocessed nodes, *2.* process that node (*e.g.*, propagate the node's solution to its neighbors), *3.* update the list of unprocessed nodes (*e.g.*, add to the list any node whose solution was updated in the previous step), and repeat the process until the list of unprocessed nodes is empty.

A high-performance simulator requires careful crafting of the above components. Of particular importance is the order in which to process nodes. Prior work [Lopes and Rybalchenko 2019] has shown that processing the nodes with better solutions first, minimizes the number of steps required to reach convergence for certain networks; ShapeShifter [Beckett et al. 2019a] adopts a similar idea. However, such optimizations traditionally targeted conventional control plane models, where the type of the computed routes admit an easily computable ordering. In our case, a "route" often has much richer structure, *e.g.*, a collection of failure scenarios and the corresponding BGP routes, making it difficult even to define an intuitive ordering, let alone a quickly computable one.

To avoid directly dealing with making optimal scheduling decisions, we propose instead to try to correct course should we end up following a bad execution order. To support this, we first change (compared to the current state of the art simulators) how we process a node scheduled for processing: instead of propagating its solution to its neighbors, it "pulls" routes from them. We augment the schedule so that it not only provides the order in which to process nodes, but each entry also provides the neighboring nodes from which the scheduled node should pull. This gives rise to a notion of *dependencies*: a node $u$ *depends* on a node $v$ if $u$ is scheduled to pull $v$'s route. Our key insight is that if node $v$ is also scheduled for an update, then it should take priority over $u$. In this way, our simulator avoids duplicated work. Importantly, scheduling is oblivious to the routing algebra under simulation, dependencies are the same across any model no matter how complex.

Algorithm 1 presents this new simulation algorithm. Initially, all nodes have a message from themselves (line 15), *i.e.*, their initial route. As long as there are unprocessed nodes (line 16), we pick the next node to process from the schedule, typically the head of the schedule (line 9). But, if one of the nodes it will pull routes from ($vs$) is also in the schedule (line 7), that node assumes priority. This process repeats until a node that does not depend on another node in the schedule is selected. In our implementation, we bound the depth of the search for the next node to process. Eventually, when the next node ($u$) is selected, the simulator processes its incoming messages (line 18), either using the `init` function if it is a message from itself or the `transfer` function for a message from a neighbor, and updates the messages that $u$ has received (line 20).

We then compute the merge of the messages received on node $u$ (line 22); if its solution changes, then we need to signal to $u$'s neighbors that there is a new route to pull from $u$ (line 23). We remark that merging all received routes on each update is (unnecessarily) costly; in our implementation we use incremental merging [Beckett et al. 2019a] to significantly reduce the number of merges.

## 6 EVALUATION

In our evaluation of ProbNV, we ask the following questions: *(i)* how well does our approach scale, *(ii)* how does the simulation algorithm of section 5 fare compared to typical simulation algorithms, and *(iii)* how fast can we check probabilistic assertions. For the evaluation, we used a CloudLab server [Duplyakin et al. 2019] with an AMD EPYC 7402P clocked at 2.8Ghz and 128GB of memory.

### 6.1 Networks Characteristics

We conduct our experiments using a diverse collection of networks. We consider a family of highly-symmetric and densely connected datacenter topologies [Al-Fares et al. 2008] running eBGP with a synthetic routing policy that uses BGP communities and route filters to disallow instances of excessive valley-routing [Beckett et al. 2019b]. Additionally, we consider wide-area topologies from the Topology Zoo [Knight et al. 2011] (BICS, Columbus, USCarrier) running OSPF or eBGP with a rich set of policies (utilizing features such as BGP communities, local preference and prefix filters) synthesized using NetComplete [El-Hassany et al. 2018]. We use a slightly modified version of NV's configuration compiler [Giannarakis et al. 2020] to translate these networks to ProbNV. For each network, we assume a single destination node that announces an eBGP or OSPF route.

### 6.2 Verification Performance

To evaluate the performance of our approach, we assume two separate failure scenarios, one where all nodes/links can fail and one where a bounded number of links fail. For each network under consideration, we compute the routes in each failure scenario and for every node in the network we compute the probability it has a route to the destination.

*All-Failures Analysis.* In the case of an all-failures analysis, we associate a symbolic boolean to every node/link in the network[7], denoting whether it has failed. The order in which decision nodes appear in BDDs can have significant impact on performance. To evaluate this impact we consider different orderings for the BDD variables: a random ordering, an ordering based on a breadth-first search from the destination node, and in the case of node failures in Fat Tree networks an ordering we picked specifically for the given network. We note that the order of decision nodes in the BDDs is determined based on the order in which symbolics are declared in the ProbNV program, so the user can easily experiment to find an ordering that works well for their network.

[7]In the all-faults scenario we consider link failures to be bidirectional.

| Topology | Routing Features | Fail Model | Symbolics Ordering | Simulation Time (s) | Assertion Time (s) | # BDD Nodes | Simulator Speedup |
|---|---|---|---|---|---|---|---|
| BICS(49/128) | BGP Prefix Filters Communities Local-Pref | Nodes | Random | 0.010 | 0.001 | 10,220 | 1.2x |
| | | | BFS | 0.01 | 0.001 | 10,220 | 1.24x |
| | | Links | Random | 15.57 | 1.32 | 8,239,364 | 1.4x |
| | | | BFS | 3.142 | 0.173 | 3,011,834 | 1.27x |
| Columbus(86/202) | BGP Prefix Filters Communities Local-Pref | Nodes | Random | 2.082 | 0.773 | 2,131,892 | 1.95x |
| | | | BFS | 0.146 | 0.034 | 287,182 | 2.41x |
| | | Links | Random | 309.4 | 20.43 | 137,397,680 | 2.29x |
| | | | BFS | 11.79 | 0.739 | 8,489,754 | 2.27x |
| USCarrier(158/378) | OSPF | Nodes | Random | - | - | - | - |
| | | | BFS | 17.07 | 0.697 | 13,430,102 | 1.92x |
| | | Links | Random | - | - | - | - |
| | | | BFS | 2176 | 22.26 | 280,939,624 | - |
| Fat(45/216) | BGP Prefix Filters Communities | Nodes | Random | 0.839 | 0.463 | 1,017,912 | 0.97x |
| | | | BFS | 0.142 | 0.094 | 272,874 | 0.89x |
| | | | FT | 0.178 | 0.089 | 252,434 | 1.12x |
| | | Links | Random | - | - | - | - |
| | | | BFS | - | - | - | - |
| Fat(80/512) | BGP Prefix Filters Communities | Nodes | Random | 52.23 | 11.23 | 19,730,732 | 1.02x |
| | | | BFS | 115.8 | 46.58 | 43,578,080 | 1.13x |
| | | | FT | 36.31 | 7.06 | 8,648,164 | 1.04x |
| | | Links | Random | - | - | - | - |
| | | | BFS | - | - | - | - |

Fig. 13. Simulation performance for all failure scenarios. We set a timeout of 50 minutes. **Topology(V/E)**: the network topology and the number of nodes (V), links (E) in it. **Routing Features**: routing protocol features used by the network. **Fail Model**: Type of failures (links or nodes). **Symbolics Ordering**: Ordering of BDD variables. **Simulation Time**: Time to compute routes (the best simulation algorithm is shown). **Assertion Time**: Time to check assertions. **# BDD Nodes**: The peak number of BDD nodes allocated during simulation. **# Simulator Speedup**: Speedup from using the new simulation algorithm instead of the classic one.

| Topology | # Failures | Simulation Time (s) | Assertion Time (s) | # BDD Nodes | Simulator Speedup |
|---|---|---|---|---|---|
| USCarrier(158/378) | 2 | 0.204 | 0.029 | 368,942 | 2.01x |
| | 3 | 6.699 | 1.299 | 5,168,254 | 1.67x |
| | 4 | 129.0 | 10.70 | 98,090,538 | 1.66x |
| Fat(45/216) | 2 | 0.017 | 0.0004 | 21,462 | 1.53x |
| | 3 | 0.08 | 0.005 | 177,828 | 1.27x |
| | 4 | 0.717 | 0.069 | 1,235,598 | 6.21x |
| Fat(500/8000) | 2 | 0.938 | 0.986 | 778,764 | 2.35x |
| | 3 | 2.190 | 0.441 | 2,620,408 | 2.51x |
| | 4 | 6.856 | 0.970 | 4,230,058 | 2.0x |

Fig. 14. Simulation performance for bounded link failure scenarios.

Figure 13 presents the results of these experiments. As expected, the majority of the time is spent on network simulation. For node failures, ProbNV often computes routes and checks probabilistic assertions within seconds. The only exception to this is the USCarrier network when using a random variable ordering. On the other hand, using the ordering motivated by the program's semantics (BFS) reduces the simulation time to a few seconds. In general, when using BDD-based techniques, one cannot stress enough the importance of a good variable ordering; the number of BDD nodes allocated throughout the simulation reflects this. Unfortunately, finding an optimal ordering for BDD variables is an NP-complete problem [Bollig and Wegener 1996]. However, heuristics based on the application on hand can effectively mitigate this problem, *e.g.*, the BFS-based ordering we use in our evaluation is a reasonable first attempt for any network.

When considering link failures, simulation is orders of magnitude more expensive. This is not surprising, as networks usually have at least twice as many links as nodes; a node failure effectively cuts many links at once. Perhaps, the surprising part is that the simulation struggles the most with datacenter networks. Prior work [Giannarakis et al. 2019, 2020] suggested that networks with highly symmetric designs, such as Fat Trees, are amenable to fault-tolerance analyses due to their high degree of redundancy. However, in our case, simulation fails to compute routes even for a relatively small Fat Tree network with 216 links —these networks can scale to hundreds of nodes and several thousand links. However, reasoning about all link failures requires a high number of BDD nodes (one for each link), even for small networks. Moreover, symmetries become less important, as eventually, with a high number of link failures, many of these symmetries are broken. Instead, the critical factor for all-links fault tolerance's performance appears to be how "dense" a network is. Networks with high vertex degree (*e.g.*, datacenter networks) not only require many BDD nodes but also give rise to more alternative paths to explore.

*Bounded Failures Analysis.* For analysis of a bounded number of failures, we focus on link failures. In this model, we want to answer questions of the form "What is the probability nodes in the network retain connectivity when there are K link failures". To answer this type of question, assuming all links have equal fail rates, we create a symbolic value of type `tedge` for each potential link failure, *e.g.*, for up to 3 link failures we create 3 symbolic values $f_0, f_1, f_2$. We then use conditional assertions of the form:

$$assert(\phi \mid f_0 <_e f_1 <_e f_2)$$
$$assert(\phi \mid f_0 <_e f_1 \; \&\& \; f_1 = f_2)$$

where $<_e$ is the standard $<$ operation applied on edge ids (natural numbers). In other words, we check that "Given that the symbolic values adhere to the given ordering what is the probability that property $\phi$ holds?". Each ordering captures the valid failure scenarios, $(f_0 <_e f_1 <_e f_2)$ describes valid three-link failures, $(f_0 <_e f_1 \; \&\& \; f_1 = f_2)$ describes the valid two link failures, and so on. The ordering constraints ensure that we do not over-approximate the probability of $\phi$ being true; without these constraints, assignments such as $(f_0 = 0, f_1 = 1, f_2 = 2)$ and $(f_0 = 1, f_1 = 0, f_2 = 2)$ are considered distinct failure scenarios, resulting in an over-approximated probability. The results of fig. 14 show that bounded fault tolerance analysis is a more realistic goal for analyzing link failures. In particular, in the case of Fat Trees, simulation takes just a few seconds, even when considering multiple failures for a network with 8000 links —in this case our analysis can leverage the symmetries built into the network. Likewise, for the USCarrier network the simulation time has dropped to just a few seconds when checking up to three link failures (instead of > 2000s for all-link failures). In USCarrier with 4 link failures, complexity (as reflected by the number of BDD nodes) and simulation time skyrockets. Unlike, the large Fat tree, the USCarrier network lacks the redundancy to sustain 4 failures; hence, more unique scenarios arise. Regardless of this, the simulation in question most likely also suffers from bad variable ordering of BDD nodes. Despite only having 4 symbolic values, those symbolic values require many decision nodes; for each symbolic edge we use $\lceil \log_2 |\text{edges}| \rceil$ bits, hence in total we have 36 decision nodes. Moreover, unlike the all-failures case, there is no semantic ordering to these variables, and finding one is not as easy. We conjecture dynamic variable reordering, a heuristic-based technique to automatically reorder variables in BDDs, would be helpful. We leave exploring this direction as future work.

*Checking Assertions.* The performance of the probability computation algorithm (section 4.3) is very sensitive to the complexity of the computed BDDs, as it traverses every path in the BDD that

| Topology | Simulator | Transfer Calls | Merge Calls | Simulation Time (s) | # BDD Nodes |
|---|---|---|---|---|---|
| USCarrier (158/378) | Classic | 9,167 | 26,745 | 32.95 | 18,529,882 |
| | New - 0 | 8,420 | 24,504 | 30.54 | 18,374,538 |
| | New - 1 | 4,165 | 11,739 | 19.43 | 12,587,974 |
| | New - 3 | 3,035 | 8,349 | 17.08 | 13,430,102 |
| | New - 6 | 3,957 | 11,115 | 18.87 | 11,063,150 |
| Fat(80/512) | Classic | 1,932 | 4,772 | 37.94 | 9,021,194 |
| | New - 0 | 1,424 | 3,248 | 36.31 | 8,648,164 |
| | New - 1 | 1,766 | 4,274 | 98.03 | 12,775,000 |
| | New - 3 | 1,876 | 4,604 | 135.6 | 23,789,094 |
| | New - 6 | 1,819 | 4,433 | 148.46 | 30,614,010 |

Fig. 15. Comparison of simulators in all node failure scenarios. **Simulator**: The simulation algorithm used and the search depth if applicable. **Transfer Calls**: Number of calls to the transfer function. **Merge Calls**: Number of calls to the merge function.

corresponds to an assertion. While we have not counted the number of paths, in large networks, such as USCarrier, the BDD of each assertion consists of several thousands decision nodes . In the case of bounded link failures, we could achieve a significant reduction in the number of decision nodes required by the assertions by using a different variable ordering; this is because our current ordering faces an exponential blowup when encoding ordering constraints such as $f_1 <_e f_2$. Still, because BDDs merge isomorphic subgraphs by construction, by memoizing computations on these subgraphs we can scale our algorithm effectively.

## 6.3 Performance of the New Simulation Algorithm

Next, we compare the performance of a simulation algorithm akin to the ones used in ShapeShifter [Beckett et al. 2019a] and NV [Giannarakis et al. 2020] (henceforth referred to as "classic"), with the simulation algorithm of section 5 (referred to as "new"). The classic simulator uses a queue for scheduling nodes, *i.e.*, routes disseminate in a breadth-first manner. For our proposed simulator, we try different bounds for the depth of search of the next node to process in the schedule. First, fig. 13 and fig. 14 already provided some insights on the performance of the two simulators; column "Simulator Speedup" shows the speedup of the fastest time that the new simulator achieved relative to the classic simulator. In most cases, the new simulator achieved a substantial speedup, sometimes exceeding 2x. Figure 15 compares the two simulators in more depth, presenting the number of steps each algorithm takes to converge, in addition to the CPU time. First of all, it is easy to see that the new simulator outperforms the classic one in the wide-area network (USCarrier): when skipping dependencies in the schedule (3rd to 5th row), it requires less than half the calls to the merge and transfer functions compared to the classic simulator, which roughly translates to half the CPU time too. On the other hand, the experiments on the Fat tree network reveal some more subtle performance issues. First, the CPU time of the new simulator is only better when we are not skipping nodes that depend on other nodes in the schedule. Moreover, despite converging in fewer steps (calls to transfer/merge), the CPU time of the new simulator is significantly worse, hinting that there is more to simulation performance than just the number of steps to convergence. In particular, comparing the number of BDD nodes, it appears that in the case of Fat Trees, the new simulator is going through intermediate states that lead to more complex BDDs.

## 7 RELATED WORK

*Probabilistic Control Plane Analysis.* The closest related work is NetDice [Steffen et al. 2020], a tool for probabilistic verification of fault-tolerance properties in distributed routing systems. The

NetDice algorithm uses an iterative process where in every round it prunes the space of failures, eliminating link failures that do not affect whether the given property holds. This strategy allows NetDice to scale well, as it only needs to examine a small subset of the possible states. Unfortunately, a direct comparison of the performance of NetDice and ProbNV is not possible; NetDice uses a custom format for their networks and focuses exclusively on iBGP, a protocol we currently do not support because the translation from router configurations we use does not support it. It appears [Steffen et al. 2020, Figure 8] that NetDice can verify properties similar to the ones we have considered here (*i.e.*, properties concerning one destination prefix) for all-link failures within an hour. It is difficult to extract exact performance numbers, but NetDice verifies most networks that consist of up to 100 links within 100 seconds, with some exceptional cases requiring up to an hour. It can also verify properties under all-link failures for networks that have hundreds or even thousand of links; it is unlikely that our system can scale to networks of this size unless the topologies are very simple (*i.e.*, a network where nodes are connected as a chain). While more scalable, NetDice's approach comes at a cost. First of all, its pruning technique is property-guided; to verify multiple properties one needs to execute the tool multiple times. In contrast, ProbNV computes precise routes for every possible failure scenario first and checks a user assertion second; a user could write multiple assertions and check different properties without incurring the cost of re-executing the network simulation itself. More importantly, to achieve scalability, NetDice trades off generality in the routing features processed. Commonly used routing features, such as OSPF areas, redistribution between protocols, eBGP and BGP local-preference and communities, are are not processed by NetDice. Integrating new protocols and features requires adapting their pruning algorithm, a feat that might be far from trivial. For features such as BGP communities, it might not be possible at all as they render their basic optimization algorithm unsound.

Bayonet [Gehr et al. 2018] is an expressive probabilistic network programming language that can model network functions with high fidelity, including details such as switch queues. Programs written in Bayonet are analyzed by a general-purpose probabilistic inference language. Although Bayonet's ability to model and verify properties of the network's control plane has not been extensively studied, its expressivity leaves little doubt that this is possible. Bayonet appears considerably more expressive than either NetDice or ProbNV. The main roadblock to adopting Bayonet for many network operators would be its inability to scale to beyond small networks (10-30 nodes).

*Deterministic Control Plane Analysis.* Batfish [Fogel et al. 2015] parses the network's configurations and simulates control plane execution to generate a concrete data plane. Unlike data plane analysis tools which operate over a concrete snapshot of the network's forwarding tables, Batfish can proactively analyze changes to router configurations and find bugs before they go live. However, Batfish's representation is completely concrete; it cannot verify the configurations' robustness against arbitrary failures. Tools such as, ARC [Gember-Jacobson et al. 2016] and Tiramisu [Abhashkumar et al. 2020], were designed specifically to deal with fault-tolerance verification. They leverage graph algorithms to scale. MineSweeper [Beckett et al. 2017] creates a symbolic model of the network using first-order logic constraints, and uses an off-the-shelf SMT solver to find properties that are logically inconsistent with the network. None of these tools can reason about probabilistic properties, and extending them to do so seems highly non-trivial, if at all possible.

ProbNV is heavily inspired by NV [Giannarakis et al. 2020]. NV also uses MTBDDs/BDDs to speed up simulation. However, unlike this work, NV could not simulate programs that included symbolic declarations. NV could translate programs that involved symbolic declarations into SMT formulae and then analyze those formula. Unfortunately, an SMT-based approach scales more poorly than BDD-based simulation. To handle symbolic declarations, ProbNV extends NV by introducing a type system to characterize and separate different kinds of data (concrete, symbolic, and multi-value). ProbNV also introduces a new, more efficient, simulation algorithm for network control planes that improves simulation time by up to a factor of two in our experiments. Finally, ProbNV introduces a notation and algorithm for processing probabilistic network programs.

*Data Plane Analysis.* There is rich literature on systems for checking and verifying properties of the network's data plane. Tools such as Anteater, HSA, and NoD [Kazemian et al. 2012; Lopes et al. 2015; Mai et al. 2011] can be used to analyze properties of forwarding tables pulled from real routers. Data plane modeling languages such as NetKat [Anderson et al. 2014], as well as probabilistic variants [Foster et al. 2016], can be used to model and subsequently verify properties complex data planes. More recently, McNetKat [Smolka et al. 2019] extended NetKat with probabilistic operations, as well as efficient BDD-based techniques to analyze them. McNetKat can reason about connectivity in the data plane, and quantitative properties such as expected congestion, but like the rest of the NetKat family, it cannot model control plane protocols such as OSPF and BGP.

*Probabilistic Inference and Model Checking.* Traditional model checking algorithms have been extended to handle probabilistic properties [De Alfaro et al. 2000; Dehnert et al. 2017; Kwiatkowska et al. 2011]. The scope of probabilistic model checkers such as Storm and PRISM goes far beyond ProbNV; they can analyze probabilistic models such as continuous-time Markov chains, and Markov decision processes. These systems also rely on MTBDDs/ADDs to scale, further reinforcing the usefulness of this data-structure in verification. However, unlike ProbNV, these systems are not designed to separate the probabilistic computation from the model exploration (*e.g.*, the route computation in our case) —for good reasons, their expressivity would not be the same if they did. They use MTBDDs/ADDs to exploit symmetries in probabilities throughout the computation, while ProbNV aims to exploit the symmetries in the different paths that arise in a network.

More closely resembling ProbNV in its front-end, are a number of probabilistic programming languages [Chaganty et al. 2013; Claret et al. 2013; Gehr et al. 2016; Geldenhuys et al. 2012; Goodman et al. 2012; Mansinghka et al. 2018; Sampson et al. 2014]. Some of these languages rely on approximate techniques such as sampling, while others rely on building symbolic models such as a Bayesian network which can then be analyzed to perform probabilistic inference.

Dice [Holtzen et al. 2020] is another PPL that uses BDDs to perform exact probabilistic inference. Dice executes the program as if it is deterministic and subsequently applies a weighted model counting algorithm. This algorithm is reminiscent of the one presented in section 4.3, but simpler as Dice does not support datatypes that must be represented using multiple bits. Dice's choices simplify the implementation of observe statements (evidence), allowing it to compute more complicated conditional probabilities than ProbNV, which only allows a single condition in an assertion. In addition, Dice's representation strategy allows it to build BDDs in a compositional manner and to exploit shared structure appearing in different execution paths without mangling the associated probabilities. Nevertheless, for some problems, such as the ones we tackle here, it is prohibitively expensive to lift all computations to a BDD-based representation as Dice does. This observation motivated the design of ProbNV and its multi-mode representation strategy.

## 8 CONCLUSIONS

This paper presented ProbNV, a language to model distributed control planes, and an accompanying symbolic execution engine to efficiently verify probabilistic properties of these models. ProbNV achieves high verification performance by distinguishing between different computation modes and by avoiding construction of an entirely symbolic model of the network. Additionally, ProbNV separates computations concerning probabilities from route computations, avoiding any overhead that probabilistic reasoning tools may induce.

### 8.1 Limitations

ProbNV was designed specifically to facilitate probabilistic reasoning about distributed control planes and many of the design choices were dictated by the specifics of this application domain and particular properties we explored. In particular, symbolic (probabilistic) variables cannot appear in arbitrary positions in a ProbNV program—the type system places a number of constraints on how they may be used. Naturally, other problems may not be compatible with the restrictions imposed by the type system. Moreover, ProbNV lacks some features that are prevalent in other PPLs such as *observe* statements, which can be used to describe conditional distributions. We did not explore adding *observe* statements to ProbNV, since for all practical purposes we were interested in the simpler interface offered by conditional assertions was sufficient.

Another limitation is that the bounded failures model in section 6 is only correct when all links fail with the same probability. We believe this limitation is not fundamental, and for practical scenarios, there are ways around it. However, in the general case, we will need the ability to define joint distributions on symbolics, which is currently outside the scope of ProbNV's design.

Finally, our approach will only work for discrete distributions. More involved network properties such as those oriented around congestion control might require traffic patterns that are described by continuous distributions, and hence cannot be validated with ProbNV.

### 8.2 Future Work

In the future, we plan to investigate further how to scale our simulation procedure, including studying new simulation algorithms and, perhaps most importantly, devising better heuristics to find good variable orderings for BDDs. One promising approach may be to use smaller networks (*e.g.*, a small fat tree) to rapidly test heuristics for the above parameters and then apply them to the (presumably larger) network in question. We are also interested in understanding how more complex failure models would be represented in ProbNV. For instance, operators often possess temporal information such as mean-time to failure for the devices of their network; expressing such high-level failure models is an exciting direction for future work.

From a programming languages perspective, an exciting direction to explore is whether it is feasible to combine the strengths of Dice and ProbNV to build a scalable and expressive PPL for discrete distributions. Another step to generalize this work would be to explore the connection between our type system and of prior work on type systems for information flow and dependency [Abadi et al. 1999; Pottier and Simonet 2003]. In particular, we would like to see whether it is possible to fit our type system into the framework of Abadi *et al.*'s core calculus of dependency [Abadi et al. 1999].

## ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers and our shepherd Arjun Guha for their feedback on improving this paper. This material is based upon work supported by the National Science Foundation under Grant #2030859 to the Computing Research Association for the CIFellows Project, and by the National Science Foundation awards CNS-1703493 and FMitF-1837030. Silva's research is partly funded by a Royal Society Wolfson fellowship and ERC consolidator grant AutoProbe (101002697).

## REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

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

Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (Seattle, WA, USA) *(SIGCOMM '08)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1402958.1402967

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.* 49, 1 (January 2014), 113–126. https://doi.org/10.1145/2578855.2535862

R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1993. Algebraic Decision Diagrams and Their Applications. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design* (Santa Clara, California, USA) *(ICCAD '93)*. IEEE Computer Society Press, Washington, DC, USA, 188–191.

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

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

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019a. Abstract Interpretation of Distributed Network Control Planes. *Proc. ACM Program. Lang.* 4, POPL, Article 42 (December 2019), 27 pages. https://doi.org/10.1145/3371110

Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (Virtual Event, USA) *(HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3422604.3425930

Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2019b. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations: Brief Reflections on Abstractions for Network Programming. *SIGCOMM Comput. Commun. Rev.* 49, 5 (November 2019), 104–106. https://doi.org/10.1145/3371934.3371965

Beate Bollig and Ingo Wegener. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on computers* 45, 9 (1996), 993–1002.

Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691.

Arun Chaganty, Aditya Nori, and Sriram Rajamani. 2013. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics*. PMLR, 153–160.

Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 92–102. https://doi.org/10.1145/2491411.2491423

Edmund M Clarke, Masahiro Fujita, and Xudong Zhao. 1996. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of discrete functions*. Springer, 93–108. https://doi.org/10.1007/978-1-4613-1385-4_4

Luca De Alfaro, Marta Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. 2000. Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 395–410.

Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*. Springer, 592–600. https://doi.org/10.1007/978-3-319-63390-9_31

Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. Netcomplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 579–594.

Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 469–483.

Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *European Symposium on Programming*. Springer, 282–309. https://doi.org/10.1007/978-3-662-49498-1_12

Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language*

*Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 586–602. https://doi.org/10.1145/3192366.3192400

Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 62–83.

Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 166–176. https://doi.org/10.1145/2338965.2336773

Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. 2019. Efficient verification of network fault tolerance via counterexample-guided refinement. In *International Conference on Computer Aided Verification*. Springer, 305–323. https://doi.org/10.1007/978-3-030-25543-5_18

Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An Intermediate Language for Verification of Network Control Planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 958–973. https://doi.org/10.1145/3385412.3386019

Nikolaos Giannarakis, Alexandra Silva, and David Walker. 2021. Appendix to ProbNV: Probabilistic Verification of Network Control Planes.

Joanne Godfrey. 2016. The Summer of Network Misconfigurations. https://blog.algosec.com/2016/08/business-outages-caused-misconfigurations-headline-news-summer.html.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

John Graham-Cumming. 2020. Cloudflare outage on July 17, 2020. https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 140 (November 2020), 31 pages. https://doi.org/10.1145/3428208

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

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

Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*. Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 499–512. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

Nuno P Lopes and Andrey Rybalchenko. 2019. Fast BGP Simulation of Large Datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 386–408. https://doi.org/10.1007/978-3-030-11245-5_18

Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 290–301. https://doi.org/10.1145/2018436.2018470

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 603–616. https://doi.org/10.1145/3192366.3192409

Kieren McCarthy. 2019. BGP super-blunder: How Verizon today sparked a 'cascading catastrophic failure' that knackered Cloudflare, Amazon, etc. https://www.theregister.co.uk/2019/06/24/verizon_bgp_misconfiguration_cloudflare/.

Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling Network Verification Using Symmetry and Surgery. *SIGPLAN Not.* 51, 1 (January 2016), 69–83. https://doi.org/10.1145/2914770.2837657

François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1, 117–158. https://doi.org/10.1145/596980.596983

Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and Verifying Probabilistic Assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/2594291.2594294

Simon Sharwood. 2016. Google cloud wobbles as workers patch wrong routers. http://www.theregister.co.uk/2016/03/01/google_cloud_wobbles_as_workers_patch_wrong_routers/.

Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 190–203. https://doi.org/10.1145/3314221.3314639

Fabio Somenzi. 1997. CUDD: CU decision diagram package. *http://vlsi.colorado.edu/$\tilde{f}$abio/CUDD/* (1997).

Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic Verification of Network Configurations. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication.* 750–764.

Yevgenly Sverdlik. 2012. Microsoft: misconfigured network device led to Azure outage. http://www.datacenterdynamics.com/content-tracks/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Hongkun Yang and Simon S. Lam. 2016. Real-Time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. Netw.* 24, 2 (April 2016), 887–900. https://doi.org/10.1109/TNET.2015.2398197