An Intermediate Language for Network Verification

Nick Giannarakis

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: David Walker

November 2020

Abstract

Computer networks have become an integral part of our daily lives, and essential infrastructure to most industries. This had led to unprecedented growth in their size and complexity. In recent years, misconfiguration-induced outages in networks have become rampant both in frequency and impact. Such misconfigurations are often found in the network's control plane, a distributed system responsible for exchanging routing information between routers. To set the routing policy, operators have to issue per-device configurations in low-level languages, while accounting for interactions with external networks, and potential device or link failures. This is a challenging task, especially for large networks which consist of millions of lines of configuration spread across thousands of devices. To aid operators, researchers have developed a range of static analyses to establish correctness properties of networks. However, developing and maintaining such tools is an enormous undertaking due to the complexity of configuration languages and the plethora of features networking protocols pack.

Inspired by intermediate verification languages, such as Boogie and Why3, this dissertation describes the design and implementation of NV, an intermediate language for verification of networks and their configurations. NV was designed to strike a balance between expressiveness, tractability and ease of use. We show that NV is sufficiently expressive via a translation from a practical subset of real protocols (and their configurations) to NV. Furthermore, we explain how NV enabled efficient implementations (often outperforming the state-of-the-art by an order of magnitude) of standard analyses such as network simulation and SMT-based verification. NV also facilitates the rapid development of new analyses; we present the key insights behind a new, highly scalable fault tolerance analysis, as well as its effortless implementation as a "meta-protocol" in NV. Finally, in a similar but orthogonal approach, we present a new take on network compression —implemented on top of NV— that significantly speeds up verification of fault-tolerance properties.

Acknowledgements

The journey towards a PhD can often become challenging and draining. I am immensely grateful to my advisor David Walker, his guidance and support have turned my journey into a very rewarding experience. Working with David was a master class in how to be a good researcher and a good mentor. I will strive throughout my career to follow the path he showed me.

I am lucky to have worked with great collaborators during my PhD. Ryan Beckett, Devon Loehr, and Ratul Mahajan, all taught me a lot and our collaboration was very helpful in completing this dissertation. I would also like to thank Andrew Appel who advised me and supported me a lot during my first years at Princeton.

I would like to thank Andrew Appel, and Jennifer Rexford for participating in my thesis committee, and Aarti Gupta, Ratul Mahajan for taking up the tedious task of serving as readers.

I wish to thank Viktor Vafeiadis and Ori Lahav for supervising my Masters Thesis and giving me the opportunity to work on interesting research problems with them. I am sincerely thankful to Catalin Hritcu. My career path would probably be very different if it wasn't for Catalin. Catalin gave me the chance to dip my toes into research while still an undergrad, and invested a lot of time on mentoring me.

I want to express my gratitude towards my friends for all the memories we shared together and for always being there when I needed them. I need to make a special mention to Alex for all the fun moments, but mainly for all 5 rocket league games we won together, Sotiris and Themis for the countless hours of basketball, and to Zoe for the endless stream of memories and adventures.

I would like to thank my family for their love, and for always trying their best to provide me with the opportunities I have been blessed with.

Finally, I would like to thank NSF for funding the work in this dissertation through grants 1837030 and 1703493!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer networks have become indispensable infrastructure for modern societies; we rely on networks for a range of daily activities, from entertainment, to communicating with friends and family, to conducting business. And yet, networks are still riddled with misconfiguration errors that adversely affect their reliability.

Network verification has been an emerging research area, seeking to apply techniques from formal methods to proactively detect network misconfigurations. But, despite the tremendous achievements over the past decade, developing such verification tools for networks remains a tedious task. One of the reasons for this, is the level of abstraction that these tools have to operate over. Router vendors, such as Cisco, Juniper, and Arista, each provide their own configuration languages that consist of an enormous number of low-level, ad-hoc commands that operators use to configure the protocols used by the network. Developing richer network models, and experimenting with new verification techniques, requires dealing with an enormous overhead pertaining to the lack of abstractions in modern networks.

The goal of this dissertation is to provide a higher-level of abstraction to developers of network analyses, through an intermediate language designed with network verification in mind. The rest of this chapter is organized as follows: section 1.1 gives

an overview of the relevant network components, section 1.2 discusses network configuration and the challenges it involves, section 1.3 briefly surveys existing work on network analyses and verification, and finally, in section 1.4 we provide an overview of the contributions of this thesis.

1.1 Structure of Networks

1.1.1 The Control Plane

Today's vast networks, such as the Internet, are structured as layers of multiple smaller networks, often operated by different organizations. Routing data packets over the Internet, then becomes a matter of routing packets within, and between these smaller networks. The process of learning routes towards new destinations is similar, independently if it takes place within the network of a single organization, or between networks belonging to different organizations: routing devices exchange *routing messages* with their neighbors (*i.e.* other, physically connected, routing devices). These messages (often called "routes") carry information about a path through the network towards a given destination. This is the primary functionality that the network's *control plane* serves.

Typically, the control plane is coordinated by a number of distributed routing protocols. The need for different routing protocols arises from their different characteristics. For example, the Border Gateway Protocol (BGP) is commonly used for routing across organizations (interdomain routing), because it can scale to large networks with thousands of destinations and it allows for complex routing policies. The latter is handy as different organizations have different routing priorities and business incentives to satisfy; a naive, shortest-path only, routing protocol would not support these needs. On the other hand, when routing within an organization (intradomain

routing), shortest-path routing usually suffices, hence simpler protocols, such as the Open Shortest Path First (OSPF) protocol, are often preferred to BGP.

1.1.2 The Data Plane

The network's data plane is the component responsible for deciding if a packet should be forwarded and if yes, out of which port. To do so it consults the routes computed by the control plane. This architecture provides a separation of concerns: the control plane can implement complex routing policies and react to changes in the environment (*e.g.*, link failures) by running computationally expensive protocols, while the data plane implements a much simpler functionality, allowing for fast processing of incoming packets.

1.2 Routing Policies

Simple routing and forwarding policies do not suffice to achieve the operational standards expected by modern networks. Network operators who manage the network of an organization have a range of specifications to adhere to, for instance, relating to:

1. **Reliability.** Operators strive to ensure connectivity between devices/other networks. The network must have sufficient redundancy to withstand a reasonable number of device or link failures.

2. **Scalability and performance.** There are a number of considerations to be made when it comes to performance: convergence time to avoid extreme packet loss, available bandwidth to be able to push traffic through, etc.

3. **Privacy and Security.** Operators have to deal with the possibility that their network is connected to a bad actor and protect it against attacks [53], such as traffic hijacking, router denial of service, etc.

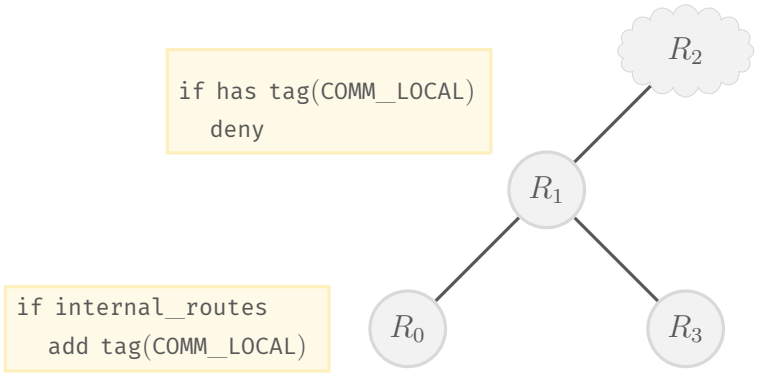
4. **Business considerations.** Businesses have monetary agreements in place that dictate routing policy. For example, a small Internet Service Provider (ISP) may prefer to carry route their traffic through AT&T's network instead of Verizon because they are charged less for it.

1.2.1 Network Configuration

To fulfill the requirements above operators issue *routing configurations* for each routing device in the network they operate. Each configuration describes the physical connectivity of the device, the routing protocols it runs and how they interact with each other. Additionally, routing protocols can be configured via a different set of policy parameters. For instance, it might be desirable to keep some routes internally in the network. Figure 1.1 illustrates how an operator may use BGP tags to signal that a route should not be exported to another network.

Unfortunately, issuing router configurations that *correctly* capture the intentions of operators has proven to be a challenging task. Over the years, network misconfiguration incidents have caused a raft of high-profile outages [47, 56, 46, 41, 28, 40, 43, 29], causing prolonged disruptions in social and business activities. Much of the difficulty in correctly configuring a network stems from the design of the configuration languages themselves. These configuration languages operate at a very low-level of abstraction, making it hard for operators to correctly express their intended routing policies, and to maintain and extend them over time. Notice, for instance, that even for the simple example of fig. 1.1 describing the BGP connections of the network requires dealing with low-level and brittle details such as device IP addresses. Moreover, there are a number of constants spread through each router configuration that serve a variety of purposes. For instance, integers are used for BGP communities, ip prefixes, sequence numbers, router IDs, even names for lists (router's R0 configuration defines a prefix

(a)



(b)

```
bgp router 0
  neighbor 172.16.0.1 remote-as 1
  neighbor 172.16.0.1 send-community
  neighbor 172.16.0.1 route-map RMO out
!
ip prefix-list 0 seq 1 permit 70.70.0.0 ge 16 le 32
!
route-map RMO permit 10
  match ip address prefix-list 0
  set community 1:1
route-map RMO permit 20
```

(c)

```
bgp router 1
  neighbor 172.16.0.0 remote-as 0
  neighbor 172.16.0.2 remote-as 2
  neighbor 172.16.0.3 remote-as 3
  neighbor 172.16.0.2 route-map RMO out
!
ip community-list standard COMM_LOCAL permit 1:1
!
route-map RMO deny 10
  match community COMM_LOCAL
```

Figure 1.1: (a) Example of a network using community tags to avoid leaking some routes to external peers. Node R_0 is considered the destination node. (b) and (c) are BGP configurations implementing this policy in CISCO-IOS.

list with the name 0!). Obviously as the network grows, it is difficult to maintain these configurations and keep them synchronized with each other.

Given the low-level configuration languages, the distributed nature of routing protocols, and the higher-rate of hardware failures compared to general-purpose com-

puters, it is rather unsurprising that network misconfigurations occur. As with all software, bugs are expected, especially when dealing with large-scale software that, inevitable, large networks require. To address network misconfigurations we need tools to *specify* and *verify* a network's routing behavior.

1.3 Network Analyses and Verification

For years, operators relied on standard tools such as traceroute to establish correctness of their networks. Such tools can generally answer questions about the forwarding behavior of a data packet in a specific *environment*, *i.e.* for a particular instance of the data plane. They cannot provide *exhaustive* guarantees, as the space of packets is generally huge —each packet has multiple fields and ideally one wants to consider all possible combinations of them— one can only try out a few combinations before eventually giving up. Moreover, these tools can answer questions about packets for the *single* data plane currently installed on the routers. In other words, they cannot answer *"what if"* questions, such as "what happens if router A crashes" or "what happens if we apply this patch to the BGP configuration of router A". These questions require recomputing the data plane, in a *proactive* fashion, *i.e.*, without actually making these changes to the live network.

1.3.1 Exhaustive and Proactive Network Analyses

To help improve network reliability, researchers have developed numerous verification and static analysis tools for networks. One wave of tools included Anteater [38], Header-Space Analysis (HSA) [31], Veriflow [32], NetKAT [5, 21, 50], NoD [36] and Bayonet [22]. These tools were designed to provide a more exhaustive analysis of the data plane; they use symbolic reasoning to analyze the forwarding behavior of *any* data packet injected in the network. The tools developed can check both deterministic

and probabilistic properties of how packets flow through the data plane, and scale to networks with thousands of devices and millions of packet-forwarding rules.

However, these tools only analyze a single snapshot of the data plane. To ask "what if" questions one needs to also model the control plane. Tools such as, Batfish [20] and FastPlane [37], parse the routers configurations and given an environment (*e.g.*, which devices or links have failed, if any) *simulate* the network to create a model of the data plane. Other approaches, such as SMT verification [8] can even reason about a *symbolic* environment, where, in contrast to Batfish and FastPlane, factors such as link failures and routing messages from peer networks are not fixed a priori to the analysis. These kind of analyses, give operators the invaluable ability to proactively reason about changes to their network without applying them to the live network.

Challenges in Developing Networking Analyses. While there have been many network analyses tools over the past decade, developing such tools requires a lot of effort. A roadblock many networking researchers stumble upon early on, is the complexity and variety of the vendor configuration languages. Cisco, Juniper, Arista, *etc.*, each have their own proprietary configuration languages that consist of an enormous number of ad hoc commands; Cisco IOS [13] alone contains over 15000 configuration commands, including over 300 for BGP alone! To analyze a network control plane, one must first parse these varied configuration languages and interpret their semantics. Fortunately, nowadays this problem is mostly solved by using Batfish [20], as its parser can parse configurations from multiple vendors, to create a common intermediate representation (IR) on top of which researchers can develop their own analyses. Indeed, researchers from a number of different institutions have built verification and simulation tools using Batfish as a front-end [8, 24, 9, 23, 18, 10, 25]. However, even when using Batfish to parse the configurations, one has to deal with the exceptional

number of low-level details that are present in these configurations, which hinders development of analyses as explained in chapter 3.

Another challenge is that verification tools need to scale to the size of modern networks. Hyper-scale datacenters have millions of lines of configuration spread across thousands of devices. Tools that give strong correctness guarantees but do not scale cannot validate the larger networks, which in proportion, handle a large volume of traffic. It is also unlikely to find a "hammer" that can verify all properties of interest for any network. Rather, we we will need different models and analyses to handle different networks and properties.

1.4 Contributions

The goal of the research presented in this thesis is to simplify the development process for new network models and new network analyses. To this end, we developed, *abstractions* and infrastructure that other researchers can build upon to develop their own network models and analyses. In particular, our contributions are summarized as follows:

1. The design of *NV*, a functional language for modeling the routing behavior of networks (chapter 3). NV provides expressive and compositional constructs that can be used to express complex network protocols, yet is sufficiently simple to allow for efficient analysis and verification of programs written in it.

2. In Chapter 4, we use NV to build models of the control plane and the data plane. We developed tools to automatically translate router configurations of control plane protocols such as BGP and OSPF, and of data plane ACLs, to the NV language. Using a higher-level declarative language, makes it easy to compose simpler building blocks to generate complex network functions. For instance, it is easy to compose a program representing the control plane, with

a program representing the data plane, and hence to describe both aspects of a network's forwarding behavior.

3. The implementation of a network simulator and an SMT-based verifier over NV programs (Chapter 5). We show how NV's design facilitated the implementation of a high-performance simulator that uses efficient datastructures such as *Multi-Terminal Binary Decision Diagrams* (MTBDDs), and relies on native code execution to interpret NV functions, allowing it to scale to large networks with thousands of routers and tens of thousands links.

   Additionally, we developed an SMT-based verifier, similar to MineSweeper [8], but more flexible as it operates over any NV program. We show how NV enables a more principled approach to implementing a sophisticated analysis such as an SMT verifier. Much like a regular compiler, NV relies on an optimizing pipeline to produce tractable constraints. Compared to MineSweeper, the state-of-the-art control plane verifier, this not only results in code that is more maintainable, but as we show, NV's systematic approach to optimizations results in improved performance too (up to 10x times faster, and scaling to networks that MineSweeper cannot handle).

4. We demonstrate that NV's expressive design can help researchers formulate new network analyses with minimal effort. By providing a *programming language* that allows users to express network models easily, one can directly construct non-standard models that correspond to new analyses. NV makes it easy to play with these non-standard models, rapidly prototyping one and then the next to see what does and does not work.

   One theme of this thesis is reasoning about fault-tolerance properties, a problem that has proven to be challenging, as existing approaches would either be incomplete or unable to scale. We present a novel fault tolerance analysis

implemented directly as an NV program (Chapter 6). Contrary to previous approaches to fault tolerance [8, 25, 24, 20], this analysis scales well, computes precise routes, does not impose restrictions on the policy or features used, and is exhaustive.

5. Finally, we developed Origami, a new tool for verifying reachability properties in the presence of failures, targeting very large networks. Our algorithm is based on the idea of control plane compression [9]. We implemented Origami as a backend to NV, and proved its correctness via a theory of control plane approximation that relates the solutions of the original and the compressed networks.

Acknowledgements This dissertation builds on published articles on NV [26] and Origami [25], and includes contributions to these papers from my coauthors: Ryan Beckett, Devon Loehr, Ratul Mahajan, and David Walker.

Chapter 2

Prelude: Network Routing

This chapter serves as a gentle introduction to routing traffic in networks. Its purpose is to describe the commonly used routing protocols and their features, especially focusing on the parts modeled in later sections. It also introduces the semantics of networks on top of which the tools developed in this thesis were built.

2.1 Forwarding Traffic Over the Data Plane

As the introductory chapter explained, the purpose of the data plane is to direct traffic from one device to another, when given a route between these two devices. More specifically, each device maintains a table called Forwarding Information Base (FIB). At a high-level, the FIB associates destinations to the link(s) over which traffic packets must be forwarded (commonly referred to as next-hop)[1]. Each data packet arriving at a router has a destination IP address associated with it. The router must determine whether it needs to forward it to another device and if so which one(s). To do so, it performs a lookup in the FIB to see if there is a known next-hop for this destination.

[1]In practice, next-hop refers to an IP address that corresponds to an interface on the routing device but these details are mostly irrelevant to the presentation and the goals of this research.

Longest Prefix Match. The challenge, however, is that there are 2^{32} unique destinations for IPv4 and 2^{64} for IPv6; storing and performing lookups in tables of that size is impractical. Instead of maintaining the next-hop for each destination IP individually, the FIB keeps track of the next-hop for a *routing prefix*. Intuitively, a routing prefix describes a set of IP addresses. In IPv4, a routing prefix consists of an IP address (a 32-bit integer, written as 4-octets `a.b.c.d`) followed by a mask written as "/n", where n is an integer number in the range $[0, 32]$. An IP address *matches* a prefix, when its m most-significant bits coincide with the IP address of the prefix. For example, 192.168.0.250 matches 192.168.0.0/24, because the first 24 bits (192.168.0) are the same.

To compute a next-hop for a given destination IP address, a router checks whether the destination IP matches an IP prefix stored in its FIB. One issue that arises by storing sets of destinations, is that these sets may overlap. For instance 192.168.0.250 matches 192.168.0.0/24 but also matches 192.168.0.251/31. Routers resolve this ambiguity via the *longest prefix match*, *i.e.* by choosing the prefix with the greater mask. In the previous example the next-hop associated with 192.168.0.25/31 will be used. This allows for a hierarchical approach to routing; the longest prefix length is preferred because it represents a more specific destination.

Filtering Data Packets. Operators can also configure packet filters at the data plane level, known as Access Control Lists (ACLs). These filters are local to a device, they do not affect the route computation[2]; they are only applied to the traffic flowing through the device and their primary use is to provide a basic level of security. An ACL describes a condition on the packet headers (*e.g.* the source/destination IP, the network protocol, etc.) and action (allow/drop) when a packet matches this condition.

[2]In practice, however, they can be used to filter routing messages from peers and hence indirectly affect the route computation

2.2 Routing Protocols Overview

The role of the control plane is to compute the FIB that the data plane uses to learn where to forward traffic for each destination. There are two common ways for the control plane to select a route for each destination: 1. by manual configuration of the next-hop for each destination (*static routing*), or 2. by running one or more distributed routing protocols in which neighboring devices exchange new routes until all devices have learned all available routes (*dynamic routing*).

Both approaches pose their own advantages and disadvantages. Static routes do not require CPU/memory intensive protocols to be executed to compute the routes. Moreover, they may be considered more secure in the sense that routes will not leak outside the network unless explicitly configured to do so. But, most importantly, they are in general very simple to set up and manage for small networks. However, as networks grow configuring and maintaining static routes quickly becomes intractable and a common cause of network misconfigurations such as routing loops. Moreover, static routes cannot adapt to dynamic changes in the network such as link or devices failures.

2.2.1 Dynamic Routing

In practice, operators often use a combination of static and dynamic routing, where the bulk of the route computation is carried out by dynamic routing protocols, and static routing is used selectively for certain parts of the network or as backup paths. Typical examples of dynamic protocols are the Border Gateway Protocol and the Open Shortest Path First protocol. As mentioned in chapter 1 the two protocols differ in how they function and what features they offer, and consequently on how they are used.

2.2.1.1 Open Shortest Path First

OSPF is a *link-state protocol*. In link-state protocols, routers exchange information about the topology (*i.e.* every device informs its neighbors about its connections) and every router constructs a weighted graph that represents the network. Each router then independently calculates the best route for each destination by running Dijkstra's shortest path algorithm. This makes OSPF a relatively simple protocol that avoids problems like non-convergence, yet allows for some policy flexibility by adjusting the weights of each link.

On the other hand, link-state protocols require sharing information about the entire topology with each node in the network, which largely dictates why OSPF should only be used for intradomain routing. Firstly, if OSPF was used to route across different organizations then details about each organization's network would leak which poses a security risk. Another factor is that it becomes intractable to flood link-state information through large networks. OSPF mitigates these issues to a certain degree via a feature called *areas* which allows an operator to split OSPF routers into groups. When OSPF areas are used, each router only exchanges topology information about the devices that belong in the same area. To communicate routes between two areas one or more devices act as *area border routers*, *i.e.* they belong to both areas. A router then typically prefers routes learned within the area it belongs to (intra-area) as opposed to routes learned from another area (inter-area). It also prefers routes originating from within the network rather than from other networks (external routes). If two OSPF routes are of the same type (e.g. intra-area), the path weight is used to pick the best one. Figure 2.1 illustrates some of these details of OSPF areas. In this OSPF network with three areas, area 0 is the *backbone area*; each other area must connect to this one. This is achieved by two area border routers (ABR in fig. 2.1), respectively connecting areas 1 and 2 with area 0. Finally, in this network, there is one router that connects with a peer network (commonly referred

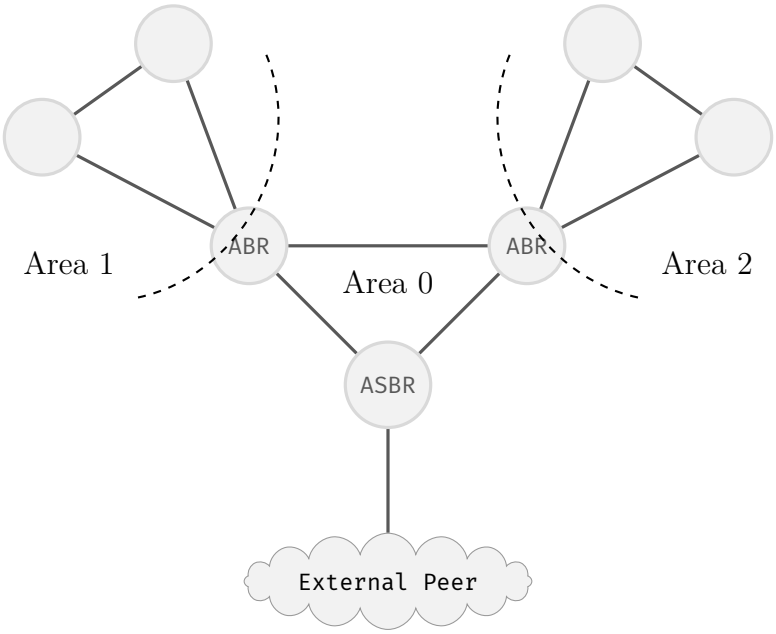


Figure 2.1: An OSPF network with three areas.

as autonomous system in networking terminology); such routers are designated as an *area system border router* (ASBR in fig. 2.1).

2.2.1.2 Border Gateway Protocol

BGP was designed primarily to exchange routing information between networks that belong to different organizations. One of the advantages of BGP in this context, is that it is a path-vector protocol; the path traversed is stored in the route and routers can make routing decisions based on it. More importantly, routers can inspect the traversed path to detect and avoid routing loops. This offers flexibility in terms of routing policy, which allows different organizations to accommodate their business needs. Another advantage, is that BGP can scale to very large networks that exchange routing information for hundreds of thousands of destinations.

Beyond the traversed path, a BGP route contains various other attributes that are used to affect routing behavior. One such attribute is the *local-preference* value, which essentially overrides the path length as the metric of how desirable a route is;

i.e. routes with higher local-preference value are preferred regardless of their length. Another commonly used attribute is the set of *communities*. Communities are tags that a router can attach, remove or test for on a route. They are used to augment the routes with "extra" information that can be used to determine routing policy. For instance, a tag may denote to a router to "not export this prefix to peers" or to "increase the local-preference value for this route". A third BGP-specific attribute is the *multi-exit discriminator* (MED) value. The MED is used to disambiguate routing when two networks have multiple connections between with them. When a router sets a lower MED value to a route over a certain link it informs the receiving network that it prefers to receive traffic over this link.

Finally we note that while the original design goal of BGP was to exchange routing information across networks operated by different organizations, its ability to scale to very large networks have led large cloud providers to adopt it in their datacenters [34].

2.3 Formal Semantics

To reason about the behavior of networks we must define a semantics for routing protocols and packet forwarding. We build on previous work on routing algebras by Sobrinho [51] and the stable paths problem (SPP) by Griffin *et al.* [30]. Their work focused on convergence properties of control planes, and characterizing the *converged* state of a network's control plane. The research in this thesis is primarily concerned with the latter; our goal is to study the behavior of the converged state of the network and not properties about transient states, whether the network converges, or how long it takes to converge. Moreover, while the original semantics proposed by Sobrinho and by Griffin were intended to model the control plane, we show that adopting a slight generalization of these semantics [16, 10] allows us to also model other functionalities,

such as meta-protocols that implement network analyses (section 6.3), or the data plane (section 4.2).

2.3.1 Network Model

In our model of the stable paths problem the network's topology is described by a graph, where the nodes V represent the routing devices and the edges E represent links between those devices. To perform network operations, nodes communicate information by exchanging messages with their neighbors. The information contained in these messages depend on the network operation performed. For instance, when modeling the data plane, nodes exchange data packets, while in a routing protocol like OSPF the messages would include the destination prefix and the route's cost. Formally, we call such messages *attributes* and the first task of defining an instance of such a network model is to define the *type* of attributes A that nodes exchange.

Given the type of attributes, the rest of the model consists of:

- The *initial* information each node bares. For instance, in the context of routing, a node may announce a route to a prefix destination, this would be captured by its initial state. Formally, the initial state is described by a function from nodes to attributes.

- A *transfer* function that describes how attributes are modified as they are communicated through the edges of the network. For instance, this function may increase the hop count of a route transmitted over a link by one. The transfer function is a function from links and attributes to attributes, as the modification applied need not be uniform across all links.

- A *merge* function that describes how a node combines the messages received from its neighbors. For instance, a merge function for a routing protocol combines two routes by comparing them and choosing the "better" one. The defini-

SPP instance $(G, A, \mathsf{init}, \mathsf{trans}, \mathsf{merge})$			**SPP solution** $\mathcal{L} : V \to A$	
V		*topology vertices*	$\mathsf{choices}(u)$	$= \{a \mid e = \langle v, u\rangle,\ a = \mathsf{trans}\ e\ \mathcal{L}(v)\}$
E	$:$	$V \times V$ *topology edges*		
G	$=$	(V, E) *network topology*	$\mathcal{L}(u)$	$= \mathsf{init}(u) \oplus a_1 \ldots \oplus a_n$
A		*attribute type*	**when**	
init	$:$	$V \to A$ *initial routes*	$\{a_1, \ldots, a_n\}$	$= \mathsf{choices}(u)$
trans	$:$	$E \to A \to A$ *transfer function*	$x \oplus y$	$\triangleq \mathsf{merge}\ u\ x\ y$
merge	$:$	$V \to A \to A$ *merge function*		

Figure 2.2: Stable Paths Definition and Solutions

tion of what is a "better" route depends on the protocol modeled, it can mean the shortest route, or it can mean a route from within the same OSPF area, etc.

2.3.1.1 Stable Solutions

To verify properties of a network, we need to compute its *stable states*, also known as its *solutions*. In general, a *state* of a network is a labelling function $\mathcal{L}$ that maps each node to a route. Such a state is *stable* when given a node and the routes associated with its neighbors, there is no incentive for the node to adopt a route that differs from the one it already has. More precisely, we define the *choices* of node u as the routes received from neighbors, *i.e.* the set of routes computed by applying the transfer function over each edge (v, u) and the label $\mathcal{L}(v)$ of the neighbor

$$\mathsf{choices}(u) = \{a \mid e = \langle v, u\rangle,\ a = \mathsf{trans}\ e\ \mathcal{L}(v)\}$$

The label of a node u can then be described as a combination of the *choices* provided from its neighbors and its initial route:

$$\mathcal{L}(u) = \mathsf{init}(u) \oplus a_1 \ldots \oplus a_n$$
$$\textbf{when } \{a_1, \ldots, a_n\} = \mathsf{choices}(u)$$
$$\textbf{and } x \oplus y \triangleq \mathsf{merge}\ u\ x\ y$$

When the equation above holds for every node u in the network simultaneously, the function $\mathcal{L}$ defines a stable state for the entire system. Our formulation is similar to that of Bonsai [9]. While not formulated in exactly the same way, these definitions reflect essentially the same notion of stability as developed in earlier work by Griffin *et al.* [30] and Sobrinho [51].

2.3.1.2 Computing Stable Solutions

Simulating Routing Protocols. The simplest way to compute a stable state is through simulation. Network simulation mimics the route exchange process in which routers engage. It computes a fixpoint of the process in which each node uses the merge function to select the best route among the received ones, then modifies that route according to the transfer function and further propagates it to its neighbors. Algorithm 1 shows a simple algorithm implementing the computation of this fixpoint. Essentially, every node in the network pulls in the routing information from its neighbors and computes a new route based on this information (lines 13-15); if a better route has emerged then it updates its label (line 3) and signals its neighbors to recompute their route (line 4). In section 5.1.1 we discuss a more sophisticated and efficient version of this algorithm.

SMT verification. A second way to compute stable solutions is to use an SMT solver. The SMT-based approach, pioneered by MineSweeper [8], does not model the

Algorithm 1 Network Simulator

```
1: procedure UPDATE(ℒ, q, u, route, E)
2:     if route ≠ ℒ(u) then
3:         ℒ(u) ← route                                  ▷ Update the label of u
4:         q ← q ∪ {v | (u, v) ∈ E}                      ▷ Add the neighbors of u to the worklist
5:
6: procedure SIMULATE(V, E, init, trans, merge)
7:     q ← {}
8:     for u ∈ V do
9:         ℒ(u) ← init(u)                                ▷ Best route of node u
10:        q ← q ∪ {u}
11:    while q ≠ empty do
12:        u ← pop q                                     ▷ Select a node from the worklist
13:        acc ← init(u)
14:        for v ∈ {(v, u)|(v, u) ∈ E} do
15:            new ← trans((v, u), ℒ(v))                 ▷ Pull-in routes from neighbors
16:            acc ← new ⊕ acc                           ▷ Update route of u
17:        UPDATE(ℒ, q, u, acc, E)
```

convergence procedure; instead, it captures the stable solutions of the network using the constraints of fig. 2.2.

While not very common, some networks may have multiple stable solutions, and the one the network converges to may depend on the order in which messages are exchanged. One advantage of the SMT-based approach is that it captures *all* stable solutions of a network. In contrast, simulation can only explore one possible message ordering and hence can only analyze at most one stable solution.

Abstraction Level. In analyses and verification there is always an interesting decision to be made; just how *detailed* the model of the system in question should be? The answer to this depends on a number of factors:

1. What kind of properties do we want to check?

2. How expensive (in terms of computational cost) is it to analyze models of the system in question?

3. How pragmatic is it to develop a sound and precise model?

The first question is a good starting point. Knowing the properties of interest, we can focus on models that capture them, avoiding models that are too simple and hence useless, or too complicated and require a lot of effort to develop and analyze. This thesis —and much of the related work— focuses on *path properties* of the *converged* state of the network, and attempts to answer questions such as:

- Can router A "speak" to router B?
- Do these routers retain connectivity if a number of links and/or routers fail?
- Will traffic from peers traverse a firewall before it enters the network?
- How many hops away is router B from router A?

These properties are very important to operators and we have developed models and analyses on top of them that allows us to check them relatively efficiently. Notice that both the semantics of fig. 2.2 and the simulation algorithm algorithm 1 ignore many of the low-level details a real implementation has to deal with; for instance, regardless of whether the protocol is a link-state protocol or a path vector protocol, they are treated identically, and this does not matter for the properties listed above.

On the other hand, the models and algorithms are not detailed enough to answer questions such as "will the network converge to a stable state?", or "how fast does the network re-converge after a failure", or "will there be congestion if some of the links/devices fail?". Some of these (the former two) are likely impossible to answer using the semantics of fig. 2.2 and the algorithms presented in chapter 5. Others, like questions about traffic congestion may fit well into this framework, although further research is required to determine this.

Chapter 3

An Intermediate Verification Language for Networks

3.1 Motivation

As seen in section 1.2.1 and chapter 2, to develop effective network analyses, one needs to model various routing protocols and to model the semantics of the different vendor-specific proprietary configuration languages used to control those protocols.

Figure 3.1 shows a configuration snippet for a single router in the Cisco IOS format; briefly, the first part (lines 1-3) describes physical connections with other devices, while the second part (lines 5-14) describes the protocols that the router is running (OSPF, Static, BGP) and their configurations. For example, line 12 tells OSPF to inject statically configured routes, such as the one configured on line 5, into its routing table. Finally, lines 16-23 define routing policies. Even in this small snippet, some of the challenges for formal analysis of such configuration languages quickly become apparent:

1. Complex instruction set: Many distinct commands perform logically similar operations. For example, line 11 assigns the value 70 to the administrative distance

```
1 interface Ethernet0 ip address 172.16.0.0/31
2
3  ip route 192.168.1.0 255.255.255.0 192.168.2.0
4  bgp router 1
5    redistribute static
6    neighbor 172.16.0.1 remote-as 2
7    neighbor 172.16.0.1 route-map RMO out
8
9  router ospf 1
10  redistribute static metric 20 subnets
11  distance 70
12  network 192.168.42.0 0.0.0.255 area 0
13
14 ip community-list standard comm1 permit 1:2 1:3
15 ip prefix-list pfx permit 192.168.2.0/24
16
17 route-map RMO permit 10
18   match community comm1
19   match ip address prefix-list pfx
20   set local-preference 200
21 route-map RMO permit 20
22   set metric 90
```

Figure 3.1: A small fragment of a router configuration.

of redistributed routes, and line 20 assigns the value 200 to the BGP local-preference parameter. Similarly, they use specific data structures, such as `prefix-list` and `community-list` to match routes, instead of generic data structures that serve multiple purposes. Analyzing router configurations at this level of abstraction is unnecessarily tedious.

2. Lack of reusable building blocks: Configuration languages lack *building blocks*. Building blocks provide flexibility in terms of creating new protocol models, and expressing transformations or abstractions of them, which often help scale verification tools to large networks. The unconventional abstract routing protocols implemented by tools such as ShapeShifter [10], or the "meta-protocols" described in later chapters of this thesis (chapter 6) cannot be expressed in the rigid configuration languages used by vendors.

3. Incomplete semantics: The semantics of the protocols are not explicit in the configuration. Instead, they are scattered throughout the configuration and the protocol RFC. For instance, how the BGP protocol selects a route is partially captured through configuration and partially specified by the RFC. This complicates the implementation of new analysis as one has to understand in detail such implicit behaviors.

3.1.1 Intermediate Languages

Batfish [20] demonstrated how colossal this task can be; even parsing the configuration languages is a lot of work that requires a team of engineers to manage.

While Batfish provides a very useful front-end for managing router configurations, it does not attempt to fundamentally change the level of abstraction present in the vendor configuration languages. Doing so has the advantage of being expedient and straightforward, but causes Batfish to inherit some of the less desirable properties of the configuration languages themselves. For instance, the Batfish IR is quite verbose: a recent examination showed that Batfish's representation of BGP routing policy uses 24 statements and more than 100 types of expressions. These include 19 different statements just to modify specific fields of a BGP message.

Besides the size and non-standard constructs, Batfish also has to deal with the lack of "'building blocks"' necessary to represent new routing protocols or features. Adding new vendor features requires adding new components to its IR, making it challenging to maintain the various analyses tools that build on top of it.

3.1.2 A Low-Level IR for Network Verification

Building sophisticated static analysis tools is no easy feat; it requires familiarity with different domains such as programming language semantics, automated reasoning techniques and, in this case, networking. Fortunately, this problem has been extensively studied in the programming languages and verification communities. Systems

like Boogie [35], Why [19], CIVL [48], have shown how to greatly simplify the task of developing such tools by introducing an intermediate verification language (IVL). In this scenario, developing analyses over an intermediate representation such as Batfish, reduces to two separate, but simpler, tasks: 1. translating the *surface-level* intermediate representation (e.g. Batfish's IR) in the intermediate verification language, and 2. building analyses over the IVL.

Such an architecture separates key concerns and simplifies each aspect of the pipeline. The front-end (*i.e.*, step 1) only needs to capture the semantics of the input program (or configurations in our case) and should be independent of what verification techniques will be used. Likewise, the various back-ends need not be concerned with the various vendor configuration languages and the intricacies of the protocols used, rather they only have to analyze the constructs of the (hopefully simpler) intermediate verification language.

The key challenge comes in the design of the intermediate verification language: it should be *succinct* to allow rapid development of new analyses, *expressive* enough to encode the semantics of the source languages, yet *tractable* to admit efficient analysis of its features.

A Functional Language for Modeling Networks. The first contribution of this thesis is the design of *NV*, a functional language for modeling the routing behavior of networks. NV uses *conventional*, *expressive* and *compositional* constructs with ordinary, broadly-understood semantics, such as integers, booleans, functions, maps, and records. Additionally, NV provides constructs specific to verification: it allows users to express unknowns, such as potential failures or actions of neighboring networks, in a general way, using *symbolic values*, and lets users specify network properties via *assertions*.

NV's design provides the building blocks to model both standard routing protocols and variations thereon. The latter is quite useful as large cloud providers have been known to tweak the behavior of standard protocols for internal use; accommodating these tweaks is easier in NV, as the language itself —and hence the analyses build on top of it— do not need to be modified. These building blocks also facilitate the implementation of network transformations, such as message [10] or topology [9, 25] abstractions, as they can be implemented as NV-to-NV transformations, independent of one another and of the back-end analysis used. In addition, standard program optimizations, such as inlining or partial evaluation, have a well-defined meaning in this conventional language and can be implemented as NV-to-NV transformations. Prior to the research presented in this thesis, designing a network verification tool involved simultaneously interpreting the low-level commands from various vendors (Cisco, Juniper, Arista, Force10, *etc.*), optimizing their representation, and converting them to the appropriate structure (*e.g.*, SMT constraints) all at once.

3.2 Language Design

3.2.1 An Example Model of a Distributed Routing Protocol

We illustrate the basic pieces of NV, and how they can be used to build and verify models of distributed control planes, through a simple example of a network running a cut-down model of BGP. To describe such a model in NV one defines:

1. a *topology*, defining the nodes and edges comprising the network,
2. a *transfer* function, defining how routes are transformed as they are propagated through the network,
3. a *merge* function, defining how a node selects a best route, and
4. an *init* function, defining the routes announced by each node.

(a)

```
type bgp = {length:int; lp:int; comms:set[int]; origin:tnode}

let transBgp (e: tedge) (x: option[bgp]) =
  match x with
  | None -> None
  | Some b -> Some {b with length = b.length+1}

let mergeBgp u x y =
  match x,y with
  | _, None -> x
  | None, _ -> y
  | Some b1, Some b2 ->
    if b1.lp > b2.lp then x else if b2.lp > b1.lp then y
    else if b1.length <= b2.length then x else y
```

(b)

```
include bgp
let nodes = 5
let edges = { 0=1; 0=2; 1=4; 2=4; 1=3; 2=3; }
symbolic route : option[bgp]

let init (u : tnode) =
  match u with
  | 0n -> Some {length = 0; lp = 100; comms = {}; origin = 0n }
  | 4n -> route
  | _ -> None

let l = solution {init = init; trans = trans; merge = merge}

let safe x =
  match x with
  | None -> false
  | Some b -> b.origin = 0n

assert (safe l[ 1n ]) && (safe l[ 2n ])
```

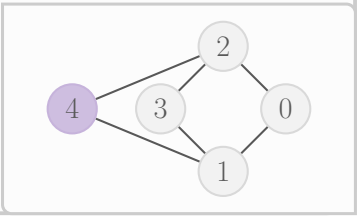


Figure 3.2: (a) shows a basic model of BGP in NV. (b) shows a network running BGP. Nodes 0-3 model an internal network and node 4 a peer announcing an unknown route to nodes 1-2. Can we verify that node 4 cannot hijack traffic from our network?

Figure 3.2a presents a model of BGP in NV. In this model, we consider a single destination and represent routes towards it using *optional* values. Optional values provide a distinction between expressions that return a value (`Some v`) and ones that do not (`None`). We use a `None` value to indicate the absence of a route; otherwise, a

BGP route consists of a record with four components: the path length (`length`), the local preference (`lp`), a set of integers representing BGP communities (`comms`), and a BGP route origin (`origin`) indicating which node initially announced the route. In our working example, the transfer function uses a *match* expression, to deconstruct the input route. If there is no route, then the output is left unchanged. Otherwise, it simply increases the path length by 1, but in general, it can be used to describe more complex BGP policies. Finally, when choosing between two routes, the *merge* function implements (part of) the BGP ranking algorithm, first comparing the local-preference and breaking any ties with the shortest path.

Having defined a model of BGP as in fig. 3.2a, we can use NV to specify and verify properties about a network running BGP. Figure 3.2b describes a network with 5 nodes, an initial route for each node and an assertion to verify.

Modeling Unknowns. Operators are often interested in verifying properties with respect to factors outside their control. Such factors could be potential hardware failures (a failed link or device), or a route sent from a peer network. Inspired from other solver-aided languages [59], NV uses *symbolic values* to model such "unknowns". A symbolic value is not bound to a single concrete value; rather it represents any possible value of its type. In the example of fig. 3.2b a symbolic value models the fact that node 4 may send arbitrary routes. Intuitively, this represents an external network (node 4) that peers with our network (nodes 0-3).

Defining the Solutions of a Protocol. Once a topology and a semantics for a distributed protocol (*i.e.*, the `init`, `transfer`, and `merge` functions) have been defined, we can compute its stable state (fig. 2.2) using a *solution* declaration. In general, an NV file can include multiple solution declarations, each using their own `init`/`trans`/`merge` functions and thus defining their own instance of the stable paths problem. However, at this point, they have to use a common topology. Moreover

there can be *dependencies* between two SPP instances, where the solutions of the first declaration are used in the functions of the second.

Specifying Properties. To specify a property of the converged (stable) state of the network we use an *assertion*. An assertion is simply a boolean expression; as all NV expressions, it can refer to previously let-bound NV expressions, including solutions to SPP instances. In the case of fig. 3.2b, our specification asserts that node 4 cannot hijack traffic from our network, *i.e.* that the edges nodes 1 and 2 prefer the route that originated from node 0.

3.2.2 Syntax

NV adopts many of the features of a typical functional language (fig. 3.3), including let-bindings, match statements, (non-recursive) functions, and data structures such as options, tuples, records, and maps. NV also has a set data type, which is implemented as a map to boolean values. The base types are booleans, integers, nodes, and edges. Furthermore, integers are parametrized by a number of bits; for example, we write `int8` for the type of 8-bit integers and `5u8` for the 8-bit representation of the value 5. Lack of an annotation means a 32-bit size. Specifying the number of bits allows for more accurate modeling of protocol semantics and enables time and space savings in MTBDD-based analyses (section 5.1.2.1). NV also supports let-polymorphism; the user can write polymorphic functions that operate over different types, though the messages exchanged between nodes must resolve to a single concrete type.

An NV program consists of a series of *declarations* (`d`). Some of these declarations are used to describe networks and their stable solutions. In particular, `nodes` and `edges` describe the topology of a network. The `solution` declaration is used to describe the stable solutions of a network as defined in fig. 2.2. In particular, `solution` takes as input a record containing the `init`, `transfer`, and `merge` functions of a stable

v	::= $\mathbb{N}u\mathbb{N}$	*fixed-width integer value*
	| `true`	
	| `false`	
	| `None`	*empty optional value*
	| `Some v`	*present optional value*
	| $(\mathtt{v}_1, \ldots, \mathtt{v}_2)$	*tuple value*
	| $\{\ell_1 : \mathtt{v}_1; \ldots; \ell_n : \mathtt{v}_n\}$	*record value*
e	::= `v`	*value*
	| `x`	*variable*
	| `let x : ty =` $\mathtt{e}_1$ `in` $\mathtt{e}_2$	*local name binding*
	| `fun (x : ty) ->` `e`	*function declaration*
	| $\mathtt{e}_1$ $\mathtt{e}_2$	*function application*
	| `Some e`	*optional expression*
	| $\mathtt{e}_1$ `op` $\mathtt{e}_2$	*binary operation*
	| `!` $\mathtt{e}_1$	*boolean negation*
	| $(\mathtt{e}_1, \ldots, \mathtt{e}_2)$	*tuple construction*
	| $\{\ell_1 : \mathtt{e}_1; \ldots; \ell_n : \mathtt{e}_n\}$	*record construction*
	| $\mathtt{e}.\ell$	*record projection*
	| `if` $\mathtt{e}_1$ `then` $\mathtt{e}_2$ `else` $\mathtt{e}_3$	*if-then-else expression*
	| `(match` $\mathtt{e}_0$ `with` | $\mathtt{p}_1 \rightarrow \mathtt{e}_1 \ldots$ | $\mathtt{p}_n \rightarrow \mathtt{e}_n$`)`	*case destruction*
op	::= $+_i$	*addition of i-bit integers*
	| $\&_i$	*bitwise and of i-bit integers*
	| $<_i$	*less-than relation for i-bit integers*
	| $<=_i$	*less-than-equal relation for i-bit integers*
	| $=$	*equality relation*
	| &&	*boolean and*
	| ||	*boolean or*
ty	::= `int`$\langle\mathbb{N}^+\rangle$	*type of fixed-width integers*
	| `bool`	
	| `node`	
	| `edge`	
	| `option[ty]`	
	| α	*type quantifier*
	| $\mathtt{ty}_1 \rightarrow \mathtt{ty}_2$	*function type*
	| $(\mathtt{ty}_1, \ldots, \mathtt{ty}_2)$	
	| $\{\ell_1 : \mathtt{ty}_1, \ldots, \ell_n : \mathtt{ty}_n\}$	*record type*
	| `dict`$[\mathtt{ty}_1, \mathtt{ty}_2]$	*map type*
d	::= `symbolic x : ty`	*symbolic value binding*
	| `require e`	*requires clause*
	| `assert e`	*assertion statement*
	| `let x : ty = e`	*top-level name binding*
	| `type t = ty`	*user-defined type*
	| `let nodes` $= \mathbb{N}$	*topology nodes*
	| `let edges` $= \{\mathbb{N} - \mathbb{N}; \ldots\}$	*topology edges*
	| `let x = solution {init =` $\mathtt{e}_1$`; transfer =` $\mathtt{e}_2$`; merge =` $\mathtt{e}_3\}$	*solutions of given SPP*

Figure 3.3: Core NV syntax

paths problem and combines them with the topology declarations to compute and return its solutions as a map from nodes to routes. An NV program may contain multiple `solution` declarations. In fact, the solutions of one stable paths problem may depend on the solutions of a previously defined stable paths problem, *e.g.* if they are used by the `transfer` function of the latter. One limitation is that currently all networks defined in an NV program must share the same topology.

NV also includes declarations to support verification. Primarily, `assert` declarations which are boolean expressions that can be used to express specifications of the converged state of the networks described by an NV program. Users may also declare variables that are bound to a *symbolic value* [59]. A symbolic value represents a class of values as opposed to a single *concrete* value. In NV, their exact interpretation depends on the analysis. SMT-based analyses treat symbolic values as representing any concrete value. Analyses based on normalization (*e.g.* a simulator) require concrete, non-symbolic values. In this case, symbolics are treated as *inputs* to the program; prior to execution, symbolic values are fixed to concrete ones, provided by the programmer or by random generation.

A user may also constrain the class of concrete values that a symbolic value represents by providing a boolean *requires clause*. Any assignment of a concrete value to a symbolic one must ensure that the requires expression evaluates to true.

3.2.3 Semantics

We give a formal semantics for the NV language in two-parts, firstly we give a semantics to expressions, and building on that we give a semantics to NV declarations. For the semantics of expressions we assume a function $\mathcal{E} : \Sigma \times \mathtt{e} \rightharpoonup \mathtt{v}$ that takes a store (a partial function from variables to values, $\Sigma = \text{Var} \rightharpoonup \mathtt{v}$) and an NV expression and returns an NV value. The expression language of NV is completely standard, and one can find an implementation of such a function in the literature [44].

To define the semantics of declarations we assume (i) a semantics of expressions as above, (ii) a function symb : $\text{Var} \times \mathtt{ty} \rightarrow \mathtt{v}$ that provides values to symbolic variables, (iii) a topology $G = V \times E$, and (iv) a function spp that given an instance of the stable paths problem computes its solutions as per fig. 2.2. In fig. 3.4, we define a semantics for NV declarations, as a function over two booleans and a store ($\mathcal{D} : \mathbb{B} \times \mathbb{B} \times \Sigma \rightharpoonup \mathbb{B} \times \mathbb{B} \times \Sigma$); the first boolean denotes the result of requires expressions and

$$
\begin{aligned}
&\mathcal{D}_{(\text{true},a,\sigma)}(\texttt{symbolic x : ty}) && = (r, a, \sigma[\texttt{x} \mapsto \text{symb}(\texttt{x}, \texttt{ty})]) \\
&\mathcal{D}_{(\text{true},a,\sigma)}(\texttt{let nodes} = \texttt{n}) && = (r, a, \sigma) \ \textit{if}\ \texttt{n} = G.V \\
&\mathcal{D}_{(\text{true},a,\sigma)}(\texttt{let edges} = \texttt{es}) && = (r, a, \sigma) \ \textit{if}\ \texttt{es} = G.E \\
&\mathcal{D}_{(\text{true},a,\sigma)}(\texttt{let x} = \texttt{solution\{init; merge; trans\}}) && = (r, a, \sigma[\texttt{x} \mapsto \text{sol}]) \\
&\mathcal{D}_{(\text{true},a,\sigma)}(\texttt{require e}) && = (r \wedge \mathcal{E}_\sigma(e), a, \sigma) \\
&\mathcal{D}_{(\text{true},a,\sigma)}(\texttt{assert e}) && = (r, a \wedge \mathcal{E}_\sigma(e), \sigma) \\
&\mathcal{D}_{(\text{false},a,\sigma)}(\texttt{d}) && = (\text{false}, \text{true}, \sigma')
\end{aligned}
$$

where $\text{sol} = \text{spp}(\text{G.V}, \text{G.E}, \text{init}, \text{merge}, \text{trans})$

Figure 3.4: Semantics of declarations in NV.

the second boolean the result of assertions. According to these semantics, symbolic declarations relate the given state to a new state where the store σ has been updated to include a new value for the given variable. This semantics provide flexibility for the analyses implementing them. A symbolic analysis, such as an SMT verifier, models the declarations *for all* instances of the symb function. On the other hand, an analysis such as a simulator which cannot handle symbolic values, interprets the declarations for a single instance of the function symb.

Node and edges declarations merely check that they match the pre-defined graph, and `solution` declarations use the spp primitive to compute the stable solutions for the given topology and network functions. Finally, note that in all but the last rule, the requires component of the state is true. Should some require clause fail, then how the declarations are interpreted does not matter; the requires component of the state will be set to false forever, indicating that the result of the analysis does not matter.

3.2.4 Syntax and Semantics of Maps

One of the more interesting aspects of NV is its treatment of maps. Giving semantics to maps is straightforward (*e.g.* as a function), but efficiently implementing them in an interpreter or as constraints is not as easy. We found that *total maps* admit efficient implementations both when interpreted and when encoded as constraints.

`create v`	$: \beta \to \text{dict}[\alpha, \beta]$	*creates a total map with default value* `v`
`get m k`	$: \text{dict}[\alpha, \beta] \to \alpha \to \beta$	*map get operation*
`set m k v`	$: \text{dict}[\alpha, \beta] \to \alpha \to \beta \to \text{dict}[\alpha, \beta]$	*map set operation*
`map f m`	$: (\beta \to \gamma) \to \text{dict}[\alpha, \beta] \to \text{dict}[\alpha, \gamma]$	*maps f over m*
`mapIte p f1 f2 m`	$: (\alpha \to \texttt{bool}) \to (\beta \to \gamma) \to (\beta \to \gamma) \to \text{dict}[\alpha, \beta] \to \text{dict}[\alpha, \gamma]$	*maps f1 over entries whose key satisfy p, f2 otherwise*
`combine f m1 m2`	$: (\beta \to \beta \to \gamma) \to \text{dict}[\alpha, \beta] \to \text{dict}[\alpha, \beta] \to \text{dict}[\alpha, \gamma]$	*combine two maps pointwise by applying f over their values*
`forall p f m`	$: (\alpha \to \texttt{bool}) \to (\beta \to \texttt{bool}) \to \text{dict}[\alpha, \beta]$	*checks if f is true of every entry for which p is*

Figure 3.5: NV supported map operations.

In fact, to scale our analyses to large networks we need to not only implement maps efficiently, but leverage them as a mean to accelerate verification.

The key principles driving our design of maps in NV are derived from their use in the context of routing:

1. Maps are typically indexed using statically-known values. For instance, routers route or block particular, known, subnets. They attach a particular tag to a BGP message. These values appear as constants in the configuration and need not be computed dynamically.

2. Map operations are performed over each map entry independently. For instance, the final solutions of routing algorithms are maps from IP subnets to routes. Computing those solutions does not require aggregation operations[1], such as folds, across those maps. On the other hand, routing algorithms do need to compute routes for *all* subnets, so we do need operations that apply functions across all elements of the map.

3. There is a lot of symmetry in networks. For instance, in routing many different IP subnets (keys) may be associated with the same route (value). Leveraging

[1]This is true for many practical purposes but certain network operations may invalidate this assumption. See section 3.2.5 for a more extended discussion.

symmetries has been crucial to scaling analyses of networks [9, 25, 10, 45]. The map data-structure should be able to leverage such symmetries by construction.

Points 1 and 2 are critical to use an optimized, tuple-based, encoding of maps in SMT (section 5.2.2). We *enforce* point 1 by requiring that the keys used in map get/set operations are constants, rather than arbitrary expressions. Point 3 motivated the use of *Multi-Terminal Binary Decision Diagrams* (MTBDDs) to implement maps (section 5.1.2.1) in our simulator.

Figure 3.5 shows the map operations in NV. The operation `create` takes as an argument a value `v` and returns a *total* map in which every key is associated with `v`. Map get (also written as `m[k]`) and set (written `m[k := v]`) return and update the value associated with key `k` of the map, respectively. The `combine` function merges the values of two maps key-wise.

Our maps support operations such as `map`, which maps a function over the values of a map, and `mapIte`, which maps one of two different functions over the values, based on a boolean function over their corresponding key. Intuitively, `mapIte` is useful to implement operations that would otherwise require the `map` operation to access both keys and values (*e.g.*, filtering routes based on their prefix). While more general, this operation —commonly known as `mapi`— cannot be efficiently implemented using our MTBDD encoding of maps (section 5.1.2.1).

Finally, the `forall` operation takes a predicate on the key of the map, and a predicate on the value of the map, and returns true if for every entry where the predicate on the key is true, the predicate on the value is also true. While in theory, such an operation goes against the principles we listed above, in practice, `forall` is not an operation that is useful when modeling routing protocols or other network functions. It is useful to express assertions over the large maps that the simulator models; the SMT solver would not be a suitable tool to verify these models. For this reason, we currently have not implemented this operation in our SMT verifier,

$$
\begin{aligned}
&\mathcal{E}_\sigma(\texttt{create e}) && = \lambda\, k.\ \mathcal{E}_\sigma(\texttt{e}) \\
&\mathcal{E}_\sigma(\texttt{get e k}) && = \mathcal{E}_\sigma(\texttt{e})\ \mathcal{E}_\sigma(\texttt{k}) \\
&\mathcal{E}_\sigma(\texttt{set e}_1\ \texttt{k}\ \texttt{e}_2) && = \lambda\, \texttt{k}'.\ \begin{cases} \mathcal{E}_\sigma(\texttt{e}_2) & ,\ \mathcal{E}_\sigma(\texttt{k}') = \mathcal{E}_\sigma(\texttt{k}) \\ \mathcal{E}_\sigma(\texttt{e}_2)\ \mathcal{E}_\sigma(\texttt{k}') & ,\ \textit{otherwise} \end{cases} \\
&\mathcal{E}_\sigma(\texttt{map f e}) && = \lambda\, k.\ \mathcal{E}_\sigma(\texttt{f})\ (\mathcal{E}_\sigma(\texttt{e})\ \mathcal{E}_\sigma(\texttt{k})) \\
&\mathcal{E}_\sigma(\texttt{mapIf p f e}) && = \lambda\, k.\ \begin{cases} \mathcal{E}_\sigma(\texttt{f})\ (\mathcal{E}_\sigma(\texttt{e})\ \mathcal{E}_\sigma(\texttt{k})) & ,\ \mathcal{E}_\sigma(\texttt{p})\ \mathcal{E}_\sigma(\texttt{k}) = \mathcal{E}_\sigma(\texttt{true}) \\ \mathcal{E}_\sigma(\texttt{e})\ \mathcal{E}_\sigma(\texttt{k}) & ,\ \textit{otherwise} \end{cases} \\
&\mathcal{E}_\sigma(\texttt{mapIte p f}_1\ \texttt{f}_2\ \texttt{e}) && = \lambda\, k.\ \begin{cases} \mathcal{E}_\sigma(\texttt{f}_1)\ (\mathcal{E}_\sigma(\texttt{e})\ \mathcal{E}_\sigma(\texttt{k})) & ,\ \mathcal{E}_\sigma(\texttt{p})\ \mathcal{E}_\sigma(\texttt{k}) = \mathcal{E}_\sigma(\texttt{true}) \\ \mathcal{E}_\sigma(\texttt{f}_2)\ (\mathcal{E}_\sigma(\texttt{e})\ \mathcal{E}_\sigma(\texttt{k})) & ,\ \textit{otherwise} \end{cases} \\
&\mathcal{E}_\sigma(\texttt{combine f e}_1\ \texttt{e}_2) && = \lambda\, k.\ \mathcal{E}_\sigma(\texttt{f})\ (\mathcal{E}_\sigma(\texttt{e}_1)\ \mathcal{E}_\sigma(\texttt{k}))\ (\mathcal{E}_\sigma(\texttt{e}_2)\ \mathcal{E}_\sigma(\texttt{k})) \\
&\mathcal{E}_\sigma(\texttt{forall p f e}) && = \forall\, k.\ \mathcal{E}_\sigma(\texttt{p})\ \mathcal{E}_\sigma(k) \Rightarrow \mathcal{E}_\sigma(\texttt{f})\ (\mathcal{E}_\sigma(e)\ \mathcal{E}_\sigma(k))
\end{aligned}
$$

Figure 3.6: Semantics of map operations in NV.

however, should one insist to use it, it can be implemented using more expensive SMT encodings (*e.g.*, with the array theory and a universal quantifier).

Giving a semantics for NV's total maps and their operations is straightforward; total maps are just functions from the key space to the value space of the map. Assuming a function $\mathcal{E}$ that gives an interpretation to the core NV expressions, we can extend it to include map operations[2] as shown in fig. 3.6.

3.2.5 Trade-offs and Limitations

When designing an intermediate verification language there is a tension between the expressivity needed to model the source language(s) and the tractability of analysis. In designing NV, we limited ourselves to constructs that admit efficient and complete verification procedures (chapter 5). In addition to our restrictions on map operations, we have omitted features such as recursive functions and recursive datatypes. These conditions, while overly restrictive for a general-purpose language, can (with few exceptions) be easily accommodated in our context.

One such exception is BGP's record of a route's path as a string. At each hop, BGP will push the name of the router onto the front of the string. A router may use

[2]For ease of presentation, this definition assumes a type of "overloading" for the function $\mathcal{E}$.

this to implement loop prevention by filtering routes that contain itself. Additionally, a user may construct route filters using regular expressions over the string. Such operations currently cannot be directly modeled in NV, though that may change in the future. Fortunately, however, *verification* (as opposed to implementation) does not usually require a precise model of such features—approximations suffice. One recent analysis [10] demonstrated such features can be overapproximated without loss of precision in real networks. For instance, for loop prevention it suffices to approximate the string with a set of nodes traversed.

Another limitation is the lack of general "aggregation" operations (*e.g.*, a fold) over maps. This stems from our representation of maps, both in the simulator, but even more so in the SMT-based backend where not all map entries are modeled. Although we have not found very compelling use cases yet, under certain restrictions, we can implement such operations even if they require a more computationally expensive SMT model. The only exception we have made, is the forall operation, which as previously explained, is not useful for modeling routing protocols, rather, it is used to facilitate writing assertions over total maps in the context of simulation.

Chapter 4

Modeling Protocols and Configurations

This chapter uses the NV language introduced in chapter 3 to build the models of the networks control and data planes that were described in chapter 2. In section 4.1 we show how to model the control plane: this includes representing the Routing Information Base (RIB), common routing protocols such as BGP, OSPF, static routes and redistribution between them. Section 4.1.3 explains the process of translating route-maps to NV expressions and the challenges we faced. Finally, section 4.2 explains how our model of the networks data plane works, *i.e.* how to encode the longest prefix match semantics for packets and access-control lists, and shows how to compose the control plane and data plane models to get an end-to-end model of a network's forwarding behavior.

4.1 Control Plane Models

A routing protocol's behavior can be broken down in two parts: 1. an implicit part that covers some *fixed* behaviors of the protocol (*e.g.* how routes are ranked) and is described by the protocol's specification and (vendor-dependent) implementation, 2.

a part that is explicit in the routers' configurations that controls aspects relating to routing policy. An NV program represents both aspects explicitly. To build a model of the control plane we show how to map elements from the router configurations and the protocols RFC to NV programs. Furthermore, we have implemented a translation from actual router configurations to NV programs, as a new backend to Batfish. We first infer the basic structure of the network (the topology and which protocols are in use), then translate the configuration *route-maps* that implement network-specific policy.

4.1.1 Modeling the Topology and Active Protocols

In the first stage, we infer the physical connectivity between routers, the different protocols that each router runs, and the prefixes (subnets) they announce.

For the topology, the translation is simple. We create one node in the graph for each router, and add edges between each pair of nodes for which Batfish has inferred physical connectivity. To represent the active protocols we adjust the type of messages exchanged by the routers. Each router in a network may run routing protocols such as BGP and OSPF, and may also contain hardcoded static or connected routes. Figure 4.1 defines the NV type of the messages that our model of the control plane uses.

Comparing Routes From Different Protocols. Each protocol stores an *administrative distance* in their routes (ospfAd, bgpAd, connectedAd, staticAd), an 8-bit integer used to compare routes for the same destination across different protocols; if a router has paths towards a destination via multiple protocols it will choose the one lowest administrative distance. In general the default preference is to use a directly connected interface (in Cisco routers connectedAd is 0), then a static route (staticAd is 1), then an eBGP route (bgpAd is 20) and finally an OSPF route (ospfAd is

```
type ospfType = {ospfAd: int8;
                 weight: int16;
                 areaType:int2;
                 areaId: int;}
type bgpType = {bgpAd: int8;
                lp: int;
                aslen: int;
                med:int;
                comms:set[int];}
type connectedType = {connectedAd: int8;}
type staticType = {staticAd: int8;}

type ribEntry = {
  ospf      : option[ospfRoute];
  bgp       : option[bgpRoute];
  static    : option[staticRoute];
  connected : option[connectedRoute];
  selected  : option[int2] }
```

Figure 4.1: Type of routes for a simple control plane model running multiple protocols.

110). Operators, however, can reconfigure these values according to their needs. For instance, a common reason to modify the administrative distance is to install a static route with increased AD to act as a backup to a dynamically computed route. Given the best route for each active protocol we select the best overall route based on the administrative distance; fig. 4.2 shows how to encode this process in NV.

OSPF Routes. For OSPF we model the path weight (a 16-bit integer), the type of the OSPF area (inter-area, intra-area and type 1/type 2 external routes), and the areaId of the route. According to OSPF semantics [42], when comparing OSPF routes, intra-area routes are preferred over inter-area routes, which are preferred over type 1 external routes. Moreover, type 1 external routes are preferred over type 2 external routes. If the area is the same, then the path weight is used to determine the best route (see fig. 4.3).

The transfer function adjusts the path weight according to the link weights set by the configuration, and uses the area id to determine whether the area type is IntraArea

```
let betterEqOption oa ob =
  match (oa,ob) with
  | (_, None) -> true
  | (None, _) -> false
  | (Some a, Some b) -> a <=u8 b

let best c s o b =
  match (c,s,o,b) with
  | (None,None,None,None) -> None
  | _ ->
      let c = match c with | None -> None
                           | Some c -> Some c.connectedAd in
      let s = match s with | None -> None
                           | Some s -> Some s.staticAd in
      let o = match o with | None -> None
                           | Some o -> Some o.ospfAd in
      let b = match b with | None -> None
                           | Some b -> Some b.bgpAd in
      (*compare static and connected *)
      let (x,px) = if betterEqOption c s then (c,0u2)
                   else (s,1u2) in
      (*compare bgp and ospf*)
      let (y,py) = if betterEqOption o b then (o,2u2)
                   else (b,3u2) in
      (*compare winners of the two groups *)
      Some (if betterEqOption x y then px else py)
```

Figure 4.2: Route comparison among different protocols.

or InterArea (external routes are determined by other factors, such as whether the route is redistributed by another protocol).

BGP Routes. For BGP, in the absence of strings/lists in NV, we represent the path by its length. This abstraction is sufficient in many cases[1] [10, 8], as it can determine the best route, and under certain conditions prevent routing loops. The abstraction can be unsound when using a non-default local-preference value as that can lead to routing loops. To restore soundness in such networks, we can also model the set of nodes traversed, and use it implement explicit loop prevention. Finally

[1]Formally, this abstraction is sound when the routing policy is monotonic[51, 16].

```
let ospfIntraArea = 0u2
let ospfInterArea = 1u2
let ospfE1 = 2u2
let ospfE2 = 3u2

let betterOspf o1 o2 =
  if o1.areaType <u2 o2.areaType then true
  else if o1.areaType >u2 o2.areaType then false
  else if o1.weight <=u16 o2.weight then true else false
```

Figure 4.3: Route comparison for OSPF routes.

```
let betterBgp b1 b2 =
  if b1.lp > b2.lp then true
  else if b2.lp > b1.lp then false
  else if b1.aslen < b2.aslen then true
  else if b2.aslen < b1.aslen then false
  else if b1.med <= b2.med then true else false
```

Figure 4.4: Route comparison for BGP routes.

we also model the local-preference and multi-exit discriminator values, and the set of community values. The merge function for BGP implements the BGP route ranking algorithm (fig. 4.4), first comparing the local-preference value, then the path length, and finally the multi-exit discriminator.

The transfer function, handles tasks such as increasing the path length at each hop, redistributing routes into BGP, and adjusting the other attributes according to the network's configuration.

Route Redistribution. We model route redistribution in the transfer function. When a router exports protocol's X route to a neighbor, it checks whether 1. redistribution from another protocol Y is configured, and 2. if Y is the "best" protocol (*i.e.*, the one with lowest administrative distance) for this route. If both checks succeed, the transfer function transforms the route of protocol Y to a route of protocol X and exports it to the neighbor.

Modeling the RIB. The type `ribEntry` represents a single entry in a router's RIB. It contains routing information learned from each protocol; the absence of a value denotes that there is no route through this protocol. Furthermore, an optional 2-bit integer is used as a "pointer" to the preferred protocol in this rib entry. A value of type `ribEntry` represents the route for a single destination. Depending on the type of analysis it may be preferable to reason about each destination separately; SMT verification or analysis that rely on symmetries such as network compression (see chapter 6) favor reasoning about each destination separately. In other cases, such as when computing routes to all destinations, one can leverage the MTBDD-based maps in NV to achieve better performance thanks to the fact that routes to different destinations are often similar. Our tool implements both translations. In the single destination case, a variable assigned a symbolic value is used to denote the destination prefix and the messages exchanged are of type `ribEntry`. Using a symbolic value allows us to model *any* single destination in the network. In the all-destinations case, we model the RIB as a map from destination prefixes to routes (as represented by `ribEntry`):

```
type ipv4Prefix = (int, int6)
type attribute = dict[ipv4Prefix, ribEntry]
```

4.1.2 Model Extensions

Before the research in this thesis, protocol models were only implicitly described: the models were baked-in the tools that analyzed them. Modifying or extending them required extensive effort, as the models were obfuscated by low-level details and optimizations relating to the analysis. Targeting a high-level language facilitates extensions to models, in a relatively compositional way.

Route Origin and Next-Hops. One such extension we have implemented in our translation from Batfish, is to optionally keep track of the node which originated the route and if this route was learned from a neighbor, the edge over which the route was transmitted, *i.e.*, an abstraction of the next-hop, as shown in fig. 4.5.

```
type ospfType = {ospfAd: int8;
                 weight: int16;
                 areaType:int2;
                 areaId: int;
                 ospfNextHop: option[tedge];
                 ospfOrigin: tnode;}
type bgpType = {bgpAd: int8;
                lp: int;
                aslen: int;
                med:int;
                comms:set[int];
                bgpNextHop: option[tedge];
                bgpOrigin: tnode;}
```

Figure 4.5: Extensions to the standard control plane model.

Abstractions. Similarly, one could opt for a less detailed model if it suffices to verify the desired properties. Beckett *et al.* [10] showed that for basic reachability properties, the BGP route decision variables such as the path length, the local-preference and the multi-exit discriminator can often be abstracted away without loss of precision. More abstract representations have significant performance advantages (see also, section 5.1.5): 1. the state space is smaller and hence network simulators converge faster, 2. more uniform attributes favor sharing in MTBDDs which we exploit in our simulator, and 3. performing route updates is faster due to their simpler structure.

4.1.3 Modeling the Policy

Network operators implement policy over a network using a mechanism called *route-maps*. A route-map is a list of statements which test and modify certain characteristics of a route. For instance, a route-map applied on a BGP connection may check if a certain tag is present and increase or decrease the local preference value accordingly. Abstractly, route-maps contain two types of statements: *conditional* statements which test properties of the route, *mutation* statements which modify attributes of the route. There are also goto statements that jump within the route-map; however, as these are always forward jumps and cannot be used to implement recursion, they may be eliminated through inlining. Figure 4.6a shows a single route-map with two clauses identified by the numbers 10 and 20 respectively. The clauses are evaluated in order until one of them succeeds; if all of them fail, then the route is dropped. For a clause to succeed, all of its match statements —which test an attribute of the route— must succeed. For instance, the first clause of `RM`, will succeed if the route has the community tag associated with `comm1` and the route's destination prefix matches the prefixes in the list `pfx`. When a clause successfully matches, any set statements are executed to modify the route and the action specified by the clause (`permit/deny`) is executed to permit or drop the route.

The route-map of fig. 4.6b includes an extra *continue* statement in the first clause. When the first clause matches, the continue statement will force evaluation to continue by jumping to the clause with the sequence number 20. Now this clause must also match; if that fails then the match of clause 10 is considered to have failed as well. Continue statements further expose the lack of flexibility of router configuration languages. One would imagine that continue statements can be eliminated by inlining the respective route-map clauses as done below:

(a)

```
route-map RM permit 10
  match community comm1
  match ip address prefix-list pfx
  set local-preference 200
route-map RM permit 20
  match community comm2
  set local-preference 100
route-map RM deny 30
```

(b)

```
route-map RM permit 10
  match community comm1
  match ip address prefix-list pfx
  continue 20
  set local-preference 200
route-map RM permit 20
  match community comm2
  set local-preference 100
route-map RM deny 30
```

Figure 4.6: Route-maps in Cisco IOS syntax.

```
route-map RM permit 10
  match community comm1
  match ip address prefix-list pfx
  match community comm2
  set local-preference 200
  set local-preference 100
route-map RM permit 20
  match community comm2
  set local-preference 100
route-map RM deny 30
```

However, according to the semantics of match statements, matches over the same attributes (*e.g.* over communities) in a clause are considered to be a logical OR, *i.e.* clause 10 now matches if `comm1` or `comm2` is attached on the route. Hence, such a transformation cannot be applied at the level of configurations because their syntax cannot support it. We deal with such commands by performing inlining at the level of the IR described next.

Intermediate Policy Representation.

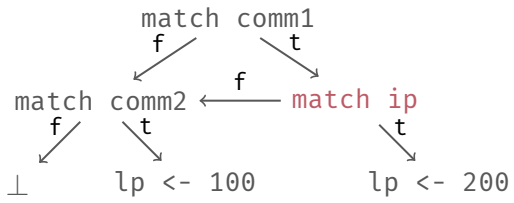
Converting route-maps to NV is complicated somewhat by the different abstraction levels – route-maps operate over a single route. For our NV encoding that uses the `dict` data type to represent all routes at the same time, we must lift route-maps to operate over such a data type. This is challenging, because our map operations separate the key and the value space, *i.e.* the `mapIf`/`mapIte` operations apply a predicate over the keys and a separate function over the values.

To convert route-maps to NV expressions, we go through a directed acyclic graph (DAG) based intermediate representation. We represent each route-map as a DAG in which non-leaf nodes correspond to conditional statements, and leaf nodes correspond to a list of mutation statements. This representation allows us to separate prefix processing (map keys) from route processing (map values) by swapping DAG nodes to reorder conditional statements.

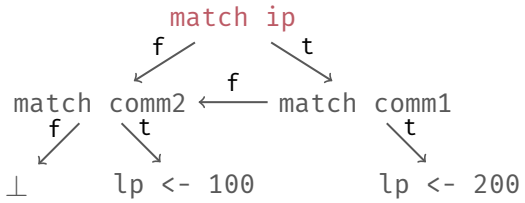
The DAG of fig. 4.7a captures the semantics of this route-map. It sets the local-preference value according to the route-map and also bumps the path length (which is implicitly done by BGP semantics). Note that if no route-map matches, the route is implicitly dropped (denoted by $\perp$).

A natural translation from this DAG representation to NV is a chain of if-then-else expressions. Indeed, this is the translation we use when modeling a single route. However, when modeling routes for all prefixes using a map, this expression needs to be mapped over routes in the RIB. Since route-maps test not only the route but also the prefix (*i.e.* the keys in the RIB), we must use the `mapIte` operation, which can perform tests over the map keys. Doing so, however, requires some extra effort: conditional statements on the prefix must be executed first, so they may appear as predicates for `mapIte`.

Fortunately, our DAG-based IR makes this easy; we need only swap nodes in the DAG until all nodes which condition on the prefix are at the top (*i.e.* any parents also



(a)



(b)

```
let transRouteMap (x : dict[ipv4Prefix,option[bgp]]) =
  mapIte (fun pre -> matchPrefix pre pfx)
    (fun ov ->
      match ov with
      | None -> None
      | Some v ->
        if matchComm comm1 v then
          Some {v with lp = 200; aslen=v.aslen+1} else
        if matchComm comm2 v then
          Some {v with lp = 100; aslen=v.aslen+1} else None)
    (fun ov ->
      match ov with
      | None -> None
      | Some v ->
        if matchComm comm2 v then
          Some {v with lp = 100; aslen=v.aslen+1} else None) x
```

(c)

Figure 4.7: (a) shows the DAG representation of fig. 4.6a. (b) shows the transformed DAG, where all conditional statements on prefixes are on the top of the DAG, and finally, (c) is the resulting NV function.

condition on the prefix). Figure 4.7b shows the result of applying this transformation to the DAG in fig. 4.7a. Translating to an NV expression using `mapIte` is easy now; the red nodes are used as predicates to `mapIte`, while the rest are translated as if-then-else chains over the map values (fig. 4.7c).

4.2 Data Plane Model

As described in earlier sections, the role of the data plane is to move data packets from one node to another. One might wonder, how does such a system fit into the

```
type packetV4 = {srcIp: int;
                 dstIp: int;
                 srcPort: int16;
                 dstPort: int16;
                 protocol: int8;
                }

type packets = set[packet]
```

Figure 4.8: A model of IPv4 packets.

stable paths model (section 2.3), *i.e.*, what kind of information is exchanged between nodes and what is the stable state of the system. For the former, analogously to our control plane model where we model either a single destination or all destinations simultaneously, we can model a single data packet or the set of all data packets. In any case, fig. 4.8 defines our model of an IPv4 packet [1]. This model includes the fields of a packet that are relevant to the node's forwarding behavior, including computing the longest prefix match and filtering using ACLs. Other fields, such as the checksum, are irrelevant to the kind of analysis we are interested in and can be ignored without loss of precision.

Our notion of stable state for the data plane is somewhat odd; one might have expected that packets would "move" from node to node, starting from a source until they reach their destination. Such a model is illustrated in fig. 4.9, where the stable state of the network is captured by the final figure in which the packet has reached the destination. While this model closely mimics the way the data plane actually works, it is not expressible in our notion of stable paths; the idea of changing a node's label when information are transmitted is not compatible with our semantics, *i.e.* it is not possible to for a packet to removed from a node's solution when this node propagates it to its neighbors. Moreover, such a model still would not be very useful for verification, as it cannot answer questions beyond packet reachability; one would

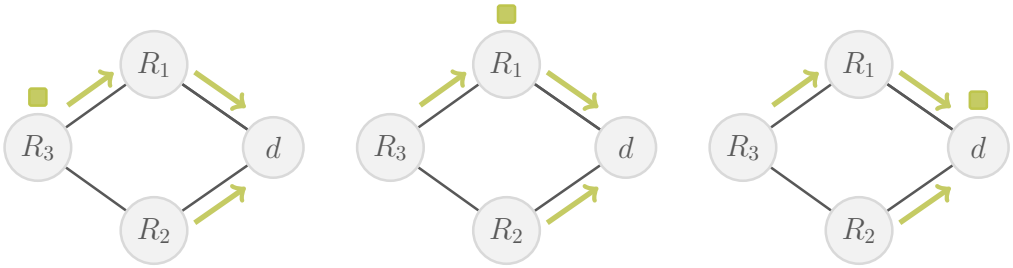


Figure 4.9: A single packet (denoted by the green box) flowing through the network, starting from node R_3 and destined for node d. The green arrows denote the edges over which each node forwards packets destined for d.

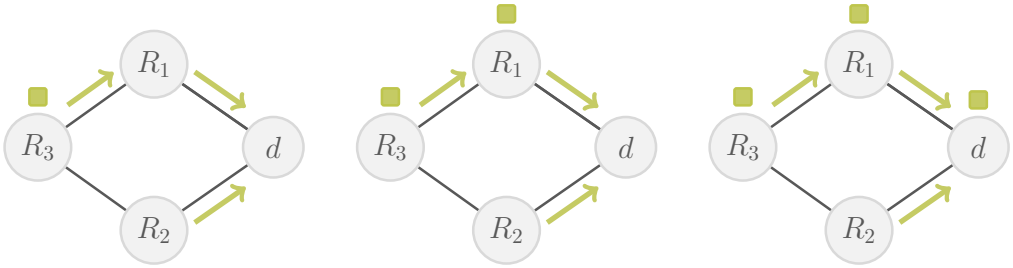


Figure 4.10: Packet flow according to our stable paths model of the data plane.

need to keep track of packet histories [5] to reason about more involved properties such as waypointing.

Instead, our notion of a stable state for the data plane can be described as "every node has seen and propagated all packets it should and no other packet can reach it", illustrated in fig. 4.10. This model can be easily expressed in NV, the merge function is simply the union of incoming packets. The transfer function propagates packets according to longest prefix matching semantics while respecting any ACLs installed on the link. In particular, it *filters* the packets that should not be forwarded, either because there is no route over this edge or because there is an ACL that blocks them:

```
let mergeData u x y = x union y

let transData fib edge ps =
  let (u,v) = edge in
    mapIf (fun p -> (!(longestPrefixMatch fib[u] edge p)
                     || (acl edge p))
          (fun v -> false) ps
```

Encoding the ACLs is straightforward; they are just boolean tests on one or more of the packet fields. For instance, the ACL

```
access-list 102 deny tcp any any eq 23
```

filtering all TCP traffic destined for port 23 simple translates to the NV expression (6 denotes TCP in IPv4 packets):

```
p.dstPort = 23 && p.protocol = 6
```

Encoding Longest Prefix Match. The function `longestPrefixMatch` (fig. 4.11) implements the main functionality of the data plane; it decides whether a packet should be forwarded or not. In practice, the router performs a lookup in its FIB to find which interface (link) it should be forwarded through. In our case, as the transfer function is applied over each link `(u,v)` of a node `u`, the lookup in the FIB to see whether this packet should be forwarded over `(u,v)` or not. The `longestPrefixMatch` function performs a linear lookup in `fib` to see if there is a route for the given packet `p` over the link `e`. It first checks if there is there is a route over this edge for the prefix matching all 32-bits of the packet's destination IP (line 20) by calling the function `getNextHop`. `getNextHop` returns `None` if the lookup was not successful (line 3) *i.e.* if there is no route for this particular prefix in the FIB, `Some None` (lines 4 and 5) if the packet is intended for this node (*i.e.* if the route is announced by this node), and `Some (Some nexthop)` where `nexthop` is the link over which traffic for this prefix should be forwarded (lines 10,13,16). Returning to `longestPrefixMatch`, if the packet was intended for this node then it should not be forwarded and the function returns false (line 21). Otherwise, if there is a next-hop over which this node should forward this packet, it checks if this next-hop matches the link over which we are forwarding traffic (line 22). Finally, if there was no route for this prefix in the FIB, it moves on to check less specific prefixes in order, starting from a prefix that

```
let getNextHop route =
  match route.selected with
  | None -> None
  | Some 0u2 -> Some None
  | Some 1u2 -> (match route.static with
                 | None -> None
                 | Some s ->
                   (match s.staticNextHop with
                    | None -> Some None
                    | e -> e))
  | Some 2u2 -> (match route.ospf with
                 | None -> None (*bogus case*)
                 | Some o -> Some o.ospfNextHop)
  | Some 3u2 -> (match route.bgp with
                 | None -> None (*bogus case*)
                 | Some b -> Some b.bgpNextHop)

let longestPrefixMatch fib e p =
  let dst = p.dstIp in
  (match getNextHop (fib[((dst & 4294967295),32u6)]) with
  | Some None -> false
  | Some (Some nexthop) -> e = nexthop
  | None ->
     (match getFwd (fib[((dst & 2147483647),31u6)]) with
      | Some None -> false
      | Some (Some nexthop) -> e = nexthop
      | None ->
         ...
         (match getFwd (fib[((dst & 0),0u6)]) with
         | Some None -> false
         | Some (Some nexthop) -> e = nexthop
         | None -> false)))
```

Figure 4.11: Forwarding Functions

matches 31-bits of the destination IP (line 24) until one that matches no bits of the destination (line 29).

Composing the Control and Data Planes. For the data plane to function we need to provide the FIB it consults to determine whether to forward a packet over a link or not. We do so by providing the solutions of our model of the control plane as described in section 4.1. For instance, assuming the control plane is described by the

```
let nodes = 4
let edges = {0=1; 0=2; 1=3; 2=3;} (*where node 0 = d *)
let fib = solution {init = initControl;
                    trans = transControl;
                    merge = mergeControl}

let p = {srcIp = ..; dstIp = ..; srcPort = ..;
         dstPort = ..;  protocol = ..;}

let initData u =
  if u = 3n then {p} else {}

let data = solution {init = initData;
                     trans = transData fib;
                     merge = mergeData}
```

Figure 4.12: Model of the network in fig. 4.10.

functions `initControl`/`transControl`/`mergeControl`, fig. 4.12 encodes in NV the behavior of the data packet moving through the network illustrated in fig. 4.10.

4.3 Supported Properties

After defining models of the control and data plane, it is interesting to look into the type of properties that can be checked using these models. Properties are specified in the NV language itself, which provides a flexible mechanism to define them. In this section, we describe some common network properties supported by our models (or slight extensions thereof). Models for checking fault-tolerance properties are studied in depth in chapter 6. We once again focus on models where all destinations are modeled but the respective properties are straightforward to express in the single destination model (unless the property in question requires reasoning about multiple destination prefixes).

Reachability. Figure 4.13 checks whether, in the computed control plane, a router `u` has a route to a given destination prefix `pre`, it suffices to check that there is *some* entry for this prefix in its RIB.

```
let rib : dict[tnode, dict[ipv4Prefix, ribEntry]] =
  solution {init = initControl;
            trans = transControl;
            merge = mergeControl}

assert(match rib[u][pre].selected with
       | None -> false
       | Some _ -> true)
```

Figure 4.13: Basic Control Plane Reachability.

One can also consider more specific questions, such as whether the destination prefix is reachable through a specific node `v`, in case it is announced by multiple nodes. This requires, modeling the origin of the route (figs. 4.5 and 4.14).

Finally, reachability can be also checked at the data plane level, taking into consideration any ACLs. Users can ask whether a packet or a set of packets `ps` can reach a specific destination node `v`, by checking that the intersection of the packets that reached node `v` and of the set of packets `ps` are all the packets in `ps`.

```
let data : dict[tnode, packets] =
  solution {init = initData;
            trans = transData;
            merge = mergeData}

let ps : packets = {..}

assert((combine (fun a b -> a && b) data[v] ps) = ps)
```

Path Length. Another common correctness property to check is that the path length does not exceed a certain bound. This can easily be done by keeping track of an additional hop count. This hop count should not influence routing/forwarding decisions and is merely propagated and increased at every hop. If done on the control

```
let rib : dict[tnode, dict[ipv4Prefix, ribEntry]] =
    solution {init = initControl;
              trans = transControl;
              merge = mergeControl}

  assert(
    let route = rib[u][p] in
    match route.selected with
    | None -> false
    | Some i ->
      (match i with
       | 0 -> u = v
       | 1 -> (match route.static with
               | None -> false
               | Some s -> s.staticOrigin = v)
       | 2 -> (match route.ospf with
               | None -> false
               | Some o -> o.ospfOrigin = v)
       | 3 -> (match route.bgp with
               | None -> false
               | Some b -> b.bgpOrigin = v)))
```

Figure 4.14: Reachability to a Specific Router.

plane level then it should a per-protocol hop count as different protocols may follow different paths. At the data plane level the model should be extended to exchange a structure richer than just set of packets. Figure 4.15 shows a model suitable to reason about whether any given packet `p` reaches a destination node `v` in less than `k` steps:

Forwarding Loops. Some routing protocols/features, such as static routes and route redistribution, might lead to forwarding loops. For each packet we can keep track of the forwarding path which is used to set a boolean that denotes there was a forwarding loop. In the end, all such booleans must be set to false as shown in fig. 4.16.

```
type packets = dict[packet, option[int]]

let mergeData u x y =
combine (fun vx vy ->
          match vx, vy with
          | _, None -> vx
          | None, _ -> vy
          | Some i, Some j -> Some (max i j)) x y

let transData fib edge ps =
  match edge with
  | (u,v) ->
    mapIte (fun p -> (!(longestPrefixMatch fib[u] edge p)
                        || (acl edge p))
          (fun v -> None)
          (fun v -> match v with
                    | None -> None
                    | Some i -> Some (i+1)) ps

let data : dict[tnode, packets] =
solution {init = initData; trans = transData; merge = mergeData}

assert(match data[v][p] with
        | None -> false
        | Some i -> i <= k)
```

Figure 4.15: Data Plane Hop Count.

4.4 The Curious Case of iBGP

The usual role of the iBGP routing protocol is to connect a network with the outside world. An organization typically uses an IGP protocol such as OSPF to route within their network, but as explained in section 2.2, such protocols are not suitable for peering with other networks because of scalability and security considerations. Instead, organizations rely on eBGP to connect with peer networks. iBGP is the protocol that glues together the two, in order to disseminate external routes within an organization's network.

We devise a small example to better illustrate illustrate the role and functionality of iBGP. In the network of fig. 4.17, routers $R_1 - R_4$ represent the network of a

```
type metadata = {path: set[tnode]; loop:bool;}
type packets = dict[packet, metadata]

let mergeData u x y =
combine (fun vx vy ->
            {path = vx.path union vy.path;
             loop = psx.loop || psy.loop}) x y

let transData fib edge ps =
  match edge with
  | (u,v) ->
    mapIte (fun p -> (!(longestPrefixMatch fib[u] edge p)
                        || (acl edge p))
        (fun p -> {path = {}; loop = false;})
        (fun p ->
            {path = p.path union {u};
             loop = p.path[u] || p.loop}) ps

let data : dict[tnode, packets] =
  solution {init = initData;
            trans = transData;
            merge = mergeData}

assert(forall (fun u -> u <=n nodes)
              (fun packets ->
                forall (fun packet -> true)
                       (fun meta -> meta.loop = false) packets)
              data)
```

Figure 4.16: Data Plane Path and Forwarding Loops Check.

single organization using OSPF to route packets internally. Additionally, router R_1 is connected with an external peer R_0 and the two rely on eBGP to exchange routes. Likewise, R_4 is connected with another external peer (R_5), and eBGP is used between them. Obviously, there is a "gap" in our network: when R_0 announces an eBGP route for 143.1.50.0/24 to R_1, this route cannot be further propagated if R_1 communicates with R_3 and R_2 only using OSPF. One could use route redistribution from BGP to OSPF, but, if R_0 and R_5 announce a high number of routes, OSPF will not be able to handle them and crash. It is this gap that iBGP fills in collaboration with the IGP (OSPF in this case).

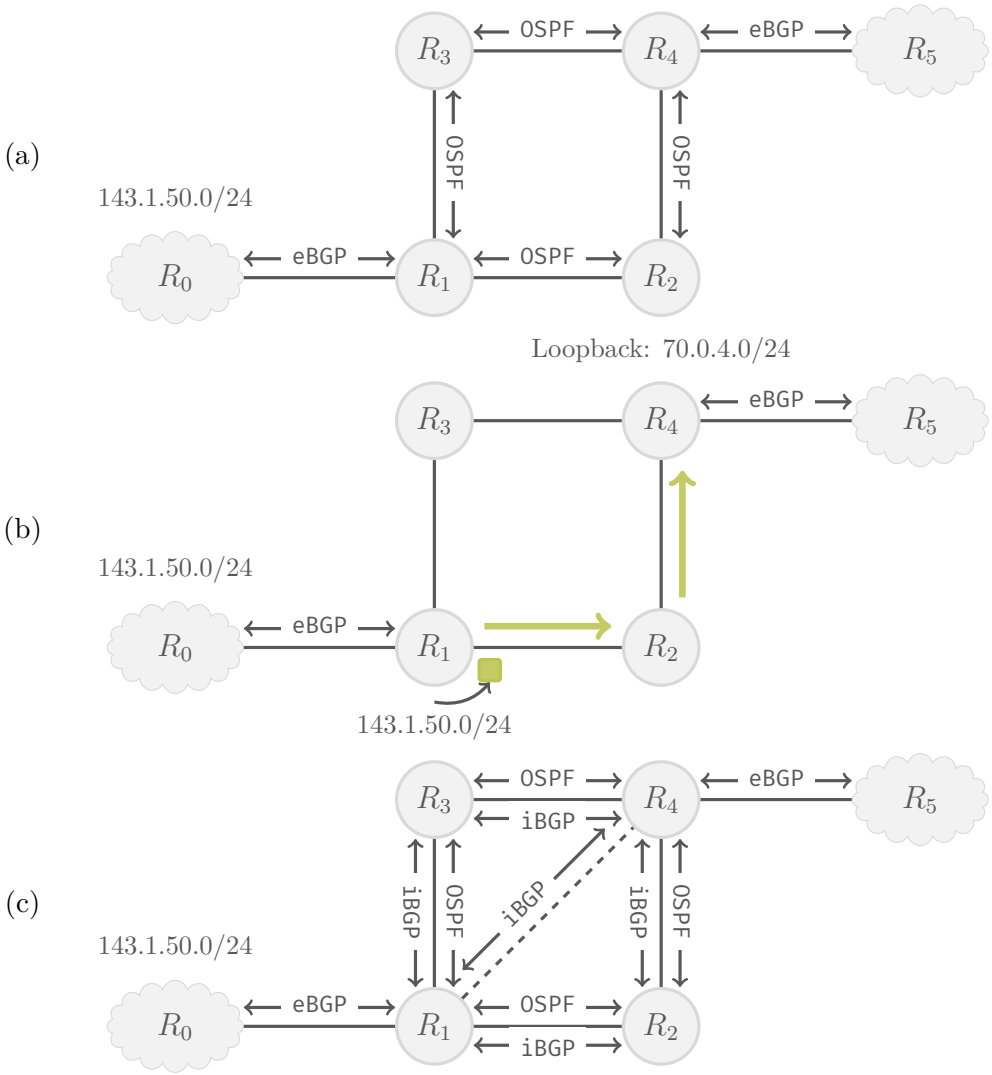


Figure 4.17: (a) shows a network running eBGP with its external peers, and OSPF internally. (b) shows iBGP's route encapsulation; the forwarding behavior of the packet carrying the route is determined by OSPF. (c) iBGP model in NV. Dashed lines represent virtual links, that do not correspond a physical connection.

iBGP Operational Model. iBGP works by encapsulating external BGP routes in data packets that are disseminated within the network according to the IGP. This way, external routes are propagated throughout the network indirectly, without over-populating the IGP's routing table. Returning to our example, if iBGP is configured in our network (R_1-R_4), R_1 will encapsulate the BGP route to 143.1.50.0/24 in data packets destined to each iBGP peer, *i.e.*, at a unique loopback address that each of them announces. How does R_1 forward the data packets to R_2-R_4? As usual, by

consulting its FIB; the IGP will have computed how to route packets to the loopback addresses used by iBGP. Upon, receiving a packet destined for its loopback, a router decapsulates the packet and installs the route in its RIB. It also further propagates the route to any external BGP peers. Note, it is important that routes learned from iBGP are not further propagated using iBGP! Since the route is encapsulated in a data packet by iBGP, the path length or the AS path of the BGP route are not altered when it is transmitted. As such, there is no loop prevention mechanism, and recirculating a route learned from iBGP would lead to routing loops. To avoid such situations, BGP routes include an extra field to distinguish whether the route was learned from an internal peer, and hence should not be further propagated internally:

```
type bgpType =
  { bgpAd: int8;
    lp: int;
    aslen: int;
    med:int;
    comms:set[int];
    bgpNextHop: option[tedge];
    bgpOrigin: tnode;
    external: bool; (*true if it's a route learned from eBGP*)
  }
```

Verification Model. Unfortunately, modeling operations such as encapsulation in NV is not quite straightforward, if at all possible, due to the restricted map operations. Instead, we work with a slightly different model that is better suited for verification tools and our language. This model was developed for modeling iBGP in MineSweeper [8].

The key ideas in this alternative model are 1. to directly connect all iBGP peers using *virtual links* (see also, fig. 4.17c), *i.e.*, links that do not necessarily exist in the actual network but can be used to exchange BGP routes in our model and 2. exchange a data packet between each iBGP neighbor to verify that there is an IGP route between them and ACLs do not block communication between two iBGP peers.

This model deviates from how iBGP actually operates; it does not encapsulate routes in data packets and assumes it can exchange routes over links that do not exist in the topology. But, it is sufficient for our verification purposes. A BGP route is transmitted over a virtual link, if the two iBGP peers can exchange data packets, hence the transfer function for iBGP must look at the dataplane. This creates a dependency, from BGP to the IGP and the traffic routed over the IGP. To deal with these dependencies, MineSweeper creates N copies of the network where N is the number of routers configured to run iBGP, *i.e.*, one for each loopback address[2]. In NV, we can express these dependencies in two ways. The first, is essentially the approach that MineSweeper takes, creating one instance of the stable paths problem, where BGP, the IGP and the dataplane are solved together:

```
(* where rib and packets as defined in the previous sections *)
type attribute = rib * packets
```

This approach has the disadvantage that all three "layers" must be solved at the same time. As a result, simulation is slower as reaching a stable state requires more steps, and SMT verification has to deal with approximately N times more SMT constraints. The second approach, takes advantage of NV's ability to define multiple SPP instances and compute their solutions separately. Hence, we can define an SPP that models the IGP and its solutions, define an SPP that models the dataplane over the IGP, and finally define an SPP that models BGP. The structure of this model can be seen in fig. 4.18. Given the solution to the dataplane, `transiBGP` (the transfer function for iBGP) uses it to exchange BGP routes via iBGP. This function operates over a link `e` (or `u v`) which corresponds to an iBGP peering; it might be a virtual link added to support our iBGP model or an actual physical link between the two devices. If an iBGP peering exists (line 14), before transmitting a route, the transfer

[2]MineSweeper can only encode one destination at a time, thus N networks are required to model N destinations.

function for iBGP checks that the route is learned from eBGP (line 17), and that the two iBGP peers can communicate with each other (lines 18-19). Notice that in this case, we do not increase the route's length.

We currently do not have an automated translation from Batfish to NV that works with iBGP. However, we have demonstrated that building such a model is not only possible, but facilitated by NV's ability to represent multiple SPP instances.

```
1  let igp = solution {init=initOspf; trans=transOspf;
2                      merge=mergeOspf;}
3
4  let dataplane = solution {init=initData; trans=transData igp;
5                            merge=mergeData}
6
7  ...
8  let transiBGP e route =
9    match route with
10   | None -> None
11   | Some route ->
12    let (u~v) = e in
13    (match (loopback u, loopback v) with
14     | (Some ipu, Some ipv) ->
15      (* Do not redistribute internal BGP routes,
16         check iBGP peers can talk with each other*)
17      if (route.external = true &&
18         dataplane[v][{srcIp=ipu; dstIp=ipv}] &&
19         dataplane[u][{srcIp=ipv; dstIp=ipu}]) then
20         Some { route with nexthop = flipEdge e;
21                length = route.length;
22                external = false}
23       else None
24     | _ -> None)
25 ...
26
27 let egp = solution {init=initBgp; trans=transBgp; merge=mergeBgp}
```

Figure 4.18: Part of the iBGP model in NV.

Chapter 5

Network Analyses Implementation

This chapter describes the implementation of common network analyses on top of the NV language. We start by explaining the components of a network simulator, the simulation algorithm (section 5.1.1) and the process of interpreting the network functions (sections 5.1.2 and 5.1.3). We continue the discussion about network simulation with an overview of related work in section 5.1.4. Finally, we conclude with an evaluation of the performance of our simulator in section 5.1.5.2.

Next, we discuss network verification using an SMT solver. Section 5.2.1 gives an overview of how to formulate the stable paths problem as SMT constraints based on the encoding suggested by MineSweeper [8]. Section 5.2.2 explains how NV expressions are translated to SMT constraints and details the design and implementation choices that we made in order for the SMT back-end to scale. We conclude by evaluating the performance and scalability of our SMT backend in section 5.2.3.

5.1 Network Simulation

A simulator is an algorithm that mimics the exchange and processing of messages as dictated by the semantics of the protocols in play. Existing simulators such as Batfish [20] and FastPlane [37] are invaluable industrial tools for testing the conse-

Algorithm 2 Network Simulator

```
1:  procedure UPDATE(ℒ, q, v, route)
2:      if route ≠ ℒ(v) then ℒ(v) ← route; q ← q ∪ {v}
3:
4:  procedure SIMULATE(V, E, init, trans, merge)
5:      q ← {}
6:      for u ∈ V do
7:          ℒ(u) ← init(u)                                  ▷ Best route of node u
8:          received(u)(u) ← init(u)                        ▷ Routes received at u
9:          q ← q ∪ {u}
10:     while q ≠ empty do
11:         u ← pop q                                       ▷ Propagate u's route
12:         for v ∈ neighbors(u) do
13:             new ← ⟦trans((u, v), ℒ(u))⟧
14:             if u ∈ received(v) then                     ▷ Is there a stale route?
15:                 old ← received(v)(u)
16:                 if ⟦merge(v, old, new)⟧ = new then      ▷ Incremental update
17:                     UPDATE(ℒ, q, v, ⟦merge(v, ℒ(v), new⟧)
18:                 else UPDATE(ℒ, q, v, ⋃_⟦merge⟧ received(v))   ▷ Full update
19:             else UPDATE(ℒ, q, v, ⟦merge(v, ℒ(v), new)⟧)
20:             received(v)(u) ← new
```

quences of various routing configurations. However, the key difference between our simulator and existing ones is that they are designed to simulate specific control plane protocols (*e.g.*, BGP, OSPF, etc.) whereas our simulator simulates the NV *programming language*. Our simulator is capable of simulating models beyond conventional control planes, such as non-traditional protocols that correspond to new analyses (section 6.3), or other components of the network such as the data plane (section 4.2).

5.1.1 The simulation algorithm

Algorithm 2 presents the core simulation algorithm. We use the notation $[\![e]\!]$ to denote execution of the functional components of NV, such as the init, transfer and merge functions. Subsequent sections describe the non-standard elements of the execution of these components.

The goal of the algorithm is to compute the solution $\mathcal{L}$ of a given instance of the stable paths problem. Recall that such a solution is a *stable state* of the system. In other words, the solution $\mathcal{L}(u)$ at every node u must be equal to the merge of its initial attribute with all attributes transferred from its neighbors (see section 2.3).

Overall, the algorithm for computing solutions is structured as a worklist algorithm that stores the nodes to be processed on a priority queue (q). Lines 6-9 initialize the solution $\mathcal{L}$ and populate the queue with all nodes in the network. Lines 10-11 select a node from the queue to process, or, if there are none left, terminate the algorithm.

Lines 12-20 explain how to process a node u. To do so, u sends its current attribute to all of its neighbors v. Each neighbor v may need to update its solution. The neighbor v checks whether it has previously received information from u (*i.e.*, if $u \in$ received(v)). If not, it can simply merge the new information with its current solution (lines 19-20). Otherwise, v's solution contains stale information from u. The simple thing to do in this case is to recompute a merge of all received messages, including the new one from u (line 18). However, this is costly, so we use an observation from ShapeShifter [10], which is that when (`merge old new`) = `new` it suffices to perform an *incremental merge*, *i.e.* to merge `new` into the existing solution rather than recomputing a merge of all received messages (lines 15-17).

Correctness and Convergence. There are some important questions that arise about this simulation algorithm:

- Does it truly compute a stable state of the network?
- Is it guaranteed to converge to a stable state if one exists?
- Does the message ordering affect the computation?

Prior research has answered these, for the more restricted setting that corresponds to the original stable paths problem. In the original formulation of the problem, the merge function is considered to be a selective function[1]. Under this condition, Griffin *et al.* [30] proved that if this algorithm converges then it computes a state that corresponds to a stable state. In [51] Sobrihno identified local properties of the transfer and merge functions that guarantee that the network and the simulation algorithm converges to a stable state. In particular, one needs to check that the transfer function is strictly monotonic. To formulate this property in our context, consider a merge function of the form:

$$\mathsf{merge}(a, b) = \begin{cases} a, \text{if } a \prec b \\ b, \text{otherwise} \end{cases}$$

where $\prec$ is a ranking function between two routes (*e.g.*, for integers one could consider $<$ as the ranking function). Then, strict monotonicity is defined as:

$$\forall e \in E.\ \forall x \in A.\ \mathsf{merge}(\mathsf{trans}(e, x), x) = x \wedge \mathsf{trans}(e, x) \neq x$$

In other words, a route becomes worse (according to the rank function) every time it traverses a link.

Recent work by Daggitt *et al.* [16] showed that these convergence guarantees hold even when dealing with an asynchronous system where messages may be dropped, re-ordered or sent more than once. More research is required to identify weaker and more general conditions that apply over our more expressive context, and guarantee properties such as convergence.

[1] A function $f(x, y) : A \to A \to A$ is selective when its output is one of its inputs. For instance, the function $f(x, y) = min(x, y)$ is selective.

5.1.2 Interpreting Network Functions

To use the simulator we need to implement the process that computes $[\![e]\!]$, *i.e.* an *interpreter* for our expression language. One non-standard aspect of NV is the presence of symbolic values. Interpreting programs with such values would require to use techniques similar to the ones used in symbolic execution [33]. Unfortunately, such interpreters scale very poorly, especially when dealing with programs that have many branch points. Instead, our simulator treats symbolic values as *program inputs*; every declaration of a symbolic, is instantiated to a concrete value (chosen by the simulator or provided by the user) before simulation.

With symbolics out of the way, an interpreter for NV could be a standard ML-based interpreter implementation that treats NV structures as ordinary functional data structures. However, a conventional implementation would have led to non-competitive performance. To accelerate performance, our interpreter implements maps as *multi-terminal binary decision diagrams*. We focus on this aspect of the interpreter in the following sections.

5.1.2.1 Representing Maps as MTBDDs

Multi-Terminal binary decision diagrams are a variant of binary decision diagrams capable of representing functions from finite domains to arbitrary values (rather than just booleans). MTBDDs have been used for symbolic model checking of domains requiring rich structure, such as probabilistic systems [4]. However, NV makes this data-structure, more accessible as its exposed to the users through a high-level maps abstraction. We leverage the fact that MTBDDs store one copy of each leaf —we have found that due to the highly symmetric nature of networks, for many practical purposes NV maps contain many repeated values, hence an MTBDD representation is quite compact. Moreover, the NV simulator frequently applies the same operation to each entry in a map —sharing of leaves means we need apply the operation only

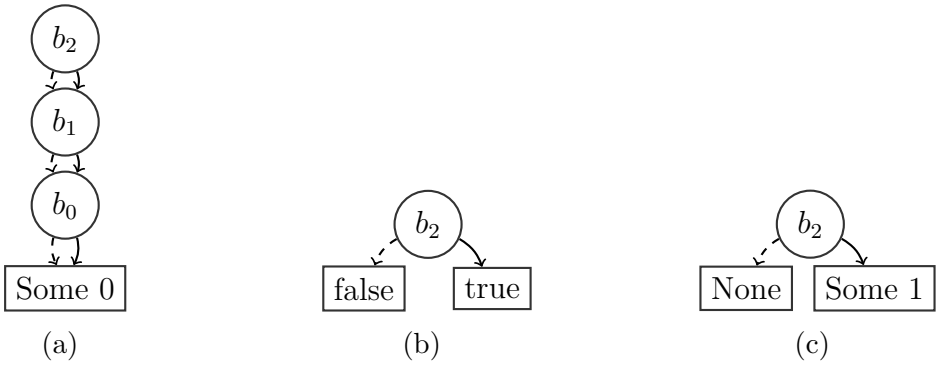


Figure 5.1: Implementation of `mapIte (fun k -> k > 3) opt_incr (fun v -> None) (create (Some 0))`, where `opt_incr = (fun v -> match v with | None -> None | Some v -> Some v+1)`. This operation increments the value of map entries whose key is greater than 3, and sets others to `None`. It is applied on a map from 3-bit integers to the `Some 0` value. (a) shows the MTBDD created by the map operation `create (Some 0)` (presented is an unreduced MTBDD). The MTBDD decision nodes are labelled with a bit (b_2 is most significant). If a bit is false, one follows the dashed line to find the corresponding structure; otherwise, the solid line. Here, any bit pattern leads to `(Some 0)`. (b) shows the encoding of the predicate `(fun k -> k > 3)` as a BDD. (c) The result of `mapIte`, by performing an MTBDD apply operation on (a) and (b) and mapping `opt_incr` and `(fun v -> None)` over the result.

once per distinct entry. Finally, the canonical structure of MTBDDs enables efficient equality tests. Fast (in)equality tests significantly improve simulator performance, as the simulator performs many such tests to see if a node's attribute changed after a merge operation.

To represent a map as an MTBDD, we must represent its domain as a series of binary decisions. With the exception of functions, types in NV are designed to be finitary, hence they can be represented in such a fashion. For instance, finite integers are represented bitwise; fig. 5.1a shows the MTBDD corresponding to a total map from 3-bit integers to values of type `option[int]`. In addition to functions, we also disallow maps to be used as keys in other maps; while it would be technically possible, it would incur a high overhead. Figure 5.2 defines a size function that determines how many binary decisions are required to represent values of each NV type. Note that we consider node identifiers to be 24-bit integers, but this can be dynamically

$$\text{size}(\text{ty}) = \begin{cases} 1 & \text{if ty} = \texttt{bool} \\ N & \text{if ty} = \texttt{int}\langle N \rangle \\ 24 & \text{if ty} = \texttt{node} \\ 48 & \text{if ty} = \texttt{edge} \\ 1 + \text{size}(\text{ty}') & \text{if ty} = \texttt{option}[\texttt{ty}'] \\ \sum_i \text{size}(\text{ty}_i) & \text{if ty} = (\texttt{ty}_1, \ldots, \texttt{ty}_2) \\ \sum_i \text{size}(\text{ty}_i) & \text{if ty} = \{\ell_1 : \texttt{ty}_1, \ldots, \ell_n : \texttt{ty}_n\} \end{cases}$$

Figure 5.2: Function determining how many boolean variables are required to represent values of a given NV type.

adapted depending on the network's topology, shrinking it to improve performance or expanding it to accommodate networks with 2^{24} or more nodes.

5.1.2.2 Map Operations over MTBDDs

$$\frac{e \rightsquigarrow \texttt{v} \qquad \text{size}(\texttt{kty}) = \text{n} \qquad \text{constant}(\text{n}, \texttt{v}) = \text{Leaf } \texttt{v}}{\texttt{create e} : \texttt{dict}[\texttt{kty}, \texttt{vty}] \rightsquigarrow \text{Leaf } \texttt{v}}$$

$$\frac{e_1 \rightsquigarrow \texttt{m} \qquad e_2 \rightsquigarrow \texttt{k} \qquad \text{val\_to\_bdd}(\texttt{k}) = \texttt{b} \qquad \texttt{m}_{|\texttt{b}} \Downarrow \{\texttt{v}\}}{\texttt{get e}_1\ \texttt{e}_2 \rightsquigarrow \texttt{v}}$$

$$\frac{e_1 \rightsquigarrow \texttt{m} \qquad e_2 \rightsquigarrow \texttt{k} \qquad e_3 \rightsquigarrow \texttt{v} \qquad \text{val\_to\_bdd}(\texttt{k}) = \texttt{b}}{\texttt{set e}_1\ \texttt{e}_2\ \texttt{e}_3 \rightsquigarrow \text{Ite}(\texttt{b}, \text{Leaf } \texttt{v}, \texttt{m})}$$

$$\frac{\begin{array}{c} e_1 \rightsquigarrow \texttt{p} \qquad e_2 \rightsquigarrow \texttt{f} \qquad e_3 \rightsquigarrow \texttt{g} \qquad e_4 \rightsquigarrow \texttt{m} \\ \text{eval\_bdd}(\texttt{p}) = \texttt{b} \qquad \text{op}(\texttt{v}_1, \texttt{v}_2) = \begin{cases} \texttt{f v}_2 & \text{if } \texttt{v}_1 = \texttt{true} \\ \texttt{g v}_2 & \text{if } \texttt{v}_1 = \texttt{false} \end{cases} \end{array}}{\texttt{mapIte e}_1\ \texttt{e}_2\ \texttt{e}_3\ \texttt{e}_4 \rightsquigarrow \texttt{p} \oplus_{\text{op}} \texttt{m}}$$

$$\frac{e_1 \rightsquigarrow \texttt{f} \qquad e_2 \rightsquigarrow \texttt{m}_1 \qquad e_3 \rightsquigarrow \texttt{m}_2 \qquad \text{op}(\texttt{v}_1, \texttt{v}_2) = \texttt{f v1 v2}}{\texttt{combine e}_1\ \texttt{e}_2\ \texttt{e}_3 \rightsquigarrow \texttt{m}_1 \oplus_{\text{op}} \texttt{m}_2}$$

Figure 5.3: Implementation of map operations in NV using MTBDDs.

Most map operations reduce to constructing BDDs and combining them with the MTBDD that represents the map. Figure 5.1 illustrates an example of this. For

instance, to implement a `mapIte` operation, we compute a BDD that corresponds to the function provided as a predicate; the BDD determines which function should be applied over which map entries. Combining this with the MTBDD that represents the map boils down to a standard MTBDD *apply* operation [14, 7]. In a nutshell, given two functions f and g, their (MT)BDD based representation B_f and B_g, and an operation $\diamond$ (on the leaves of the MTBDDs), an apply operation, recursively descends the two tree structures to compute an (MT)BDD s.t.:

$$\text{apply}(\diamond, B_f, B_g)(x) = f(x) \diamond g(x)$$

As a shorthand for apply we write $B_f \oplus_\diamond B_g$.

Figure 5.3 provides a more detailed view of the (MT)BDD manipulations underlying our map operations. In the following paragraphs, we use a red font to denote (MT)BDDs/operations over (MT)BDDs.

- Creating a map, uses the size function to allocate the right number of decision variables for the MTBDD and otherwise just allocates a leaf node with the value given as argument. All possible assignments to the decision variables lead to this single leaf.

- The `get` operation demonstrates one way BDDs are combined with the MTBDD representing the map. In particular, the function val_to_bdd creates the BDD representation of an NV value, which is then combined with the MTBDD using a *constrain* operation. Constrain is a standard BDD operation and simple constrains the domain of a given BDD/MTBDD using another BDD. For instance, given a function $F(x, y, z)$ (represented as an MTBDD), we can constrain its domain s.t. $F(x, y, 0)$ simply by dropping the decision variable z and replacing it with the false branch for z. We write $\mathtt{m}_{|\mathtt{b}}$ to mean the MTBDD `m` constrained by the BDD `b`.

The final step is to retrieve the leaves that correspond to the now restricted MTBDD. We write `m`$\Downarrow S$ for the library operation that returns the set of valid leaves in `m`. In this case, the set will be a singleton as we have constrained the domain of `m` to a specific *concrete* value.

- For `set`, we again compute the BDD that corresponds to the value of the key being updated. But, in this case, we use an if-then-else (Ite) combinator from the BDD library to update the map entry for the "path" in the tree specified by the key that is being updated.

- `mapIte` is perhaps the most interesting and complicated in its implementation map operation. The key component is the function eval_bdd which constructs a BDD that captures the semantics of an NV predicate, *i.e.*, of a function from the map keys to booleans. Section 5.1.2.3 describes this procedure in more detail. To compute the resulting map, the computed BDD is combined with the MTBDD representing the map using an apply operation.

- `combine` does a pointwise combination of two maps, by a straightforward apply operation of the user-provided function.

Constructing BDDs and combining them can be computationally expensive. To amortize the cost of the operations shown in fig. 5.3 we cache them (for example, we cache the result of procedures such as eval_bdd). In practice, cache hits are likely to occur frequently during simulation since multiple nodes have similar configurations (*e.g.*, filtering the same communities), and since a node may execute the same operation multiple times before it converges to a stable state.

5.1.2.3 Interpreting NV Expressions as BDDs

One of the challenges with the MTBDD implementation of maps is that we have to implement operations of a rather expressive functional language using BDDs. The

eval_bdd function essentially acts as a symbolic interpreter for NV; it computes the result of an NV function when applied to *any* possible key, *i.e.* a symbolic key represented as a BDD. Hence, with eval_bdd we implement an interpreter for the NV language that can handle non-concrete values represented by BDDs. As an initial attempt, we could treat all values as BDDs and all operations as operations over those BDDs. Since, this function is only used to compute a BDD for the `mapIte` operation the overhead will be somewhat restricted. That said, BDD operations are rather expensive. For example, consider the cost of an addition operation implemented by a machine-level instruction versus the same operation performed in a bitwise fashion over vectors of booleans. Moreover, we have already observed that the predicate of a `mapIte` expression may encode very complex operations; for example, in our data plane encoding (section 4.2) the function encoding the longest matching prefix procedure is encoded as a BDD using eval_bdd. To maximize performance, we take an approach combining concrete operations (over concrete values, or collections of concrete values) whenever possible, and resorting to symbolic operations (over BDDs) when it is necessary. We detail this approach in the following paragraphs.

Types of Values. Roughly, there are three different type of values used by our BDD-based interpreter: (i) BDD values that can represent symbolic values, (ii) concrete NV values, and (iii) collections of concrete NV values. To illustrate the difference between these type of values, we show how they manifest in a concrete example in fig. 5.4. In the predicate of the `mapIf` expression, the argument `k` will be a BDD value, denoting all integers in the map's domain. The variables `u` (representing a node) and `x` (representing a map) are free variables from the context and are regular NV values. However, as interpretation proceeds values might change form; the masking operation `[k & 3]` between `k` (a BDD) and 3 (a concrete value) results in a BDD, and the map get operation `x[k & 3]` between the concrete value `x` and the BDD value `k &`

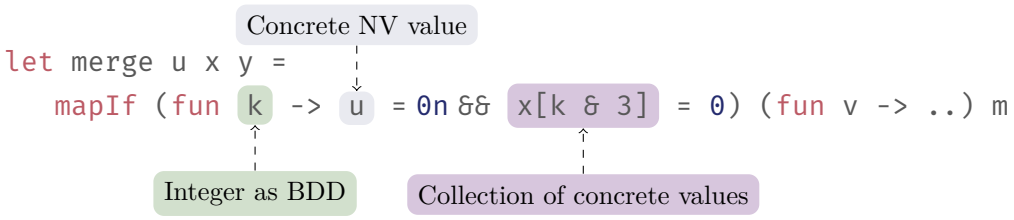


Figure 5.4: NV code demonstrating the different types of values. The variables `x` and `u` are free variables whose value is provided by the context. For example, in this case the `mapIf` expression is part of a merge function and `x` and `u` are two of its arguments.

`3` produces a collection of concrete values: the set of values that correspond to the map entries where all but the two least-significant bits of the key are set to 0.

The exact representation of these different type of values in our OCaml implementation is shown in fig. 5.5. There are two types of base BDD values, booleans and integers. Booleans are represented by a single BDD, while integers are represented bitwise by an array of BDDs. An optional value uses an extra BDD to represent whether the value is `None` or not. Concrete values are regular NV values (fig. 3.3), and collection of concrete values are represented by MTBDDs; just like NV maps but it is technically convenient to separate the two in the implementation. Tuples are lists of any of these values.

OCaml

```
type t =
  | BBool of Bdd.vt (* A BDD representing a boolean *)
  | BInt of Bdd.vt array (* Integer as an array of BDDs *)
  | BOption of Bdd.vt * t (* Option of BDD *)
  | Tuple of t list (* Tuple of elements, not necessarily BDDs *)
  | Value of Syntax.value (* Conventional NV value *)
  | BMap of BddMap.t (* Collection of concrete NV values *)
```

Figure 5.5: The type of values used by BDD-based interpreter as declared in our OCaml implementation.

Reducing Expressions. The next step is to define how NV expressions reduce to the values defined above. The idea is to use the conventional reduction rules whenever possible, *e.g.*, for expressions that use concrete NV values; lift these reduction rules

$$\frac{\texttt{e}_1 \rightsquigarrow \texttt{Value v}_1 \qquad \texttt{e}_2 \rightsquigarrow \texttt{Value v}_2 \qquad \texttt{op} = \{+, \&, \&\&, ||, <, \leq, =\} \qquad \texttt{v}_1\ \texttt{op}\ \texttt{v}_2 \rightsquigarrow_\beta \texttt{v}}{\texttt{e}_1\ \texttt{op}\ \texttt{e}_2 \rightsquigarrow \texttt{Value v}} \text{ VAL-OP}$$

$$\frac{\texttt{e}_1 \rightsquigarrow \texttt{BMap m} \qquad \texttt{e}_2 \rightsquigarrow \texttt{Value v} \qquad \texttt{op} = \{+, \&, \&\&, ||, <, \leq, =\} \qquad \texttt{f} = \lambda.\ \texttt{v}_1\ \texttt{op}\ \texttt{v}_2 \qquad \texttt{m} \oplus_\texttt{f} \text{Leaf v} = \texttt{m}'}{\texttt{e}_1\ \texttt{op}\ \texttt{e}_2 \rightsquigarrow \texttt{BMap m}'} \text{ MTBDD-VAL-OP-L}$$

$$\frac{\texttt{e}_1 \rightsquigarrow \texttt{BInt b}_1 \qquad \texttt{e}_2 \rightsquigarrow \texttt{Value v}_2 \qquad \texttt{op} = \{+, \&, <, \leq, =\} \qquad \texttt{b}_1\ \widehat{\texttt{op}}\ (\texttt{value\_to\_bdd}(\texttt{v}_2)) = \texttt{b}}{\texttt{e}_1\ \texttt{op}\ \texttt{e}_2 \rightsquigarrow \texttt{BInt b}} \text{ BDD-INT-OP-L}$$

$$\frac{e_1 \rightsquigarrow \texttt{Value m} \qquad e_2 \rightsquigarrow \texttt{BInt bs} \qquad \texttt{k} = \texttt{And}(\texttt{m(i)} = \texttt{bs[i]}) \qquad \texttt{m}' = \texttt{m}_{|\texttt{k}}}{\texttt{get}\ \texttt{e}_1\ \texttt{e}_2 \rightsquigarrow \texttt{BMap m}'} \text{ BINT-MAP-GET}$$

Figure 5.6: Example reduction rules for binary operations.

over collection of values, *e.g.*, for operations between a concrete value and a collection of concrete values; as a final resort, we lift NV operations to operations over BDDs, *e.g.*, when computing the result of a binary operation between a BDD and a concrete value, the concrete value is lifted to a BDD and reduction proceeds as an operation between two BDDs.

There are many formal reduction rules; we present a subset of them that are representative of the key ideas and the design decisions we made.

Figure 5.6 shows reduction rules for various binary operations:

- Rule VAL-OP applies to binary operations between concrete NV values and simple uses the standard NV interpreter to reduce (denoted as $\rightsquigarrow_\beta$).
- Rule MTBDD-VAL-OP-L shows how a binary operation is lifted when applied over a collection of values. Each leaf of the MTBDD stores a concrete NV value, and the NV operation is applied on each one of them by constructing another MTBDD with a single leaf and combining the two using an MTBDD apply operation.

- BDD-Int-Op-l demonstrates reduction over BDDs. To reduce an operation over a symbolic integer (represented bit-wise as an array of BDDs) and a concrete integer, the concrete integer is lifted to a BDD representation and then the NV operation is performed as a lifted BDD-based operation (denoted in the rule as $\widehat{op}$). For this, we have to implement the NV operations over BDDs; as NV operations are standard arithmetic/boolean operations, we can refer to work on digital circuits [39] for their implementation. To give some intuition, we show the implementation of the simplest operation over two symbolic integers, `&` (bitwise and):

```OCaml
let uand (xs: Bdd.vt array) (ys: Bdd.vt array) =
  BInt (Array.init (Array.length xs)
                   (fun i -> Bdd.dand xs.(i) ys.(i)))
```

  Lifted operations are computationally more expensive compared to their concrete counterparts, as they have to manipulate BDDs, and operate over each bit separately.

- Rule BInt-Map-Get shows how a map get operation works when the key used is symbolic (*e.g.*, an integer represented using BDDs). Firstly, we compute a single BDD (`k`) associating each decision variable in the map (`m(i)`) with the respective bit of the symbolic integer (`bs[i]`). This BDD is then used to constrain the MTBDD, reducing the collection of values according to domain defined by `k`. This is the same idea as the one we used in the map get operation where the MTBDD was constrained by a BDD the key (see, section 5.1.2.2), but in this case the BDD may describe more than a single key.

Figure 5.7 shows the reduction rules for conditional expressions such as if-then-else:

$$\frac{\texttt{e}_1 \rightsquigarrow \texttt{Value v}_1 \qquad \texttt{e}_2 \rightsquigarrow \texttt{v}_2 \qquad \texttt{e}_3 \rightsquigarrow \texttt{v}_3 \qquad \texttt{v} = \begin{cases} \texttt{v}_2 & \text{if } \texttt{v}_1 = \texttt{true} \\ \texttt{v}_3 & \text{if } \texttt{v}_1 = \texttt{false} \end{cases}}{\texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3 \rightsquigarrow \texttt{v}} \textsc{Ite-Val}$$

$$\frac{\texttt{e}_1 \rightsquigarrow \texttt{BBool b}_1 \qquad \texttt{e}_2 \rightsquigarrow \texttt{BInt b}_2 \qquad \texttt{e}_3 \rightsquigarrow \texttt{Value v}_3 \qquad \texttt{value\_to\_bdd}(\texttt{v}_3) = \texttt{b}_3 \qquad \texttt{b}[\texttt{i}] = \text{Ite}(\texttt{b}_1, \texttt{b}_2[\texttt{i}], \texttt{b}_3[\texttt{i}])}{\texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3 \rightsquigarrow \texttt{BInt b}} \textsc{Ite-Bdd-Int-L}$$

$$\frac{\texttt{e}_1 \rightsquigarrow \texttt{BBool b}_1 \qquad \texttt{e}_2 \rightsquigarrow \texttt{Value v}_2 \qquad \texttt{e}_3 \rightsquigarrow \texttt{Value v}_3 \qquad \oplus(\texttt{b}, \texttt{x}, \texttt{y}) = \begin{cases} x & \text{if } b = \texttt{true} \\ y & \text{if } b = \texttt{false} \end{cases} \qquad \texttt{m} = \oplus(b_1, \text{Leaf } \texttt{v}_2, \text{Leaf } \texttt{v}_3)}{\texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3 \rightsquigarrow \texttt{BMap m}} \textsc{Ite-Bdd-Val}$$

Figure 5.7: Example reduction rules for if-then-else expressions.

- Ite-Val applies when the guard of an if-then-else expression is a concrete value. It simple reduces to the true or false branch regardless of the type of value of each branch.

- When the guard and one of the branches of the if-then-else expression reduce to a BDD, then the operation must be lifted to a symbolic operation over BDDs. In rule Ite-Bdd-Int-L, the guard has reduced to a BDD, and the true branch to a symbolic integer. On the other hand, the false branch has reduced to a concrete value, hence we lift it to a BDD using `value_to_bdd` and use a BDD `Ite` to compute the final result. Note, that this must be done in a bitwise fashion for integers.

- Ite-Bdd-Val shows an alternative to the above rule that applies when the guard has reduced to a BDD value (*i.e.* `BBool b`$_1$) but both branches have reduced to concrete values. In this case, we can represent the result as a collection of values, *i.e.*, an MTBDD that returns the value of the true branch when the BDD is true and the value of the false branch when the BDD is false.

Beyond the reduction rules presented, there are a few dozen more to handle all the possible combinations of values, and more complicated data-structures such as tuples and options. Reducing `match` expressions is particularly complicated because the guard may contain multiple type of values (*e.g.*, a tuple with a BDD value, an MTBDD value, and a concrete value). In contrast, the `if-then-else` only has a single type of value as a guard.

Trade-offs and Limitations. There are certain limitations in our current implementation. In particular, some operations between collections of values and symbolic values are currently not supported. One such example is an addition operation between an MTBDD with integers in the leaves and a symbolic integer represented bitwise using BDDs. One way to implement this operation is to "lift" the MTBDD to a BDD representation. Intuitively, an MTBDD comprises of a set of values (leaves) and the paths (decisions) that lead to these. We can construct a symbolic integer (an array of BDDs) by converting the integers stored in the MTBDD leaves to BDDs using `value_to_bdd`, and then combine them in a single symbolic integer based on the paths that led to them, using a BDD if-then-else (`Ite`) combinator. Addition then is performed between two BDDs as per usual.

Obviously such operations are computationally expensive, and more importantly they deviate from the philosophy of separation between symbolic reasoning and concrete execution that we are seeking to enforce. That said, if the MTBDD has few leaves, we expect the analysis to scale even when these operations are permitted. The rule Ite-Bdd-Val highlights another design choice to be made; should we prefer collections of concrete values (represented by MTBDDs) or symbolic values (represented by BDDs)? Currently, we stick to collections of concrete values whenever possible, but it is possible that when these collections get sufficiently large a completely symbolic approach (*i.e.*, using BDDs instead) might perform better. In the future, as we

tackle more complex systems, it will be interesting to explore techniques to estimate the computational cost of programs evaluated in this mixed mode (between concrete execution, execution over collections of values and symbolic execution), and to devise strategies to pick an optimal representation (collection of values vs. symbolic values) either dynamically or statically.

5.1.3 Native Execution

Past implementations of control plane simulators ([20, 37, 10]) have relied on *interpreting* computational elements of control plane protocols. Of course, interpreters are generally much slower than *compiled* programs. In our case, because we have already mapped the ad hoc vendor configuration languages into a conventional functional programming language, the work required to compile NV to machine code is significantly reduced. Indeed, we can translate NV functions to OCaml, use OCaml's compiler to obtain assembly code, and link the compiled binary to the simulator. The key takeaway is that simulation can now proceed as in algorithm 2, but using the compiled functions instead of interpreting NV syntax. In the following paragraphs, we explain the technical aspects of simulation using native execution.

Architecture. The translation from NV to OCaml is straightforward for many constructs; functions, options, integers, booleans, and tuples all have corresponding constructs in OCaml. However, there are non-standard aspects of NV, such as symbolic values, solution declarations that need to simulate an SPP, and the MTBDD-based maps which OCaml has no built-in support for. We link compiled NV programs to a *runtime system* that provides the necessary functionalities to implement these constructs, and also, to be able to convert solutions computed as OCaml values to NV values for further processing and presentation to the user.

Figure 5.8 provides an overview of the structure of a compiled NV program. An NV program is translated to an "open" OCaml module that needs to be linked with a module that provides concrete values for symbolics, and a module that provides a simulator. This design allows us to compile the NV program to assembly once (an expensive process, as files are often thousands of lines) and provide different instantiations to symbolic values without having to recompile the full NV program. Similarly, we could have included a simulator in the compiled program, but this would increase compilation time and would require us to recompile an NV program every time improvements are made to the simulator. Once an NV program is translated to OCaml and compiled to machine code using OCaml's compiler, the NV runtime is responsible for *dynamically linking* the compiled program with the modules of type `Symbolics` and `SimulatorSig`; once the modules are linked together, the compiled program is natively executed, calling the simulator provided by the `SIM` module whenever it needs to compute stable solutions to a given SPP instance.

Communication Between OCaml and NV. It is technically convenient to be able to move between OCaml values and NV values:

- It allows us to reuse parts of our MTBDD-based maps library for NV values, without having to re-implement it for arbitrary OCaml values.
- The results of the simulation can be examined, manipulated and presented to the user, using existing functions for NV values.

We implemented functions for converting a subset of OCaml into NV (an *embedding*) and, conversely, for converting NV values back into OCaml (an *unembedding*). Obviously, to embed an OCaml value into an NV value during runtime, we need some kind of structural information for this value; types provide this information. Both the embed and unembed operations are indexed by an NV type (although, the latter for reasons relating to our implementation). Moreover, as we have already type-checked

```
let nodes = ...
let edges = ...
symbolic s: (int, int);

let init u = ...
let trans e x = ...
let merge u x y = ...

let sol = solution {init; trans; merge}
```

OCaml

```
module FatTree (S: Symbolics) (SIM: SimulatorSig) : NativeSPP =
struct
  (* Helpers for (un)embedding values between NV and OCaml. *)
  let record_cnstrs x = ...
  let record_proj x = ...

  (* Get the value of the first symbolic in the program's text*)
  let s = S.get_symb record_cnstrs record_proj 0

  let init u = ...
  let trans e x = ...
  let merge u x y = ...

  (* Given the attribute type, a name for the SPP solved, and an
   SPP, simulates the SPP and stores the result.*)
  let sol = SIM.simulate record_proj ~type_id:3 ~sol_name:"sol"
   init trans merge
end
```

Figure 5.8: Structure of Compiled Programs.

the program we know the type of every value we might want to embed/unembed. For example, the `SIM.simulate` function, takes as input a type id that corresponds to the attribute type of the SPP instance solved (the runtime system keeps track of these relations), such that it can convert the computed solutions to NV values to be further processed, or to be presented to the user.

Figure 5.9 shows part of the implementation of the function embedding OCaml values into NV. The function matches on the the given type and returns a function that a given an OCaml value will construct an NV value. This is straightforward,

for boolean or integer values that have a simple structure, but what about more complex values, such as tuples? Given an OCaml tuple of arbitrary size, we have no way to "deconstruct" it. This is where `record_proj` comes in handy; it provides a projection function for every tuple type that appears in the given NV program. Specifically, `record_proj (i,n) v` projects the i-th element of a tuple of size n. We use `record_proj` to deconstruct an OCaml tuple and to complete the implementation of `embed_value`.

```OCaml
let rec embed_value (record_proj: (int*int) -> 'a -> 'b)
                    (typ: Syntax.ty) : 'v -> Syntax.value =
  match typ with
  | TBool -> fun v -> Syntax.vbool (Obj.magic v)
  | TInt n -> fun v -> Syntax.vint (Obj.magic v)
  | TTuple ts -> ...
```

Figure 5.9: Embedding OCaml values to NV.

Handling Maps. Implementing the MTBDD-based maps as part of an OCaml program poses its own set of challenges. OCaml has no built-in support for MTBDD-based maps; to use MTBDDs with OCaml values we have two solutions: 1. either develop an MTBDD-based map library that operates directly over arbitrary OCaml values, or 2. reuse our existing library for NV values, embedding an OCaml value to an NV value before storing it in the map and unembedding it when retrieving it. The latter is technically easier, as the untyped (and thus unsafe) code is restricted to the embedding and unembedding functions. However, it induces a lot of embeddings and unembeddings that add an unnecessary overhead to the simulation. We have implemented both approaches, and in our experience the improved performance of maps that operate directly over OCaml values justifies the extra implementation complexity.

Another issue that arises is the implementation of the `mapIf` operation. When we execute an expression such as `mapIf p f m`, the predicate `p` on the keys of the map has a non-standard, symbolic, interpretation: its evaluation results in a BDD that is used to restrict the domain of the map, *i.e.* by the process presented in section 5.1.2.2. Obviously, if we translated the predicate `p` to an OCaml function, this would not be possible; OCaml's semantics do not include symbolic interpretations using BDDs. Instead, when translating to OCaml, we opt to retain the original NV expression via an expression id. To relate the expression id with the original NV expression, the runtime maintains an array of NV expressions, one for each unique predicate that appears in the program. If these predicates were closed (no free variables), we would be done. However, in general, we must also be able to get our hands on the values bound to the free variables inside such functions. We do so by recording the free variables (aka, the function's environment) at the compiled call to `mapIf`. Given both the free variables and the code (*i.e.*, the complete closure), we are able to translate the function to a BDD at run time. As with interpreted simulation, to avoid recomputing BDDs multiple times, the runtime also caches the results of this operation. As an example, consider the merge function from fig. 5.4:

```
let merge u x y =
    mapIf (fun k -> u = 0n && x[k & 3] = 0) (fun v -> ...) m
```

The resulting OCaml function will be:

```OCaml
let merge u x y =
    NativeBdd.mapIf 0 (u, x) (fun v -> ...) m
```

where `NativeBdd.mapIf` is a library call to a function implementing the `mapIf` operation, 0 is the index of the NV expression that corresponds to the predicate `(fun k -> u = 0n && x[k & 3] = 0)`, and `(u,x)` represents the free variables that appear in the predicate. The function applied over the leaves is compiled to an ordinary OCaml function over OCaml values. Note, to interpret the predicate as a

BDD, our implementation of `NativeBdd.mapIf` embeds the free variables into NV values. Note, that if the approach of storing NV values in MTBDD leaves is chosen, then the leaves are unembedded to apply the OCaml functions and then embedded again to be stored in the map (hence the high overhead).

5.1.4 Related Work

Batfish Beyond its ability to parse router configurations from different vendors, Batfish also has its own simulator to compute routes. Originally [20], this simulator was based on a variant of Datalog called LogiQL. This approach bears some similarities with NV, in the sense that the protocol semantics and the router configurations have to be translated to LogiQL, a high-level declarative language. But, the computation engine of LogiQL is very different than an interpreter for a language like NV. In fact, its performance was one of the reasons it was later replaced with a more conventional, imperative interpreter.

FastPlane FastPlane [37] is a high-performance control plane simulator that relies on a generalized variant of Dijkstra's algorithm. The correctness of their simulation algorithm relies on the insight that the routing policy in datacenter networks is monotonic, *i.e.*, that route advertisements become less preferable as they are propagated throughout the network. Consequently, compared to Batfish or NV, FastPlane is less general, as it only works on monotonic networks, but its simulation engine is orders of magnitude faster than Batfish. Unfortunately, FastPlane is not publicly available so we cannot benchmark against it.

ShapeShifter ShapeShifter[10] is a simulator that uses *abstract interpretation* to scale simulation performance to very large networks. The key observation the authors make, is that to validate important properties like reachability to a destination along *some* path, it is not necessary to model all the details of the protocols in play.

For instance, for BGP, decision variables such as the local-preference value and the multi-exit discriminator can often be completely dropped without loss of precision. Additionally, they suggest that the BGP route path can be safely approximated by its length or even by a single boolean variable denoting whether there is a route or not.

Besides, abstractions of the route components, ShapeShifter's simulator greatly affected the design of the NV language and NV's simulator. In particular, ShapeShifter uses a BDD data-structure to represent "maps" from destination prefixes to routes. This is the idea that inspired our MTBDD implementation of maps. MTBDDs are likely the better choice because they do not require the map values (*i.e.* in this case routes) to be lifted to a BDD based representation which incurs a high computational cost. Another feature of ShapeShifter's simulator implemented in NV is incremental merging as explained in section 5.1.1.

5.1.5 Evaluation

In this section, we evaluate the performance of our simulator, the performance difference between interpreted and native simulation, and how our simulator fares compared to others, such as Batfish and ShapeShifter. We run all experiments on a 2015 Mac with a 4Ghz i7 CPU and 16GB of memory.

5.1.5.1 Networks Studied

For data center topologies, we focus on FatTree [3] designs, commonly used to interconnect large numbers of servers while providing fault tolerance and high bisection bandwidth. Fattree designs are parameterized by k, the number of "pods", and by varying k, one can explore the impact of topology[2] size on analysis time. For routing,

[2]SP(k) and FAT(k) each have $(5/4)k^2$ nodes and k^3 edges.

modern datacenter designs use the eBGP routing protocol [34] for its scalability and policy-rich configurability, coupled with variants of shortest path routing.

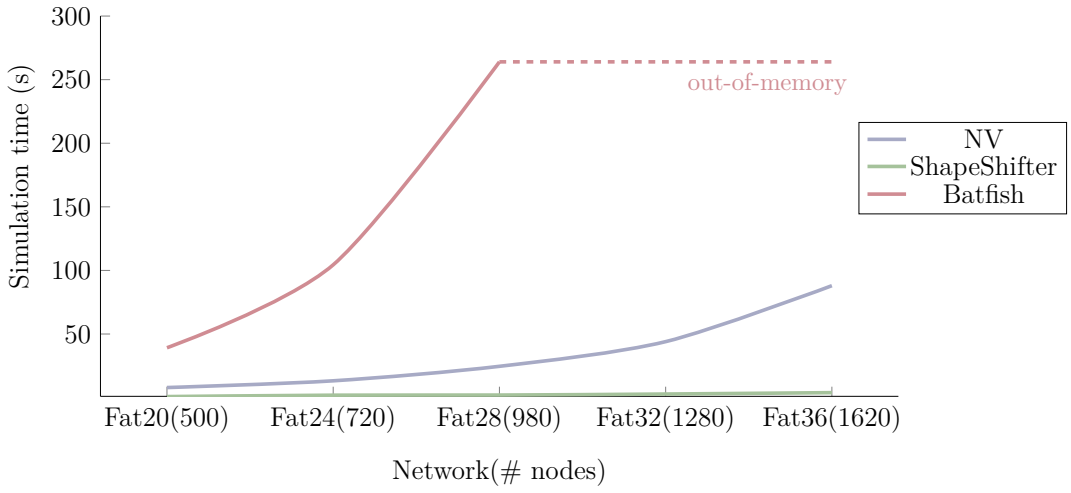
We consider two different policies described in the literature: a pure shortest-path routing policy (denoted using `SP`), and a variant that uses tagging and filtering to disallow "valley routing" [11], *i.e.* dropping routes that go through the same layer of the fat tree multiple times (denoted `FAT`).

5.1.5.2 Simulation Performance

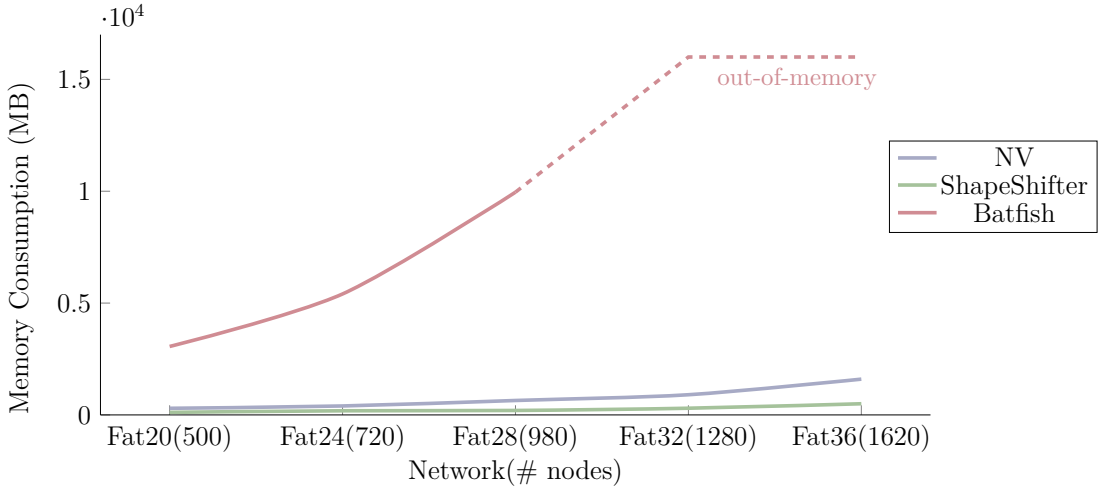
To gauge the performance of our simulator, we measure how it fares compared to Batfish and ShapeShifter when simulating the network for all destination prefixes[3] announced in fat tree networks ranging in size from 500 to 1620 nodes, and announcing between 200 and 648 destination prefixes. Figure 5.10 and fig. 5.10b show the computation time and memory usage respectively for the simulators. Firstly, our MTBDD-based simulator appears to be an order of magnitude faster than Batfish. More importantly, as the network size grows, the runtime of NV only marginally increases, while Batfish follows a much steeper trend. The same is true of the memory consumption of the two; NV peaks at `~1.6GB` for `Fat36` while Batfish runs out of memory (16GB) on the smaller `Fat32` network. Although Batfish's route model is more detailed, these differences in scaling trends are mainly attributed to the MTBDD representation of a router's RIB in NV; these networks advertise hundreds of prefixes and the ability to compactly represent them and process them in bulk is key to scaling the simulation as the network size increases.

The advantages of a BDD-based representation of a router's RIB are amplified in ShapeShifter. ShapeShifter employs even more aggressive abstractions, significantly reducing the state space, achieving better leaf "sharing" in BDDs, and faster convergence of the simulation. As a result, both the simulation time and memory

[3]For `Fat(k)` there are $k^2/2$ destinations announced through BGP.



(a)



(b)

Figure 5.10: The plots compare Batfish, ShapeShifter when BGP path length has been abstracted to a boolean, and NV when the interpreter is used for simulation. (a) shows the time to compute routes for all destination prefixes in the network. (b) shows the memory consumption.

consumption are practically flat as the network size increases. Note, however, that ShapeShifter's abstractions are not precise; they are only meant to be used to answer reachability questions very quickly. In fact, when ShapeShifter uses less abstract models, such as the ones NV uses, its performance deteriorates significantly (worse than Batfish). That is because, ShapeShifter represents routes as BDDs (and not

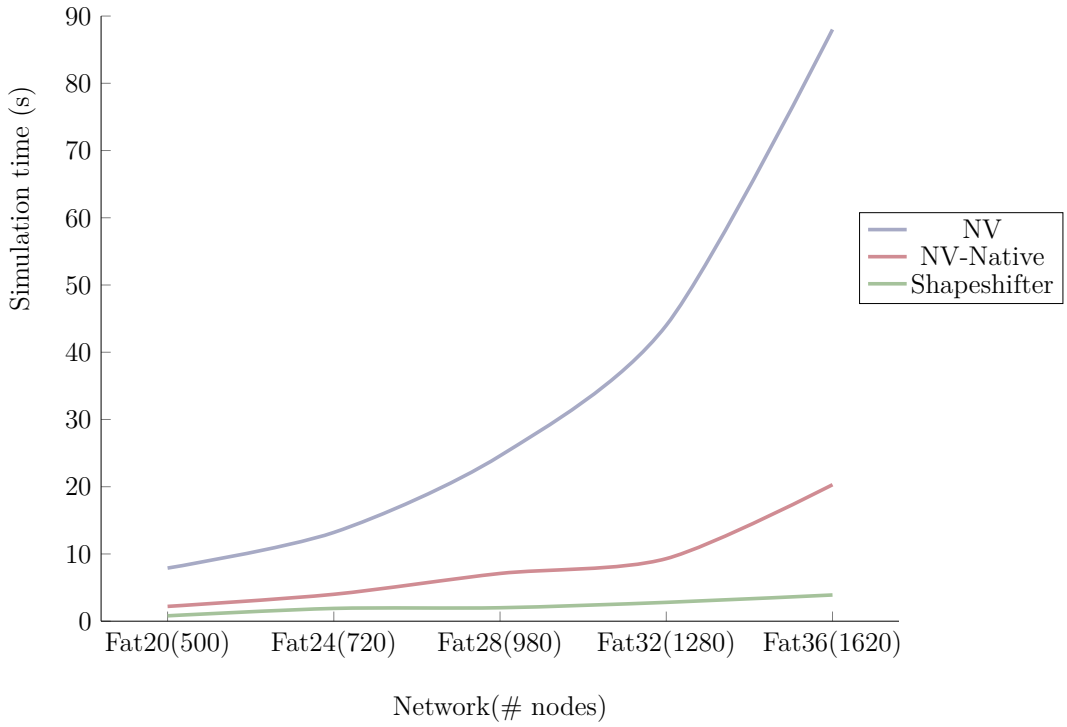


Figure 5.11: Comparison between interpreted, native simulation, and Shapeshifter for All-Prefixes analysis. NV represents the time for the interpreted simulation, and NV-Native the time for native simulation **excluding** OCaml compilation time.

using MTBDDs) and hence all operations are performed over BDDs which are very slow.

5.1.5.3 Native Simulation Performance

In the all-prefix simulations of fig. 5.11, we observe that execution time of the native simulator is substantially faster (roughly, 3-4x faster) than that of the interpreted simulation. This is not surprising, we know from general-purpose programming languages that interpreters are much slower than executing machine code. In fact, the networks we used have relatively simply policy; we expect the performance gap to widen for networks with more complex policy.

Cost of Native Simulation. Native execution can deliver significant performance improvements compared to conventional interpreter-based simulators, but it also has

inherent overheads. Embedding and unembedding values between NV and OCaml also induces some overhead, although caching these operations, and using MTBDDs with OCaml values in the leaves (see relevant discussion in section 5.1.3) minimizes this overhead.

The largest overhead, however, is induced by the OCaml compiler. Compiling a large OCaml program to assembly is a time consuming process, more so than the simulation itself. In the fat tree examples studied, OCaml's compilation time ranged between 30s and 1450s! It might seem that native simulation is useless in practice. But, despite the high compilation time, there are still cases where native simulation outperforms even when taking compilation time into account. For instance, as we show in section 6.3 simulating a more complex protocol is much faster using native simulation. Another use-case would be to perform multiple simulations by instantiating symbolic values with different concrete values, amortizing the compilation cost. But more importantly, we know that with some extra engineering effort we can drastically reduce the compilation time. OCaml's compiler was not designed to handle files with hundreds of thousands lines of code; if our translation from NV to OCaml generated multiple files (*e.g.*, by separating the transfer function for each link) instead of a single file, we anticipate that OCaml's compiler would scale far better. Moreover, we would be able to compile these files in parallel significantly reducing the total compilation time. Finally, this approach would allow us to benefit from incremental compilation; changing the configuration of a single router would not require us to recompile all other configurations. Similarly, changing the assertion function would only require us to recompile the (very small) assertion function —although assertion functions are usually simple, one could just interpret them once native simulation has concluded.

5.2 SMT-Based Verification

A second way to check properties of converged network states is through SMT [8]. The SMT-based approach does not model the convergence procedure; instead, it captures the stable solutions of the network using constraints. The challenge in creating an efficient SMT encoding is that high-level programs introduce abstractions that do not favor SMT reasoning. Much like a regular compiler, NV relies on an optimizing pipeline to produce tractable constraints. Section 5.2.3 shows that NV's systematic approach to optimizations results in improved performance compared to state-of-the-art control plane verifiers like MineSweeper.

The strength of the SMT-based analysis lies in its ability to perform symbolic reasoning. With respect to NV, this means that it can indeed reason about all possible assignments to a symbolic value. For instance, an operator can reason about what happens in their network when it receives any possible route from an external peer. In contrast, as section 5.1.2 explains, exhaustive analysis using approaches based on normalization, such as simulation, is often impractical.

5.2.1 Stable Paths as Constraints

Like the simulator, the SMT verifier, can also be split into two components. The first component, as shown in the MineSweeper paper [8], describes the structure of the constraints: if the converged states are specified by a formula `N` (*i.e.* the definition of a solution as shown in fig. 2.2) and we wish to verify a property `P` holds in those states, it suffices to show that $\texttt{N} \wedge \neg \texttt{P}$ is unsatisfiable; that is, there is no converged state of our network in which `P` does not hold. On the other hand, if the SMT solver deems that formula as satisfiable, then it will provide a *model*, *i.e.* an assignment of values to the SMT variables, that make this formula satisfiable; this model constitutes a counterexample and can be used to debug the network.

One advantage of this SMT encoding is that by describing only the notion of the stable state and not how its computed, the SMT solver can explore all possible stable states. In contrast, simulation uses one message ordering and may only explore at most one stable state. As such, this approach provides more exhaustive correctness guarantees. On the other hand, if the network has no stable state, the SMT solver will determine that the the formula $\mathtt{N} \wedge \neg \mathtt{P}$ is unsatisfiable because $\mathtt{N}$ is unsatisfiable, hence one needs to separately ensure that the network has a stable state for the result to be meaningful.

5.2.2 Compiling to Constraints

The second component of the SMT verifier concerns the translation of NV expressions (fig. 3.3) to SMT constraints. The NV language is sufficiently restricted that one could come up with a straightforward translation to SMT. In the absence of recursion, most constructs have straightforward representations in SMTLIB2 using complex SMT theories such as the array and datatype theory. However, such an encoding is far too inefficient to be practical; SMT solvers simply cannot deal with these theories as efficiently as they can with simpler constraints that only involve booleans and integers. Instead, much like an optimizing compiler, our translation from NV expressions to SMT is staged as a pipeline eliminating complex structures and simplifying the program before translating to SMT.

5.2.2.1 Transformations

Renaming and Inlining. The first stage involves renaming variables to ensure all variable bindings have unique names. Name binding is done via let and match expressions, and function declarations. For instance, the program:

```
let f x =
  match x with
  | None -> None
  | Some (x,y) -> x + y

let g y =
  let x = 2 in
  f (Some (x,y[0]+y[1]))
```

will be transformed by assigning a fresh binder for each binding:

```
let f~0 x~0 =
  match x~0 with
  | None -> None
  | Some (x~1, y~0) -> Some (x~1 + y~0)

let g~0 y~1 =
  let x~2 = 2 in
  f~0 (Some (x~2,y~1[0] + y~1[1]))
```

Further, after this phase we inline function definitions, completely eliminating any functions and function applications, other than the ones that appear in `solution` declarations, so the above program would be translated to:

```
let g~0 y~1 =
  let x~2 = 2 in
  match Some (x~2,y~1[0] + y~1[1]) with
  | None -> None
  | Some (x~1, y~0) -> Some (x~1 + y~0)
```

Map Unrolling. Map operations are chosen to allow programmers to specify maps with huge domains, such as the domain of all 2^{32} IP addresses, but to only pay a cost proportional to the subset that is used, such as the (much smaller) set of IP addresses that appear in the network. In particular, because there are no aggregation operations over the the entire map, such as a fold, entries that are never accessed via a `get` operation need not be represented. We implement such sparse maps as tuples, via a *map unrolling* process, where an element of the tuple is reserved for each key

that accesses the map. Map accesses then become tuple projections. Continuing with our working example, we apply map unrolling:

```
let g~0 y~1 =
  let x~2 = 2 in
  let a = match y~1 with (a,_) -> a in
  let b = match y~1 with (_,b) -> b in
  match Some (x~2, a+b) with
  | None -> None
  | Some (x~1, y~0) -> Some (x~1 + y~0)
```

When the keys used to access a map are constants, implementing the map as a tuple is straightforward — collect all the n constant keys used in the program and create a n-tuple with an entry for each constant. However, we also allow indexing maps with *symbolic values*, whose value is unknown at compile time. To accommodate both n constant keys $c_1, \ldots, c_n$ and m symbolic keys $s_1, \ldots, s_m$, we use a tuple of size $n+m$ where the value associated with c_i appears in element i and the value associated with s_j appears in element $n + j$. Of course, symbolic key s_j may actually resolve to the constant c_i. In this case, the computation must be as if the constant c_i was used in s_j's place. To do this, we encode map get on a symbolic key as follows (map set is similar). Here, assume we have one constant key `c` and two symbolic keys `s1, s2`:

```
encode(m[s]) =
  let (m0, m1, m2) = m in
  if s = c then m0 else
  if s = s1 then m1 else m2
```

Option Unboxing. Optional values are ubiquitous in functional programming languages and NV is no exception. SMT solvers, on the other hand, do not have special support for options. One has to encode them using the generic datatype theory, which has severe performance implications. Expressions of type `option[A]` are translated to pairs of type `(bool, A)` where the first component is `false` for `None` (and the

second component is irrelevant, we use a default value for the type), and `true` for `Some`.

```
let g~0 y~1 =
  let x~2 = 2 in
  let a = match y~1 with (a,_) -> a in
  let b = match y~1 with (_,b) -> b in
  match (true, (x~2, a+b)) with
  | (false, _) -> (false, 0)
  | (true, (x~1, y~0)) -> (true, (x~1 + y~0))
```

Tuple Expansion and Flattening.

After maps and options have been eliminated, the only complex data type left are tuples. We simplify their structure by flattening nested tuples and expanding variables of tuple type. For instance, we transform a variable `x` of type $[\alpha_1, [\alpha_2, \alpha_3]]$ to an expression `(x1,x2,x3)` of type $[\alpha_1, \alpha_2, \alpha_3]$. After flattening, tuples are just collections of base-type expressions (*i.e.* expressions of integer or boolean type) and we can encode them pointwise, as independent variables/expressions. A function that takes a tuple as an argument will be transformed to its *curried* form:

```
let g~0 y~2 y~3=
  let x~2 = 2 in
  let a = match (y~2, y~3) with (a,_) -> a in
  let b = match (y~2, y~3) with (_,b) -> b in
  match (true, x~2, a+b) with
  | (false, _, _) -> (false, 0)
  | (true, x~1, y~0) -> (true, x~1 + y~0)
```

Partial Evaluation. Our transformations often lead to an explosion in program size as expressions are inlined and new intermediate expressions are introduced. To mitigate this, we partially evaluate the program and apply some additional simplifications before SMT encoding, normalizing away most of the clutter introduced by language abstractions and transformations. Partial evaluation will evaluate let-expressions with values on the right hand side, but not ones with expressions as we

have found that it often helps the SMT solver to avoid creating large terms. For instance, in this example, the bindings to `a` and `b` were not eliminated (although in this case a subsequent optimization phase over the SMT constraints will eliminate them):

```
let g~0 y~2 y~3 =
  let a = y~2 in
  let b = y~3 in
  (true, 2 + (a+b))
```

5.2.2.2 Translation of expressions

After applying the transformations above, the remaining expressions have a relatively easy translation to the quantifier-free core and linear arithmetic fragment of SMTLIB2, as shown in fig. 5.12. We use red color to denote an SMT expression/value as opposed to an NV one. The tuple flattening and expansion transformation allows us to easily deal with tuples by treating them as lists of expressions to be translated separately in a pointwise fashion. This is the reason that our SMT compilation function computes lists of SMTLIB2 expressions for each NV expression. Constructs such as binary operations and if-then-else expressions are straightforward to translate as SMTLIB2 supports the corresponding expressions. Let-expressions are also straightforward to compile thanks to the fact that our programs have unique binders after the renaming transformation. Match expressions are simple compiled as chains of if-then-else constraints.

Note that there are no functions or function applications as they have been inlined. The only functions remaining in the program are the `init/transfer/merge` functions that are part of solution declarations, and these are treated specially. Their arguments are fixed and are applied before translation begins. For example, for the transfer function over a link $(\mathtt{u}, \mathtt{v})$ we partially evaluate the expression `trans` $(\mathtt{u}, \mathtt{v})$ `lab`$(\mathtt{u})$ (where `lab`$(\mathtt{u})$ is the variable denoting the solution of the node u) and then translate

$$\frac{}{\mathcal{C}_{\mathsf{env}}(\mathsf{x}) = \langle [\mathsf{x}], \mathsf{env}\rangle}\ \textsc{EVar} \qquad \frac{\mathsf{v} \in \{\mathsf{true}, \mathsf{false}, \mathbb{N}\}}{\mathcal{C}_{\mathsf{env}}(\mathsf{v}) = \langle [\mathsf{v}], \mathsf{env}\rangle}\ \textsc{EVal}$$

$$\frac{\mathsf{v} = [\mathsf{v}_1, \ldots, \mathsf{v}_n] \qquad \dfrac{\mathcal{C}_{\mathsf{env}_{i-1}}(\mathsf{v}_i) = \langle [\mathsf{c}_i], \mathsf{env}_i\rangle}{\mathsf{c} = [\mathsf{c}_1, \ldots, \mathsf{c}_n]}}{\mathcal{C}_{\mathsf{env}_0}(\mathsf{v}) = \langle [\mathsf{v}], \mathsf{env}_n\rangle}\ \textsc{EVal-Tuple}$$

$$\frac{\mathsf{op} \in \{\&\&, ||, +, \&<, \leq, =\} \qquad \mathcal{C}_{\mathsf{env}}(\mathsf{e}_1 = \langle [\mathsf{c}_1], \mathsf{env}_1\rangle) \qquad \mathcal{C}_{\mathsf{env}_1}(\mathsf{e}_2 = \langle [\mathsf{c}_2], \mathsf{env}_2\rangle)}{\mathcal{C}_{\mathsf{env}}(\mathsf{v}) = \langle [\mathsf{op}\ \mathsf{c}_1\ \mathsf{c}_2], \mathsf{env}_2\rangle}\ \textsc{EOp}$$

$$\frac{\begin{array}{c}\mathcal{C}_{\mathsf{env}}(\mathsf{e}_1) = \langle [\mathsf{c}_1], \mathsf{env}_1\rangle \\ \mathcal{C}_{\mathsf{env}_1}(\mathsf{e}_2) = \langle [\mathsf{c}_{20}, \ldots, \mathsf{c}_{2n}], \mathsf{env}_2\rangle \qquad \mathcal{C}_{\mathsf{env}_2}(\mathsf{e}_3) = \langle [\mathsf{c}_{30}, \ldots, \mathsf{c}_{3n}], \mathsf{env}_3\rangle\end{array}}{\mathcal{C}_{\mathsf{env}}(\mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ \mathsf{e}_2\ \mathsf{else}\ \mathsf{e}_3) = \langle [\mathsf{ite}\ \mathsf{c}_1\ \mathsf{c}_2\ \mathsf{c}_3, \ldots, \mathsf{ite}\ \mathsf{c}_1\ \mathsf{c}_2'\ \mathsf{c}_3'], \mathsf{env}_3\rangle}\ \textsc{EIte}$$

$$\frac{\begin{array}{c}\mathcal{C}_{\mathsf{env}}(\mathsf{e}_1) = \langle [\mathsf{c}_1], \mathsf{env}_1\rangle \\ \mathcal{C}_{\mathsf{env}_1}(\mathsf{e}_2) = \langle [\mathsf{c}_{20}, \ldots, \mathsf{c}_{2n}], \mathsf{env}_2\rangle \qquad \mathsf{env}_3 = \mathsf{env}_2 \uplus (\mathsf{x}, \mathsf{ty\_to\_sort}\ \mathsf{ty}) \uplus (= \mathsf{x}\ \mathsf{c}_1)\end{array}}{\mathcal{C}_{\mathsf{env}}(\mathsf{let}\ \mathsf{x} : \mathsf{ty} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2) = \langle [c_{20}, \ldots, c_{2n}], \mathsf{env}_3\rangle}\ \textsc{ELet}$$

$$\frac{\begin{array}{c}\mathcal{C}_{\mathsf{env}}(\mathsf{e}) = \langle [\mathsf{g}_0, \ldots, \mathsf{g}_n], \mathsf{env}_1\rangle \\ \mathcal{C}_{\mathsf{env}_1}(\mathsf{e}_1) = \langle [\mathsf{c}_{10}, \ldots, \mathsf{c}_{1n}], \mathsf{env}_2\rangle \\ \mathcal{B}_{\mathsf{env}_2}(\mathsf{p}_1, [g_0, \ldots, g_n]) = \langle [\mathsf{p}_{20}, \ldots, \mathsf{p}_{2n}], \mathsf{env}_3\rangle \\ \vdots \\ \mathcal{C}_{\mathsf{env}_{2m-1}}(\mathsf{e}_m) = \langle [\mathsf{c}_{m0}, \ldots, \mathsf{c}_{mn}], \mathsf{env}_{2m}\rangle \\ \mathcal{B}_{\mathsf{env}_{2m}}(\mathsf{p}_m, [g_0, \ldots, g_n]) = \langle [\mathsf{p}_{m0}, \ldots, \mathsf{p}_{mn}], \mathsf{env}_{2m+1}\rangle \\ \mathsf{r}_i = \mathsf{ite}\ \mathsf{p}_{1i}\ \mathsf{c}_{1i}\ (\mathsf{ite}\ \mathsf{p}_{2i}\ \mathsf{c}_{2i} \ldots \mathsf{c}_{mi})\end{array}}{\mathcal{C}_{\mathsf{env}}(\mathsf{match}\ \mathsf{e}\ \mathsf{with} \mid \mathsf{p}_1 \to \mathsf{e}_1 \ldots \mid \mathsf{p}_m \to \mathsf{e}_m) = \langle [r_1, \ldots, r_n], \mathsf{env}_{2m+1}\rangle}\ \textsc{EMatch}$$

$$\frac{}{\mathcal{B}_{\mathsf{env}}(\_, [\mathsf{g}]) = \langle [\mathsf{true}], \mathsf{env}\rangle}\ \textsc{PWild}$$

$$\frac{\mathsf{env}_2 = \mathsf{env} \uplus (\mathsf{x}, \mathsf{ty\_to\_sort}\ \mathsf{ty}) \uplus (= \mathsf{x}\ \mathsf{g})}{\mathcal{B}_{\mathsf{env}}(\mathsf{x}\ :\ \mathsf{ty}, [\mathsf{g}]) = \langle [\mathsf{true}], \mathsf{env}_2\rangle}\ \textsc{PVar}$$

$$\frac{\mathsf{v} \in \{\mathbb{N}, \mathsf{true}, \mathsf{false}\}}{\mathcal{B}_{\mathsf{env}}(\mathsf{v}, [\mathsf{g}]) = \langle [=\ \mathsf{g}\ \mathsf{v}], \mathsf{env}\rangle}\ \textsc{PConst}$$

$$\frac{\mathcal{B}_{\mathsf{env}_{i-1}}(\mathsf{p}_i, g_i) = \langle [\mathsf{c}_i], \mathsf{env}_i\rangle \qquad \mathsf{c} = \bigwedge_{i=1\ldots n} \mathsf{c}_i}{\mathcal{B}_{\mathsf{env}_0}((\mathsf{p}_1, \ldots, \mathsf{p}_n), [\mathsf{g}_1, \ldots, \mathsf{g}_n]) = \langle [\mathsf{c}], \mathsf{env}_n\rangle}\ \textsc{PTuple}$$

Figure 5.12: The function $\mathcal{C}$ takes an environment and an NV expression and produces the constraints and SMTLIB2 expressions that capture its semantics. $\mathcal{B}$ takes an NV pattern (used in match expressions) and a list of constraints that correspond to the guard of the match expression and implements the pattern-matching logic.

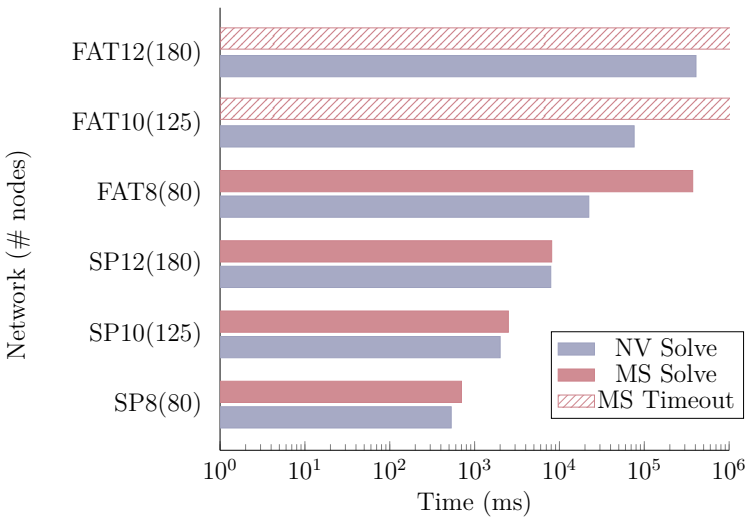


Figure 5.13: SMT time to solve the constraints for NV and MineSweeper (MS). MineSweeper timeouts after 30 minutes for FAT10 and FAT12.

it to SMT. Constraints are generated to combine the resulting SMT expression with the messages from the rest of v's neighbors and to characterize its stable solution as per usual, *i.e.*, the solution of a node is equal to merging the incoming messages and the initial route of the node.

5.2.3 Evaluation

SMT queries compiled from high-level, general purpose languages often lack the performance of hand-tuned queries, as layers of abstractions complicate the final result. We show that in NV we can sidestep this issue thanks to its restricted scope and a careful representation of the language in SMT.

We compare our SMT-based analysis with MineSweeper, the state-of-the-art SMT verifier for control planes. For the experiment we use six FatTree networks running the routing policies described in section 5.1.5.1, and ranging in size from 80 to 180 routers (nodes). MineSweeper and NV solve slightly different problems; Minesweeper asserts facts about a data packet in the network, which requires modeling the part of the control plane that affects this data packet. On the other hand, NV currently

only models the control plane and does not consider data packets. However, since there are no data plane access control lists (ACLs) in the examples, the encodings are similar [8]. Each leaf in the fat tree announces a destination prefix; we pick a node at random (called the *destination*). For MineSweeper, we assert that a data packet sent from any node in the network, with a concrete destination IP that matches the prefix of our destination, reaches the destination. Similarly, in NV, we assert that every node has a route to the prefix announced by the destination node.

Discussion. Figure 5.13 compares the SMT time for the constraints generated by MineSweeper and NV. Our goal is not to compare the absolute verification time, but rather to observe differences in performance trends.

Networks based on shortest-path routing have similar verification time and scaling pattern. The `FAT` networks, which route based on more complex policy, provide more interesting data. For NV, SMT time is 40-50x slower compared to the `SP` networks. On the other hand, MineSweeper only verified the smaller 80 node network, with a slowdown of more than 500x. It is difficult to pinpoint the cause of this difference between the two; however, it is likely that some of the optimizations MineSweeper performs do not kick in when dealing with more complex policy. For instance, MineSweeper also performs some forms of partial evaluation. However, unlike NV, MineSweeper reduction rules are rather ad-hoc, as they are defined over a language that was designed for neither partial-evaluation nor translation to constraints.

Unsurprisingly, MineSweeper computes the SMT encoding faster than NV (not shown in fig. 5.13). This is because MineSweeper builds on top of the original structure of the problem, while NV requires many transformations to reduce the abstractions introduced. Despite that, MineSweeper is still slower in overall verification time, and fails to verify larger networks with more complex routing policy. The bottleneck remains the constraint solver; the encoding procedure can be further optimized, and

even parallelized, but it is not obvious how to improve the performance of the SMT solver.

An important point to be made, is that while the results of fig. 5.13 show NV can generate efficient constraints, in the end, the user's formulation is crucial to the performance of verification. For instance, our initial attempt at expressing the merge function for BGP compared the route attributes and directly returned the best route:

```
let merge u x y =
  match x,y with
  | _, None -> x
  | None, _ -> y
  | Some b1, Some b2 ->
    if b1.lp > b2.lp then x
    else if b2.lp > b1.lp then y
    else if b1.len < b2.len then x
    else if b2.len < b1.len then y
  ...
```

This created unnecessarily many intermediate variables when compiled to an SMT query, as each branch returned a route and created hence multiple SMT variables and constraints to relate them. Instead, rewriting the merge function to return a boolean indicating which of the routes is preferred, significantly reduced the complexity of the encoding and resulted in orders-of-magnitude faster solving time:

```
let isBetter x y =
  match x,y with
  | _, None -> true
  | None, _ -> false
  | Some b1, Some b2 ->
  | Some b1, Some b2 ->
    if b1.lp > b2.lp then true
    else if b2.lp > b1.lp then false
    else if b1.len < b2.len then true
    else if b2.len < b1.len then false
    ...

let merge u x y = if isBetter x y then x else y
```

Chapter 6

Reasoning About Fault Tolerance

This chapter pertains to reasoning about *hardware failures*, such as node or link failures, discussing the impact of failures on a network (section 6.1) and how to analyze the network's behavior in the presence of failures. In particular, section 6.2 discusses how to exploit symmetries using *network compression*, as a means to scaling verification of fault-tolerance properties to large networks. Section 6.3 builds on the idea of exploiting symmetries to analyze the impact of failures in a network, through a different (orthogonal) approach to network compression. Finally, in section 6.4 we discuss related work and other approaches to scaling fault-tolerance verification.

6.1 The Impact of Failures

Hardware failures in networks are not a rare phenomenon; commodity switches in datacenters show a failure rate[1] between 5% and 10% [27]. To avoid severe service disruptions, operators add *redundancy* to their network designs, by adding more devices and links, and thus creating backup paths. This approach while obviously effective, does not magically solve all problems. Adding more devices to the network

[1]Defined as the number of commodity switches that observed a failure divided by the number of devices of that type during a one year period.

increases the configuration burden; these devices must be configured correctly to take over (extra) traffic when necessary due to other device failures. One of the reasons that hardware failures can cause havoc in a network is that they trigger *latent bugs*, bringing down seemingly correct networks. In other words, a network might have been operating as intended until a specific combination of failures triggered a previously unreachable piece of buggy configuration. For instance, when a router in a Microsoft Azure data center shut down, previously unknown configuration errors on other devices surfaced, disrupting traffic to West Europe for over 2 hours [56].

The bottom line is that hardware failures significantly complicate the task of reasoning about a network's behavior. Both humans (*e.g.*, the network operators) and computers (*e.g.*, a network simulator) have trouble dealing with the immense number of failure scenarios that have to be considered. For example, even constraining the possible failures space to a single link failure, a small Fat Tree network with 20 nodes and 64 links[2] gives rise to 64 "different" networks. If we consider two potential link failures, then the number of combinations explode to 4032, already too many for a human to go through manually.

6.2 Scaling Fault Tolerance Verification via Network Compression

In theory, failures lead to a combinatorial explosion in the number of possible network behaviors. However, despite the high number of failure scenarios, in practice most real networks have highly *symmetric* designs, making many of these scenarios irrelevant, in the sense that they do not affect the network's routing behavior. For instance, consider the Fat Tree of section 6.2.1; it is easy to see that, barring any misconfigurations, every red node (ToR) can reach the destination under a single

[2]Note that links are unidirectional.

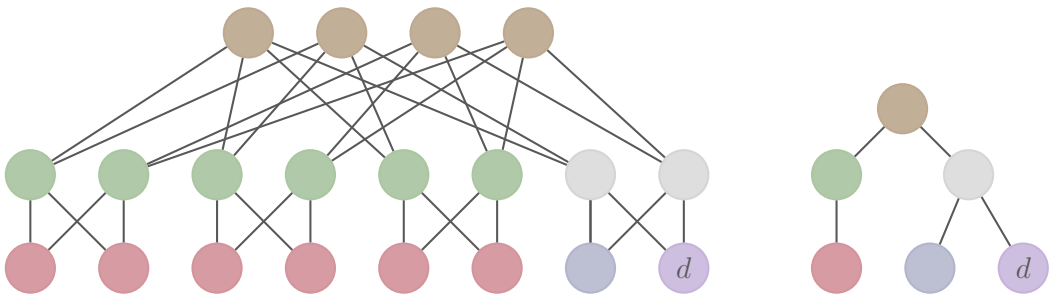


Figure 6.1: A FatTree network (on the left) and the compressed network Bonsai computes.

link failure, because they have two symmetric and disjoint paths to the destination. Of course, when considering larger networks, more complicated failure scenarios and how they interact with policy, this kind of reasoning is impossible for a human to do. Nonetheless, even if the network's topology and policy are not perfectly symmetric, there must be some symmetries; to enable scalable analyses we must be able to discover and exploit them. Indeed, prior work [9, 45] has shown that exploiting symmetries can improve verification performance by orders of magnitude. However, past work did not account for failures, as by their nature, they tend to break these symmetries.

In this section we present, Origami, a tool to compress networks given a bounded number of link failures (section 6.2.7), and an associated network approximation theory (section 6.2.4) we used to prove the soundness of Origami.

6.2.1 Background on Network Compression

Beckett *et al.* [9], introduced the idea of *control plane compression*, defining a theory of *control plane equivalence*, and using it to prove the correctness of a network compression algorithm. Given a network (referred to as the concrete network), the algorithm produces a smaller one (referred to as the abstract network) with equivalent behavior. Informally, their notion of equivalence says that every concrete node

routes traffic "in the same direction" as their abstract counterpart. This notion of equivalence, preserves properties such as reachability, path length, and loop freedom. To reason about these properties, analysis tools (quickly) operate over the smaller, compressed network, rather than the original large network. To identify nodes with similar behavior, Bonsai's compression algorithm relies on a set of *local* conditions, called *effective abstraction conditions*; these conditions can be checked efficiently without actually computing the solutions of the network, and when true, the original and compressed networks are guaranteed to be equivalent.

Bonsai is very capable of compressing large networks —it can shrink a Fat Tree running variants of shortest-path routing and consisting of several thousand nodes to a Fat Tree with 6 nodes— however, as its compression algorithm does not preserve information such as the number of neighbors, links, or paths, the abstract network is unsuitable for reasoning about fault tolerance properties of the concrete network. Figure 6.2 illustrates some of the issues that arise with Bonsai's compression when there can be link failures. First, notice that in the concrete network (fig. 6.2a) every node has at least two disjoint paths to the destination, whereas their corresponding abstraction only has a single path to the destination (fig. 6.2b), and thus, fault-tolerance properties are obviously not preserved. Moreover, notice in fig. 6.2c the impact of a link failure on the routing behavior of the concrete network: nodes b_1 and b_2, b_3 no longer "route in the same direction", violating the invariant connecting the concrete and abstract networks.

6.2.2 Key Ideas

To achieve compression in the presence of failures, we need to rethink how compression works. Addressing the problems described above, our approach revolves around two key ideas.

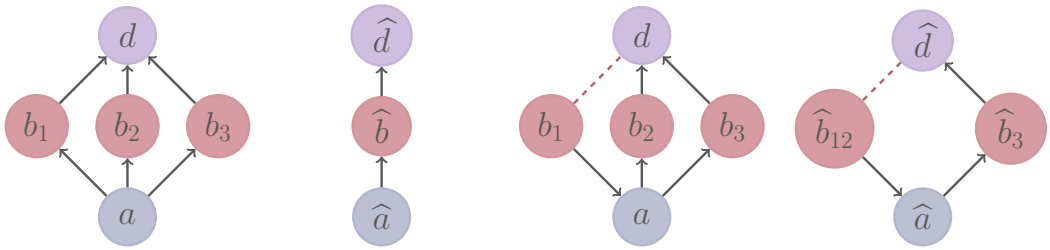


Figure 6.2: All graph edges shown correspond to edges in the network topology, and we draw edges as directed to denote the direction of forwarding eventually determined for each node by the distributed routing protocols for a fixed destination d. In (a) nodes use shortest path routing to route to the destination d. (b) shows the compressed network that Bonsai computes; this abstraction precisely captures the forwarding behavior of the concrete network. Figure (c) shows how forwarding is impacted by a link failure (shown as a red, dashed edge). Finally, (d) shows a sound approximation of the original network for any single link failure.

Lossy Compression. To achieve compression given a bounded number of link failures, we *relax the notion of similarity* between concrete and abstract nodes: A node in the abstract network may merely *approximate* the behavior of concrete nodes. This makes it possible to compress together nodes that, in the presence of failures, may route differently. In general, when we fail a single link in the abstract network, we are *over-approximating* the failures in the concrete network by failing multiple concrete links, possibly more than desired. Nevertheless, the paths taken in the concrete network can only deviate so much from the paths found in the abstract network:

Property 1. *If a node has a route to the destination in the presence of k link failures then it has a route that is "at least as good" (as prescribed by the routing protocol) in the presence of k' link failures for k' < k.*

This relation suffices to verify important network reliability properties, such as reachability, in the presence of faults. Just as importantly, it allows us to achieve effective network compression to scale verification. Revisiting our example, consider the new abstract network of fig. 6.2d. When the link between $\widehat{b}_{12}$ and d has failed,

$\widehat{b}_{12}$ still captures the behavior of b_1 precisely. However, b_2 has a better (in this case better means shorter) path to d. Despite this difference, if the operator's goal was to prove reachability to the destination under any single fault, then this abstract network suffices.

Computing Good Abstractions. It is not too difficult to find abstract networks that approximate a concrete network; the challenge is finding a valid abstract network that is *small enough* to make verification feasible and yet *large enough* to include sufficiently many paths to verify the given fault tolerance property. Rather than attempting to compute a single abstract network with the right properties all in one shot, we search the space of abstract networks using an algorithm based on *counter-example guided abstraction refinement* [15].

The CEGAR algorithm begins by computing the smallest possible valid abstract network. In the example above, this corresponds to the original compressed network in Figure 6.2b, which faithfully approximates the original network when there are no link failures. However, if we try to verify reachability in the presence of a single fault, we will conclude that nodes $\widehat{b}$ and $\widehat{a}$ have no route to the destination when the link between $\widehat{b}$ and $\widehat{d}$ fails. The counterexample due to this failure could of course be spurious (and indeed it is). Fortunately, we can easily distinguish whether such a failure is due to lack of connectivity or an artifact of over-abstracting, by calculating the number of corresponding concrete failures. In this example a failure on the link $\langle \widehat{b}, \widehat{d} \rangle$ corresponds to 3 concrete failures. Since we are interested in verifying reachability for a single failure this cannot constitute an actual counterexample.

The next step is to *refine* our abstraction by splitting some of the abstract nodes. The idea is to use the counterexample from the previous iteration to split the abstract network in a way that avoids giving rise to the same spurious counterexample in the next iteration. Doing so results in the somewhat larger network of Figure 6.2d. A

second verification pass over this larger network might take longer, but will succeed. Section 6.2.7 discusses the ideas our compression algorithm leverages in order to quickly compute a suitable abstraction.

6.2.3 Stable Paths with Failures

We adapt the stable paths semantics defined in earlier sections (see fig. 2.2), to account for link failures, defining a class of SPP instances, called *stable paths problems with faults* (SPPFs). SPPFs differ from the traditional definition of the stable paths problem that we used throughout the thesis, in a few ways:

1. the type of routing messages are assumed to include a special value denoting the absence of a route. We write $A_\infty = A \cup \{\infty\}$, where ∞ means no route.

2. SPPFs admit the possibility of a bounded number of link failures.

3. A ranking function $\prec \subseteq A_\infty \times A_\infty$ is used to merge two messages. We write $a \prec b$ to mean that route a is preferred over route b. Any route is preferred to ∞.

4. There is a single destination node $d \in V$ announcing an initial route $a_d \in A$.

The routing protocols mentioned in chapter 2, like eBGP and OSPF, can be formulated as an SPPF instance.

SPPF Solutions. To define the solutions of an SPPF instance, we adapt the definitions of section 2.3 to this current setting. More specifically, we define a *solution* $\mathcal{S}$ of an SPPF to be a pair $\langle \mathcal{L}, \mathcal{F} \rangle$ of a node labelling $\mathcal{L}$ and a failure scenario $\mathcal{F}$. Like before, the labelling $\mathcal{L}$ represents the chosen route for each node. The failure scenario $\mathcal{F}$ is an assignment of 0 (has not failed) or 1 (has failed) to each edge in the network.

The definition of *choices* of node u is adjusted[3] to take into account the possibility of link failures; a valid choice is available only through a link that is not failed:

$$\mathsf{choices}(u) = \{(e, a) \mid e = \langle u, v\rangle,\ a = \mathsf{trans}(e, \mathcal{L}(v)),\ \mathcal{F}(e) = 0\}$$

A solution $\mathcal{S} = \langle \mathcal{L}, F\rangle$ to an $\mathsf{SPPF} = (G, A, a_{\mathrm{d}}, \prec, \mathsf{trans}, k)$ is a stable state satisfying the usual stability conditions:

$$\mathcal{L}(u) = \begin{cases} a_{\mathrm{d}} & u = d \\ \infty & \mathsf{choices}(u) = \emptyset \\ min_{\prec}(\{a \mid (e, a) \in \mathsf{choices}(u)\}) & \mathsf{choices}(u) \neq \emptyset \end{cases}$$

$$\textbf{subject to} \ \sum_{e \in E} \mathcal{F}(e) \leq k$$

Additionally, failure scenarios are constrained so that the sum of the failures is at most k.

6.2.4 A Theory of Network Approximation

Given a concrete SPPF and an abstract $\widehat{\mathsf{SPPF}}$, a network abstraction is a pair of functions (f, h) that relate the two. The topology abstraction $f : V \to \widehat{V}$ maps each node in the concrete network to a node in the abstract network, while the attribute abstraction $h : A_\infty \to \widehat{A}_\infty$ maps a concrete attribute to an abstract attribute. The latter allows us to relate networks running protocols where nodes may appear in the attributes (*e.g.*, in the AS-path component of BGP).

The goal of Origami is to compute compact $\widehat{\mathsf{SPPFs}}$ that may be used for verification. These compact $\widehat{\mathsf{SPPFs}}$ must be closely related to their concrete counterparts.

[3]Note that unlike the previous chapters, in this section, we assume that routing messages over a link $\langle u, v\rangle$ travel "backwards" from v to u. This is only a matter of convention; originally in Origami we assumed this convention, and we decided to stick to it here, to stay consistent with already published manuscripts.

Otherwise, properties verified on the compact $\widehat{\mathsf{SPPF}}$ will not be true of their concrete counterpart. We define such a relation, called *label approximation*, which provides an intuitive, high-level, semantic relationship between abstract and concrete networks. We also explain some of the consequences of this definition and its limitations. Section 6.2.5 continues our development by explaining two *well-formedness* requirements of network policies that play a key role in establishing label approximation. Unfortunately, while this broad definition serves as an important theoretical objective, it is difficult to use directly in an efficient algorithm. Section 6.2.6 defines *effective SPPF approximation* for well-formed SPPFs. This definition is more conservative than label approximation, but has the advantage that it is easier to work with algorithmically and, moreover, it implies label approximation.

Label Approximation. Intuitively, we say the abstract $\widehat{\mathsf{SPPF}}$ label-approximates the concrete SPPF when SPPF has at least as good a route at every node as $\widehat{\mathsf{SPPF}}$ does.

Definition 1 (Label Approximation)**.** *Consider any solutions $\mathcal{S}$ to SPPF and $\widehat{\mathcal{S}}$ to $\widehat{\mathsf{SPPF}}$ and their respective labelling components $\mathcal{L}$ and $\widehat{\mathcal{L}}$. We say $\widehat{\mathsf{SPPF}}$ label-approximates SPPF when $\forall u \in V.\ h(\mathcal{L}(u)) \preceq \widehat{\mathcal{L}}(f(u))$*

If we can establish a label approximation relation between a concrete and an abstract network, we can typically verify a number of properties of the abstract network and be sure they hold of the concrete network. However, the details of exactly which properties we can verify depend on the specifics of the preference relation ($\prec$). For example, in an OSPF network, preference is determined by weighted path length. Therefore, if we know an abstract node has a path of weighted length n, we know that its concrete counterparts have paths of weighted length of at most n. More importantly, since "no route" is the worst route, we know that if a node has any route to the destination in the abstract network, so do its concrete counterparts.

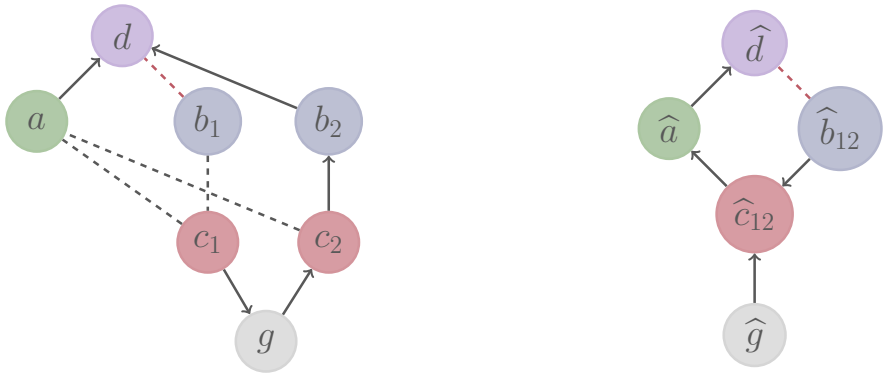


Figure 6.3: Concrete network (left) and its corresponding abstraction (right). Nodes c_1, c_2 prefer to route through b_1 (resp. b_2), or g over a. Node b_1 (resp. b_2) drops routing messages that have traversed b_2 (resp. b_1). Red lines indicate a failed link. Dashed lines indicate a topologically available but unused link.

Limitations. Some properties are beyond the scope of Origami (independent of the preference relation). For example, our model cannot reason about quantitative properties such as bandwidth, probability of congestion, or latency.

6.2.5 Well-formed SPPFs

Not all SPPFs are well-behaved. For example, some never converge and others do not provide sensible models of any real network. To avoid dealing with such poorly-behaved models, we demand henceforth that all SPPFs are *well-formed*. Well-formedness entails that an SPPF is strictly monotonic and isotonic:

$$\forall a, e.\ \ a \neq \infty \Rightarrow a \prec \mathsf{trans}(e, a) \qquad \textit{strict monotonicity}$$

$$\forall a, b, e.\ \ a \preceq b \ \Rightarrow\ \mathsf{trans}(e, a) \preceq \mathsf{trans}(e, b) \qquad \textit{isotonicity}$$

Monotonicity and isotonicity properties are often cited [16, 51, 37] as desirable properties of routing policies because —among other things— they guarantee network convergence and prevent persistent oscillation. In practice too, prior studies have revealed that almost all real network configurations have these properties [24, 37].

In our case, these properties help establish additional invariants that tie the routing behavior of concrete and abstract networks together. To gain some intuition as to why, consider the networks of Figure 6.3. The concrete network on the left runs BGP

with the routing policy that nodes c_1 and c_2 prefer to route through node g instead of a, and that b_1 (resp. b_2) drop announcements coming from b_2 (resp. b_1) — this can be implemented in BGP using a community tag. In this scenario, the similarly configured abstract node $\widehat{b}_{12}$ can reach the destination—it simply takes a route that happens to be less preferred by $\widehat{c}_{12}$ than it would if there had been no failure. However, in the concrete analogue, b_1, is *unable* to reach the destination because c_1 only sends it the route through b_2, which it cannot use. In this case, the concrete network has more topological paths than the abstract network, but, counterintuitively, due to the network's routing policy, this turns out to be a disadvantage. Hence having more paths does not necessarily make nodes more accessible. As a consequence, in general, abstract networks cannot soundly overapproximate the number of failures in a concrete network—an important property for the soundness of our theory.

The underlying issue here is that the networks of Figure 6.3 are not isotonic: suppose $\mathcal{L}'(c_1)$ is the route from c_1 to the destination through node a, we have that $\mathcal{L}(c_1) \prec \mathcal{L}'(c_1)$ but since the transfer function over $\langle b_1, c_1 \rangle$ drops routes that have traversed node b_2, we have that $\mathsf{trans}(\langle b_1, c_1 \rangle, \mathcal{L}(c_1)) \not\prec \mathsf{trans}(\langle b_1, c_1 \rangle, \mathcal{L}'(c_1))$. Notice that $\mathcal{L}'(c_1)$ is essentially the route that the abstract network uses *i.e.* $h(\mathcal{L}'(c_1)) = \widehat{\mathcal{L}}(\widehat{c}_{12})$, hence the formula above implies that $h(\mathcal{L}(b_1)) \not\prec \widehat{\mathcal{L}}(\widehat{b}_{12})$ which violates the notion of label approximation.

A similar problem arises when the routing policy is not monotonic. Consider the topology of Figure 6.4 running eBGP with a routing policy that dictates that node c prefers to route through nodes x_1, x_2 over b and nodes x_1, x_2 prefer to route through c. The abstract node $\widehat{x}_{12}$ has a (unique) choice to route through $\widehat{c}$. On the other hand, depending on how the concrete network convergences, x_2 may route directly to d and c may route through x_2. In this scenario x_2 ends up having a route to the destination that is worse than the one of its abstract counterpart $\widehat{x}_{12}$.

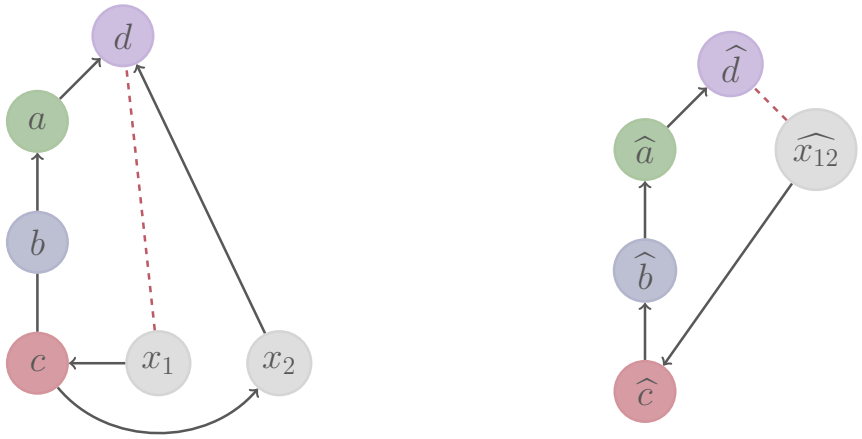


Figure 6.4: Concrete network (left) and its corresponding abstraction (right). Node c prefers to route through x_1, x_2 and nodes x_1, x_2 prefer to route through c.

Fortunately, if a network is strictly monotonic and isotonic, such situations never arise. Moreover, often, these properties can be checked via an SMT solver using a local and efficient test.

6.2.6 Effective SPPF approximation

We seek abstract networks that label-approximate given concrete networks. Unfortunately, to directly check that a particular abstract network label approximates a concrete network one must effectively compute their solutions. Doing so would defeat the entire purpose of abstraction, which seeks to analyze large concrete networks *without the expense of computing their solutions directly.*

In order to turn approximation into a useful computational tool, we define *effective approximation*, a set of simple conditions on the abstraction functions f and h that are *local* and can be checked efficiently. When true those conditions imply label approximation. Intuitively effective approximations impose three main restrictions on the abstraction functions:

1. The topology abstraction conforms to the $\forall\exists-\mathsf{abstraction}$ condition; this requires that there is an abstract edge $(\widehat{u},\widehat{v})$ iff for every concrete node u such that $f(u)=\widehat{u}$ there is some node v such that $f(v)=\widehat{v}$ and $(u,v)\in E$.

2. The abstraction preserves the rank of attributes *(rank-equivalence)*:

$$\forall a,b.\ a \prec b \iff h(a) \mathrel{\widehat{\prec}} h(b)$$

3. The transfer function and the abstraction functions commute *(trans-equivalence)*:

$$\forall e,a.\ h(\mathsf{trans}(e,a)) = \widehat{\mathsf{trans}}(f(e),h(a))$$

Relating Abstract and Concrete Choices. To relate the solutions of the two networks, we must relate the routing choices between abstract and concrete nodes. The examples of section 6.2.5 demonstrate that, in general, the solutions of the concrete network and the solutions of its effective abstractions are not necessarily label-approximate. At the same time, they provided us with useful insights about the class of networks for which effective abstractions imply label-approximate. For the purpose of relating the routing choices of the two networks, we assume that if a protocol uses the path traversed to implement loop prevention (*e.g.*, as does BGP), then this is exposed in a node's solution as $\mathcal{L}(u)$.path. If it's not used (*e.g.*, as in OSPF), then this component can be ignored.

Definition 2. *Given an SPPF and its abstraction* $\widehat{\mathsf{SPPF}}$ *defined by* (f,h) *we say that the two networks are choice approximate if their solutions satisfy the following property:*

$$\begin{aligned}\forall\langle u,v\rangle\in E, \widehat{a}\in\widehat{A}.\ (f(\langle u,v\rangle),\widehat{a}) \in min(\mathsf{choices}_{\widehat{\mathcal{S}}}(f(u))) \implies \\ (\exists a.\ h(a) \mathrel{\widehat{\preceq}} \widehat{a} \ \wedge\ (\langle u,v\rangle,a)\in\mathsf{choices}(u)) \vee \\ (u\in\mathcal{L}(v).\text{path})\end{aligned}$$

Our initial intuition is that if an abstract node $f(u)$ has an optimal routing choice through the link $\langle f(u), f(v)\rangle$ then the concrete node u has a routing choice over $\langle u, v\rangle$ that is at least as good. Formally this means that $\mathsf{trans}(\langle u, v\rangle, \mathcal{L}(v)) \preceq \mathsf{trans}(\langle f(u), f(v)\rangle, \widehat{\mathcal{L}}(f(v))$. Thinking inductively, one may wonder: *given that v and $f(v)$ are label-approximate, can we prove the same of u and f(u)?* As we saw in section 6.2.5 the transfer function must be isotonic for this to hold.

The example of Figure 6.4 illustrated the case where using protocols that have a loop detection mechanism, such as BGP, a concrete node may not have a choice that its abstract counterpart had. This is captured in definition 2 by the case where $u \in \mathcal{L}(v)$.path. To establish label-approximate in this case we need to resort to monotonicity. To understand how monotonicity is used in this case, imagine there was no loop-detection (i.e. the route causing a loop is not dropped), then according to monotonicity, the routing choice v would offer to u would be strictly worse, as v's route uses u and every application of the transfer function produces a worse attribute.

We prove that when these conditions hold, we can approximate any solution of the concrete network with a solution of the abstract network.

Theorem 1. *Given a well-formed SPPF and its effective approximation $\widehat{\mathsf{SPPF}}$, for any solution $\mathcal{S} \in$ SPPF there exists a solution $\widehat{\mathcal{S}} \in \widehat{\mathsf{SPPF}}$, such that their labelling functions are label approximate.*

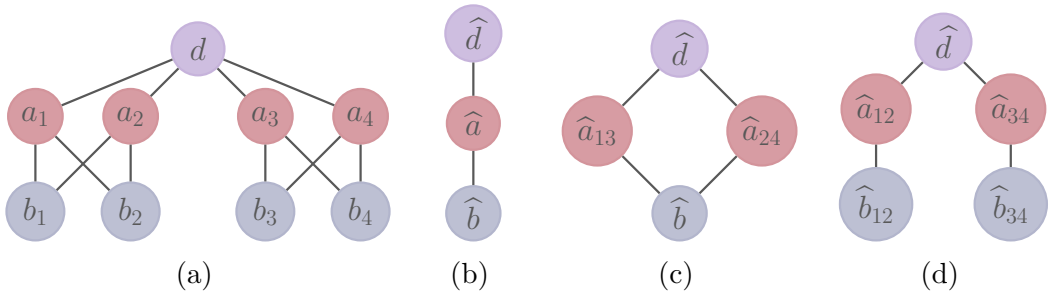
6.2.7 Computing Fault-Tolerance Preserving Approximations

The first step towards verification is to compute a small abstract network that satisfies our SPPF *effective approximation* conditions. We do so by grouping network nodes and edges with equivalent policy and checking the forall-exists topological condition, using an algorithm reminiscent of the one used in Bonsai [9]. Typically, however,

this minimal abstraction will not contain enough paths to prove any fault-tolerance property. To identify a finer abstraction for which we can prove a fault-tolerance property we repeatedly:

1. Search the set of candidate refinements for the smallest *plausible* abstraction.
2. Check if the candidate abstraction satisfies the desired property, and if so terminate the procedure. We have successfully verified our concrete network.
3. If not, examine whether the returned counterexample is an actual counterexample. We do so, by computing the number of concrete failures and check that it does not exceed the desired bound of link failures. (If so, we have found a property violation.)
4. If not, use the counterexample to *learn* how to expand the abstract network into a larger abstraction and repeat.

Implementation. We implemented Origami, as a backend to NV. Describing instances of SPPFs in NV is straightforward. The topology and routing semantics can be defined as per usual, *e.g.* one can use the per-destination translation of section 4.1. To account for the possibility of link failures, we modify a given instance of SPP (*i.e.*, an NV program), by adding boolean symbolic values, one for each edge, denoting whether this edge is failed or not. The transfer function is modified to drop the route if the link is failed. Finally, a requires clause is added bounding the number of link failures to the user-provided bound k. To check whether the given property holds, we use NV's SMT-based verifier (described in chapter 5). Both the search for plausible candidates and the way we learn a new abstraction to continue the counterexample-guided loop are explained below.



(a) (b) (c) (d)

Figure 6.5: Eight nodes in (a) are represented using two nodes in the abstract network (b). Pictures (c) and (d) show two possible refinements of the abstract network (b).

6.2.7.1 Searching for plausible candidates

Though we might know an abstraction is not sufficient to verify a given fault tolerance property, there are many possible refinements. Consider, for example, Figure 6.5a presents a simple concrete network that will tolerate a single link failure, and Figure 6.5b presents an initial abstraction. The initial abstraction will not tolerate any link failure, so we must refine the network. To do so, we choose an abstract node to divide into two abstract nodes for the next iteration. We must also decide which concrete nodes correspond to each abstract node. For example, in Figure 6.5c, node $\widehat{a}$ has been split into $\widehat{a}_{13}$ and $\widehat{a}_{24}$. The subscripts indicate the assignment of concrete nodes to abstract ones.

A significant complication is that once we have generated a new abstraction, we must check that it continues to satisfy the effective approximation conditions, and if not, we must do more work. Figure 6.5c satisfies those conditions, but if we were to split $\widehat{a}$ into $\widehat{a}_{12}$ and $\widehat{a}_{34}$ rather than $\widehat{a}_{13}$ and $\widehat{a}_{24}$, the forall-exists condition would be violated—some of the concrete nodes associated with $\widehat{b}$ are connected to the concrete nodes in $\widehat{a}_{12}$ but not to the ones in $\widehat{a}_{34}$ and vice versa. To repair the violation of the forall-exists condition, we need to split additional nodes. In this case, the $\widehat{b}$ node, giving rise to the network of fig. 6.5d.

Overall, the process of splitting nodes and then recursively splitting further nodes to repair the forall-exists condition generates many possible candidate abstractions to consider. A key question is which candidate should we select to proceed with the abstraction refinement algorithm?

One consideration is size: A smaller abstraction avoids taxing the verifier, which is the ultimate goal. However, there are many small abstractions that we can quickly dismiss:

Definition 3 (Plausible Abstraction)**.** *We say an abstraction is plausible if all nodes of interest have at least* $k+1$ *disjoint (topological) paths to the destination node.*

Obviously, in an implausible abstraction the destination will be unreachable with k failures, as all available paths can be cut-off. We can efficiently check whether an abstraction is plausible, by computing the *min-cut* of the graph. Figure 6.5d is an example of an implausible abstraction that arose after a poorly-chosen split of node $\widehat{a}$. In this case, no node has 2 or more paths to the destination and hence they might not be able to reach the destination when there is a failure.

Clearly verification using an implausible abstraction will fail. Instead of considering such abstractions as candidates for running verification on, the refinement algorithm tries refining them further. A key decision the algorithm needs to make when refining an abstraction is *which abstract node to split*. For instance, the optimal refinement of the network of Figure 6.5b is the network of Figure 6.5c. If we were to split node $\widehat{b}$ instead of $\widehat{a}$ we would end up with a sub-optimal (in terms of size) abstraction. Intuitively, splitting a node that lies on the min-cut and can reach the destination (*e.g.*, $\widehat{a}$) will increase the number of paths that its neighbors on the unreachable part of the min-cut (*e.g.*, $\widehat{b}$) can use to reach the destination.

To summarize, the search for new candidate abstractions involves (1) splitting nodes in the initial abstraction, (2) repairing the abstraction to ensure the forall-exists condition holds, (3) checking that the generated abstraction is *plausible*, and

if not, (4) splitting additional nodes on the min-cut. This iterative process will often generate many candidates. The *breadth* parameter of the search bounds the total number of plausible candidates we will generate in between verification efforts. Of all the plausible candidates generated, we choose the smallest one to verify using the SMT solver.

6.2.7.2 Learning from counterexamples

Any nodes of an abstraction that have a min-cut of less than $k+1$ definitely cannot tolerate k faults. If an abstraction is plausible, it satisfies a *necessary* condition for source-destination connectivity, but not a *sufficient* one—misconfigured routing policy can still cause nodes to be unreachable by modifying and/or subsequently dropping routing messages. For instance, the abstract network of Figure 6.5c is plausible for one failure, but if $\widehat{b}$'s routing policy blocks routes of either $\widehat{a}_{13}$ or $\widehat{a}_{24}$ then the abstract network will not be 1-fault tolerant. Indeed, it is the complexity of routing policy that necessitates a heavy-weight verification procedure in the first place, rather than a simpler graph algorithm alone.

In a plausible abstraction, if the verifier computes a solution to the network that violates the desired fault-tolerance property, some node could not reach the destination because one or more of their paths to the destination could not be used to route traffic. We use the generated counterexample to learn edges that could not be used to route traffic due to the policy on them. To do so, we inspect the computed solution to find nodes $\widehat{u}$ that (1) lack a route to the destination (*i.e.* $\widehat{\mathcal{L}}(\widehat{u}) = \infty$), (2) have a neighbor $\widehat{v}$ that has a valid route to the destination, and (3) the link between $\widehat{u}$ and $\widehat{v}$ is not failed. These conditions imply the absence of a valid route to the destination not because link failures disabled all paths to the destination, but because the network policy dropped some routes. For example, in picture Figure 6.5c, consider the case where $\widehat{b}$ does not advertise routes from $\widehat{a}_{13}$ and $\widehat{a}_{24}$; if the link between $\widehat{a}_{13}$ and $\widehat{d}$

fails, then $\widehat{a}_{13}$ has no route the destination and we learn that the edge $\langle \widehat{b}, \widehat{a}_{13} \rangle$ cannot be used. In fact, since $\widehat{a}_{13}$ and $\widehat{a}_{12}$ belonged to the same abstract group $\widehat{a}$ before we split them, their routing policies are equal modulo the abstraction function by trans-equivalence. Hence, we can infer that in a symmetric scenario, the link $\langle \widehat{b}, \widehat{a}_{24} \rangle$ will also be unusable.

Refining Using Learned Paths. Given a set of unusable edges, learned from a counterexample, we restrict the min-cut problems that define the plausible abstractions, by disallowing the use of those edges. Essentially, we enrich the refinement algorithm's topological based analysis (which is based on min-cut) with knowledge about the policy; the algorithm will have to generate abstractions that are plausible without using those edges. With those edges disabled, the refinement process continues as before.

6.2.7.3 Optimizations.

Optimizing Refinement. To speed-up the refinement process, we implemented some basic optimizations:

- If the min-cut between the destination and a vertex u is less than or equal to the desired number of disjoint paths, then we do not need to compute another min-cut for the nodes in the unreachable portion of vertices T (as computed by min-cut); we know nodes in T can be disconnected from the destination. This significantly reduces the number of min-cut computations.
- We stop exploring abstractions that are larger in size than the smallest plausible abstraction computed since the last invocation of the SMT solver.

Minimizing Counterexamples. When the SMT solver returns a counterexample, it often uses the maximum number of failures. This is not surprising as

maximizing failures simplifies the problem for the SMT solver. Unfortunately, it also confounds our analysis to determine whether a counterexample is real or spurious. For instance, if the failure bound is 2, failing 2 abstract links could result in 4 concrete failures, which exceeds the failure bound, and leads the analysis to believe the counterexample is spurious, causing another round of refinement. However, it could also be the case that there exists another failure scenario that results in just 2 concrete failures. This latter scenario might even use just a subset of the links from the initial scenario. If we do not detect such issues, they will slow our analysis by performing unnecessary iterations of the refinement loop.

To mitigate the effect of this problem, we *could* ask the solver to minimize the returned counterexample, returning a counterexample that corresponds to the fewest concrete link failures. We could do so by providing the solver with additional constraints specifying the number of concrete links that correspond to each abstract link and then asking the solver to return a counterexample that minimizes this sum of concrete failures.

Of course, doing so requires we solve a more expensive optimization problem. Instead, given an initial (possibly spurious counter-example), we simple ask the solver to find a new counterexample that (additionally) satisfies the constraint on the number of concrete failures. If it succeeds, we have found a real counterexample. If it fails, we fall back to refining our abstraction.

Optimizing the SMT Representation. Finally, we found that using Z3's support for pseudo-boolean functions to encode the bound on the number of link failures significantly improved verification time. As NV does not offer an expression that corresponds to pseudo-booleans constraints, we added special support in the SMT backend for Origami.

Topo	V/E	Fail	$\widehat{\mathbf{V}}/\widehat{\mathbf{E}}$	Ratio	Abs	SMT Calls	SMT Time
FT20	500/8000	1	9/20	55.5/400	0.1	1	0.1
		3	40/192	12.5/41.67	1.0	2	7.6
		5	96/720	5.20/11.1	2.5	2	248
		10	59/440	8.48/18.18	0.9	-	-
FT40	2000/64000	1	12/28	166.7/2285.7	0.1	1	0.1
		3	45/220	44.4/290.9	33	2	12.3
		5	109/880	18.34/72.72	762.3	2	184.1
SP40	2000/64000	1	13/32	153.8/2000	0.2	1	0.1
		3	39/176	51.3/363.6	30.3	1	2
		5	79/522	25.3/122.6	372.2	1	22
FbFT	744/10880	1	20/66	37.2/164.8	0.1	3	1
		3	57/360	13.05/30.22	1	4	18.3
		5	93/684	8/15.9	408.9	-	-

Figure 6.6: Compression results. **Topo**: the network topology. **V/E**: Number of nodes/edges of concrete network. **Fail**: Number of failures. $\widehat{\mathbf{V}}/\widehat{\mathbf{E}}$: Number of nodes/edges of the best abstraction. **Ratio**: Compression ratio (nodes/edges). **Abs**: Time taken to find abstractions (sec.). **SMT Calls**: Number of calls to the SMT solver. **SMT Time**: Time taken by the SMT solver (sec.).

6.2.8 Evaluation

We evaluate Origami on a collection of synthetic data center networks that are using BGP to implement shortest-paths routing policies over common industrial datacenter topologies. Data centers are good fit for our algorithms as they can be very large but are highly symmetrical and designed for fault tolerance. Data center topologies are typically organized in layers, with each layer containing many routers. Each router in a layer is connected to a number of routers in the layer above (and below) it. The precise number of neighbors to which a router is connected, and the pattern of said connections, is part of the topology definition. We focus on two common topologies: fat tree topologies (labelled FT20, FT40 and SP40 below) and a different fat tree used at Facebook (labelled FbFT). These are relatively large data center topologies ranging from 500 to 2000 nodes and 8000 to 64000 edges.

SP40 uses a pure shortest paths routing policy. For other experiments (FT20, FT40, FbFT), we augment shortest paths with additional policy that selectively drops routing announcements, for example disabling "valley routing" in various places which allows up-down-up-down routes through the data centers instead of just up-

down routes. The pure shortest paths policy represents a best-case scenario for our technology as it gives rise to perfect symmetry (in terms of policy) and makes our heuristics especially effective.

Experiments were done on a Mac with a 4GHz i7 CPU and 16GB of memory.

6.2.8.1 Compression results

Figure 6.6 shows the level of compression achieved, along with the required time for compression and verification. In most cases, we achieve a high compression ratio especially in terms of links. This drastically reduces the possible failure combinations for the underlying verification process. The cases of 10 link failures on FT20 and 5 link failures on FbFT demonstrate another aspect of our algorithm. Both topologies cannot sustain that many link failures, *i.e.* some concrete nodes have less than 10 (resp. 5) neighbors. We can determine this as we refine the abstraction; there are (abstract) nodes that do not satisfy the min-cut requirement and we cannot refine them further. This constitutes an actual counterexample and explains why the abstraction of FT20 for 10 link failures is smaller than the one for 5 link failures. Importantly, we did not use the SMT solver to find this counterexample. Likewise, we did not need to run a min-cut on the much larger concrete topology. Intuitively, the rest of the network remained abstract, while the part that led to the counterexample became fully concrete.

6.2.8.2 Verification performance

The verification time of Origami is dominated by abstraction time and SMT time, which can be seen in fig. 6.6. In practice, there is also some time taken to parse and pre-process the configurations but it is negligible. The abstraction time is highly dependent on the size of the network and the abstraction search breadth used. In this case, the breadth was set to 25, a relatively high value.

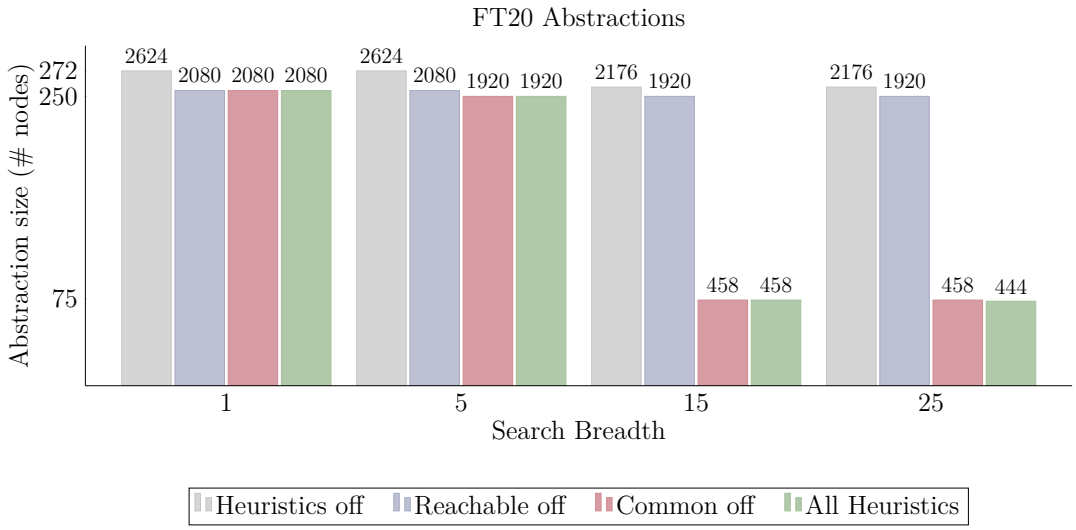


Figure 6.7: The initial abstraction of FT20 for 5 link failures using different heuristics and search breadth. On top of the bars is the number of edges of each abstraction.

While the verification time for a high number of link failures is not negligible, we found that verification without abstraction is essentially impossible. We used Minesweeper[8], the state-of-the-art SMT-based network verifier, to verify the same fault tolerance properties and it was unable to solve any of our queries. This is not surprising, as SMT-based verifiers do not scale to networks beyond the size of FT20 even without any link failures.

6.2.8.3 Refinement effectiveness

We now evaluate the effectiveness of our search and refinement techniques.

Effectiveness of Search To assess the effectiveness of the search procedure, we compute an initial abstraction of the FT20 network suitable for 5 link failures, using different values of the search breadth. On top of this, we additionally consider the impact of some of the heuristics described in section 6.2.7. Figure 6.7 presents the size (the number of nodes are on the y axis and the number of edges on top of the bars) of the computed abstractions with respect to various values for the breadth of search and sets of heuristics:

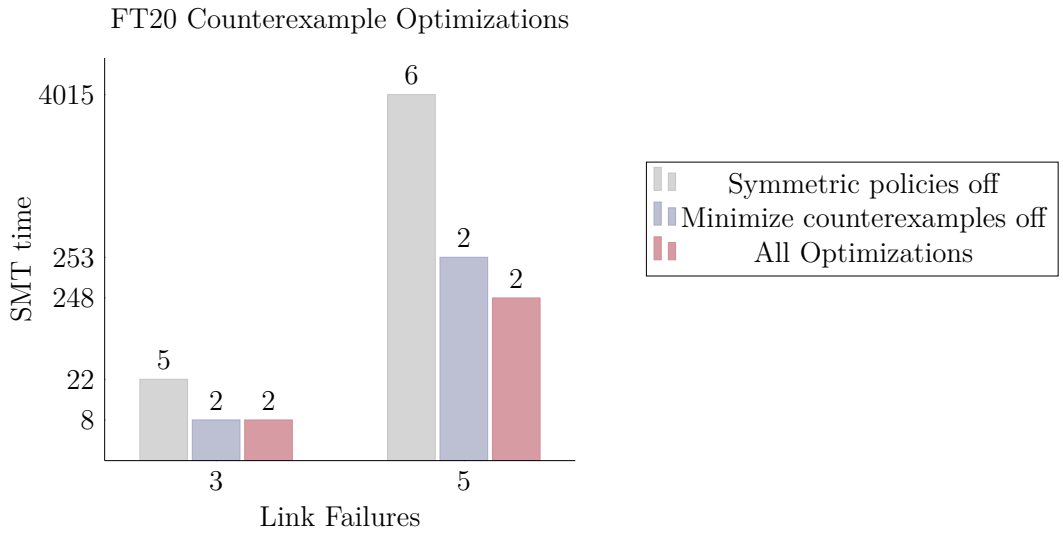


Figure 6.8: Effectiveness of minimizing counterexamples and of learning unused edges. On top of the bars is the number of SMT calls. The refinement time is insignificant so we omit it.

- Heuristics off means that (almost) all heuristics are turned off. We still try to split nodes that are on the cut-set.
- Reachable off means that we do not bias towards splitting of nodes in the reachable portion of the cut-set.
- Common off means that we do not bias towards splitting reachable nodes that have the most connections to unreachable nodes.

The results of this experiment show that in order to achieve effective compression ratios we need to employ both smart heuristics and a wide search through the space of abstractions. It is possible that increasing the search breadth would make the heuristics redundant, however, in most cases this would make the refinement process exceed acceptable time limits.

Use of Counterexamples. We now assess how important it is to 1) use symmetries in policy to infer more information from counterexamples, and 2) minimize the counterexample provided by the solver.

We see in Figure 6.8 that disabling them increases number of refinement iterations. While each of these refinements is performed quickly, the same cannot be guaranteed of the verification process that runs between them. Hence, it is important to keep refinement iterations as low as possible.

6.3 Discovering Symmetries Dynamically

In section 6.2, we introduced the idea of exploiting symmetries based on network compression. Origami, performed a topological transformation. Intuitively, Origami groups together nodes that route in a "similar" way under any bounded failure scenario. As demonstrated in section 6.2.8, this technique can significantly reduce the size of a network, making it feasible to verify networks under multiple failures. However, by design, approaches that rely on network compression have certain limitations that will be difficult to ever overcome:

- Firstly, relating the behavior of different nodes under any failure scenario will always be challenging. For illustration purposes, consider node failures. Two nodes u and v that are identical (*i.e.*, similar connectivity and policy) will always have at least one different behavior; in the failure scenario where u has failed its behavior will be different from v. Analogously, for the failure scenario where v has failed. This quickly makes it obvious, that maintaining precise routing behavior through Origami-style compression is futile; approximation —thus, loss of ability to reason about certain properties— is necessary.

- Compression techniques like the ones proposed by Bonsai and Origami, rely on a static, syntactic check to determine nodes that have similar routing policy. However, real network configurations may have small discrepancies, even between devices that supposedly fulfill the same "role" in the network (*e.g.*, two

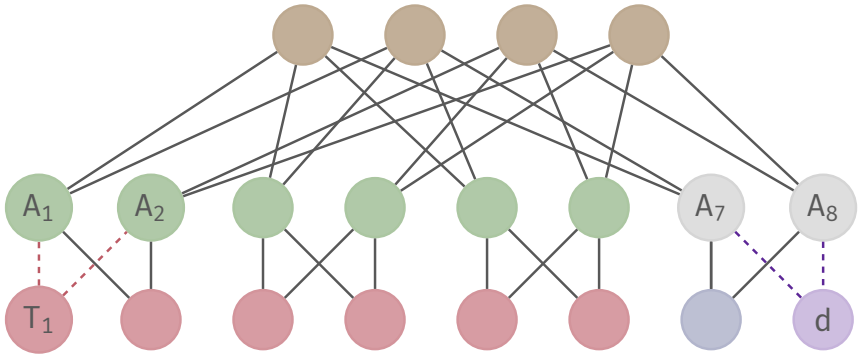


Figure 6.9: A FatTree network, showing the two 2-link failure scenarios (in red, and purple respectively) that can affect the routing behavior of the T_1 ToR router.

Top-Of-Rack routers). As a result, compression may be unable to achieve the necessary reduction in the size of the network, for verification to be possible.

- Finally, for compression to be sound, conditions are required of the network's policy. Origami, is stricter than Bonsai in this matter, where the policy needs to be both monotonic and isotonic.

6.3.1 Symmetries Among Failure Scenarios

In this section, we present a different, orthogonal to network compression, approach to scalable analysis of fault tolerance by exploiting symmetries. Guided by the limitations above, we do not attempt to relate the behavior of two different nodes; instead, we focus on relating different failure scenarios with respect to one node. Returning to our Fat Tree example, consider the impact of two link failures to the routing behavior of the Top-of-Rack routers. For exposition purposes, we focus on a single ToR router ($\mathtt{T}_1$). Assuming a typical shortest-path routing scheme, it is relatively easy to verify that $\mathtt{T}_1$ only has two routing behaviors: i) either it can route to the destination in 4 hops, or ii) it does not have a route to the destination. More specifically, the only two failure scenarios where T_1 cannot reach the destination are, when $\mathtt{T}_1$ is disconnected from the aggregation layer (the green nodes in fig. 6.9), or the destination is disconnected from its aggregation layer (gray nodes). Hence, while in theory we would need

to consider 4096 scenarios (4032 to account for 2-link failure combinations, plus 64 cases for each 1-link failure), the number of scenarios we have to consider in practice is far smaller.

To recap, so far, relying on per-node symmetries among failure scenarios, has obvious advantages: i) we can drastically reduce the state space, ii) we do not lose any precision, the actual routes are computed and not some approximation, iii) the correctness of this approach is fairly obvious and does not require a complex theory, and more importantly it does not impose restrictions on the routing policy of the network.

6.3.2 NV Programs as Analyses

```
let transFail (noRoute: a) (transBase: tedge -> a -> a)
              (e: tedge) (x : dict[(tedge, tedge), a]) =
  mapIte (fun key -> let (e1, e2) = key in e = e1 || e = e2)
         (fun v -> noRoute)
         (fun v -> transBase e v) x

let mergeFail (mergeBase: tnode -> a -> a -> a)
              (u: tnode) (x y : dict[(tedge,tedge), a] =
  combine (mergeBase u) x y

let initFail (initBase: tnode -> a) (u: tnode) =
  createDict (initBase u)
```

Figure 6.10: Meta-protocol defining routes for up to two link failures.

But, how do we compute the failure scenarios that matter for the given network and property? Once again motivated by the limitations of network compression, we avoid static checks that may be overly conservative and abolishing some of the performance gains to be made thanks to symmetries, and instead, we adopt a *dynamic* approach. As a first step, we focus on specifying what it means to compute the stable solutions of a network under all bounded failure scenarios, without worrying about

symmetries. We can easily do this in NV, by defining a *meta-protocol*, an NV program that takes as input a SPP instance and generates a new SPP instance that considers bounded failures. Figure 6.10, defines such a meta-protocol, which given, an init, transfer, and merge function (dubbed as `initBase`, `transBase`, and `mergeBase` in the program's text) and a value denoting the absence of a route (`noRoute`), defines a protocol that exchanges *maps* associating a failure scenario to a route. In this particular implementation, failure scenarios consist of two edges, and given that the two edges can be equal (*e.g.*, for every edge e_1 in the network, (e_1, e_1) is a valid map key), it intuitively describes all failure scenarios of up to two link failures. More complicated scenarios, involving more link failures or combination of node failures are easy to imagine. The core functionality is implemented in the transfer function; the transfer function over an edge e uses an if-then-else operation over the map (`mapIte`) to drop the route for every map entry (failure scenario) that includes e, and to otherwise, for scenarios that do not include e, apply the transfer function of the underlying protocol. The merge function combines the routes for each scenario. Since link failures do not affect the routes announced by each node, the init function simply creates a total (as, in NV maps are total) map from all failure scenarios to the initial route announced by each node.

We have explained how to compute the routes under all given failure scenarios, but still, we have not discussed at all about how to keep track and use symmetries among those scenarios. The reference to NV's total maps may have provided a hint at what is going on. As explained in section 5.1.2.1, NV's simulator implements maps using multi-terminal binary decision diagrams, a data structure that compactly represents map entries with common values, allocating a single value (leaf) in its internal representation. In other words, the symmetries that we seek in order to speed up fault-tolerance verification, are automatically discovered and used by NV's simulator!

Running the Analysis. Returning to the Fat Tree example, we assume a simple shortest path routing protocol, where routes are optional values:

```
let nodes = 20
let edges = ...

let init u = if u = d then Some 0 else None

let trans e x =
  match x with
  | None -> None
  | Some x -> Some (x+1)

let merge u x y =
  match (x,y) with
  | (_, None) -> x
  | (None, _) -> y
  | (Some x, Some y) -> if x <= y then x else y
```

Assuming the above network is defined in a file called "fattree.nv", and the failures meta-protocol is defined in a file called "failures.nv", we can define an NV program that combines them:

```
include "fattree.nv"
include "failures.nv"


let label = solution {init = initFail init;
                      trans = transFail None trans;
                      merge = mergeFail merge}
```

The NV simulator will compute a solution for router $\mathtt{T}_1$ that corresponds to the following simple function from 2-link failure scenarios to routes:

$$\mathtt{label}_{\mathtt{T}_1}(f) = \begin{cases} \mathtt{None} & , f \in \{\langle(\mathtt{T}_1, \mathtt{A}_1), (\mathtt{T}_1, \mathtt{A}_2)\rangle, \langle(\mathtt{d}, \mathtt{A}_7), (\mathtt{d}, \mathtt{A}_8)\rangle\} \\ \mathtt{Some\ 4} & , \mathit{otherwise} \end{cases}$$

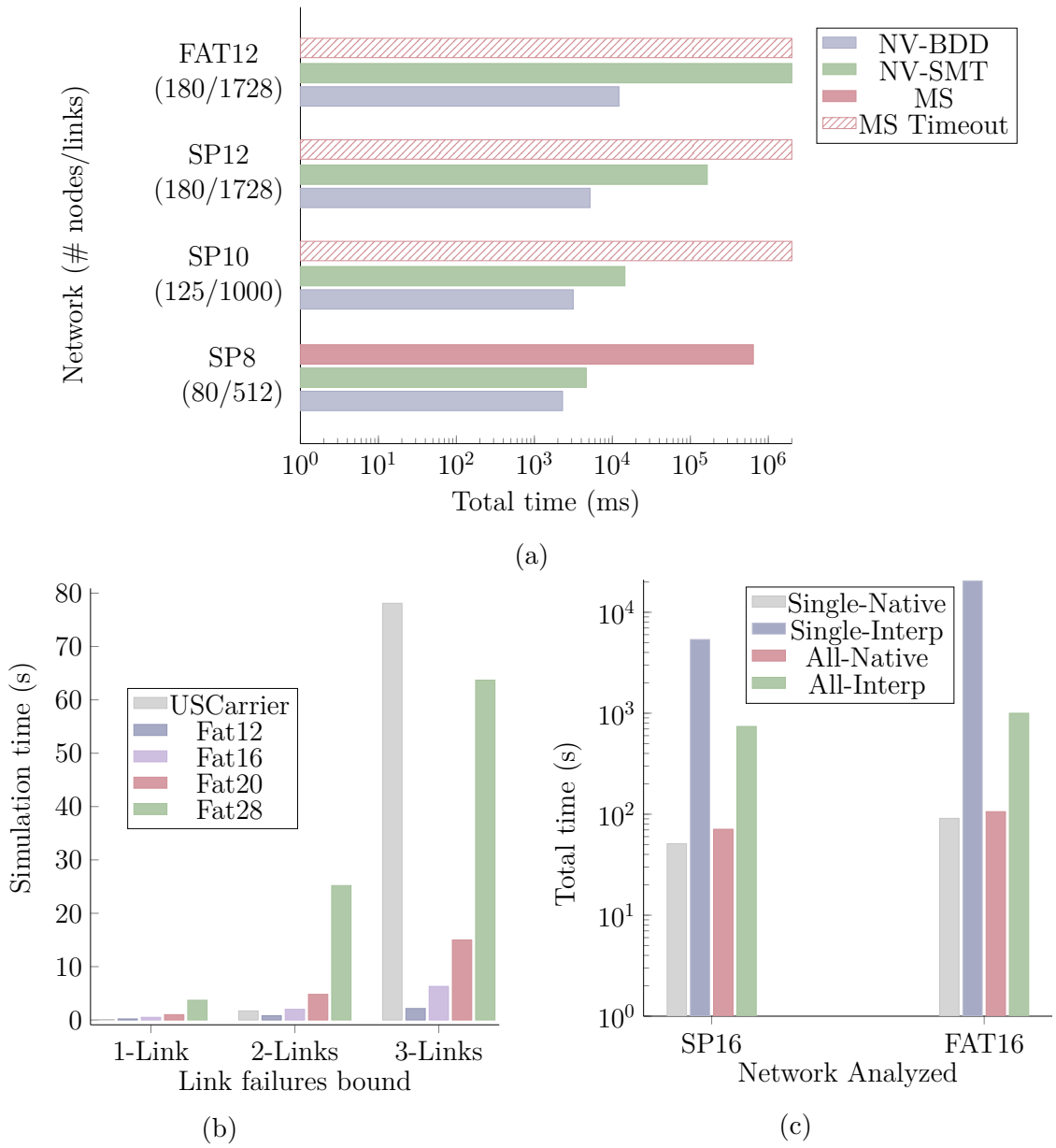


Figure 6.11: (a) compares the total time (including compilation time) of our fault tolerance analysis using native simulation, and the SMT approaches of NV and MineSweeper for a single-prefix. (b) shows how our fault tolerance analysis scales (excludes compilation time) as the size of the network and the failures increase. (c) compares the total time (including compilation time) to do fault tolerance analysis over all prefixes simultaneously or each prefix separately, using both, interpreted and native simulation.

6.3.3 Evaluation

Next, we examine the performance of our map-based fault tolerance analysis that simulates the given failure scenarios simultaneously. For the purposes of evaluation, we again use the Fat Tree networks as explained in section 5.1.5.1; as a brief reminder, these are Fat Trees that run eBGP with a shortest-path policy (`SP` networks), or a shortest-path policy plus no "valley-routing" (`FAT` networks). In addition, we consider one more network, a wide area network from Topology Zoo, running a policy synthesized using NetComplete [17], called USCarrier. At 174 nodes and 410 links, this network is smaller than some of the larger FatTrees, however, it is much less symmetric and can only tolerate a single link failure, and thus we expect it to strain analyses based on symmetries.

Comparison with SMT-based Approaches. Currently, the only other approach that provides exhaustive fault-tolerance verification of networks with expressive policy is the SMT-based analyses. Figure 6.11a compares the verification performance of our analysis, MineSweeper, and the SMT backend of NV, when verifying single link fault tolerance. It is well known, and also demonstrated in section 5.2.3, that SMT-based techniques have certain scaling limits even when failures —which largely increase the state space— are not considered. Unsurprisingly, in the presence of failures the performance of the SMT-based analysis deteriorates even faster before eventually timing out. In contrast, our simulation based analysis relies on MTBDDs to exploit symmetries in failure scenarios, and computes the routes for any possible failure in a matter of seconds.

Besides performance, the two approaches differ in what they compute too. The simulation computes the solutions of all nodes for all valid link failure scenarios; those solutions are presented to operators or checked against various assertions. On the other hand, the SMT approach looks for a violation to a specific assertion provided

a priori. This is both a blessing and a curse: the SMT solver may quickly determine that the property in question is violated and terminate, *e.g.* if we introduce a large number of failures that partition the network. At the same time, the SMT approach stops after finding a violation to the property and returns a single counterexample (as opposed to all counterexamples); once one bug is fixed one has to repeat the procedure with a presumably more difficult problem at every iteration.

Failure Analysis Scaling. We further evaluate how the fault tolerance analysis scales as we increase the size of the networks and the number of failures (fig. 6.11b). We note that precise fault tolerance analysis of some of these networks is out of reach of existing network analysis tools[4].

In the highly symmetric fat tree topologies, the analysis scales linearly with the number of link failure combinations. For instance, considering a single link failure, scaling is practically linear in the number of links. Of course, the number of unique failure combinations grows exponentially as we increase the bound on link failures, as demonstrated by the slowdown when considering 2 or 3 link failures for a network like FAT28, which has roughly 22,000 links.

On the other hand, USCarrier faces greater impact when the number of failures is increased; this is because the network is less symmetric and lacks redundancy to sustain multiple failures; the network gets partitioned. As more edges fail, the network's behavior changes significantly. Hence, the routes computed for each scenario can vary wildly, reducing the sharing that MTBDDs exploit.

Efficient Fault-Tolerance Analysis. Finally, in fig. 6.11c we compare the performance of the fault-tolerance analysis when done separately for each prefix vs. simultaneously for all prefixes, and when using the interpreted simulator vs. the native simulator. First, notice that native simulation can be at least an order of magnitude

[4]ARC [24] can scale, but is not compatible with rich policy (*e.g.*, tagging)

faster even when compilation time is taken into account. Another interesting thing to notice, is that doing the analyses per-prefix is slightly faster in the case of the native simulator, but much slower in the case of interpreted simulation. It is difficult to pinpoint the reasons for this. On the one hand, the symmetries in failure scenarios and the symmetries in routes to different prefixes do not perfectly align, so we expect more symmetries with respect to failure scenarios to be present when analyzing each prefix separately. Yet, it is much slower to interpret each prefix separately, most likely an artifact of high-penalty on the performance of our interpreter of some expressions that are often executed (*e.g.*, match expressions).

6.4 Related Work

ARC [24] is another system that is used to perform fault-tolerance analyses of networks. ARC's models are graphs and it performs network analysis by executing standard graph algorithms, such as shortest path computations and min-cuts, over these graphs. One of the advantages of this approach is that many graph algorithms are guaranteed to be polynomial, whereas the BDDs and SAT/SMT encodings used by NV result in exponential algorithms in the worst case. On the other hand, ARC can only reason about a very limited subset of routing policies; it is incapable of representing control plane features such as BGP local preference and communities.

Tiramisu [2], takes a mixed approach, where it uses different verification algorithms for different protocols and properties. For instance, for protocols and properties that standard graph algorithms suffice, and hence Tiramisu uses an approach similar to ARC. For richer policies and properties, such as when the network is using BGP local-preference values, Tiramisu relies on ILP reasoning. Finally, for some networks and properties (*e.g.*, when computing the exact paths is necessary) Tiramisu falls back to a simulation algorithm reminiscent of the one we use. The combination

of all these techniques and of domain-specific optimizations, often allows Tiramisu to scale better than generic approaches, such as simulation or SMT verification.

QARC [55] is an extension to ARC that focuses on doing *quantitative* reasoning under failures, *i.e.* finding failure scenarios that given a traffic load will overload some links. QARC can also synthesize repairs, by computing upgrades to link capacities in order to avoid load violations. QARC relies on an ILP solver, so naturally, it does not scale as well as ARC or even our simulator-based approach to fault-tolerance analysis, however, it attacks a more complicated problem so that is expected.

Finally, other simulation approaches resort to testing for fault-tolerance properties; Batfish generates random link failures, and simulates each resulting network independently. Obviously, this approach can only give weak guarantees, as one has to limit the number of simulations.

Chapter 7

Conclusions and Future Directions

Over the past decades, computer networks have become an integral part of our daily lives, and essential infrastructure to most industries. This had led to unprecedented growth in their size and complexity. Despite that, the means to configure and manage them have not caught up. Although we have been steadily making progress towards more automation, network configuration is still a manual and error-prone task as demonstrated by recent network misconfigurations [6, 28, 47, 43, 40, 46, 56, 41, 57, 58, 29]. To mitigate this problem, a number of automated static analyses have emerged to proactively detect such misconfigurations. This dissertation proposes some such tools: 1. a high-performance network simulator that relies on native code execution and Multi-Terminal Binary Decision Diagrams, allowing it to simulate complex protocols on top of large networks, an 2. optimizing SMT back-end that often outperforms existing verifiers by an order of magnitude, and finally, 3. new techniques to reason about fault-tolerance properties based on symmetry reductions.

More importantly, this dissertation explored how to simplify and facilitate the development of new network models and network analyses. Towards this goal, we presented NV a high-level declarative language to describe network models for verification purposes. We demonstrated that it is capable of expressing useful network

models (*e.g.*, widely used control plane protocols such as BGP and OSPF, and data plane operations such as longest prefix matching), while being amenable to efficient analysis. We hope that the network verification community will find NV useful when it comes to building new models of network protocols and to developing novel analyses.

7.1 Future Directions

There are various avenues to explore concerning NV and network verification in general. We highlight some of them in this section.

7.1.1 Network Simulation

The simulator used by NV is largely based on existing algorithms for control plane simulation. However, NV programs go beyond typical routing protocols; one can define meta-protocols that exchange richer structures, such as maps containing routes (sections 4.1 and 6.3.2), or data plane traffic (section 4.2). Along with other design choices, such as the use of MTBDDs, the added expressiveness and flexibility lead to a number of open questions.

Correctness and Convergence. Existing work [30, 51, 16] has identified conditions that provide correctness and convergence guarantees for the more restrictive class of the stable paths problem, where the merge function is restricted to a ranking function that selects the best of two routes. However, these conditions do not (always) apply in our context, and it can be difficult to know whether an NV program has a stable state. Moreover, one can write NV programs that despite having a stable state, standard simulation algorithms (*e.g.*, the ones used by NV or the one proposed in the original stable paths [30] work) cannot compute it, as they do not converge:

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

let init u =
  if u = 2n then Some 0 else None

let merge u x y =
  if u = 2n then None
  else (*pick shortest path*)

let trans e x =
  match x with
  | None -> None
  | Some x -> Some (x+1)
```

In this example, all nodes have a stable state where their labels are equal to `None`, and the SMT backend of NV successfully computes it. However, the simulator diverges: Node 2 sends a message (`Some 1`) to nodes 0 and 1. When node 2 receives a message (either from 0 or 1), its merge functions "invalidates" its existing route, setting it to `None`. Since node's 2 label changed, it will send a new message (`None`) to its neighbors. Node 1 and 2, will now recompute their label, and will see the route from each other as more preferred (compared to the empty route offered by node 0). Node 1, will select a route with cost 2 from node 0, and will sent a route with cost 3 to node 2. At this point, nodes 1 and 0 will alternate in selecting a new route and sending it to their neighbors, and their label will increase by 1 forever.

Devising conditions that guarantee that an NV program has a stable state as well as conditions that guarantee that the simulator will compute this stable state is interesting future work to pursue.

Optimizations. Existing simulators often employ domain-specific optimizations to achieve high performance. For instance, both ShapeShifter and FastPlane optimize the order in which nodes are "executed" based on the protocols they simulate (*e.g.*, they first disseminate BGP routes that are more preferred), ensuring faster conver-

gence to a stable state. However, unlike ShapeShifter and FastPlane, NV's simulator has to deal with arbitrary NV programs rather than a small, fixed number of selected routing protocols. Moreover, heuristics that work well for standard routing protocols such as BGP, do not apply to other NV programs, such as the fault-tolerance meta-protocol of section 6.3. It is unlikely to find heuristics that fit all problems. Therefore it might be sensible to give the user more control over the execution order in a simulation.

Another technique that some simulators (*e.g.*, Batfish) and other network verification tools (*e.g.*, QARC, McNetKat [55, 50]) use in order to scale is parallelism. There are multiple ways in which we can parallelize our simulator:

i) Compute solutions to different instances of the stable paths problem in parallel. For instance, in some cases we can compute solutions for each destination prefix separately and in parallel.

ii) Compute the messages sent to each neighbor in parallel. This might be helpful in very large networks, where each node has dozens of neighbors.

iii) Parallelize the MTBDD/BDD operations. This approach will likely yield the best results. Simulations that take a long time, likely involve large maps (MTBDDs with many leaves); applying operations over these leaves in parallel may significantly reduce simulation time.

In theory a parallel implementation of our simulator can improve its performance, however, there are engineering challenges to overcome. Most BDD/MTBDD libraries (including CUDD[52], which we use), are neither parallel nor thread-safe. Moreover, until recently OCaml did not support shared-memory concurrency, which might be useful when implementing a parallel simulator. The latter is slowly changing, as newer versions of OCaml are expected to support such features [49]. For the former, Sylvan [60] is a decision diagram framework that uses multi-core implementations of

BDD/MTBDD operations. Whether it is possible to redesign our simulator to be thread-safe and to achieve speed-up through parallelism remains an open question.

7.1.2 Richer Network Models

This thesis —like a lot of the research on control plane verification— focuses on relatively simple path properties, such as reachability (existence of a path), the length of the path, and so on. Yet service disruptions are often characterized by *quantitative* properties, such as *link load* and *latency*. Recent work on QARC [55] showed that, for networks with simple routing policy, an integer-linear programming approach can be used to verify quantitative properties in the presence of a bounded number of failures. A related direction, is to consider probabilistic properties: NetDice [54] performs probabilistic fault-tolerance analysis providing *soft* guarantees about a range of network properties.

To capture such properties we may need to enrich the network model used by NV. A question that naturally arises, is whether the current tools (*e.g.*, the simulator) and the datastructures supporting them (*e.g.*, MTBDDs) will be able to *efficiently* analyze such richer models.

7.1.3 Network Specification and Repair

While automated network verification tools have made leaps in the past years, deploying such tools in practice still requires a lot of manual work.

Network Specifications. One of the reasons for this is that *specification* of network properties remains largely manual and the community has not invested as heavily in tools to support this task. An exception to this trend is Config2Spec [12] which aims to automatically infer properties of the network. In particular, for each prefix announced in the network, they are looking to find which nodes satisfy a fixed set of

properties (reachability, load-balancing, waypointing, and isolation) under a bounded number of failures. To do so, they rely on a counter-example guided abstraction refinement approach that interleaves simulation with SMT verification. The inferred specifications act as a useful starting point for network operators and can be checked whenever the network is updated. The main drawback of their approach is that it relies on a complicated CEGAR loop which uses an SMT verifier to reason about failures. As such, their approach will not scale to larger networks with more than a few hundred nodes, or to networks that have very complex routing policy. An alternative that avoids the expensive CEGAR loop, would be to use our simulator and the fault-tolerance analysis of section 6.3; this would streamline the process as it would compute precise routes for every failure scenario, which we can then inspect to infer properties that hold.

Network Repair. Even when current network analyses and verification tools suffice to analyze a network, fixing the reported problems remains a mostly manual, and extremely tedious task. A fix may require changes across different devices, and routing policies, and it might affect different traffic classes, hence it is difficult to ensure that a repair works and it does not break some other network property. So far, the only tool that attempts to find fixes to network misconfigurations is CPR [23]. CPR builds on top of ARC [24] to automatically synthesize minimal repairs for the violations found. However, building on top of ARC means it also inherits the limitations of ARC's graph-based approach, *i.e.*, no support for policy attributes, such as BGP communities and local-preference.

It would be interesting to explore whether NV can be used to synthesize network repairs. Once again, the ability to use a simulator to compute routes —unlike other tools that focus on building abstract models to verify specific properties— for all failure scenarios, might come in handy. However, we also anticipate challenges unique

to NV; NV's low-level representation has abandoned some of the original structure of the problem, and recovering that will require some effort. Aside from the fact that NV operates on top of a representation that differs from the original router configurations (and we currently have no way of "decompiling" from NV to router configurations), we will likely need new language constructs to facilitate this task. For instance, currently, assertions are just boolean expressions, providing no provenance as to why an assertion failed.

Bibliography

[1] Internet Protocol. RFC 791, September 1981.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast and general network verification. *arXiv preprint arXiv:1906.02043*, 2019.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[4] Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. Symbolic model checking of probabilistic processes using mtbdds and the kronecker representation. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, TACAS '00, pages 395–410, 2000.

[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014.

[6] M Anderson. Time warner cable says outages largely resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved, 2014.

[7] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *SIGCOMM*, August 2017.

[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. SIGCOMM '18, pages 476–489, 2018.

[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. In *Proceedings of the ACM on Programming Languages*, volume 4, January 2020. Article 42.

[11] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *SIGCOMM*, 2016.

[12] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Config2spec: Mining network specifications from network configurations. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 969–984, 2020.

[13] Cisco. Cisco ios master command list, all releases. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/mcl/allreleasemcl/all-book.html, 2019.

[14] Edmund M Clarke, Masahiro Fujita, and Xudong Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of discrete functions*, pages 93–108. Springer, 1996.

[15] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV, Proceedings*, pages 154–169, 2000.

[16] Matthew L. Daggitt, Alexander J. T. Gurney, and Timothy G. Griffin. Asynchronous convergence of policy-rich distributed bellman-ford routing protocols. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 103–116, 2018.

[17] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *USENIX NSDI'18*, Renton, WA, USA, 2018.

[18] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI*, 2016.

[19] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. 2003.

[20] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[21] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In *European Symposium on Programming*, pages 282–309. Springer, 2016.

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

[23] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 359–373. ACM, 2017.

[24] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, 2016.

[25] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. Efficient verification of network fault tolerance via counterexample-guided refinement. In *International Conference on Computer Aided Verification*, pages 305–323. Springer, 2019.

[26] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. Nv: An intermediate language for verification of network control planes. *PLDI. ACM*, 2020.

[27] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.

[28] Joanne Godfrey. The summer of network misconfigurations. https://blog.algosec.com/2016/08/business-outages-caused-misconfigurations-headline-news-summer.html, 2016.

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

[30] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Networking*, 10(2), 2002.

[31] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[32] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[33] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[34] P. Lapukhov, A. Premji, and J. Mitchell. Use of BGP for routing in large-scale data centers. Internet draft, 2015.

[35] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[36] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.

[37] Nuno P Lopes and Andrey Rybalchenko. Fast bgp simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 386–408. Springer, 2019.

[38] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[39] M. Morris Mano and Michael D. Ciletti. *Digital Design (4th Edition)*. Prentice-Hall, Inc., USA, 2006.

[40] Hugo Martin and Samantha Masunaga. United airlines blames grounding of hundreds of flights on computer glitch. https://www.latimes.com/business/la-fi-united-flights-grounded-20150708-story.html, 2015.

[41] Kieren McCarthy. 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/, 2019.

[42] John Moy. OSPF Version 2. RFC 2328, April 1998.

[43] Ben Mutzabaugh. Unions want southwest ceo removed after it outage. https://www.usatoday.com/story/travel/flights/todayinthesky/2016/08/01/unions-want-southwest-ceo-removed-after-outage/87926582/, 2016.

[44] Benjamin C Pierce and C Benjamin. *Types and programming languages*. 2002.

[45] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. *SIGPLAN Not.*, 51(1):69–83, January 2016.

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

[47] Jonathan Shieber. Facebook blames a server configuration change for yesterday's outage. https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage/, 2019.

[48] Stephen F Siegel, Manchun Zheng, Ziqing Luo, Timothy K Zirkel, Andre V Marianiello, John G Edenhofner, Matthew B Dwyer, and Michael S Rogers. Civl: the concurrency intermediate verification language. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[49] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Retrofitting parallelism onto ocaml. *arXiv preprint arXiv:2004.11663*, 2020.

[50] Steffen Smolka, Praveen Kumar, David M Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. Scalable verification of probabilistic networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–203, 2019.

[51] João Luís Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Trans. Netw.*, 13(5):1160–1173, October 2005.

[52] Fabio Somenzi. Cudd: Cu decision diagram package. *http://vlsi.colorado.edu/fabio/CUDD/*, 1997.

[53] Kotikalapudi Sriram and Douglas C Montgomery. Resilient interdomain traffic exchange: Bgp security and ddos mitigation. 2019.

[54] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Probabilistic verification of network configurations.

[55] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. Detecting network load violations for distributed control planes. In *PLDI*, pages 974–988, 2020.

[56] Yevgenly Sverdlik. 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, 2012.

[57] Andree Toonk. Large scale bgp hijack out of india. https://www.bgpmon.net/large-scale-bgp-hijack-out-of-india/, 2015.

[58] Andree Toonk. Massive route leak causes internet slowdown. https://www.bgpmon.net/massive-route-leak-cause-internet-slowdown/, 2015.

[59] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152. ACM, 2013.

[60] Tom van Dijk and Jaco van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19(6):675–696, 2017.