1-1-2013

Automated Formal Analysis of Internet Routing Configurations

Anduo Wang
University of Pennsylvania, anduo@cis.upenn.edu

Automated Formal Analysis of Internet Routing Configurations

Abstract
Today's Internet interdomain routing protocol, the Border Gateway Protocol (BGP), is increasingly complicated and fragile due to policy misconfigurations by individual autonomous systems (ASes). To create provably correct networks, the past twenty years have witnessed, among many other efforts, advances in formal network modeling, system verification and testing, and point solutions for network management by formal reasoning. On the conceptual side, the formal models usually abstract away low-level details, specifying what are the correct functionalities but not how to achieve them. On the practical side, system verification of existing networked systems is generally hard, and system testing or simulation provide limited formal guarantees. This is known as a long standing challenge in network practice --- formal reasoning is decoupled from actual implementation. This thesis seeks to bridge formal reasoning and actual network implementation in the setting of the Border Gateway Protocol (BGP), by developing the Formally Verifiable Routing (FVR) toolkit that combines formal methods and programming language techniques. Starting from the formal model, FVR automates verification of routing models and the synthesis of faithful implementations that carries the correctness property. Conversely, starting from large real-world BGP systems with arbitrary policy configurations, automates the analysis of Internet routing configurations, and also includes a novel network reduction technique that scales up existing techniques for automated analysis. By developing the above formal theories and tools, this thesis aims to help network operators to create and manage BGP systems with

correctness guarantee.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Boon Thau Loo

Second Advisor
Andre Scedrov

Keywords
Formal Verification, Internet Routing, Reduction, Routing Algebra, Safety Analysis

Subject Categories
Computer Sciences

AUTOMATED FORMAL ANALYSIS OF INTERNET ROUTING CONFIGURATIONS

Anduo Wang

A DISSERTATION
in
Computer and Information Science
Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2013

Boon Thau Loo
Associate Professor,
Computer and Information Science
Supervisor of Dissertation

Andre Scedrov
Professor,
Computer and Information Science
Co-Supervisor of Dissertation

Val Tannen
Professor, Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Oleg Sokolsky (Chair), Research Associate Professor, Computer and Information Science

Rajeev Alur, Zisman Family Professor, Computer and Information Science

Jennifer Rexford, Gordon Y.S.Wu Professor, Computer Science, Princeton University

Jonathan M. Smith, Olga and Alberico Pompa Professor, Computer and Information Science

Carolyn L. Talcott, Computer Science Laboratory, SRI International

AUTOMATED FORMAL ANALYSIS OF INTERNET ROUTING CONFIGURATIONS

Dedicated to Wuxi, my hometown.

Acknowledgments

My vague thought of leading an entertaining and useful life started when I was an undergraduate at Tianjin University, the former Pei-yang University funded in 1895, China's first modern University, a copycat of its western peers (or ancestors) in an (failed) attempt of restoring the nation's glory. With this thought, I enrolled in the master program of Tsinghua University, now the nation's leading institution, first funded in 1911 as a preparatory school for students sent by the government to study in the United States. At Tsinghua, my vague thought took in the form of an attempt of starting the academic career — an application for Penn's graduate program. Thus I began my journey of a PhD student at Penn six years ago.

These six years turned out to be very hard. The six-years-hard work will be summarized in the rest of this thesis, this section, however, is reserved for those people whom I am so fortunate to have met over the years. Many thanks go to these people who support me and keep me going towards the PhD degree, those who foster my career and open my eyes to the exciting academic world. Indeed, working with them have already became the most entertaining and useful part of my life.

First, I want to thank Prof. Oleg Sokolsky and Prof. Boon Thau Loo, for both of whom, I can never fully express my gratitude. Oleg, with open-minded, strong-willed, and above all, kind and generous support, effectively brought me to Penn, guided me through my first year, and later chaired my WPEII exam, thesis defense.

Boon, whom I'm as fortunate to have as my major advisor in the next five year, has shaped my interdisciplinary PhD work. His appreciation for a broad spectrum of problems (e.g. bridging theory and system research in general) will continue to influence my research in future years. It will take even longer for me to "absorb" his other influence, e.g. his disciplined but agile, determined but kind personality. Only by then will I be able to view Boon a role model. I was once overwhelmed by Boon's art of advisory so much that I almost lost hope in pursuing faculty positions. Fortunately, I recovered a bit in the past year, and have been in the midst of "deciphering" and "internalizing" his masterful working style ever since.

Next, I want to thank Prof. Andre Scedrov, Dr. Carolyn Talcott, and Prof. Jennifer Rexford. Andre has been my major advisor since the end of my second year. From the very beginning, Andre impressed me with the depth and width of his research, and excited me of how much one could achieve over the years.[1] Andre brought me a new circle of cool theory researchers, including Carolyn, who was later my mentor at SRI in the past three years. Carolyn, together with Boon, molded the second half of this thesis. In addition to this, Carolyn marked the turning point of my research in the sense that, with Carolyn, I gradually picked up the "collaboration skills". I believe this is partly because she is so smart that even my awkward communication skill at that time did not stop an effective collaboration, which further entailed improved communication. On the other hand, Jennifer, the open-minded sharp system researcher, helped carve the first half of my thesis. Though experts of different domains, like Andre, both Jennifer and Carolyn are the perfect examples of combining big-picture and low-level details, activeness and experience. Jennifer also offered brilliant, detailed and in-depth comments on my job search.

I also want to thank my other committee members from Penn: Prof. Rajeev

[1] By the way, since the train ride from Baltimore to Philadelphia where he found us students a great place in a seemingly full train, I am totally convinced of every suggestion Andre made about public transportation and anything else.

Alur and Prof. Jonathan M. Smith. Rajeev's comments on the importance of rigor will remind me for the rest of my career that every small piece of technical detail reveals the researcher's understanding of the problem and the depth of the adopted approach. On the other hand, Jonathan's lecture on scoping and framing the problem, and involving junior students comes at such a high level that reminds me of the ultimate goal of research.

In addition, I want to thank post-doc researchers at Penn: Alexander Gurney, Limin Jia, and Vivek Nigam. Through close collaboration with them, I observe the same quality of discipline and rigor, which is also the goal I set for my own post-doc job. Besides, Alex has an unusual understanding and interests of computer science ranging from the hard-core networking system to the theoretical type system. He is also a kind mentor one could rely on for general research instruction. Limin is smart, sharp and open-minded lady with expertise in programming language and verification. Her patience and hands-on help leaded to the major breakthrough for my PhD study and thesis work. Without her help, my thesis will not reach its current rigor and depth. In addition to tremendous help in research, she also offered personal advices. Vivek is an happy, easy-going fellow with deep understanding of the most fundamental part of computer science — logic. I admire Limin and Vivek for their research depth as well as their very different personalities. I cannot feel more lucky to have the chance to closely work with them.

At computer science lab, SRI, I want to thank Dr. Mark-Oliver Stehr, Dr. Minyoung Kim for being my orientation host, Dr. Steven Eker for his patient help on Maude, Ms. Lori Truitt for her kind administrative help, Dr. Natarajan Shankar for holding the coolest "Crazy idea session", Sam Owre for his suggestions on Maude and PVS tool, Dr. Bruno Dutertre for bringing us interns to the lunch of great meal and chat, Dr. Patrick Lincoln for his great home party, Phillip Porras for his illustrative introduction of the emerging software-defined networks.

My special thanks to fellow Penn students: Huang Liang, Peng Li, Wenchao

Zhou, and Changbin Liu. Huang and Peng gave me invaluable advices in my first year. Wenchao and Changbin both helped me a lot on research projects. Wenchao also offered valuable comments on my WPEII exam, dissertation, and job search. My special thanks also goes to fellow student interns and office mates at SRI: Leila Jalali (2011), Gabriel Gelman, and Dongting Yu (2012). Leila offered to drive me to work, and we enjoyed chats on painting and Chinese thinker Confucius. Dongting and I took advantage of the great Chinese foods around SRI, and aside from food, I learned of operational networks form him. Besides, I want to thank fellow students I worked closely with, including, Yifei Yuan, Xianglong Han, Jinyan Cao, Yiqing Ren, and Salar Moarref. Many thanks to other friends, including but not limited to: Mengmeng Liu, Zhuowei Bao, Shiv Muthukumar, Yang Li, Shaohui Wang, Zhihao Jiang, Mingchen Zhao, Santosh Nagarakatte, Dong Lin, Alwyn Goodloe, and Tao Tao.

My research was funded in part by the following funding agencies and sources: NSF CAREER CNS-0845552, AFOSR Young Investigator Award, AFOSR MURI grant FA9550-08-1-0352, and the NSF ITR-1138996 ExCAPE Expeditions Project.

Last but not least, I want to thank my parents, Xicheng Wang and Dongzhen Zhang. In the past six years, I have witnessed the weakness and strength of the genes you folks have passed on to me. I want to thank you for their gift, and I am proud to tell that with the helps of all the people at Penn, SRI, and others, in my purpose of the following thesis, I was closer than ever in making the best out of my parents' genes.

ABSTRACT

AUTOMATED FORMAL ANALYSIS OF INTERNET ROUTING CONFIGURATIONS

Anduo Wang

Boon Thau Loo
Andre Scedrov

Today's Internet interdomain routing protocol, the Border Gateway Protocol (BGP), is increasingly complicated and fragile due to policy misconfigurations by individual autonomous systems (ASes). To create provably correct networks, the past twenty years have witnessed, among many other efforts, advances in formal network modeling, system verification and testing, and point solutions for network management by formal reasoning. On the conceptual side, the formal models usually abstract away low-level details, specifying what are the correct functionalities but not how to achieve them. On the practical side, system verification of existing networked systems is generally hard, and system testing or simulation provide limited formal guarantees. This is known as a long standing challenge in network practice — formal reasoning is decoupled from actual implementation.

This thesis seeks to bridge formal reasoning and actual network implementation in the setting of the Border Gateway Protocol (BGP), by developing the Formally Verifiable Routing (*FVR*) toolkit that combines formal methods and programming language techniques. Starting from the formal model, *FVR* automates **verification of routing models** and the **synthesis of faithful implementations** that carries the correctness property. Conversely, starting from large real-world BGP systems with arbitrary policy configurations, *FVR* automates the **analysis of Internet routing configurations**, and also includes a novel network reduction technique that **scales up existing techniques for automated analysis**. By developing the above formal theories and tools, this thesis aims to help network operators to create and manage BGP systems with correctness guarantee.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today's Internet is a network of networks, the constituting networks under a single administrative domain are called domains. To exchange reachability information across and within the domains, the Internet runs a single inter-domain routing protocol, called the Border Gateway Protocol (BGP). BGP allows each domain to independently configure its BGP policy to influence routing for economical reasons. As a result, the system-wide behavior of the Internet is determined by the implementation of the BGP protocol, as well as how each network domain configures its BGP *routing policies*. Given a BGP network (or BGP system), we refer to its network topology, routing mechanisms (e.g. how the BGP nodes exchange reachability information), and BGP routing policies as *configurations* (we use BGP configurations, policy configurations, Internet routing configurations interchangeably); and broadly refer to any of the non-converging BGP behaviors as *routing anomalies* (or BGP anomalies, anomalies).

Several studies such as [36, 42, 44] have shown that BGP mis-configurations result in routing anomalies including route oscillation, delayed convergence, or in severe cases, prolonged periods of network disruptions. These routing anomalies incur serious performance disruptions and router overhead. In response, the net-

working community has proposed several BGP extensions [59, 72, 74], alternative Internet architectures [69], and policy configuration guidelines (or "safety guidelines") that help the network operator to achieve global convergence if universally adopted [18, 19]. Despite all these efforts, BGP policy-based routing configurations have become a main source for routing anomalies ranging from the frquent route oscillation to the less common but more sever Internet outage. Indeed BGP configurations has been recognized Internet's most complicated and fragile components, and are the focus of this thesis.

1.1 Motivation

To build a more reliable, predictable, and secure network, the last twenty years have witnessed great efforts in modeling, static reasoning and dynamic anomaly detection for BGP configurations. These efforts include formal network modeling, system verification and testing, and point solutions through formal reasoning.

On the conceptual side, abstract formal models of BGP allow researchers to explore how local policies affect BGP stability [25, 66, 30, 67, 70, 16, 28, 33, 21, 20, 17, 73]. Algebraic models such as *routing algebra* are used to formalize routing protocols with convergence guarantees. Combinatorial models such as *stable path problem* enable researchers to study the dynamic behavior of BGP and identify policy guidelines that, if universally adopted by ISPs, could ensure global convergence [19, 18, 23, 14, 22, 64, 13, 63]. However, these formal models usually abstract away low-level details, specifying only some correct functionalities but not how to achieve them. The correct-by-construction approach of routing algebra also leaves out useful semantics, which cannot be expressed algebraically.

Concurrently, on the practical side, several diagnosis tools for anomalies are developed, ranging from static configuration checker and model checking tools, to runtime debugging tools of deployed systems. *Static checking* tools include *rcc* [50],

which statically checks "potential" configurations for possible faults. *Model checking* tools use a collection of algorithmic techniques for checking temporal properties of system instances based on exhaustive state space exploration. Recent advances in model checking detect anomalies in network protocol systems by imposing constraints on network implementations. Other model-checking tools for BGP anomalies include [34, 11, 56]. In addition, several practical software tools and testing platforms have been proposed to facilitate the verification of existing networked systems, such as *runtime verification* platform [15, 32] *Pip* [61]), which provides mechanisms for checking at runtime that a system does not violate expected properties. However, system verification of existing networked systems is generally hard, and system testing or simulation provide limited formal guarantees.

There are also hybrid approaches — point solutions such as BGP configuration guidelines [18, 19], extensions [59, 72, 74], alternative Internet architectures [69]. However, the development of such guidelines and architectures requires deep understanding and insights, relying on the reasoning process that is often manual, tedious, and error prone. Worse, once developed, to make the approach effective, universal deployment is also required, incurring additional overhead.

This is known as a long standing challenge in network practice — the formal reasoning process is decoupled from the actual implementation.

1.2 Research Challenges

This thesis aims to address the long standing challenge — the decoupling of formal reasoning and actual implementation. To achieve this in the setting of BGP routing configuration, we must address the following challenges.

- **Automation.** Abstract BGP models enable researchers to explore how local

policies affect BGP stability [25, 66, 30, 67, 70, 16, 28, 33, 21, 20, 17, 73]. However the reasoning process is a tedious and manual one, based on large number of low-level variables. To lower the bar of adoption of these model-based reasoning tools, and to free network operators to better focus on high-level network-wide properties, we need tool support that automates the reasoning process.

- **Generating property-preserving implementation.** The formal models usually abstract away low-level details, specifying what are the correct functionalities but not how to achieve them. To enforce the formal reasoning result on actual implementation, a solution is to compile a verified formal model into executable implementation.

- **Applicability.** Existing correct-by-construction formal models allow us to verify formal models and generate correctness-preserving implementations. However, these models do not capture all "well-behaved" policy configurations since they rely on sufficient correctness conditions that rule out some "correct" yet semantically useful policy configurations. The static verification output is also restrictive. To better address actual policy configurations, we need more general formal models and analysis techniques that provide more informative analysis output.

- **Scalability.** Finally, due to the size of the Internet, and the NP-hard nature of the global convergence property — the general problem of analyzing BGP systems are hard. State-based analysis (e.g. model checking) will likely suffer an exponential blow-up as network size increases. Therefore, we need networking-theory that exploits the redundancy in policy configuration to scale up existing analysis.

1.3 Contributions

In bridging formal reasoning and the actual implementation for BGP systems, in this thesis I designed and implemented ***F**ormally **V**erifiable **R**outing* (*FVR*) toolkit. Its methods and tools can be broadly classified into two categories.

First, to verify formal models and synthesize faithful implementations, *FVR* frees network operators from manual analysis and lower the bar of applying formal tools, and enforces the formal reasoning result in the actual distributed routing-protocol implementation. Thus *FVR* addresses the research challenges **automation** and **generating implementation**, and the specific contributions are:

C1. ***Verifying a routing policy using a SMT solver [80, 81].*** I map various policy configuration models into logical constraints; and formulated verification of routing convergence property as a constraint solving problem solvable in existing SMT-solvers — the policy configurations constraints are checked against convergence conditions drawn from previous work [67].

C2. ***Verifying a routing policy with a theorem prover [82, 79].*** In addition to SMT solving, I formalize in theorem prover the algebraic routing policy model that allows construction of compound policy configurations from atomic ones. By using the theorem prover's (e.g. PVS) abstract data-types, I develop a suite of verified atomic component algebras, based on which network operators can compose compound policy configurations with correctness guarantees.

C3. ***Convert formal policy model into* NDlog *programs [81, 80].*** *NDlog* is a datalog like language for specifying declarative networking. I develop logical formulations for both policy models (routing algebra and SPP formalism) and *NDlog* programs, based on which the formal model is mapped into a logically equivalent *NDlog* program. I also prove the translation is sound and implemented it for the BGP protocol.

Second, *FVR* automates formal analysis of actual Internet routing systems, including the policy configurations and the routing mechanisms. *FVR* also mitigates the inherent state explosion problem in exhaustive search based formal analysis by including a ***network reduction technique*** that scales up analysis. Central to network reduction is a rewriting calculus that reduces input network size prior to analysis. Thus *FVR* addresses the research challenges of **Applicability** and **Scalability**, and the specific contributions are:

C4. ***Extract the formal model [85].*** I design a formal model in Maude that captures a BGP system's dynamic behavior given its topology and policy configurations. The dynamic representation is essentially a transition system of concurrently updating network objects — the routers and routing messages. I also implement a Maude library that automatically extracts the formal model for a BGP system.

C5. ***Dynamic analysis on formal model [85, 84].*** I develop a Maude library that automatically catches conflicting policy configurations in the formal model. This is achieved by detecting route oscillations in the dynamic routing system updates by utilizing Maude's high-performance exhaustive search capability.

C6. ***Verifying routing protocols in NDlog programs [76, 83].*** The core of reasoning about *NDlog* program is to encode the semantics of *NDlog*— a variant of datalog — in logical statements recognizable in PVS. To achieve this, I develop a mapping that compiles *NDlog* program into logically equivalent PVS recursive definitions, hence lowering the barrier to using theorem provers for verifying *NDlog* routing protocols.

C7. ***Network Reduction [84, 77].*** I develop two reduction rules that transform policy configurations by only requiring neighboring routing information. The transformation proceeds by repeatedly merging network node pairs and the

relevant portion of configurations in the rest of the network. I prove that the rewriting rules are sound with regarding to the network property being analyzed. I also prove that they are symmetric to each other, and study how the order of node merging affects the reduction result.

C8. ***Case studies and evaluation [84, 78].*** I develop a prototype of network reduction in Maude, evaluated it on a variety of network configurations (e.g. full mesh, route reflectors, confederations) on network topologies ranging from the AS-level CAIDA to router-level Rocketfuel dataset [68, 4]. The experiments show that network reduction enables us to analyze networks up to 500 nodes that are otherwise not tractable, while incurring only a small overhead.

Chapter 2

Background: BGP Anomalies and Formal Models

This chapter, we provide brief background information on today's Internet and the BGP routing protocol. We review two formal BGP models that our analysis adopts. We also present a taxonomy of BGP anomalies, including *safety*, the standard property for BGP convergence across ASes; *acyclic preference*, the property for ensuring a consistent configuration within an AS; and *IGP-iBGP consistency*, the property for avoiding intra-AS oscillation. This taxonomy of anomalies form the problem space our methods and tools aim to address.

2.1 Background on BGP

Today's Internet has grown from a collection of loosely cooperated networks, called the *Autonomous Systems (AS)* (or domains), each of which is typically owned and administered by an independent entity such as an Internet Server Provider (*ISP*). To exchange inter-Autonomous System (or *inter-domain*) network reachability information, a single routing protocol called the *Border Gateway Protocol* or *BGP* [6,

60, 5, 71] is used. As shown in Figure 2.1, BGP exchanges the routing information across ASes using external BGP (eBGP), while distributes those external ones within each AS using internal BGP (iBGP).

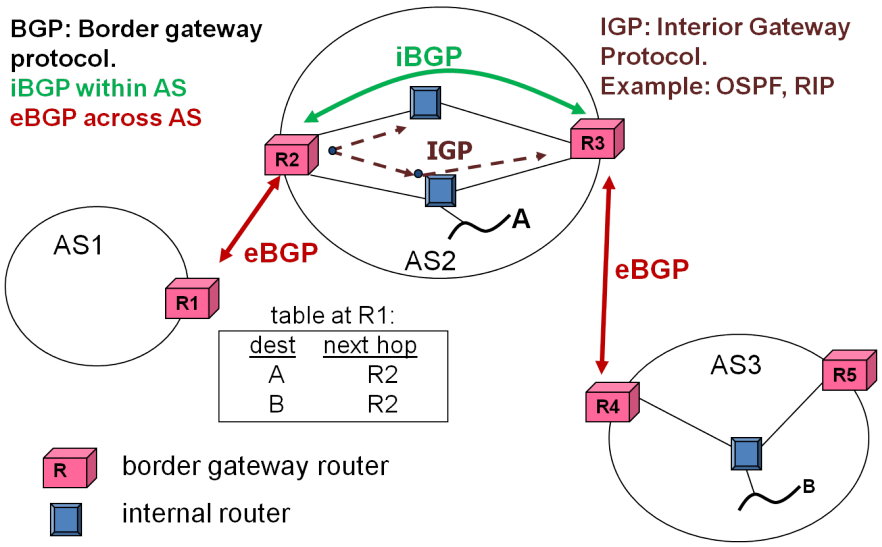

Figure 2.1: Overview of BGP system

Note that unlike the intra-domain routing protocols used solely within an AS, BGP adds routing policy policy, which is used by AS administrator to overwrite the shortest-path behavior. While this policy expressiveness contributes to BGP's popularity since it enables the ASes to express their own economic purposes, it also introduces routing anomalies ranging from route oscillations, slow convergence, to routing inconsistency [24, 26].

Since the internal routers do not communicate directly with routers outside their own AS, but through the border gateway routers, to maintain a consistent routing view within an AS, the original BGP protocol requires each AS to form a full-mesh BGP sessions among border gateway routers and internal routers. This full-mesh requirement does not scale up well. As an alternative to full-mesh, route reflectors are proposed. *Route reflectors* [5] are routers that serve a special role,

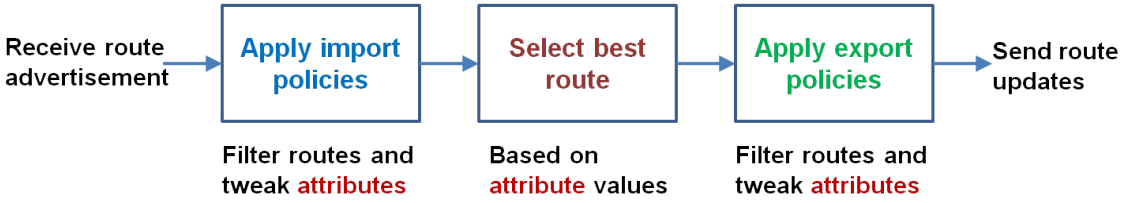

Figure 2.2: Policy-based routing process

they collate and distribute route advertisements on behalf of other internal routers called their *clients*.

In both eBGP and iBGP sessions, an individual BGP speaker (i.e. a network node that supports BGP protocol) exchanges route information (i.e. routing advertisements) with its neighbor using a *path-vector* protocol. Every BGP route is endowed with *attributes* that describe it. From the point of view of each BGP speaker, the *policy-based* BGP routing process, shown in Figure 2.2, in the following three steps: (1) Upon receiving a route advertisement, a BGP speaker may choose to accept or ignore the advertisement based on its *import policy*. Import policy includes filters which are used to deny certain routes. If the route is accepted, the BGP speaker may also tweak the path attributes associated with the route, and the stores it as a possible routing candidate in its routing table; (2) For each destination, the BGP speaker selects among all candidate routes one single best route based on their attribute values according to its *local routing policy*; (3) After the route selection process, if a new best route is selected, it may be re-advertised to the BGP speaker's neighbors according to its *export policy*. An example export policy is a filter which forbids re-advertisement of routes to certain neighbors if the BGP speaker is unwilling to carry traffic.

The determination of all the three kinds of policies is up to the network operator: BGP allows considerable flexibility. Conflicting policies, within or between

Stage	BGP route selection step
eBGP	**1. Highest LOC_PREF**
	2. Lowest AS path length
	3. Lowest origin type
iBGP	**4. Lowest MED (with same NEXT-HOP AS)**
	5. Closest exit point (lowest IGP cost)
	6. Lowest router ID (break tie)

Table 2.1: Key attributes in BGP route selection

ASes, are the cause of protocol oscillation, as the protocol struggles and fails to satisfy all policies at once.

In general, administrators implement the policies by configuring the path attributes associated with each route. These path attributes are summarized in Table 2.1. We also characterize these by whether they are primarily associated with eBGP- or iBGP-level routing decisions. In the second step of BGP policy-based routing, i.e., route selection step, the BGP speaker selects the single best route by comparing the attributes of its current available routes (for a given destination) with the new route, and then decides whether the new route is selected and promoted to be the best route. The attributes are listed in the order in which they are compared during route selection: if the routes are tied at any stage, then BGP proceeds to consider the next attribute on the list.

The most important attribute in eBGP route selection is local preference (`LOC_PREF`). This is a value set by each router on routes it receives, according to (arbitrary) rules established by the network operator. If two routes have the same local preference, then the next tiebreaking attribute is the AS path length—the number of ASes through which this route passes—followed by the 'origin' code. The next step is to use the multi-exit discriminator (`MED`) attribute, the most important attribute in iBGP route selection, which says which individual link is preferred, out of the many links between this AS and a given neighbor. If that was not enough to determine a single best route, BGP breaks ties by examining the shortest-path dis-

Table 2.2: Operations on Path Attribute in BGP Route Selection

	Path attribute	Receive from (before selection)	Send to (after selection)
1	LOCAL_PREF	External (NA): Set by local policy	External: NA
		Internal (LOCAL_PREF) : Unchanged	Internal: Unchanged
2	AS_PATH	External (P): Unchanged	External: Append local AS to P
		Internal (P): Unchanged	Internal: Unchanged
4	MED	External (MED/NA): Unchanged	External: MED (local-set) /NA
		Internal (NA): Unchanged	Internal : NA
5	NEXT_HOP	External (NEXT_HOP): Unchanged	External: Set self
		Internal (NEXT_HOP): Unchanged	Internal: Unchanged

tance to the relevant border router. Finally, if all else fails, it uses the value of each router's unique identifier. This final step is meant to ensure that all possible routes can be placed in a total order, with no two routes being equivalent in preference.

Finally, we summarize the operations on path attributes in Table 2.2 and Table 2.3. For example, for attribute `LOCAL_PREF`, import policy is specified in the first row of Table 2.2 and 2.3. If a router receives a new path from external nodes, then set `LOCAL_PREF` according to its local policy, otherwise unchanged; When a router advertises a new path to a neighbor, if it is sent to an external node, then removes `LOCAL_PREF` and sets its value to `NA` (because normally, AS would not like to expose their local policy to neighbor ASes), otherwise unchanged. Finally, when a router receives a path originated from an internal neighbor or advertises a path that it originates, always leaves `LOCAL_PREF` unspecified, i.e. sets its value to `NA`.

2.2 BGP Models

Routing anomalies are caused by routing misconfigurations. To represent the routing configurations for diagnosis, this thesis utilizes two formal models: *stable path*

Table 2.3: Operations on Path Attribute When sender originates route

	Path attribute	Before Send to
1	LOCAL_PREF	External: NA
		Internal: NA
2	AS_PATH	External: Set local AS
		Internal: Empty path
4	MED	External: MED (local-set)
		Internal : NA
5	NEXT_HOP	External: Set self
		Internal: Set self

problem (SPP) and *routing algebra*. For each model, we present a sufficient correctness condition that ensures protocol convergence (devoid of anomaly).

2.2.1 Stable Path Problem

Stable Paths Problem (SPP) [24] is a well-known formalism for BGP policy configuration, where the entire configuration instance is modeled in terms of the router-level topology and each router's policy-induced route preferences. SPP is a well-established combinatorial model of BGP configurations that captures the outcomes of routing policy—which paths are preferred over which other paths, at each router—while avoiding the need for detailed modeling of the BGP decision process in all its complexity.

Specifically, SPP models routing policy as an combinatorial structure: an instance of the SPP G is a tuple (V, E, d, P, Λ), where V and E are network nodes and arcs respectively, d is the specific destination node [1], P is the set of permitted (usable) paths available for each node to reach d, and Λ is the ranking functions for each node. For each node v, λ^v is its ranking function, mapping its routes to natural numbers (ranks), and P^v are its permitted paths, the set of available paths to reach o. A path assignment is a function π that maps each network node $v \in V$ to a

[1] Assuming the Internet is symmetric, we can study its routing behavior by studying routing to a single destination.

path $\pi(v) \in P^v$. A path assignment is *stable* if each node u selects a path $\pi(u)$ which is (1) the highest ranked path among its permitted paths, and (2) is consistent with the path chosen by the next-hop node. Consistency requires if $\pi(u) = (uv)P$ then for the next-hop node v, we must have $\pi(v) = P$. A solution to the SPP is a stable path assignment.

For a given SPP instance G as above, and a node i in V, write P^i for the subset of P consisting of paths from i to d. In this thesis, we will use the symbol '$\circ$' for concatenation of arcs and paths. If (i, j) is an arc in E, and p is a path from j to d, then their concatenation $(i, j) \circ p$ is a path from i to d. Similarly, if p is a path from i to j, and q is a path from k to l, and (j, k) is an arc in E, then the concatenation is $p \circ (j, k) \circ q$ or just $p \circ q$.

One major benefit of SPP formalism is that researchers have identified and proved various sufficient conditions for BGP convergence [67, 24]. In this thesis, we adopt the sufficient safety condition based on a structure called the *path digraph* [67] . For a SPP instance (V, E, d, P, Λ), its path digraph is defined as follows:

Definition 1. *Let $G = (V, E, d, P, \Lambda)$ be an SPP instance. The path digraph is a graph whose nodes are the elements of P, and where there is an arc (p, q) from p to q if either of these two cases holds:*

- *If $q = r \circ p$ for some path r, there is a 'transmission arc'.*
- *If p and q are two paths in P^i and $\lambda(p)$ is smaller than $\lambda(q)$ (i.e. p ranks higher and is more preferred), there is a 'preference arc'.*

If the digraph is acyclic then the SPP is safe, that is, the SPP instance has a unique stable solution, which can be found by iteration from any starting state. We will call an SPP instance *cyclic* (or *acyclic*) if its path digraph is cyclic (or acyclic).

For example, disagree gadget which exhibits divergence behavior, has a cyclic path digraph, as shown in Figure 2.3

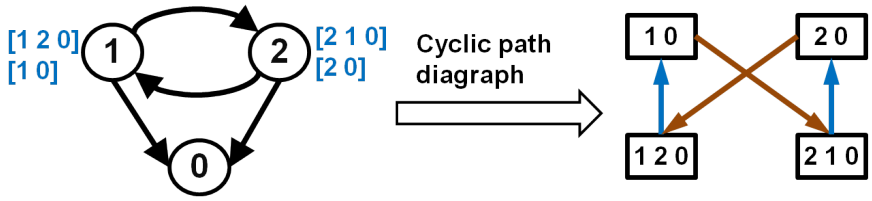

Figure 2.3: The path digraph for Disagree has a cycle

Route preference and Route selection In addition to the above ranking function Λ, we review two useful alternative notions: route preference relation $\prec$ and route selection function σ. Consider two paths p and q.

Conveniently, use $p \prec q$ to denote that p is preferred over q, i.e. if $\lambda(p)$ ranks higher $\lambda(q)$. Similarly, use $\sigma(\{p, q\}) = p$ to denote that among two routes $\{p, q\}$, p is the best and selected. Note that, among the three notions, route selection function σ is most general: it applies to a set of paths; besides, it does not imposes a total ordering (ranking) among a router's permitted paths.

2.2.2 Routing Algebra

Routing algebra[67] is an abstract structure that describes how BGP speakers calculate route preferences. An abstract routing algebra is a tuple $\langle \Sigma, \preceq, \mathcal{L}, \oplus, \mathcal{O} \rangle$.

- **Path signatures Σ:** It contains signatures for paths, which describe relevant attributes of the paths so that routes can be ranked. In particular, there is a special element ϕ in Σ, representing the signature for prohibited paths.

- **Path preference relation $\preceq$:** It is a total order relation over the elements of Σ and is used by the routing protocol algorithm to select the most desirable

path. To capture the property that prohibited paths are never selected, for any signature α that is not ϕ, $\alpha \prec \phi$.

- **Link attributes $\mathcal{L}$:** It is a set of labels describing attributes of links between immediate neighbors.

- **Path concatenation $\oplus$:** It is the concatenation function that takes a label and a signature and returns a signature. The purpose of $\oplus$ is for ASes to calculate the attributes of a new route received from its neighbor based on the link it comes from and the attributes of the received route.

- **Initial path signatures $\mathcal{O}$:** It is a subset of Σ called *origination* that represents signatures associated with initial routes, which are the paths containing only one node.

An important properties of a routing algebra is *monotonicity*, which ensures that as a path grows longer, it does not become more preferred. Formally:

Monotonicity: $\forall l \in \mathcal{L}, \forall s \in \Sigma, s \preceq l \oplus s$

Monotonicity ensures that a path $uv \circ P_v$, where uv is a link from u to v, and P_v is a route from v to the destination, should be less preferred than the path P_v. In the definitions above, s represents the signature of P_v and l is the label for uv. Notice that $uv \circ P_v$ and P_v are actually routes for different source nodes u and v. In a real protocol implementation, each AS only has a local view of its routing policies; which means that neither u nor v need to compare the preference between these two routes. However, convergence is a global property concerning routing policies at all nodes; and therefore, the route preference relation $\preceq$ is a total function for all routes in the entire system. The monotonicity property on the global routing policy will impact the convergence of the system as a whole.

Complex policies can be represented as compositions of simpler policies [25]. For example, ASes often rank routes based on multiple attributes (e.g., the next-

hop AS, the path length, and so on) in a series of "tie-breaking" steps. This is naturally captured by the *lexical product* operator, where $A \otimes B$ denotes the lexical product of algebras $A = \langle \Sigma_A, \preceq_A, \mathcal{L}_A, \oplus_A \rangle$ and $B = \langle \Sigma_B, \preceq_B, \mathcal{L}_B, \oplus_B \rangle$. Each link label in the resulting algebra is a pair, consisting of the labels for uv in A and B. Similarly, each signature for a path P is a pair composed of signatures from A and B. The concatenation function is the pairwise concatenation of the labels and signatures. The preference relation is also pairwise in lexical order: the first components are compared using $\preceq_A$, if equal then the second components are compared using $\preceq_B$. For instance, the widest shortest hop-count policy is the lexical product of a policy that prefers higher bandwidths with a policy that prefers shorter paths.

Example: Shortest Hop Count As a first example to show the mappings between the abstract algebra and a concrete policy, we consider shortest hop count policy. In this policy, routes with fewer hop counts are preferred.

- **Link labels and path signatures.** Since each node's immediate neighbor is one hop away, each link's attribute is set to 1: $\mathcal{L} = \{1\}$, whereas Σ is the set of natural numbers, which describes the hop count of a path. ∞ is the signature for prohibited path, i.e. all paths with cost ∞ are excluded from consideration.

- **Preference relations.** To select paths of fewest hop count, the less than equal ($\leq$) relation on natural numbers is used as $\preceq$.

- **Concatenation.** To compute the hop counts of paths, $\oplus$ is the plus (+) function on natural numbers for summing up the cost of a link and an existing path.

- **Initial path signatures** The hop count of a path containing one node is 0.

Example: Gao-Rexford Guideline As a second example, we illustrate the use of algebra to specify policy guideline. We consider the well-known Gao-Rexford policy guideline. Gao-Rexford guideline is based on the three types of business relationships between neighboring ASes: peers, customers, and providers. The policy guideline captures the constraint that an AS prefers routes through its customers over routes through peers or providers (called "guideline A" in [19]).[2]

Links and paths are distinguished based on their attributes, mapping naturally to label set $\mathcal{L}$, and signature set Σ, respectively. Consequently, the representation of the policy guideline in algebra is straightforward:

- **Link labels and path signatures.** Routes are classified based on the business relationship between neighboring ASes. Routes received from a customer, provider, or peer are classified with path signatures *C*, *P*, and *R*, respectively. In addition, the signature ϕ explicitly denotes all prohibited routes. Therefore, $\Sigma = \{C, P, R, \phi\}$. Likewise, labels $c/p/r$ denote three classes of links to customers, providers, and peers, and $\mathcal{L} = \{c, p, r\}$.

- **Preference relations.** Each AS prefers routes via customers over those via providers or peers, which is straightforwardly encoded as $C \prec P$ and $C \prec R$. To have a total ordering on the signatures, we must define a preference relation between provider (P) and peer (R) routes. Using $P = R$ implies that an AS can decide which routes are preferred based on other tie-breaking methods. So, our encoding uses the following three constraints: $C \prec P$, $C \prec R$, and $P = R$.

- **Concatenation.** The signature of a new route depends only on the node's relationship with its neighbor, as captured by the link label; for example $p \oplus C = P$, $p \oplus R = P$, and $p \oplus P = P$. However, an AS does not export routes

[2] A router is motivated to obey this guideline because it got paid by the customers.

learned from one peer or provider to other peers and providers, as illustrated in Figure 2.4. The figure shows a node u deciding whether to export (to its neighbor v) a route to destination d. Figure 2.4(a) shows that u is a provider of v, making uv a provider link and vu a customer link. Node u can export customer routes (C) to v, but any peer (R) and provider (P) routes are filtered.

- **Initial path signatures** The initial signature of a route is determined by the business relationship between the two end routers: it is set to $c/p/r$ if the route is from a customer, peer, or provider.

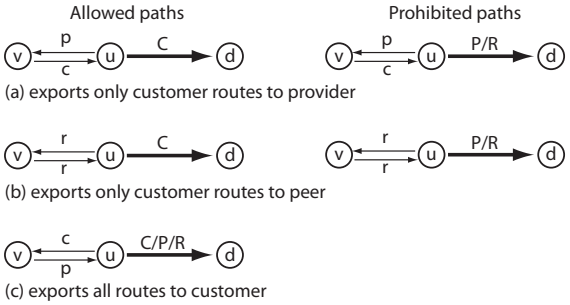

Figure 2.4: Export policy for Gao-Rexford guideline. The bold line indicates a route to destination d, with an associated route signature. Each unidirectional link between nodes u and v has a link label.

Note that in algebra, route filtering is expressed by generating a prohibited path (ϕ). For import policies, if v decides not to import a path of signature s from u, we can encode this import policy as $l \oplus s = \phi$ where l is the label for link vu. Our example policy involves *export* filtering. An export filter at node u can be modeled as an import filter at the receiving node v. The export filters in Figure 2.4(a) can be represented by $c \oplus P = \phi$ and $c \oplus R = \phi$, where the customer v filters any routes that u learned from its own peers or providers. The complete definition of the concatenation operator is:

$\oplus$	C	R	P
c	C	ϕ	ϕ
r	R	ϕ	ϕ
p	P	P	P

Similar algebraic encodings have been presented in prior work [66], but our illustration here serves to highlight a shortcoming of existing algebraic representations. To address this limitation, we propose extension to the original algebraic, and we will revisit this example in Chapter 4.1.2.

Finally, the lexical product [25] can then be used to compose multiple policies, for instance, combining the Gao-Rexford guideline with a policy that excludes particular paths by AS.

2.3 Taxonomy of BGP Anomalies

In this thesis, we identify three families of routing oscillation anomalies and the routing attributes involved in oscillation.

- In **eBGP anomalies**, routing policy conflicts occur at an inter-AS level. The typical causing attribute is `LOC_PREF`, because it is set arbitrarily at each AS, independently of any other.

- **iBGP anomalies** are limited to a single AS, and associated with `MED`. Due to a quirk in the decision procedure, it is possible for there to be three routes p, q, and r such that p is preferred to q, q to r, and r to p. The router will be unable to settle on a single choice, if there is feedback where its actions cause the visibility of those three routes to change.

- **iBGP-IGP anomalies** result from inconsistency between the semantics of route reflectors, and particular IGP distance values.

2.3.1 eBGP Anomaly: Policy Conflicts Across ASes

Policy conflicts across ASes are caused by the tension between (1) autonomy requirement that each router sets its policy independent of its neighbor; (2) the *routing tree* that the BGP system converges to, imposes dependency between the routing choice (policy) of neighboring routers. Given a group of connected ASes and the corresponding routing policies, for destination d, a routing tree is a spanning tree rooted at d such that each router's selected route is given by the path from the root to the router node.

When the routers are from different ASes, their routing policies can be set in an arbitrary way such that no unique policy compliant routing tree can be formed. We call this scenario policy conflicts across ASes.

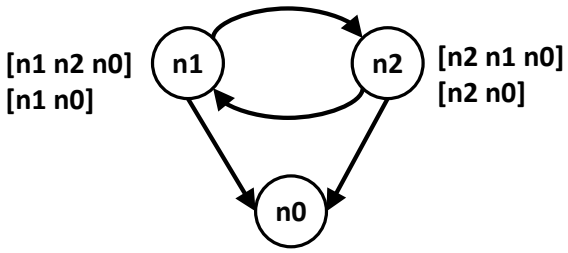

Figure 2.5: Disagree Gadget

For example, consider an eBGP system called the Disagree "gadget", whose network topology is shown in Figure 2.5. The available routes for each node are listed besides the corresponding node. The order in which the paths are listed illustrates each node's routing policy: Nodes prefer higher ranked routes, e.g. node `n1` prefers route `[n1 n2 n0]` over `[n1 n0]`. Disagree has two stable path assignment solutions: `([n1 n2 n0], [n2 n0])` and `([n2 n1 n0], [n1 n0])`. However, Disagree is not guaranteed to converge because there exists an execution trace where route assignments keep oscillating. Consider the execution where node `n1` and `n2` update and exchange routing messages in a synchronized manner, and

their network states oscillate between two unstable path assignments `([n1 n0]) ([n2 n0])` and `([n1 n2 n0] [n2 n1 n0])` forever.

The related eBGP correctness property we consider is safety [67, 26]. The progress of the BGP algorithm towards a solution depends on the timing of messages and other non-deterministic factors: we want to ensure that every execution schedule will result in a routing solution being found, regardless of the asynchronous nature of the protocol. The final state is characterized by *stability*, meaning that no future messages will affect which best paths are selected by each router.

Definition: A BGP instance is **safe**, if under all possible executions, it converges to a stable state, where the best routes selected by all the routers form a policy-compliant routing tree.

2.3.2 iBGP Anomaly: Inconsistent Policies

An unfortunate outcome of the BGP decision procedure is a corner case with the 'MED' attribute, where not all route preferences can be modeled by a total binary relation. This is because, with MED, one can have three routes p, q and r where p is preferred over q, q over r, and r over p, all at the same router: the preferences are cyclic. This phenomenon is associated with a specific family of protocol oscillations.

Consider the scenario in Figure 2.6: The routing policies of the instance are set by the following route selection functions:

$$\sigma_{iBGP}(\{AB, \mathbf{AC}\}) = \sigma_{igp}(\{AB, \mathbf{AC}\}) = \mathbf{AC} \tag{2.1}$$

$$\sigma_{iBGP}(\{AB, ADE\}) = \sigma_{igp}(\{AB, ADE\}) = AB \tag{2.2}$$

$$\sigma_{iBGP}(\{AC, ADE\}) = \sigma_{med}(\{AC, ADE\}) = ADE \tag{2.3}$$

$$\sigma_{iBGP}(\{\mathbf{AB}, AC, ADE\}) = \sigma_{igp}(\{AB, \sigma_{igp}(\{AC, ADE\})\}) \tag{2.4}$$

$$= \sigma_{igp}(\{AB, ADE\}) = \mathbf{AB} \tag{2.5}$$

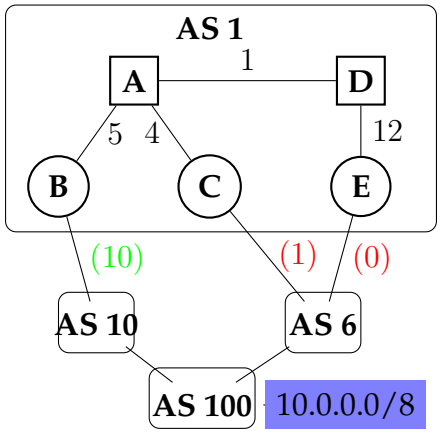

Figure 2.6: Cyclic Route Preference Causes Oscillation

Here the function σ_{iBGP} is inconsistent in lines (1) and (5). It prefers AC when given both AB and AC, but if it learns about ADE as well, its best route switches to AB, rather than sticking with AC or adopting the new ADE.

The cyclic nature of the preferences in this example is revealed from the first three lines: (1) says AC is better than AB, (2) says AB is better than ADE, but (3) says ADE is better than AC. The inconsistency arises from the fact that different attributes are being used to establish the preferences: MED in (3) overrides IGP distance.

We will now formalize this idea by showing that an inconsistent σ_{iBGP} function always leads to such a 'cycle'. Conversely, if σ_{iBGP} is consistent then its preferences can be implemented by a $\prec$ relation.

Given a selection function σ, derive a binary relation $\prec$ by $p \prec q$ if and only if $p = \sigma(\{p, q\}) \neq q$. This relation is called *cyclic* if there are paths p_1 through p_n where

$$p_1 \prec p_2 \prec p_3 \prec \cdots \prec p_n \prec p_1.$$

Otherwise, it is *acyclic*.

2.3.3 IGP-iBGP Anomaly

While BGP can choose the correct egress point in an AS, for each destination, establishment of the intra-AS path to that border router is the responsibility of another protocol (an interior gateway protocol or IGP). Problems can occur if the iBGP configuration does not match the distance values used in the IGP [73, 17].

Consider the scenario in Figure 2.7. The iBGP instance consists of four nodes: the dashed lines denote the BGP links, the solid lines the IGP links, and the numbers along the IGP links denote the link costs. The IGP links carry traffic, whereas the iBGP links exist in order to propagate routing information, so that the IGP can select the correct border router as the egress point for each packet.

The nodes RR_1 and RR_2 are 'route reflectors', and the nodes C_1 and C_2 are clients of RR_1 and RR_2 respectively. This means that they will always choose a route through their local reflector, if possible. However, IGP route decisions are based on minimizing the sum of link weights: so for C_1 to reach RR_1, the actual path taken will be $C_1C_2RR_1$ with a total cost of 2. Similarly, C_2 will try to reach RR_2 by using the path $C_2C_1RR_2$.

The interaction between the IGP forwarding and the BGP routing policy results in a forwarding loop. Packets from C_1 to an external destination will be routed first to C_2, on the assumption that this is the best way to reach RR_1. But C_2 is trying to reach RR_2 instead, and so it forwards the data to C_1. The end result is that the data never gets to either reflector, and does not reach its destination.

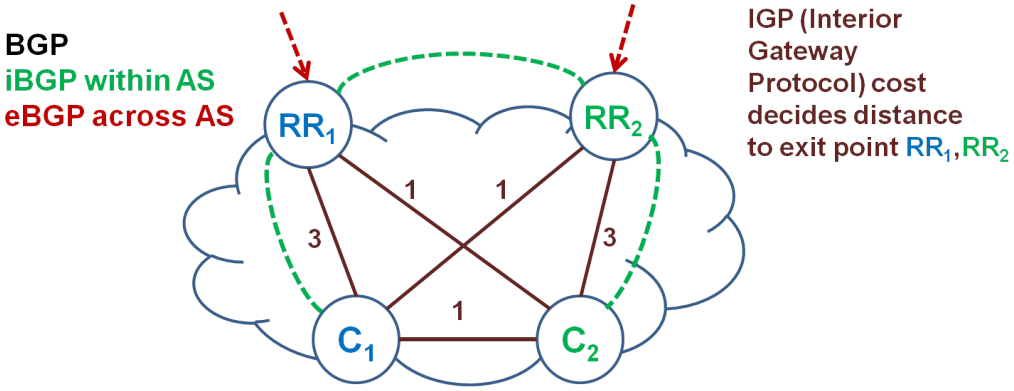

Figure 2.7: IGP-BGP Inconsistency.

Chapter 3

Overview

We have reviewed the algebraic and combinatorial models for BGP systems, and located three sources of anomalies (Chapter 2.3). This chapter presents the overview of *F*ormally *V*erifiable *R*outing (*FVR*) — the toolkit and its underlying theories that address these anomalies with these models.

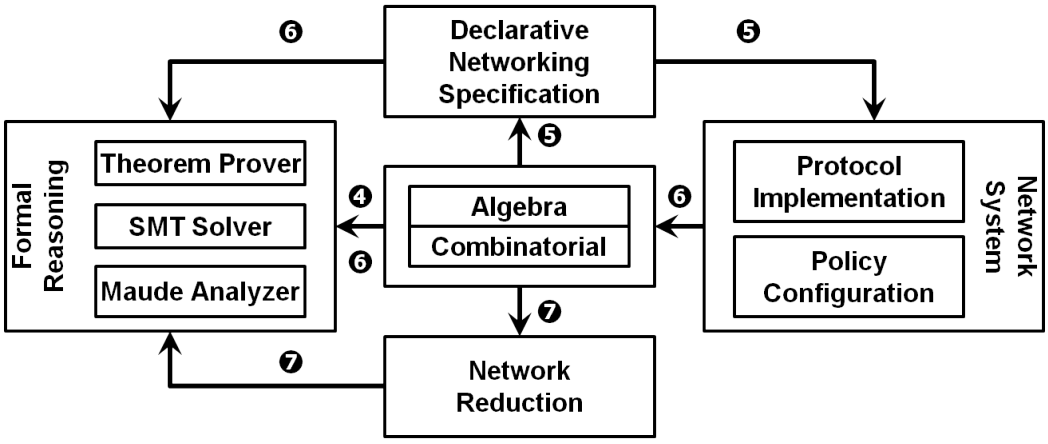

Figure 3.1: Policy-based routing process

As shown in Figure 3.1, the overall goal is to bridge formal reasoning (left) and the actual implementations (right) in the setting of BGP systems. The formal reasoning can be carried out in either an inductive theorem prover, a fully auto-

matic SMT solver, or the Maude analyzer that combines both; The actual systems consist of the implementation of the protocol and the policy configurations. What bridges them are the middle boxes: In the center, the *formal model* box consisting of routing algebra and combinatorial model of SPP, plays a central role. The top middle *declarative network specification* box is the key programming language technique that enables program synthesis and actual system verification. Finally, the bottom *network reduction* box is the scalability technique.

More specifically, starting from the formal model, we designed and implemented Formally safe routing (*FSR*) toolkit [80] that automates **verification of routing models** (arc 4, *Chapter* 5) and **synthesizes faithful implementations** (arc 5) that carry the correctness property [1, 80, 81, 82]. Reversely, starting from the large real-world BGP systems with arbitrary policy configurations, we designed and implemented the two component tool-kits [84, 78, 77] for detecting anomalies in larger BGP systems: the reasoning engine that automates the **analysis of Internet routing configurations** (arc 6), and the scalability component that implements a novel **scalability technique for analysis** (arc 7).

The rest of the thesis will describe in details the four key enabling techniques outlined as follows.

- **Verifying formal network models.** (arc 4) *Chapter* 4 shows the automatic verification of BGP policies that are expressible in the routing algebra model. The verification is achieved by constraint solving using Yices SMT-solver.

- **Generating faithful implementations from verified models.** (arc 5) *Chapter* 5 shows, given a verified routing algebra model, how to convert it into a network datalog (*NDlog*) program, which can be further compiled into actual distributed implementation with the existing declarative networking engine. Correctness proof of the generated *NDlog* programs are checked through inductive reasoning using the PVS theorem prover.

- **Verifying actual routing systems.** (arc 6) *Chapter* 6 presents the verification of actual network systems beyond those expressible in routing algebra models. We develop a more general BGP model in the formal tool Maude with exhaustive-search based analysis functions. This Maude extension allows us to analyze BGP systems with arbitrary policies that are not captured for correctness-by-construction models. The dynamic analysis functions also provide more informative analysis output in the form of a trace that demonstrates routing anomalies.

- **Scalability techniques for analysis.** (arc 7) *Chapter* 7 presents a *configuration rewriting calculus* that transforms network configurations while preserving routing properties. We proved the correctness properties and demonstrate its effectiveness by evaluation on real-world networks. Our evaluation results show that reduction effectively scales up analysis by incurring low overhead. Reduction also uncovers redundancy in network configurations.

Chapter 4

Verifying Formal Models

This chapter presents *FVR*'s ability to verify routing configurations that can be expressed in the routing algebra model. These include general policy guidelines and concrete BGP instances. Hence *FVR*'s automatic analysis serves two important communities: For researchers, *FVR* automates important parts of the design process and provides a common framework for describing, evaluating, and comparing new safety guidelines. For network operators, *FVR* automates the static analysis of internal router (iBGP) and border gateway (eBGP) configurations for detecting safety violations.

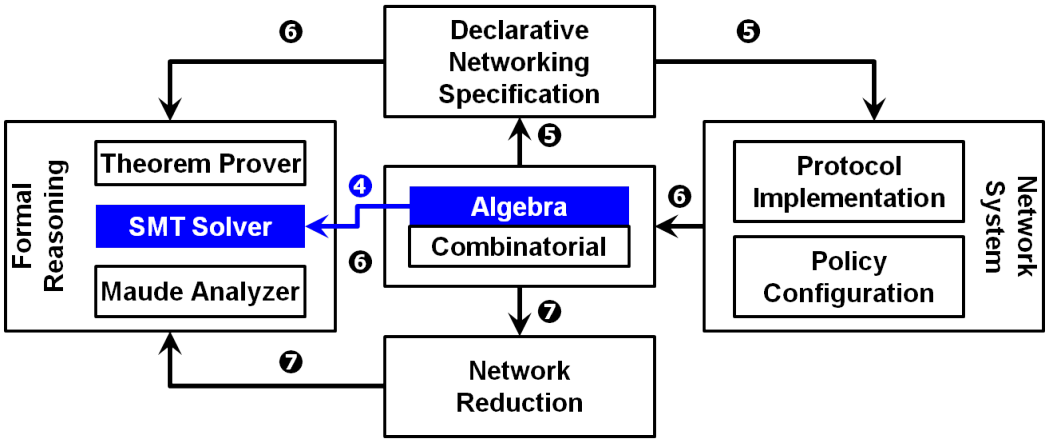

Figure 4.1: *FVR* Architecture: static analysis.

As shown in Figure 4.1, given a policy configurations, *FVR* uses routing algebra model as the representation for analysis, and automates the analysis through constraint solving in SMT solver. These two main enabling technologies are as follows.

- **Policy configuration as algebra:** We first extend routing algebra [25, 66], enabling researchers and network operators to express policy configurations in the abstract algebraic form. These configurations can be anything from high-level *policy guidelines* (e.g., proposed constraints that a researcher wants to study) or a completely specified *policy instance* (e.g., an iBGP configuration or a multi-AS network that an operator wants to analyze). Router configuration files can be automatically translated into the algebraic representation, easing the adoption of *FVR*.

- **Safety analysis:** We then automate the reasoning process of routing algebra by formulate the safety analysis as a *constraint satisfaction* problem, solvable with existing SMT (Satisfiability Modulo Theories) solver [86]. Specifically, we use Yices solver to determine whether it is possible to jointly satisfy the policy configuration and the safety requirement of *strict monotonicity* (the left-most top input in Figure 4.1, drawn from previous work [66]). If all constraints can be satisfied, the routing system is provably safe; otherwise, the solver outputs the smallest subset of the constraints that are not satisfiable to aid in identifying the problem and fine-tuning the configuration.

4.1 Unified Policy Specification

FVR statically checks policy configurations for routing anomalies, and support a wide range of policy configurations inputs: ranging from high-level *guidelines* to specific *instances*, as summarized in Table 4.1. For example, the shortest hop-count

Policy	Topology	Preferences	Filters
Hop-count	General	Specific	None
Gao-Rexford	General	Constrained	Constrained
IGP-cost	Specific	Specific	Constrained
SPP instance	Specific	Specific	Specific

Table 4.1: Spectrum of policy configurations

routing policy does not specify the network topology but *completely specifies* the path preferences; in contrast, the Gao-Rexford guideline merely *constrains* the preferences and filters based on business relationships. In other settings, a researcher may analyze specific BGP "gadgets" that violate a proposed safety guideline; similarly, network operators may verify the safety of their network configuration. In these settings, the topology, preferences, and permitted paths are much more concrete, and can be expressed naturally as instances of the Stable Paths Problem (SPP) [24].

While the existing algebra describe routing policies in terms of the network arcs, described in Chapter 2.2.2, it is not sufficient for specifying policies in terms of the distributed policies on network nodes. For example, the concatenation operator $\oplus$ that expresses the combined results of route filtering at adjacent network nodes, does not indicate which node performs which route filtering—the importing node, the exporting node, or a combination of the two. Such distinction is also crucial in distributed implementation which is also node-based (Chapter 5). To bridge this gap, in this section, we extend the original routing algebra by splitting the original concatenation operator $\oplus$ into three operators: separate operators for import filtering $\oplus_I$, export filtering $\oplus_E$, and the basic concatenation $\oplus_P$.

Using the extended routing algebra, we first show how to specify typical policy guidelines. In the second half of this section, we propose an automatic way to specify policy configuration instances, given its SPP representation: we describe how to translate SPP instances to an algebraic representation. Together, these ex-

tensions enable *FVR* to automatically analyze safety for a wide range of policy configurations.

4.1.1 Extending Routing Algebra

The routing algebra in Chapter 2.2.2 does not distinguish whether routes are filtered during export or import—an important distinction when generating distributed protocol implementations. To specify the two filters separately, we replace the original $\oplus$ operator with three concatenation functions for *export* ($\oplus_E$) and *import* ($\oplus_I$) filtering, and a *simple concatenation* function for route generation $\oplus_P$. The result of $l \oplus_E s$ is either E (export), or F (filtered). For example, if node u does not export routes with signature s to node v, we can encode the export filter as $l \oplus_E s = F$, where the label of link uv is l; otherwise $l \oplus_E s = E$. The result of $l \oplus_I s$ is either I (import) or F (filtered). For example, if u does not import a path with signature s from w, we can encode this import filter as $l \oplus_I s = F$ where l is the label for link wu; otherwise $l \oplus_I s = I$. Whenever an incoming route advertisement is received, the import filter ($\oplus_I$) is first applied. If accepted, a new route is generated ($\oplus_P$), and exported after filtering ($\oplus_E$). [1]

In general, for safety analysis, we need to combine the import and export filters into a single concatenation operator ($\oplus$). At a high level, this is as simple as assigning the signature ϕ (for prohibited paths) to any path filtered by *either* the import or the export policy. However, for a path $vu \circ P_{ud}$, the import filter at v depends on the label l of the link vu, but the export filter at u depends on the label $\bar{l}$ of the reverse link uv. We can generate $\oplus$ as follow: for each label l and signature s, if $\bar{l} \oplus_E s = F$ or $l \oplus_I s = F$, then $l \oplus s = \phi$; otherwise $l \oplus s = l \oplus_P s$.

[1] A similar extension that distinguishes between import and export labels was proposed in [70]. Their approach is equivalent to ours, for the purpose of safety analysis. We chose ours because it provides a straightforward translation to declarative networking implementations.

4.1.2 Converting Policy Guidelines to Algebra

In the Gao-Rexford guideline example, neighboring ASes have a bilateral business relationship, leading to link labels of $\bar{p} = c$, $\bar{c} = p$, and $\bar{r} = r$ and the combined $\oplus$ table shown earlier in Chapter 2.2.2.

Revisiting the Gao-Rexford example from Chapter 2.2.2, the three concatenation operators $\oplus_I$, $\oplus_P$, and $\oplus_E$ are defined as follows:

$\oplus_I$	C	P	R
c	I	I	I
r	I	I	I
p	I	I	I

$\oplus_P$	C	P	R
c	C	C	C
r	R	R	R
p	P	P	P

$\oplus_E$	C	P	R
c	E	F	F
r	E	F	F
p	E	E	E

Each row of the leftmost (rightmost) table corresponds to one import (export) policy in Figure 2.4, from top to bottom; for example, in the rightmost table, the first row exports only customers routes to a provider. Since there are no import restrictions, the leftmost table has I for all its entries. The center table shows the $\oplus_P$ operator, where new routes have their signatures (C, R, and P) set according to the labels (c, r, and p) respectively.

In addition to the above guideline example, our algebra extensions can be used to specify a variety of import and export filter guideline. For example, if the signature includes the entire AS path, we can easily specify an import (export) policy that disallows routes that traverse a particular AS, by expressing $\oplus_E$ ($\oplus_I$) to output F values whenever a route passes through a particular AS.

4.1.3 Converting SPP Instances to Algebra

Researchers and network operators often want to analyze concrete policy configurations to explore small "gadgets" that violate a policy guideline or verify a real network configuration is safe. And we use such "gadgets" as example configurations to show how to represent policy instances in algebra.

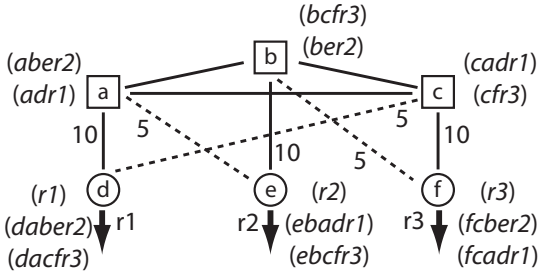

Figure 4.2: iBGP configuration instance

We assumed that each "gadgets" is given in terms of its SPP representation. A SPP instance consists of a topology, where each node has a ranked list of "permitted paths" that it could learn from its neighbors, as described in Chapter 2.2.1. We propose a general process to convert an arbitrary SPP instance into algebra form.

As an illustrative example, Figure 4.2 as presented in [16], shows an gadgets that captures the internal BGP (iBGP) configuration, where the squares (a, b, and c) are route reflectors and the circles (d, e, and f) are egress nodes that each have an externally-learned route (r_1, r_2, and r_3) to the destination. The solid lines denote iBGP sessions (labeled with its IGP cost) and dotted lines denote additional (IGP) links. Each node has an ordered list of permitted paths, ranked from most to least preferred; for instance, node a prefers the route $aber_2$ over adr_1.

To convert an SPP instance to an algebraic representation, we need an automatic way to construct the equivalent link labels, path signatures, preference relations, and concatenation operator. [33] has shown a formal translation of SPP to routing algebra . While the goal of our translation is the same as theirs, the algebra we use is of a slightly different form. At a high level, we assign a unique label to each link, and a unique signature to each path. Then, we convert the ranking of permitted paths into a series of preference relations, and define the concatenation function to connect the permitted paths and exclude any filtered routes.

Using the SPP instance in Figure 4.2 as an example:

Labels and signatures. Since a concrete configuration does not have any meaningful classification of links and routes, we assign each link uv a unique label constant l_{uv}, and each permitted path $p = u_n \cdots u_0$ a unique signature r_p. In our iBGP example, the label set $\mathcal{L}$ is $\{l_{ab}, l_{ac}, l_{ad}, l_{ba}, l_{bc}, l_{be}, l_{ca}, l_{cb}, l_{cf}, l_{da}, l_{eb}, l_{fc}\}$, and the signature set Σ is $\{r_1, r_2, r_3, r_{aber2}, r_{adr1}, r_{bcfr3}, r_{ber2}, r_{cadr1}, r_{cfr3}, r_{daber2}, r_{dacfr3}, r_{ebadr1}, r_{ebcfr3}, r_{fcber2}, r_{fcadr1}\}$.

Preference relations. Each node has a ranked list of permitted paths, of the form $r_1, r_2, ...r_n$. We translate this list into the equivalent pairwise preferences: $r_1 \prec r_2, r_2 \prec r_3, ...r_{n-1} \prec r_n$. For instance, at node a, $r_{aber2} \prec r_{adr1}$. The preference relation is defined as the collection of preference relations at each node.

Concatenation. The $\oplus$ operator constrains the relationship between label and signature constants. In particular, for any permitted path r_{uvp} at node u, $r_{uvp} = l_{uv} \oplus r_{vp}$; for instance, $r_{aber2} = l_{ab} \oplus r_{ber2}$. Any other paths are disallowed by assigning the signature ϕ; for instance, $l_{cb} \oplus r_{ber2} = \phi$ because path $cber_2$ is not listed as a permitted path.

Using the above process, we have also encoded various eBGP gadgets [24] in algebra as SPP instances.

4.2 Automated Safety Analysis

Given any algebra, *FVR* fully automates the process of safety analysis, relieving users from the manual and error-prone process of proving safety for each new algebra. The key insight is that the safety analysis can be translated automatically into integer constraints checkable by a standard SMT solver. Applying our technique of encoding SPP instances using algebra (Section 4.1.3), *FVR* can check safety for both high-level policy guidelines and concrete configurations. After a brief review of safety analysis based on routing algebra, we explain how to generate the

integer constraints and present three examples that illustrate the conversion process and resulting safety analysis.

4.2.1 Strict Monotonicity Implies Safety

FVR uses the safety requirement of *strict monotonicity*, in order to automatically check that a given policy configuration converges. This is an important property of a routing algebra, which ensures that a path does not become more preferred as it grows longer. Formally:

Monotonicity: $s \preceq l \oplus s,\ \forall l \in \mathcal{L}, \forall s \in \Sigma$

Monotonicity ensures that a path P_v from v to the destination is not less preferred than a longer path $uv \circ P_v$, where uv is a link from u to v. In the definitions above, s represents the signature of P_v and l is the label for uv.

FVR uses the stricter form of monotonicity, where l and $l \oplus s$ cannot be equally preferred, defined as follows:

Strict Monotonicity: $s \prec l \oplus s,\ \forall l \in \mathcal{L}, \forall s \in \Sigma$

Sobrinho has proved the following theorem [66].

Theorem 2. *If the routing algebra is strictly monotonic, then the path-vector protocol converges.*

Theorem 2 reduces the convergence analysis of protocols to modeling the routing policies in a routing algebra, and proving that the algebra is strictly monotonic. Note that strict monotonicity is a *sufficient*, not *necessary* condition. Hence, there are safe systems that cannot be specified as a strictly monotonic algebra. Consequently, *FVR* will report these systems as unsafe (i.e. false positives). However, this sufficient condition is still very useful in practice to analyze the safety of BGP systems.

Note that the strict monotonicity is actually the most general condition known that guarantees safety regardless of the network topology. This enables researchers

and network operators using *FVR* to benefit from this theoretical result by providing automated tool support.

4.2.2 Converting Policies to Yices Constraints

Policy configurations expressed in routing algebra have a natural representation as integer constraints. Path signatures can be mapped to integers, and path preferences can be expressed as comparisons ($\leq$) between these integers. By definition [66], the preference relation $\preceq$ needs to be a total relations, and $\leq$ is indeed a total order. This mapping is also complete because we can always map the signatures onto the integer domain, when the $\preceq$ is a total order. Strict monotonicity imposes additional constraints on the preference relation, also naturally captured by comparing integers. This observation allows *FVR* to leverage SMT solvers, which determine whether a set of constraints (i.e., first-order logic formulas) are satisfiable based on a set of theories (e.g., integer theory). Translating from algebraic input to SMT constraints is straightforward, making this approach preferable to other alternatives (e.g., SAT solvers) that would require greater effort to generate encoding.

In addition, an SMT solver produces valuable output, beyond the basic "yes/no" answer. If the constraints can be satisfied, the solver returns a concrete instance (example) that satisfies all of the constraints. For instance, we consider the simple constraint $x < 2$ when x is an integer. An SMT solver can prove that there exists a value instantiating x that makes $x < 2$ true, and returns $x = 1$ as an example. If the constraints cannot be satisfied, the solver returns the smallest subset of constraints that are not satisfiable—an invaluable aid in identifying the problematic parts of the policy configuration. In our *FVR* implementation, we utilize the Yices [86] SMT solver, although the technique we present here can be applied to SMT solvers in general. Our technique generalizes to other SMT solvers well because our encod-

ing only requires the basic integer theory which is readily included in most SMT solvers.

Input to SMT solver: Given a policy configuration written in routing algebra, *FVR* generates integer constraints for safety analysis recognizable by the solver. *FVR* generates two kinds of constraints based on the sufficient conditions required for safety in Section 4.2.1: (1) route preference constraints based on $\preceq$ relation (2) strictly monotonic constraints based on $\oplus$ function. *FVR* automatically generates these integer comparison constraints, allowing us to leverage Yices built-in integer support for enforcing total ordering. More concretely, we generate constraints from the algebraic specification via the following three steps:

- **Step 1:** For each signature, we define a variable of the type positive integer.
- **Step 2:** For each $s_1 \preceq s_2$ in the specification, we generate a constraint $s_1 \leq s_2$. Since signatures are integers, the $\leq$ relation imposes a total ordering.
- **Step 3:** For any signature s and s', and label l, for each definition of $s' = l \oplus s$ in the specification, a constraint $s < s'$ is generated. This constraint enforces strict monotonicity. To check for (non-strict) monotonicity, we could generate $s \leq s'$ instead.

SMT solver output: The conjunctions of all constraints are checked by Yices for satisfiability. If Yices returns `sat`, an assignment of integers to variables (signatures) exists that satisfies all of the constraints. This means that the algebra is strictly monotonic, and by Theorem 2, any path-vector protocol that implements the policy configurations converges.

On the other hand, if Yices returns `unsat`, specific input constraints that form an *unsatisfiable core* are provided. Unsatisfiable core (or unsat core) is a minimal set of inputs constraints that cannot be conjunctively satisfied. It is often significantly smaller than the set of input constraints.

Given the natural mapping of the original input specifications in algebra and Yices constraints, one can easily identify the preference relation for each violating constraint. The user can use these violating preferences as hints to identify (and fix) specific problematic parts of the policy configuration. Note that, there can be multiple unsatisfiable cores (i.e. many configuration conflicts), and Yices only outputs one of them at each invocation. To fix all the configuration problems, the user can attempt removing all unsatisfiable cores one by one in an iterative fashion.

Policy compositions: The lexical product (Chapter 2.2.2) of a monotonic algebra and a strict monotonic algebra is strictly monotonic [25]. For policy configurations in the form of a lexical product over algebras, safety analysis can be performed by analyzing each algebra separately. Consider the lexical product $A \otimes B$ of two algebras A, B. Analysis starts from algebra A, and if it is strictly monotonic, the composed policy is safe. If A is monotonic, then B is checked. If B is strictly monotonic, then the composed algebra is safe, otherwise it is deemed unsafe. If A is not even monotonic, then the composed policy is deemed unsafe.

4.2.3 Yices Examples

We present several examples to demonstrate the three-step process of generating Yices constraints from algebraic input and the analysis process.

Shortest Hop-Count: We start with the simplest example using shortest hop-count. The algebraic specification of this policy is presented in Chapter 2.2.2. We show the Yices encoding of the constraints below:

```
(define-type Sig (subtype (n::nat) (> n 0)))
(assert (forall (s::Sig) (< s s+1)))
```

The first line declares a type (`Sig`) for signatures, which is the subset of positive integers. Yices provides the built-in type `nat` for integers. Yices uses prefix syntax, so $n > 0$ is encoded as $(> n\ 0)$. Step 1 and 2 are omitted since the signatures are already integers, and the preference relation $\preceq$ is already specified using $\leq$.

The second line corresponds to step 3, and encodes the strict monotone constraint. `assert` is the keyword to tell Yices to insert this constraint into the logical context to be checked for satisfiability. Since the domain of the signatures is infinite, we cannot enumerate all strict-monotonicity constraints; instead, we universally quantify using `forall` over all signatures.

As expected, Yices returns `sat` for this policy.

Gao-Rexford Guideline A: Our second example analyzes the safety of Gao-Rexford guideline A, with the routing algebra presented earlier in Section 2.2.2. The constraints are expressed in Yices as:

```
(define-type Sig (subtype (n::nat) (> n 0)))
(define C::Sig)    (define P::Sig)    (define R::Sig)
;; preference relations
(assert (< C R))   (assert (< C P))   (assert (= R P))
;; strict monotonicity
(assert (< C C))   (assert (< C R))   (assert (< C P))
(assert (< R P))   (assert (< P P))
```

The first four statements define the three classes of signatures—customer (C), provider (P), and peer (R)—as positive integers (step 1). The next three constraints correspond to step 2, encoding the route preference constraints of $C < R$, $C < P$ and $R = P$. The next five constraints correspond to entries in the combined concatenation operator in Section 2.2.2 after omitting constraints in the form $S < \phi$, which are already ensured to be true because any signature is strictly preferred

over the signature for prohibited path ϕ by definition. This corresponds to step 3, which encodes strict monotonicity of the algebra.

Interestingly, Yices returns `unsat` for the above input, indicating that the algebra is not strictly monotonic. One of the violating constraints is resulted from $c \oplus C = C$, which states that a customer route that is sent from a customer link is still a customer route. This is a known property of the Gao-Rexford guideline, which requires an additional constraint on acyclicity in the customer-provider relationship for safety.

Another approach to guaranteeing safety in Gao-Rexford is to use another algebra that is strictly monotonic as the tie breaker, in the event of a tie between two equally preferred route classes (e.g., provider and peer, or routes from same classes). As an example of policy composition, we first use Yices to prove that the algebra encoding guideline A is *monotonic* ($s \preceq l \oplus s$ $\forall l \in \mathcal{L}, \forall s \in \Sigma$), and then compose guideline A with a strictly monotonic algebra such as shortest hop-count; the resulting protocol converges.

To perform the above analysis, we modify the strict monotonicity constraints in the above Yices encoding to check for monotonicity constraints. This requires changing each $<$ to $\leq$, e.g. `(assert (<= C C))`, etc. When we check the constraints, Yices returns `sat`, and provides a possible instantiation `C=1`, `P=2`, `R=2`.

In addition to guideline A, we have applied *FVR* to analyze a number of guidelines including Gao-Rexford guideline B [19] and also guidelines that ensure safe backup routing [18].

Internal BGP Configuration Instance: As our final example, we use *FVR* to analyze the six-node iBGP configuration in Figure 4.2, using our technique for encoding SPP instances in algebra (Section 4.1.3). In Section 7.3, we present our experiences analyzing a larger network based on the Rocketfuel [68] dataset, and also the analysis of well-known eBGP gadgets.

We use the same three-step process to generate solver constraints. First, each permitted path in the SPP instance is mapped to an integer variable. Second, a constraint is generated for each route preference, derived from the per-node rankings in SPP. Finally, for each entry of the concatenation function, we generate a strict monotonic constraint as described in step 3. All in all, eighteen constraints are generated.

The number of constraints depends on the number of permitted paths which, in turn, depends on the network topology. In contrast, the previous two examples are independent of the network topology. These differences reflect the broad applicability of *FVR*, in analyzing policy configurations that range from partially to fully specified (see Table 4.1).

For these input constraints, Yices returns `unsat`, meaning that the algebra violates strict monotonicity. In fact, this iBGP system is known to be unsafe [16].

However, given eighteen constraints, pinpointing the problem manually is quite difficult. For larger networks with even more constraints, manual analysis becomes even harder. Fortunately, Yices can generate the minimal set of constraints (unsat core) that cannot be satisfied. More details on pinpointing configuration problems with unsat core are in Section 5.4.2.

Here, the unsat core includes the rankings of nodes a, b and c's and strict monotonicity constraints involving available routes of those nodes, but does *not* include constraints for the route preferences of node d, e, and f. This leads to the conclusion that there are potential problems with the configurations of the route reflectors a, b, and c. In fact, each reflector prefers other reflector's client over its own, which causes an oscillation [16].

Once the problem is identified, the network operator can change the network settings such as topology, so that the route preferences of nodes a, b, and c are changed, and use *FVR* to analyze the safety of the new configuration. As vali-

dation, we rerun Yices with a modified configuration that does not include the preference cycle among the reflector nodes, and the solver returns `sat`.

Soundness of SPP Safety Analysis For an SPP instance, each AS only knows the preference relation among the routes that are in its own routing table, and the policy configurations do not enforce any order among routes that originate from different nodes. However, the safety analysis of the routing algebra requires a total ordering of all routes. The output of `sat` means that there exists one strictly monotonic algebra that extends the route preference relation specified in policy configurations to be a total order. Applying Theorem 2 directly, we know that a protocol that implements this extended algebra is safe. We argue that a protocol that implements the extended algebra has exactly the same behavior as a protocol that implements the original policy configurations where the preference relation is only a partial order. The reason is that additional preferences in the extended relation describes preferences between routes that have different sources, which are not relevant to the route selection process in practice anyway; and therefore will not affect the protocol behavior.

4.3 Evaluation

We present several case studies of using *FVR* to automatically generate a proof of safety or pinpointing configuration problems of both policy guidelines and specific instances.

4.3.1 Pinpoint iBGP Configuration Errors

We emulate a scenario where a network operator uses our *FVR* toolkit to study the safety properties of an existing iBGP network configuration. As the input topology, we utilize the intra domain topology (with inferred link weights) of AS 1755

from the Rocketfuel [68] dataset, which contains 87 routers and 322 links. Pairwise IGP costs are computed a priori based on the shortest paths. The iBGP reflector-client topology is synthetically configured as a 6-level hierarchy with 53 reflectors. Given the above input topology, on all 87 routers, and have each router within the AS compute the best route to a remote destination outside the AS, under the condition that several egress routers are aware of external routes to this particular destination. At each router, the route preference is based on the IGP cost from the router to the egress routers, i.e., the route with the lowest IGP cost is selected. This policy is similarly configured using routing algebra. The algebraic representation of the SPP instance of the network is extracted and analyzed for safety.

In the absence of real router configurations, we extract the per-node rankings from *NDlog* implementation runs as follows. We execute the GPV protocol in *NDlog* on all 87 routers, and populate the permitted paths of each router based on its incoming route advertisements. These permitted paths are then sorted based on IGP costs described above, to generate per-node rankings. *FVR* directly translates these per-node rankings expressed in algebra into constraints used by our SMT solver to perform safety analysis.

In total, the extracted SPP instance contains 259 constraints generated for strict monotonicity, and 292 constraints for per-node rankings. On a quad-core machine in our testbed, the SMT solver returns `unsat` within 100 ms, and reports a minimal unsatisfiable core consisting of six constraints. Interestingly, these six constraints not only form a dispute wheel, but are also directly attributed to the routers in the embedded gadget that we deliberately introduced earlier. This provides a "hint" for network operators to fix the configuration error starting from the errant constraints.

4.3.2 eBGP Gadget Analysis

FVR's applicability extends beyond high-level guidelines and iBGP configurations. We briefly summarize our experiences of using *FVR* to analyze well-known eBGP gadgets: GOODGADGET, BADGADGET and DISAGREE [24]. These experiments highlight the use of *NDlog* implementations generated from *NDlog* implementations of SPP instances.

The input algebra for these three gadgets are SPP instances described in Section 4.1.3, where the algebra is used to encode per-node permitted paths and rankings. Our analysis results are as expected: GOODGADGET is safe, while BADGADGET and DISAGREE are unsafe. These results match the manual proofs in prior work [24], but are obtained automatically by our solver.

4.4 Summary

This chapter present *FVR*'s ability to automate the verification of a wide range of routing configurations that can be expressed in routing algebra model. These include high-level guidelines to specific router configurations. We show that routing algebra has a natural translation to both *integer constraints* (to perform safety analysis with SMT solvers). We also preformed extensive experiments with realistic topologies and policies, showing that by SMT solving, *FVR* can detect problems in an AS's iBGP configuration, prove sufficient conditions for BGP safety.

Chapter 5

Synthesizing Faithful Implementations

Once the routing algebra is verified (Chapter 4), *FVR* compiles it into a provably-correct emulated implementation, as shown in Figure 5.1. The resulting implementation can be used for further empirical study prior to actual deployment. We also prove that the generated implementation is faithful to the algebra.

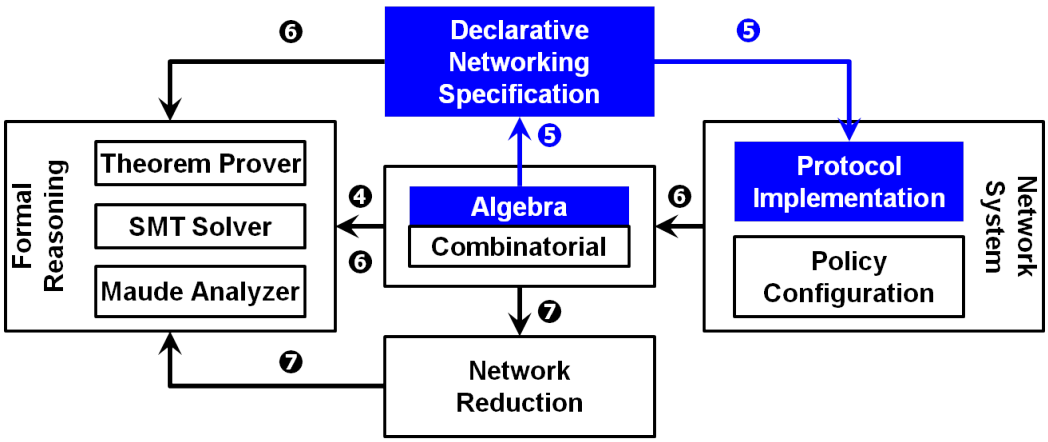

Figure 5.1: *FVR* Architecture: Implementation-based Analysis.

The basic idea is to synthesize a provably-correct implementations of safe in-

ter domain routing by unifying research in *routing algebras* [25, 66] with recent advances in *declarative networking* [39, 38, 41, 40, 53, 48] . Specifically, given the verified configuration's algebra form and a formal description of the path-vector mechanism, *FVR* maps it to a *Network Datalog* (*NDlog*) specification, which is then executed using the RapidNet declarative networking engine [2, 49], thus *FVR* automatically generate a distributed routing-protocol implementation that matches the policy configuration—avoiding the time-consuming and error-prone task of manually creating an implementation.

5.1 Background: Declarative Networking

FVR uses a declarative networking language called *Network Datalog* (*NDlog*) as an intermediary language to bridge the gap between the abstract routing algebra and efficient distributed implementations. Our choice of *NDlog* is motivated by the following. First, the declarative features of *NDlog* allows for straightforward translation from the algebra to *NDlog* programs. Second, *NDlog* enables a variety of routing protocols and overlay networks to be specified in a natural and concise manner. In fact, *NDlog* specifications are orders of magnitude less code than imperative implementations. For example, traditional routing protocols such as the path vector and distance-vector protocols can be expressed in a few lines of code [41], and a more complex protocol such as the Chord distributed hash table can be expressed in 47 lines. This makes possible a clean and concise proof (via logical inductions) of the correctness of the generated *NDlog* programs with regard to the algebra. The compact specifications also makes it easy to incorporate alternative routing mechanisms to the basic path-vector protocol, as we will later demonstrate in our evaluation section. Finally, when compiled and executed, these declarative protocols perform efficiently relative to imperative implementations [39].

In *FVR* prototype, we use the open-source RapidNet [2] declarative network-

ing engine as a basis for executing *NDlog* programs. *NDlog* programs are compiled by RapidNet into distributed execution plans that are based on the Click [35] execution model. Our generated *NDlog* implementation is composed of two components: one implements the routing mechanisms, the other implements the routing policies. *FVR* provides a built-in module implementing the path-vector mechanism, which we discuss in detail in Section 5.1. The component implementing policies is directly translated from the algebra. In Section 5.2, we show how *FVR* translates the algebra into *NDlog* programs.

Generalized Path Vector Mechanism *FVR* takes as input, a *generalized path-vector* protocol, as its default routing mechanism. The *NDlog* implementation is shown below, and for the rest of this thesis we refer to it as the GPV program. GPV implements a path-vector protocol that computes the most preferred path based on a routing algebra.

NDlog is a distributed variant of Datalog. An *NDlog* program is composed of several *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and `...` and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1, q2,...,qn` is a list of *predicates* that constitutes the *body* of the rule. A rule is triggered (evaluated) once all the body predicate values (tuples) are generated. Once triggered, the head tuple is generated. Rule execution is done in a continuous, long-running fashion using a distributed query processor, where rule head tuples are continuously updated (inserted or deleted) in an incremental fashion [45] as the body tuples are updated.

```
//GPV program
gpvRecv sig(@U,SNew,PNew) :- msg(@U,V,D,S,P),
PNew=f_concatPath(U,P), V=f_head(P),
SNew=f_concatSig(L,S), label(@U,V,L),
f_import(L,S)=true.
```

```
gpvStore route(@U,D,S,P) :- sig(@U,S,P), D=f_last(P).

gpvSelect localOpt(@U,D,a_pref<S>,P) :- route(@U,D,S,P).

gpvSend msg(@N,U,D,S,P) :- localOpt(@U,D,S,P),
label(@U,N,L), f_export(L,S)=true.
```

In *NDlog*, the names of predicates, function symbols, and constants begin with a lower-case letter, while variable names begin with an upper-case letter. Similar to most implementations of Datalog, *NDlog* includes a limited set of function calls beginning with "f_", and user-defined arithmetic functions beginning with "a_". These functions include boolean predicates, arithmetic computations, and simple list operations.

The above program manipulates the following tuples. `label(@U,V,L)`[1] tuples, where each tuple represents an edge from the node itself (`U`) to one of its neighbors (`V`) of attribute `L`. A set of computed routes, stored as `sig(@U,S,P)` tuples at each source node `U`, where `S` and `P` are the signature and path of the route respectively. Route advertisement messages exchanged among nodes are represented by `msg(@U,V,D,S,P)` tuples. Each tuple denotes a message that is sent by node `V` to `U`, and the advertised route is for destination `D` with path `P` and signature `S`. We provide a high-level description of the above program, broken down by rules:

- **Receiving routes.** Rule `gpvRecv` is triggered upon receiving a route advertisement (`msg` tuple) from a neighboring node. Based on the route advertisement, the rule generates a new route with a new path `PNew` and a new signature `SNew`. The `f_concatSig` implements the simple concatenation function $\oplus_P$, while the function `f_import(L,S)`, implements the import filter $\oplus_I$ in algebra. It evaluates to true if and only if $\texttt{L} \oplus_I \texttt{S} = I$.

[1] *NDlog* supports a *location specifier*, expressed with "@" symbol followed by an attribute. This attribute is used to denote the source location of the corresponding tuple. For example, `label` tuples are stored based on the value of the `U` attribute.

- **Storing routes.** Rule `gpvStore` builds a `route` table at each node, which stores all the candidate routes to the destination, by using the information in its locally maintained `sig` table.

- **Selecting routes.** Rule `gpvSelect` computes the optimal route (represented as `localOpt` tuples) based on the `route` table. The user-defined aggregate function `a_pref` computes the optimal route by using the route preference function `f_pref` (as its comparison function), which implements the $\preceq$ relation in algebra.

- **Sending routes.** Rule `gpvSend` propagates new routes to neighbors. Whenever a node's local optimal routes `localOpt(@U,D,S,P)` to destination `D` is updated, the updated route is re-advertised to all neighbors `N`. Similar to import policies, we use the `f_export` function to filter out routes: rule `gpvSend` only generates a message if the route is not filtered by the export policy. `f_export(L,S)` implements $\oplus_E$, and it returns true if and only if `L`$\oplus_E$`S`$= E$.

GPV provides a template for users to plug in customize policy configurations. One of the advantages of using *NDlog* is its ease of incorporating routing policies in algebraic form with routing mechanisms (e.g. GPV). Signature generation is achieved by performing a predicate unification of labels and signatures recursively in *NDlog* rules, and applying the appropriate function (`f_concatSig`) for generating new signatures. The recursive signature generation (from other signatures) is encoded in only 4 rules in *NDlog*. Import and export filters are simply boolean functions (`f_export`, `f_import`) in rule bodies which are triggered when true. While it is certainly possible to use an imperative language instead, *NDlog* provides the right balance of features in terms of compact specifications, ease of proofs and translation from algebra.

Algebra	*NDlog* Predicates / functions
$\preceq$	f_pref
$\oplus_P$	f_concatSig
$\oplus_I$	f_import
$\oplus_E$	f_export

Table 5.1: Algebra and *NDlog* Mapping.

5.2 Generating Faithful NDlog Implementation

Table 5.1 summaries the correspondence between definitions in algebra, and the function names in the generated *NDlog* programs. We use the extended algebra introduced in Section 4.1.1, which distinguishes between simple concatenation function $\oplus_P$, import filter $\oplus_I$, and export filter $\oplus_E$. Functions `f_pref`, `f_concatSig`, `f_import`, and `f_export` are directly generated from input routing algebra as follows:

- **Step 1.** For each $s_1 \preceq s_2$ in the specification, add a statement to `f_pref(S1,S2)` that returns true if `S1`$= s_1$ and `S2`$= s_2$.
- **Step 2.** For any signature s and s', and label l, for each definition of $s' = l \oplus_P s$ in the specification, generate a statement in `f_concatSig(L,S)` that returns s' if `L`$= l$ and `S`$= s$.
- **Step 3.** For any signature s and label l, if $l \oplus_E s = F$, generate a statement in `f_export(L,S)` that returns `false` if `L`$= l$ and `S`$= s$. Similarly define `f_import(L,S)` for import filter $\oplus_I$.

To deploy the *NDlog* implementation on a concrete topology, each router takes additional configuration information automatically generated from the topology:

- **Step 4.** For each link in the input topology, generate a corresponding `label` tuple (assigned a value from the set $\mathcal{L}$). A `sig` tuple is also generated for

each one-hop path to the destination. Signatures associated with these one-hop paths are typically known as the *origination set* [25], a subset of Σ defined as part of the input algebra.

Note that the above steps can be generated on a *per-node* basis, based on each node's input algebra. If the algebra directly uses functions and relations that *NDlog* has built-in support (e.g. integer arithmetic), then steps 1 to 3 can simply use *NDlog*'s built-in functions.

Policy Composition: If the network designers choose to use the compositional feature of the routing algebra, the compositional operators can be straightforwardly mapped to *NDlog* templates as well. In particular, the lexical product of two policy algebras can also be concisely represented in *NDlog* by encoding the labels and signatures as a pair, and customizing the `f_pref` comparator function to check the first attribute, and then the second attribute in the case of a tie-breaker.

NDlog Examples We present examples to demonstrate the process of generating *NDlog* programs from input algebra.

Shortest Hop-Count: For shortest hop-count, the label for each link is `1`, so for a node `u`, for each of `u`'s neighbor `v`, *FVR* generates a tuple `label(@u,v,1)`. If `u` has a direct link to the destination `d`, then a tuple `sig(@u,1,[ud])` is defined. This completes Step 4.

Next the concatenation and preference function are generated (Step 1 and 2). The concatenation function is defined as integer addition, and the preference relation is integer $\leq$ relation.

```
#def_func f_concatSig(L,S) { return L+S }
#def_func f_pref(S1,S2) { return S1 <= S2 }
```

Finally, the shortest hop-count policy does not have any import or export filtering, so they are the constant `true` function (Step 3).

```
#def_func f_export(L,S) { return true }
#def_func f_import(L,S) { return true }
```

Gao-Rexford Guideline A:

Based on the network topology, for a node `u`, *FVR* generates a `label(@u,v,ch)` tuple for each of its neighbor `v`, and `ch` is 'c' if `v` is `u`'s customer; 'p', if `v` is `u`'s provider; and 'r', if `v` is `u`'s peer. Similarly, for each initial route of length 1, `sig(@u,ch,[ud])` is defined and `ch` is 'C' if the link `ud` is a customer link; 'P', if `ud` is a provider link; and 'R', if `ud` is a peer link. This corresponds to Step 4.

Next, in Step 1 and 2, definitions for functions implementing $\oplus_P$ and $\preceq$ are generated as follows.

```
#def_func f_concatSig(L,S) {
if (L=='c') && (S=='C') return 'C'
if (L=='c') && (S=='P') return 'C'
if (L=='c') && (S=='R') return 'C'
if (L=='p') && (S=='C') return 'P'
.... }

#def_func f_pref(S1,S2) {
return (S1=='C' && S2=='R') || // C < R
(S1=='C' && S2=='P') // C < P }
```

`f_concatSig` returns the signature `S` based on the link `L`, as defined by the earlier input algebra $c \oplus_P * = C$, $p \oplus_P * = P$, $r \oplus_P * = R$, where $*$ stands in for any signature C, P, or R. For each entry in $\oplus_P$, *FVR* generates an `if` clause, and we omit the rest of the definitions. `f_pref` returns `true` if `S1` is a customer route (C). This forces a customer route to be preferred over a peer/provider routes (R and P respectively). This is a direct translation from the earlier input algebra for Gao-Rexford, namely $C \prec P$ and $C \prec R$.

Finally, import and export functions are generated based on the filters $\oplus_I$ and $\oplus_E$. Since guideline A does not specify import filters, `f_import` is the constant function that always returns `true`. The export function returns `true` if the route is not filtered ($l \oplus_E s = E$); `false` if the route is filtered ($l \oplus_E s = F$).

```
#def_func f_import(L,S) { return true }
#def_func f_export(L,S) {
if (L=='c' && S=='P') return false
if (L=='c' && S=='R') return false
if (L=='r' && S=='P') return false
if (L=='r' && S=='R') return false
return true }
```

SPP Instances: Since SPP imposes explicit rankings, the `f_pref` function would compare signatures for a given source/destination pair. Based on per-node rankings of paths, `f_pref(S1,S2)` will return true if `S1` is preferred over `S2`, and false otherwise. To speed up the comparison process, one possible optimization (enhancement to step 2) is to store the per-node rankings in an ordered table for fast retrieval. Similarly, for export filters, one can maintain a table of permitted paths to be exported, and the `f_export` simply checks that a particular path is in the permitted export list, before it is exported. Import filters can be implemented similarly.

5.3 Correctness of NDlog implementation

In order to apply Theorem 2 and show that the *NDlog* implementation of a strictly monotonic algebra converges, we need to show the correctness of the *NDlog* implementation. The correctness depends on two conditions: first, the *NDlog* program correctly implements the path-vector protocol, and second, the *NDlog* program correctly implements the input algebra. Prior work has experimentally val-

idated [39] and formally proved [76] the correctness of an *NDlog* implementation of the path-vector protocol. We will revisit the verification of declarative network in Section 6.2. In addition, [52] has formally proved correct *NDlog*'s operational semantics. We hence focus on the second condition.

We introduce several notations to set up our proofs. We define ι to be a function that maps the set of links in the network topology to the set of labels in $\mathcal{L}$. Given a concrete network topology, ι is the correct assignment of labels to links, i.e. $\iota(uv) = l$ if the label of link uv is l. The function σ_0 maps initial routes (route of length 1) to their signatures. σ_0 is the correct signature assignments to initial routes. Given a destination d, $\sigma_0([ud]) = s$ if the signature of route $[ud]$ is s.

Given ι, σ_0 and an algebra $\mathcal{A}$, function $\sigma_{\iota,\sigma_0,\mathcal{A}}$ maps each route to its signature. When it is clear from the context, we omit the subscripts, and write σ.

$$\sigma(p) = \begin{cases} \sigma_0(p) & p = [\text{ud}] \\ \iota(uv) \oplus \sigma(p') & p = uv \circ p' \end{cases}$$

Finally we define a function $nd(t)$ that returns the *NDlog* term that represents t. A key aspect is to prove that *NDlog* computes the signatures for routes correctly, formally:

Theorem 3 (Correctness of *NDlog* translation)**.** *Given any path p, if $\texttt{sig}(nd(u), nd(s), nd(p))$ is generated by prog, and $s \neq \phi$, then $s = \sigma(p)$.*

Proof of correctness To prove Theorem 3, we make a few assumptions. First, ι and the complement of the label operation $\bar{l}$ has the property that the label assigned to a link vu is the complement of the link assigned to uv.

(**Property A**): $\overline{\iota(uv)} = \iota(vu)$.

For example, in the Gao-Rexford guideline, the reverse direction of a customer link is a provider link.

We also assume that generated functions and predicates faithfully implement the algebraic specifications, which are formally stated below.

(**Property B**):

- `f_export`(*nd*(l), *nd*(s)) = `true` iff $l \oplus_E s = E$.
- `f_import`(*nd*(l), *nd*(s)) = `true` iff $l \oplus_I s = I$.
- `f_concatSig`(*nd*(l), *nd*(s)) = *nd*(s') iff $l \oplus_P s = s'$.
- `f_pref`(*nd*$(s1)$, *nd*$(s2)$) = `true` iff $s1 \preceq s2$.
- `label`(@*nd*(u), *nd*(v), *nd*(l)) `:-` `.` is in *prog* iff $\iota(uv) = l$. `sig`(@*nd*(u), *nd*(s), *nd*(p)) `:-` `.` is in *prog* iff $\sigma_0(p) = s$.

Given an algebra $\mathcal{A}$, and a network topology represented by ι and σ_0, let *prog* be the *NDlog* program that is translated from $\mathcal{A}$, and ι and σ_0.

We first prove the following lemma (containing two parts i and ii), which state that the generated signature tuples are correct; and that if a route update message is generated, then the route's signature is correctly computed, and the export policies have been applied:

Lemma 4.
(i) Given any path p, if `sig`*(nd*(u), *nd*(s), *nd*(p)*) is generated by prog, and $s \neq \phi$, then $s = \sigma(p)$.*
(ii) Given any path p, if `msg`*(nd*(u), *nd*(v), *nd*(d), *nd*(s), *nd*(p)*) is generated by prog, and $s \neq \phi$, then $s = \sigma(p)$, and $\iota(vu) \oplus_E s = E$.*

Proof (sketch): By induction of the length of p.

We abbreviate *nd*(t) to `t` when it is clear from the context that *NDlog* representation of t is required.

In the base case, the length of p is 1. We know that $\sigma(p) = \sigma_0(p)$ and by our assumptions (Property B), we know that if `sig(u,s,p)` is generated by *prog*, then $s = \sigma_0(p)$. So part (i) holds.

By examining the GPV program, `msg(n,u,d,s,p)` tuple is only generated when `gpvExport` is applied. So we know that $s = \sigma(p)$ (s and p comes from the `sig` tuple), and that `f_export(l,s) = true` and `label(u,n,l)` is true. Use Property B again, we know that $l \oplus_E s = E$, and $\iota(un) = l$, so part (ii) holds.

In the inductive case, to prove part (i), we examine the `gpvSig` rule. The new path and signature is generated from the tuple `msg(u,v,d,s,p)`. Using induction hypotheses, we know that $s = \sigma(p)$, and $\iota(vu) \oplus_E s = E$. If a new `sig(u,snew,pnew)` is generated, then it must be the case that `f_import(l,s) = I` where `l` = $\iota(uv)$. Using property A, we know that $\overline{\iota(uv)} = \iota(vu)$. By examining the way we generate $\oplus$ from $\oplus_p$, $\oplus_E$ and $\oplus_I$, we know that `snew` = $\iota(uv) \oplus \sigma(p)$, which is equal to $\sigma(pnew)$ (Property B). We can prove part (ii) in similar ways as we prove part (i) in the base case. □

Lemma 4 implies Theorem 3.

5.4 Evaluation

We present several case studies of using *FVR*: (1) empirically evaluating protocol dynamics and temporal properties that cannot be easily checked in formal analysis, and (2) deploying and evaluating alternative routing mechanisms.

In all cases, the inputs required to our tool for analysis and experimentation are the routing mechanism, input policies (specified in the form of algebra), and a network topology (synthetically generated or obtained from either CAIDA [9] or Rocketfuel [68]).

Evaluation environment. FVR provides an interface for users to specify policy configurations using algebraic specifications, which are compiled into *NDlog* programs.

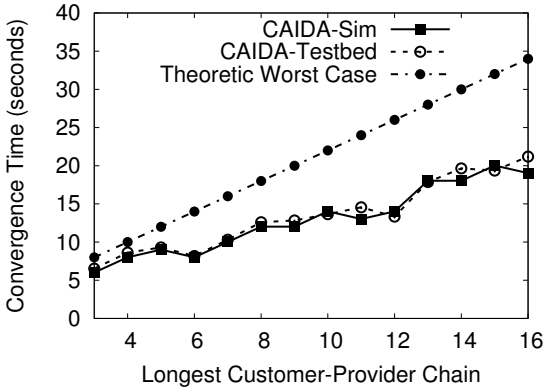

Figure 5.2: Convergence time (seconds) for BGP against longest customer-provider chain.

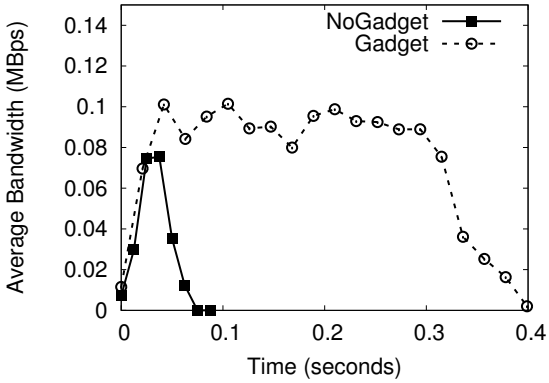

Figure 5.3: Average per-node bandwidth utilization (MBps) for iBGP with gadget.

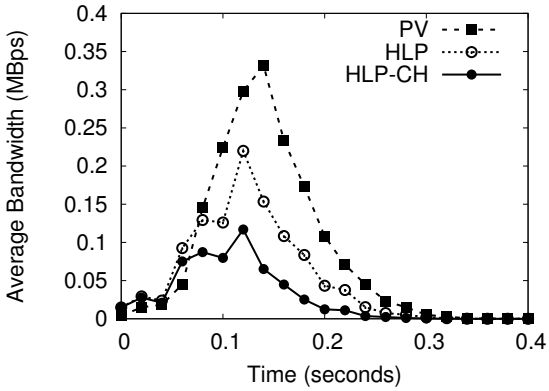

Figure 5.4: Average per-node bandwidth utilization (MBps) for HLP.

FVR uses the RapidNet [2, 49] declarative networking engine to compile the *NDlog* programs into applications (with an execution model similar to Click [35]) executable in ns-3 [51], an emerging discrete event-driven simulator similar to the popular ns-2. Like its predecessor, ns-3 emulates all layers of the network stack, supporting configurable loss, packet queuing, and network topology models. It also allows for a *simulation mode*, enabling a comprehensive examination under various network topologies and conditions, as well as an *deployment mode* where different hosts in a testbed environment execute the deployed system over a real network. The ability to run the same application in these two modes enables us to execute each *NDlog* program at scale in simulation and in an actual implementation running on a testbed, providing two avenues for augmenting the formal analysis.

5.4.1 Convergence Time vs. Network Size

Our first case study presents a scenario where a researcher empirically evaluates policy guidelines using the distributed *NDlog* implementation automatically generated from the algebraic specifications. To ensure strict monotonicity, we compose the basic Gao-Rexford guideline A policy with the shortest hop-count as the tie-breaker (using algebra's composition operator). The researcher has already analyzed the composed policy for its safety properties using Yices, but would like to measure the convergence time with respect to the depth of the AS hierarchy. A prior study [63] proved that, the worst case upper-bound of the convergence time for Gao-Rexford guideline is $2 \times (d + 1)$ *phases* (rounds of route advertisements), where d is the length of the longest customer-provider chain. The researcher can use the implementation that *FVR* generates from GPV (Section 5.1), and policy configurations. We present our results using RapidNet's simulation and deployment modes.

Simulation Mode Our first experiment is carried out in RapidNet's simulation mode. As our input topologies, we utilize the AS-level network graph (with annotated customer-provider relationships) provided in the CAIDA dataset [9]. The simulation is performed in a quad-core machine with Intel Xeon 2.33GHz CPUs and 4GB memory running Linux 2.6. In the simulation setup, all links have 100 Mbps in bandwidth and 10 ms latency.

To fit the simulation into memory and use a similar network size for our subsequent testbed evaluation, we extract sub-graphs from CAIDA's global network topology as follows: we remove all stub ASes[2], randomly select an AS *R* as the *root*, and then extract the AS hierarchy (transitively) provided by the AS. We choose 14 such sub-graphs with the length of the longest customer-provider chains ranging from 3-16. For each sub-graph, we executed the GPV protocol with guideline A, and measured the convergence time (from start of protocol until all nodes have computed routes to all destinations).

Figure 5.2 (CAIDA-Sim) shows the protocol convergence time as the length of the longest provider-customer chain increases. As a basis of comparison, we plot the theoretical worst-case convergence time [63]. Our protocol mechanism is configured to batch and propagate routes every second, a feature easily achieved using *NDlog*'s time-based predicates [40]. For instance, given the longest customer-provider chain of 10, the execution should converge within at most $2 \times (10 + 1)$ phases, namely 22 seconds. We make the following two observations from our simulation results. First, the convergence time increases linearly with the length of the longest customer-provider chain, validating the trend shown in the prior theoretical results. Second, we observe that, in practice, the protocol converges faster than the theoretical worst case. Upon further investigation based on execution logs, we realize the faster convergence is because customers at the "leaves" of the customer-provider tree typically have multiple paths to the root providers and can

[2]The pruned topology contains 5220 ASes and 23101 links.

leverage peer-to-peer links, and hence rarely require the full depth to propagate routes.

Deployment mode Our second experiment validates our simulation results using RapidNet's deployment mode. Here, we utilize 32 quad-core machines with a similar hardware/software configuration as our simulation experiment. The machines are connected using high-speed Gigabit Ethernet. We run up to 5 RapidNet instances per machine, and configure the neighbor links among RapidNet instances to be consistent with the earlier CAIDA setup in simulation. As before, we set the propagation period to 1 second.

Figure 5.2 (CAIDA-Testbed) shows that the convergence time obtained in the testbed closely mirrors that of our earlier simulation results. Our tool can switch between simulation and deployment based evaluation easily. Simulation and deployment modes of RapidNet uses the same compiled code base, with a configuration flag indicating running the network stack in simulation or using actual sockets. In the rest of this section, we primarily present results obtained in the simulation mode.

All in all, our first set of experiments based on the Gao-Rexford guideline is encouraging. Not only are we able to use Yices to check the guideline for safety, we are able to (with minimal effort) generate distributed implementations that provide additional performance insights using actual Internet topologies.

5.4.2 Pinpoint iBGP Configuration Errors

We emulate a scenario where a network operator uses our *FVR* toolkit to study the safety properties of an existing iBGP network configuration. As the input topology, we utilize the intra-domain topology (with inferred link weights) of AS 1755 from the Rocketfuel [68] dataset, which contains 87 routers and 322 links. Pairwise IGP

costs are computed a priori based on the shortest paths. The iBGP reflector-client topology is synthetically configured as a 6-level hierarchy with 53 reflectors.

Given the above input topology, we execute a GPV protocol on all 87 routers, and have each router within the AS compute the best route to a remote destination outside the AS, under the condition that several egress routers are aware of external routes to this particular destination. At each router, the route preference is based on the IGP cost from the router to the egress routers, i.e., the route with the lowest IGP cost is selected. This policy is similarly configured using routing algebra, and compiled into *NDlog* implementations.

To experiment with *FVR*'s ability to detect configuration errors, we embed a gadget similar to Figure 4.2 into the iBGP topology. This embedding is achieved by selecting three neighboring routers from the graph and setting their IGP cost to the egress routers the same as those in Figure 4.2. One goal of our experiment is to see whether our tool can detect this unsafe gadget embedded in a larger network instance.

Experimentation: Upon fixing the configuration errors, we experimentally evaluated both iBGP configurations implemented using *NDlog*. Similar to the earlier CAIDA experiments, all the links are set to 100 Mbps bandwidth, 10 ms latency, and up to 3ms jitter. Figure 5.3 shows our comparison of average per-node bandwidth utilization over time for the iBGP protocol with and without the embedded gadget (shown as *Gadget* and *NoGadget* after the fix). Compared with *Gadget*, we observe a 91% decrease in communication overhead, and 82% decrease in convergence time in *NoGadget*.

5.4.3 eBGP Gadget Analysis

FVR's applicability extends beyond high-level guidelines and iBGP configurations. We briefly summarize our experiences of using *FVR* to analyze well-known eBGP

gadgets: GOODGADGET, BADGADGET and DISAGREE [24]. These experiments highlight the use of *NDlog* implementations generated from *NDlog* implementations of SPP instances.

Experimentation We further experimentally evaluate the gadgets using the automatically generated *NDlog* implementation. In all cases, we provide an input topology, which contains one or more gadgets on a subset of the nodes. For GOODGADGET, as the number of gadgets increases, both the convergence time and communication cost increase. The increase is due to route re-computation, which occurs when a previously computed best path is overwritten by a longer path with a higher local preference. Nevertheless, all GOODGADGET scenarios converge as expected. On the other hand, the BADGADGET execution never converges—the protocol continued to transmit a high rate of update messages indefinitely. For DISAGREE, a gadget that can temporarily oscillate between two stable states before eventually converging, the protocols takes a longer time to converge as the percentage of *conflicting links* increases[3].

5.4.4 Alternative Routing Mechanism

While we have adopted GPV as the default mechanism, given that *FVR* is an extensible framework, other routing mechanisms can also be used, as long as they are implemented in *NDlog*. In our final case study, we demonstrate how researchers can supply *FVR* with a *different* routing mechanisms to study their impact on convergence behavior. We consider the *Hybrid Link-State and Path-Vector* (HLP) [69] protocol that has been proposed as an enhancement to the path-vector protocol. HLP capitalizes on the assumption that the ASes running BGP can be partitioned into domains that form a customer-provider hierarchy. HLP uses the regular link-

[3]A conflicting link is a link where the two adjacent nodes always prefer to route through each other.

state protocol within each customer-provider hierarchy, and a path-vector protocol (called *Fragment Path-Vector*, where paths that are internal to the hierarchy are hidden) across different hierarchies. We implement HLP in *NDlog* by using just 10 rules (11 rules if we also specify that internal paths are hidden).

We configure the network topology as a 10-domain network. Each domain is a 20-node acyclic hierarchical structure rooted by a top provider, where each node (with the exception of the top provider) has 1 or 2 providers. We configure the topology and policies within a domain based on the Gao-Rexford guideline A. Link latencies within one domain are set to 10 ms. In addition, there are a total of p84 cross-domain links throughout the network; these links are configured to have 50 ms latency. In all cases, links are set to have a bandwidth of 100 Mbps. For cost hiding, we set 5 as the threshold.

Figure 5.4 shows the bandwidth utilization of HLP over time, with and without cost hiding (shown as *HLP-CH* and *HLP*, respectively). As a basis of comparison, we execute the path-vector protocol (shown as *PV*). We note that as expected, *HLP* converges faster than *PV*, requiring 0.35 seconds compared to 0.4 seconds for *PV*. Moreover, the per-node communication cost for *HLP* and *PV* is 1.09 MB and 1.75 MB, respectively. *HLP-CH* further reduces the communication cost to 0.59 MB per node.

Beside HLP, other possibilities include multi-path routing protocols, and neighbor-specific BGP mechanisms, which typically require further customization to user-defined functions, for instance, propagating the top-k paths instead of the current best. Such comparisons are tremendously useful for researchers to study the full design space in both policies and mechanisms.

5.5 Summary

This chapter presents *FVR*'s ability to generate provably-correct implementation from the verified algebraic representation of routing policy. We show that routing algebra has a very natural translation to declarative networking programs. By generating the actual distributed implementation, *FVR* has combined safety analysis and experiments with the protocol implementations to pinpoint configuration errors and gain insights into the performance of the generated implementation. This allows research on inter-domain routing to leverage mature technologies, such as SMT solvers and RapidNet, to automate complex and error-prone tasks for researchers and practitioners alike.

Chapter 6

Verifying Actual Routing Systems

The routing algebra we verified enable us to synthesize provably-correct routing configurations, as shown in the previous chapter. However, the model, due to its correctness-by-construction nature is too restricted and does not capture all correct and useful configurations in real world. For example, routing algebra assumes total ordering of all available routes in the BGP network system, which may not be available in a actual configuration (shown in Figure 2.6). Moreover, when dealing with iBGP-eBGP interaction, routing algebra can only specify the BGP networks where all the component iBGPs are configured in the same way, due to the inherent limitation in the semantics of lexical product [25]. On the other hand, the dynamic behavior of the BGP system also depends on the underlying routing mechanism — path vector protocol. Thus, to reason about an actual routing system, it is necessary to be able to verify arbitrary routing configuration and the path vector routing protocol.

This chapter presents *BGPVerif*'s ability to detect routing anomalies in actual configurations, and verify routing mechanism. As shown in Figure 6.1, based on the combinatorial SPP model, *FVR* includes Maude library [7, 43] that automatically detects routing anomalies in policy configurations; Simultaneously, *FVR* uti-

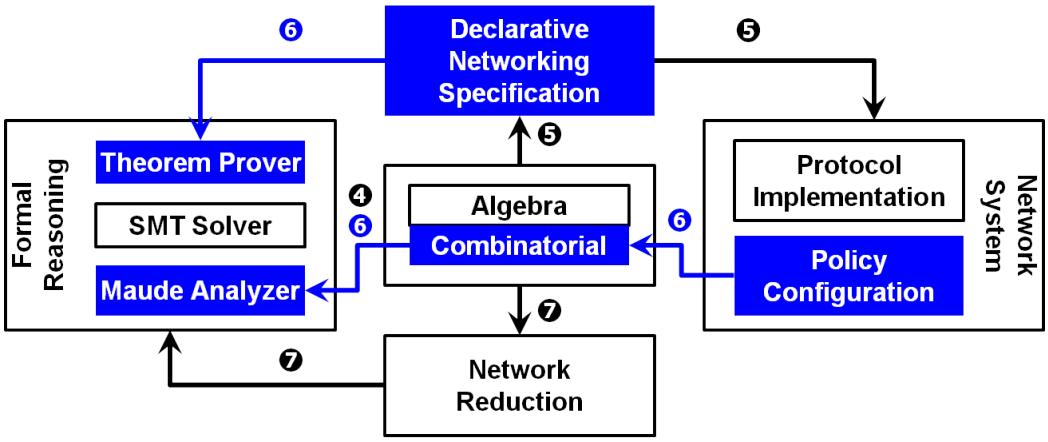

Figure 6.1: *FVR* Architecture: Dynamic Analysis.

lizes theorem prover to verify the routing mechanism implemented in declarative networks.

6.1 Detect Anomalies in Policy Configurations

Background: Rewriting Logic and Maude Rewriting logic [46] is a logical formalism that is based on two simple ideas: states of a system can be represented as elements of an algebraic data type, and the behavior of a system can be given by transitions between states described by *rewrite rules*. By algebraic data type, we mean a set whose elements are constructed from atomic elements by application of constructors. Functions on data types are defined by equations that allow one to compute the result of applying the function. A rewrite rule has the form $t \Rightarrow t'$ *if* c where t and t' are patterns (terms possibly containing variables) and c is a condition (a boolean term). Such a rule applies to a system state s if t can be matched to a part of s by supplying the right values for the variables, and if the condition c holds when supplied with those values. In this case the rule can be applied by replacing the part of s matching t by t' using the matching values for variables in t'.

Maude [7] is a language and tool based on rewriting logic. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. Given a specification S of a concurrent system, Maude can execute this specification and allows one to observe possible behaviors of the system. One can also use the search functionality of Maude to check if a state meeting a given condition can be reached during the system's execution. Furthermore, one can model-check S to check if a temporal property is satisfied, and if not, Maude will produce a counter example. Maude also supports object-oriented specifications that enable the modeling of distributed systems as a multiset of objects that are loosely coupled by message passing. As a result, Maude is particularly amenable to the specification and analysis of network routing protocols. We choose Maude because: (1) It comes with a full-fledged automated tool support for analyzing distributed system [8, 54, 12]. Based on Maude's object-oriented specification language and underlying rewriting logic, we encode in Maude the BGP protocol as a transition system driven by rewriting rules. Consequently, we can use the high-performance rewriting engine provided by Maude to analyze BGP instances automatically.

6.1.1 A Maude Library for BGP Systems

This section presents our Maude library for analyzing BGP instances. This library provides specification of the protocol dynamics that are common to BGP instances, and defines a routing policy template in terms of the Stable Path Problem (SPP) so that network designers can customize it to analyze a specific instance. Our library also provides support for detecting route oscillation.

Our library is organized into a hierarchy of Maude modules. Table 6.1 presents the correspondence between concepts in BGP protocols and the Maude code. We first show how our library represents a single network state of BGP system (Sec-

BGP system	Maude interpretation
Network nodes	(Router) objects
Routing messages	Terms of type `Msg`
Global Network	Multiset of router objects and terms representing messages
Protocol dynamics	Local rewriting rules
Global network behaviors	Concurrent rewrites using local rules
Route oscillation support	A Logger object recording the histories of route assignments and rewriting rules updating the Logger object

Table 6.1: Overview and Interpretation of Maude Library

tion 6.1.1). Then we explain how to capture the dynamic behavior of a local BGP router using rewrite rules. In doing so, the global network behaviors can be viewed as concurrent applications of the local rewriting rules (Section 6.1.1). Finally, we discuss the component in the library that detects route oscillation (Section 6.1.1).

Network State Maude provides a built-in sort `Configuration` for the global state of a concurrent object-based system. In our library, we model BGP system state by defining two sub-sorts of `Configuration`: `Msg` and `Network`. `Msg` is the type for routing messages, and `Network` the type for the collection network nodes. To construct the `Network Configuration` from the constituting network nodes, we further introduce sort `Node`. `Node` is a sub-sort of `Object`, and is the type of each network node (routers) in the BGP system.

```
sorts Msg Network .
subsorts Network Msg < Configuration .
sort Node .
subsort Node < Object .
```

A network state is represented by a multiset of network nodes (routers) and routing messages used by routers to exchange routing information. Given the

above sorts, we can represent a BGP system state by a multiset of routing messages terms (of sort `Msg`) and network nodes objects (terms of sort `Node`). To distinguish a network node object from other Maude objects, we introduce a special Cid (class identifier) called `Router`. With `Router` Cid, we introduce the constructor of a network node (router) object. In Maude, `op` is the keyword defining constructors:

```
op Router : Oid .
op [_:_|_,_,_] : Oid Router RoutingTable BestPath Nb -> Node .
```

The constructor `[_:_|_,_,_]` takes five arguments: The first two are a unique identifier (of sort `Oid`) and a class identifier (the Cid constant `Router`). The `Oid` denotes the node's unique name in the network, the class identifier denotes the class type of the node. The last three arguments are object attributes: routing table (of sort `RoutingTable`), best path (of sort `BestPath`), and neighboring table (of sort `Nb`). Given the argument terms, a router object term (of sort `Node`) is constructed. Next we discuss a `Node` object's three attributes.

RoutingTable and BestPath attributes. At a given state, the routing table and best path attributes store node v's available paths P^v to reach destination 0, and its current best path that ranks the highest among P^v respectively. They are constructed as follows:

```
op routingTable:_ : List{Path} -> RoutingTable .
op bestPath:_ : Path -> BestPath.
```

The sort `Path` and its constructors are defined as follows:

```
sort Path .
op emptyPath : -> Path [ctor] .
op source:_,dest:_,pv:(_),metric:_ :
     Oid  Oid  ListOid  Metric -> Path .
```

There are two ways to construct a term of sort `Path`: a path is either an empty path (`emptyPath`), or a path from a source node (`Oid`) to a destination node (`Oid`) via a path vector (the list of intermediary nodes denoted by `List{Oid}`) at some cost of sort `Metric`. Sort `Metric` specify how paths are measured and compared. For example, a path metric in the SPP formalism is its rank defined in Λ. We interpret the metrics over Maude's built-in natural numbers sort `Nat`:

```
sort Metric .
subsorts Nat < Metric .
```

Neighbor attribute. A node's neighboring table holds its list of link information to reach its direct neighbors:

```
sort Neighbor .
op mkNeigh(_,_) : Oid Metric -> Neighbor.
```

Here the constructor `mkNeigh(_,_)` takes two arguments: the direct neighbor's identifier, and the cost (`Metric`) to reach this neighbor.

Example node object. Based on the above definition, each network node is then represented by a Maude object, whose attributes consist of its routing table, best path and neighboring table. We omit the detailed Maude sort definitions, but provide an example encoding of the network node `n1` in Disagree gadget show in Figure 2.5 as follows.

```
[n1 : router |
      routingTable: (source: n1,dest: n0,pv:(n1 n0),metric: 2),
      bestPath: (source: n1,dest: n0,pv:(n1 n0),metric: 2),
      nb: (mkNeigh(n0,2) mkNeigh(n2,1))]
```

The constructor for a node is `[_:_|_,_,_]`. The first two elements (`n1:router`) specify the node's id `n1`, and its object class `router`. The next three elements are the attributes. At a given state, the routing table attribute constructed from `routingTable:_` contains `n1`'s current available routes. Each routing table entry

stores the routing information for one particular next-hop. Here, the routing table attribute only contains one entry `(source: n1, dest: n0, pv:(n1 n0), metric: 2)`. This route is specified by its source (`source: n1`), destination (`dest: n0`), the path vector that contains the list of nodes along the path (`pv: (n1 n0)`), and the cost of the route (`metric: 2`). This route is also used for the best path attribute, constructed from `bestPath:_`, which contains `n1`'s current best path. The last attribute is the neighbor table, constructed from `nb:_`. To extract a node's local neighbor table from the network topology, we further introduce an operator `mkNeigh`. The first argument of `mkNeigh` is the identifier of the neighboring node, and the second argument the metric associated with the link to that node. Node `n1` has two neighbors, node `n0`, the cost to which is 2 (`mkNeigh(n0,2)`); and node `n2`, the cost to which is 1 (`mkNeigh(n2,1)`).

Besides `Router` objects, the second part of a network state are routing messages in the network. Typically, network nodes exchange routing information by sending each other routing messages carrying newly-learned routing paths. In our library, it is specified as follows:

```
op sendPacket(_,_,_,_,_) : Oid Oid Oid Metric ListOid -> Msg
```

The first two `Oid` arguments denotes the sender's Oid and the receiver's Oid respectively. The rest of the arguments specify the destination Oid, metrics of the advertised path, and the path-vector (the list of network nodes along the path) of the routing path the message carries.

For example, in the disagree gadget, the initial routing message sent by node `n1` to its neighbors `n2` carrying its direct path to `n0` are is: `sendPacket(n1,n2,n0,2,n1 n0)`.

Protocol Dynamics We now show how to specify network system dynamics in Maude. By modeling a BGP system as a concurrent system consisting of router

objects (and the routing messages), to specify the global BGP evolution, we only need to specify the local rewrite rules governing the state transition of each BGP router.

A BGP node's dynamics can be captured by various equivalent state transitions. To reduce search space in analysis, we adopt a one-state transition: for each BGP node `N`, when it receives routing messages from a neighbor `S`, `N` computes the new path from the received message, updates `N`'s routing table and re-selects best path accordingly, and finally sends out routing messages carrying its new best path information if a different best path is selected. This state transition is encoded as a single rewrite rule of the following form:

```
rl [route-update] :
  sendPacket(S, N, D, C, PV)
  [ N : router | routingTable: RT, bestPath: Pb, nb: NB ]
 =>
if  (case 1) then best path re-selects (promotion)
else (if (case 2) then best path remains same
      else (if (case 3) then best path re-selection (withdraw)
            else error processing
            fi) fi) fi.
```

Here, `r1` is the identifier of this rule, and `route-update` is the name of this rule. Rule `r1` is fired when the left-hand side is matched; that is, when a node `N` consists of routingTable `RT`, bestPath `Pb`, and neighboring table `NB` receives a route advertisement message from neighbor `S`. The result of applying the rule is shown on the right-hand side: the routing message is consumed, and attributes of router `N` are updated. Based on the result of the re-selected bestPath attribute, there are three different cases for `N` to update its state as specified in the three branches. Next, we explain these three cases.

Best path promotion. In any case, node `N` needs to first compute the new path

based on its neighbor `S`'s message asserting that `S` can reach `D` via a path `PV`. We define a function `newPath` that takes a routing message and the neighbor table as arguments, and returns the new path by first prepending `N` to the path announced by `S`, setting the new path attribute according to the local ranking function `lookUpRank`, and then imposing the import policy by modifying the path metric according to BGP routing policy configuration (`import` function). Here `import` and `lookUpRank` are unspecified routing policy functions. Together with `export` that we will introduce shortly, they constitute our library's specification interface for defining BGP routing policy. To specify a particular BGP instance's routing policy, the user only needs to specify `import, lookUpRank` and `export` accordingly.

The first branch (case 1) is specified below. The newly computed path is compared with the current bestPath `Pb`, if the new one is preferred over the old value `Pb`, the `bestPath` attribute will be updated to this new path. Furthermore, if the export policy allows, the new best path value will be re-advertised to all of `N`'s neighbors by sending them routing messages.

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   prefer?(newPath(sendPacket(S,N,D,C,PV),NB),Pb)==true
then
([ N : router |
   routingTable: updatedRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   bestPath: newPath(sendPacket(S,N,D,C,PV),NB),
   nb: NB ]
 multiCast(NB, export(newPath(sendPacket(S,N,D,C,PV),NB))))
```

Here the new state of `N` is obtained by updating the old `routingTable` attribute `RT` (`updateRT` function), and updating the `bestPath` attribute by setting it to the new value of `bestPath`. The `updateRT` function recursively checks the routing table, and for each next-hop entry, it either inserts the new path (`newPath(...)`) if no available route is presented; or replaces the old value with the new path.

To complete the state transition, for all N's neighbors, routing messages carrying the new path are generated by multiCast function. To impose the export routing policy, before sending the new best path, export is applied to the new path to filter out the routes which are intended to be hidden from neighbors. Similar to import, export is to be instantiated by the user when analyzing a particular BGP instance. If the export routing policy prohibits the new path to be announced, export will transform it to emptyPath, which multiCast will not generate any message.

Best path remains the same. In the second branch (case 2), a new path newPath(...) is computed from the received message as before. However, the new path is no better than the current bestPath Pb. But the next-hop node of the new path and Pb are different, implying that the new path is just an alternative path [1] for N to reach the destination. As a result, the current bestPath value Pb is unchanged, and only the routingTable will be updated with this alternative path (newPath(...)). No routing messages will be generated:

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   getNext(newPath(sendPacket(S,N,D,C,PV),NB))=/=getNext(Pb) and
   prefer?(Pb,newPath(sendPacket(S,N,D,C,PV),NB))==true
then
[ N : router |
   routingTable: updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   bestPath: Pb,
   nb: NB ]
```

Best path withdraw. The same as in the second branch, in case 3, the newly computed path newPath(...) is worse than the current bestPath Pb, but it is now routed through the same next-hop S as current bestPath Pb. The fact that S now sends a less preferred path indicates that the previous learned route Pb is no longer

[1] Different next-hop implies the route is learned from a different neighbor.

available at S. Therefore, we need to withdraw Pb by dropping Pb from routing table, shown as follows:

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   getNext(newPath(sendPacket(S,N,D,C,PV),NB))==getNext(Pb) and
   prefer?(Pb, newPath(sendPacket(S,N,D,C,PV),NB))==true
then
([ N : router |
   routingTable: updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   newBest(newPath(sendPacket(S,N,D,C,PV),NB),
            updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT)),
   nb: NB ]
 multiCast(NB,export(newBest(newPath(sendPacket(S,N,D,C,PV),NB),
                      updateRT(newPath(sendPacket(S,N,D,C,PV),NB),
                               RT))))
```

Here, updateRT replaces (therefore removes) the outdated Pb with the new path
(newPath(...)), and newBest function re-computes the best path from newPath(...) and the remaining paths in routing table. As in case 1, to complete the state transition, the newly selected best path is sent to its neighbors by multiCast(...).

Auxiliary Functions. In the following code snippet, we show the auxiliary functions used in computing new network state. To compute a new path, concat is defined as follows:

```
op concat : Oid Oid Oid Metric ListOid ListNeighbor ->  Path .
eq concat(S, N, D, C, PV, NB)
     = (source: N, dest: D, pv:(N PV), metric: lookUpRank (N PV)) .
```

eq is a Maude keyword preceding equation definitions. Here the new path of N's to reach D is simply by prepending N to PV: N PV. And the metric of (N PV) is determined by N's routing policy, i.e., route ranking λ^N implemented by function

`lookUpRank`. The definition of this function is specific to each BGP configuration instance, and is an interface between the protocol dynamics and the routing policies. We will revisit the specification of specific BGP instance's routing policies by `lookUpRank` in appendix 6.1.2.

Given the newly computed route, to update a node's routing table, we use `updateRT` as follows:

```
op updateRT : Path ListPath -> ListPath .
eq updateRT (P, nil) = (P) .
eq updateRT (P, (P' RT)) =
 if ((getDest (P) == getDest (P')) and
     (getNext (P) == getNext (P')))
    then (P RT)
    else (P' updateRT (P, RT)) fi .
```

Note that, `updateRT` ensures that the routing table always keeps exactly one path from one particular next-hop neighbor.

Finally, if the best path attribute of the node changes, we use `multiCast` to generate routing messages:

```
op multiCast : ListNeighbor Path -> Configuration .
eq multiCast((NBentry NB'), (source: S, dest: D, pv:(PV), metric: C))
    = sendPacket (S, getOid(NBentry), D, C, (PV))
      multiCast (NB', (source: S, dest: D, pv:(PV), metric: C)) .
```

For each neighbor `NBentry` in `N`'s neighboring table, `multiCast` recursively generates the routing message `sendPacket(S,getOid(NBentry),D,C,(PV))`. `getOid` is an auxiliary-function that extracts the neighbor `NBentry`'s Oid.

Route Oscillation Detection Support Our library also provides extra definitions to help detect route oscillation. Our method is based on the observation that if route oscillation occurs during network system evolution, there is at least one *path*

assignment (at a given state for a BGP system, we define the path assignment to be the collection of best paths currently selected by all network nodes) that is visited twice. Therefore, we use the following simple heuristic: we maintain a record of all path assignments for all visited states in BGP execution, and check for recurring path assignment. Note that a path assignment (best path attribute of `router` object) only constitutes a sub-set of the entire system state (the `router` objects attributes and routing messages), consequently our heuristic based on this partial system state can have false positives: our analysis may report a false route oscillation when two states are identical only in path assignments, but not the entire system states. Nevertheless, our heuristic is sound and is still helpful in detecting all potential route oscillation: when route oscillation occurs, a recurring path assignment state must occur.

More concretely, in our Maude library, we create a global *logger object* to keep track of the history of path assignments. For each snapshot of the network state, i.e. whenever a network node makes a local state transition and updates its best path attribute, the logger object is synchronized to create a new path assignment entry that corresponds to the updated best path. We then provide a function that checks for recurring entries in the list of visited path assignments, which can be used directly in Maude's exhaustive search to detect route oscillation.

Logger object. The logger object consists of only one attribute: a history (list) of path assignments, each entry of which corresponds to the list of best paths for all nodes in the network at a given state. The Maude code for defining the logger object is shown below.

```
sort Logger .
op Logger : -> Cid .
op {_:_|_} : Oid Cid AttributeSet -> Logger  .
```

```
op history:_ : List{PathAssignment} -> Attribute .
op {_} : List{BP} -> PathAssignment .

sort BP .
op [_] : List{Oid} ->  BP .
```

The first three lines declares a special `Logger` object, and the constructor of the Logger, which takes three arguments, the first one is the identifier for the logger, similar to the identifiers for the router objects; the second argument is the class identifier of logger object; and the last one is the attribute of the logger. The next two lines declare the only attribute of `Logger`, which is a list (history) of `PathAssignment` elements, each of which denotes one path assignment in the network at a given state. A path assignment is a list of best path selected by each node in the network. The last two lines defines each entry in one path assignment, which is simply the best path for some network node: i.e. the list of nodes in the best path.

By utilizing above definitions, the global logger is represented by an object `pa` of `Logger` class which has one attribute `history`. At a given state, this attribute contains a list (history) of path assignments, each entry of which contains the snapshot of the network's collection of best paths in a visited state. An example logger object for the disagree gadget is the following:

```
{pa : Logger | history: ({[n1 n2 n0] [n2 n0]}
                         {[n1 n2 n0] [n2 n1 n0]}
                         {[n1 n2 n0] [n2 n0]}
                         {[n1 n0]   [n2 n0]})}
```

The above logger records four snapshots of the Disagree's best paths. For example, the first path assignment `{[n1 n2 n0] [n2 n0]}` denotes the network latest state where node `1`'s best path to `0` is `[n1 n2 n0]` and node `2`'s best path is `[n2 n0]`. And line 4 `{[n2 n0] [n2 n0]}` records Disagree's path assignment at its initial

(oldest) state. Note that, this object content actually exhibits route oscillation (line 1 and line 3) described in Section 6.1.1.

Synchronized logging. To log all path assignment changes, we only need to slightly modify the single rewrite rule for route update, such that whenever the rule is fired to apply local state transition for some node, the global object `pa` is synchronized and its path assignment is updated to reflect changes in the local node's best path attribute, shown as follows:

```
rl [route-update-logging] :
  sendPacket(S, N, D, C, PV)
  [ N : router | routingTable: RT, bestPath: Pb, nb: NB ]
  { pa : Logger | history: HIS }
=>
*** first branch: bestPath re-selects (promotion)
if ... then ...
 { pa : Logger | history:
   historyAppend(updateAt(index(N),
           [getPV(newPath(sendPacket(S,N,D,C,PV),NB))],
           head(HIS)),HIS)})
else ... fi .
```

On the left-hand side, two objects: a router `N` and the global logger `pa` are matched to trigger the transition. As described in 6.1.1, in the first branch of route update where the node's best path attribute is set to `newPath(...)`, the logger `pa` updates its path assignment attribute as follows: First, it creates a new path assignment entry to record `newPath(...)` by function `updateAt(...)`. Then, the new entry `updateAt(...)` is inserted into the list of previous path assignments `HIS` by function `historyAppend`. Here, the new path assignment entry `updateAt(...)` is computed by updating the latest path assignment entry `head(HIS)` with `newPath(...)`. The rest of branches 2 and 3 are modified similarly.

Route oscillation detection. A network state is now a multiset of router objects, routing messages, and one global logger object. The function `detectCycle` detects re-curring path assignments, as follows:

```
eq detectCycle([ N : router | routingTable: RT,
                              bestPath: Pb,nb: NB] cf)
  = detectCycle (cf) .
eq detectCycle(message cf) = detectCycle (cf) .
eq detectCycle({ pa : Logger | history: HIS } cf)
  = containCycle? (HIS) .
```

The first two equations ignore router objects and routing messages in the network state, and the last equation examines logger `pa` by function `containCycle?` to check for recurring path assignment entries in `HIS`. We will revisit the use of `detectCycle` to search for route oscillation in Section 6.1.3.

6.1.2 Specifying BGP Instance

Given a BGP instance with its *network topology* and *routing policies*, we show how to specify the instance as a SPP in our library. We discuss examples for both eBGP and iBGP.

eBGP instance An eBGP instance can be directly modeled by an SPP instance $S = (G, o, P, \Lambda)$: G, o specifies the instance's *network topology*, and P, Λ specifies the resulting per-node route ranking function after applying the eBGP instance's *routing policies*. Our library provides Maude definitions for each SPP element.

Network topology.

The network topology G, o is represented by three constants: `n0`, `top-Nodes`, `top-BGP`:

```
op n0 : Oid .
op top-Nodes : -> ListOid .
```

```
op top-BGP : -> Topology .
```

`n0` is the specific destination o, `top-Nodes` the set of network nodes, and `top-BGP` the set of BGP links. Our library has pre-defined sort `Topology` to capture that a network topology is a set of labeled network links:

```
sorts Link Topology .
op (_,_:_) : Oid Oid Metric -> Link .

subsort Link < Topology .
op __ : Topology Topology -> Topology .
```

The first line of Maude code declares `Topology` and `Link`; The second line says a `Link` is constructed from its two end nodes, and the associated metric. The last two lines specify how `Topology` is constructed: a topology is either a single link, or recursively constructed from existing topologies.

Our Maude library automatically generates an eBGP instance's initial state based on its topology:

```
op gadget : -> Configuration .
eq gadget = init-config (top-Nodes, top-BGP) .
```

By using above Maude library, an eBGP instance's initial network state is generated from its network topology, which is represented by a list of network nodes and links. Our library declares two constants `top-Nodes` and `top-BGP` to represent network nodes and links. For example, to specify the topology of the Disagree gadget, the user defines `top-Nodes, top-BGP` as follows:

```
eq top-Nodes = n1 n2 .
eq top-BGP = (n1,n0 : 2) (n1,n2 : 1) (n2,n1 : 1) (n2,n0 : 2) .
```

Here, `n0` is the identifier of the destination node (o). Each link is associated with its cost. Based on the value of `top-Nodes` and `top-BGP` that are input by the

user, our library automatically generates Disagree's initial state by `init-config` function:

```
eq gadget = init-config (top-Nodes, top-BGP) .
```

The resulting `gadget` is a network state which consists of the two network router objects `n1,n2`, the four initial routing messages, and the initial logger `pa`, as shown in Section 6.1.3. In this initial state, the three attributes of each network node – the routing table and best-path and neighbor tables are computed as follows: `init-config` parses the BGP links in network topology (`top-BGP`), for each link `(ni,nj : M)`, a new routing table entry for `nj` with cost `M` is created, and if `nj == n0`, then set `ni`'s best path to the one-hop direct path `ni n0`, and its routing tables containing this one-hop direct route; otherwise if there is no direct link from `ni` to `n0`, set `ni`'s best path and the routing table to `emptyPath`. Initial routing messages and logger `pa` are computed in a similar manner.

Routing policy. The route ranking function Λ and permitted paths P are the result of applying three BGP policies functions: `import`, `export` and `lookUpRank`. As we have discussed in Section 6.1.1, `import,export,lookUpRank` are three user-defined functions that serve as the specification interface for routing policies.

Functions `import` and `lookUpRank` are used to compute new routing paths from a neighbor's routing message: `import` filters out un-wanted paths, and `lookUpRank` assigns a rank to the remaining permitted paths. Note that the metric value `lookUpRank (N PV)` assigned by `lookUpRank` also determines the route's preference in route selection. `export` is used to filter out routes the router would like to hide.

As an example, the policy functions for Disagree are defined as follows.

```
eq export (P) = P . eq import (P) = P .
eq lookUpRank (n1 n2 n0) = 1 . eq lookUpRank (n1 n0) = 2 .
eq lookUpRank (n2 n1 n0) = 1 . eq lookUpRank (n2 n0) = 2 .
```

The first line says Disagree does not employ additional import/export policies. Whereas the second and third line asserts that Disagree's two nodes prefers routes through each other: For example the second line encodes node `n1`'s ranking policy that it prefers path `(n1 n2 n0)` (with higher rank 1) through `n2` over the direct path `(n1 n0)` (rank 2).

iBGP Instance The main differences between an iBGP and eBGP instances are: (1) iBGP network topology distinguishes between internal routers and gateway routers. Gateway routers runs eBGP to exchange routing information with (gateway routers of) other ISPs, while simultaneously running iBGP to exchange the external routing information with internal routers in the AS. (2) iBGP routing policy utilizes a separate IGP protocol to select best route. Internal to an AS, the ISP uses its own IGP protocol to compute shortest paths among all routers. The shortest path distance between internal routers and gateway routers are used in iBGP route selection: iBGP policy requires the internal routers to pick routes with shortest distance to its gateway router.

As a result, iBGP requires encoding two types of topologies: a *signaling* topology for gateway routers and internal routers to exchange routes within the AS, and a *physical* topology on which the IGP protocol is running. Further, an additional *destination router* denoting the special SPP destination o is added as an external router which is connected with all gateway routers. In our library, we implement and run separately in Maude an IGP protocol (for computing all-pairs shortest paths) and pass the resulting shortest path distances to iBGP protocol.

An iBGP configuration instance $C = (G_P, G_S, X)$ is defined by its physical topology G_P, signaling topology G_S, and gateway (egress) BGP nodes X. G_P represents the underlying network topology that runs a separate IGP protocol, therefore we also call G_P the *IGP topology*. G_S represents the network topology that runs iBGP. The iBGP links in G_S can be partitioned into three classes `over`,`down`,`up`: an

`over` iBGP link represents a vanilla iBGP link, a `down` iBGP link represents a iBGP session from a iBGP *reflector*[2] node to its client, and an `up` link represents that from a client to its reflector server. X represents the gateway (egress) BGP nodes from which external routes (routes to destinations outside the AS) are learned.

While the eBGP instance is usually given in the form of SPP, we need one additional translation [27] to transform iBGP instance into SPP. Because we are interested in the behavior of an iBGP instance $C = (G_P, G_S, X)$ in distributing external routing information learned from iBGP gateway routers, we define an iBGP instance's corresponding SPP representation $S = (G, o, P, \Lambda)$ as follows: o is an additional network node outside the iBGP instance, and it represents the external common destination; G is the signaling topology G_s with the additional node o and (non-BGP) links between o and egress nodes X; Λ is the function that computes IGP-distance. This is because, within an AS, for a common external destination o, all routes' AS-level metrics are same, as a result, a route can uses its IGP-distance alone to decide its rank.

To automate an iBGP instance specification in Maude, for each iBGP instance $C = (G_P, G_S, X)$, we provide additional Maude definitions to generate its SPP reorientation (G, o, P, Λ). We describe network topology G, o and routing policy Λ respectively.

Network topology. Similar to eBGP, iBGP network topology G, o is represented by constants `top-BGP, nd`. Rather than asking the user to manually input top-BGP as in eBGP instance, our library generates `top-BGP` from iBGP signaling topology as follows:

```
op top-iBGP-signal : -> Topology .
ops top-Nodes top-Xset : -> ListOid .
eq top-BGP = addExternal (top-iBGP-signal, top-Xset, nd) .
```

[2]To solve scalability problem in full-mesh iBGP configuration, some iBGP nodes are elected to be route reflectors that act as focal point in iBGP sessions: the reflectors form a smaller full-mesh, and the rest of the nodes become their clients.

`top-iBGP-signal, top-Xset` stores the iBGP instance's signaling topology G_S and egress nodes X. Note that, while the metric of each link in an eBGP network topology represents the associated link cost (e.g. the IGP distance between the two nodes), the metric of link in signaling topology denotes its class: over, up, or down. So our library further includes three metric constants:

```
ops up down over : -> Metric .
```

`addExternal` is a function that takes the external destination `nd`, egress nodes `top-Xset` as input, and generates the network topology `top-BGP` by adding to the signaling topology `top-iBGP-signal` additional links between each egress node in X and external destination o:

```
op addExternal : Topology ListOid Oid -> Topology .
eq addExternal(top, (X Xset), D) =
      addExternal (top, Xset, D) (X,D : 1) .
eq addExternal (top, nil, D) = top .
```

The resulting `top-BGP` is then used to initialize network state as in eBGP instance.

Based on the above library support, to specify the network topology of iBGP instance, the user only need to specify G_P, X by customizing constants `top-Nodes, top-iBGP-signal, top-Xset`. For example, to specify a 6-node iBGP instance [16], we write:

```
eq top-Nodes = n0 n1 n2 n3 n4 n5 .
eq top-Xset = n3 n4 n5 .
eq top-iBGP-signal =
  (n0,n1 : over) (n0,n2 : over) (n0,n3 : down)
  (n1,n0 : over) (n1,n2 : over) (n1,n4 : down)
  (n2,n0 : over) (n2,n1 : over) (n2,n5 : down)
  (n3,n0 : up) (n4,n1 : up) (n5,n2 : up) .
```

Routing policy

Like eBGP instance, routing policy Λ is given by customizing `import`, `export`, and `lookUpRank`. But unlike eBGP instance where `lookUpRank` simply assigns each path its rank, iBGP routing policy is more complex.

An iBGP policy consists of two parts: First, a valid iBGP path consists of a set of (can be empty) `up` links, which is followed by zero or one `over` link followed by a set (can be empty) of `down` links. Second, for routes with same AS-level attribute, a node always prefers routes with lower IGP distance, i.e., routes with shorter distance to a egress node.

The first policy is achieved by imposing an export policy at each node, such that only routing updates from a client will be exported to all neighbors. In Maude library, this is achieved by defining `export` in the iBGP module as follows:

```
*** for routes learned from internal nodes
*** only allow routing path of the form:
***   ... up ... up (over) down down ...
eq export ((source: S, dest: D, pv:(S N N' PV), metric: C)) =
    if (getLinkMetric (S,N,top-iBGP) == over and
        getLinkMetric (N,N',top-iBGP) == over)
    then emptyPath
    else (source: S, dest: D, pv:(S N N' PV), metric: C) fi .

*** for routes learned from egress nodes, do nothing
eq export ((source: S, dest: D, pv:(S N), metric: C)) =
  (source: S, dest: D, pv:(S N), metric: C) .
```

The second iBGP policy is achieved by set a path's rank to its IGP distance:

```
eq lookUpRank (PV) =
computeIGP (head (PV), last(front(PV)), top-IGP) .
```

Here `head(PV)` is the source of the path `PV`, and `last(front(PV))` is the egress node `head(PV)` used to reach destination `last(PV)`. And `top-IGP` is the underly-

ing IGP topology. The function `computeIGP` computes the IGP distance between the source and the egress node according to the underlying IGP topology. In our Maude library, we implement `computeIGP` by implementing a separate IGP protocol called shortest-path protocol. In shortest-path protocol, the IGP distance between two nodes is the cost of the shortest path between them.

Based on the above library support, to specify iBGP `lookUpRank`, the user only needs to specify the underlying IGP topology G_P by providing proper definition of `top-IGP`. For example, to specify the 6-node iBGP instance, we write:

```
eq top-IGP =
  (n0,n3 : 10) (n0,n4 : 5) (n1,n4 : 10)
  (n1,n5 : 5) (n2,n5 : 10) (n2,n3 : 5) .
```

Here the link metric `5,10` specifies the IGP distance between neighboring nodes.

6.1.3 Detecting Anomalies

To analyze BGP instances, our library allows us to (1) execute the Maude specification to simulate possible execution runs; and (2) exhaustively search all execution runs to detect route oscillation.

Network Simulation *Network initialization.*

For any analysis, we need to first generate a BGP instance's initial network state. For a given BGP instance, we have shown how to generate its initial state `gadget` from its network topology and routing policy, as described in section 6.1.2. For example, the initial state generated for Disagree is as follows:

```
{pa : Logger | history:{[n1 n0] [n2 n0]}}
[n1 : router | routingTable: (source: n1, dest: n0,
                              pv:(n1 n0), metric: 2),
               bestPath: (source: n1, dest: n0,
                          pv:(n1 n0), metric: 2),
```

```
                nb: (mkNeigh(n0,2) mkNeigh(n2,1))]
[n2 : router | ... ]
sendPacket(n1,n0,n0,n2,n1 n0) sendPacket(n1,n2,n0,n2,n1 n0)
sendPacket(n2,n0,n0,n2,n2 n0) sendPacket(n2,n1,n0,n2,n2 n0)
```

This state consists of Disagree's initial logger object `pa` that holds the initial path assignment `[n1 n0] [n2 n0]`, two router objects `n1`,`n2`, and four initial routing messages.

Execution.

Unlike many formal specification paradigms used in static network analysis, a Maude specification is executable. To explore *one* possible execution run from a given initial state `gadget`, we can directly use Maude's `rewrite` and `frewrite` (fair rewriting) commands. For example, we could tell Maude to execute the Disagree gadget with the following command: `frew gadget .` This command terminates and returns the following final state:

```
{pa : Logger |
    history: ({[n1 n0] [n2 n1 n0]} ... {[n1 n0] [n2 n0]})}
[n1 : router |...
    bestPath: (source: n1,dest: n0,pv:(n1 n0),metric: 2), ...]
[n2 : router |...
    bestPath: (source: n2,dest: n0,pv:(n2 n1 n0),metric: 1),...]
```

Note that this final state corresponds to one of the stable path assignments of Disagree described in Section 1, where node `n1` sets its best path to `[n1 n0]`, and node `n2` sets its best path to `[n2 n1 n0]`.

On the other hand, with the `rew` command which employs a different rewriting strategy, divergence scenario is simulated and route oscillation is observed in the simulation. This is because `frewrite` employs a depth-first position-fair rewriting strategy, while `rewrite` employs a left-most, outer-most strategy that coincides with the execution trace that leads to divergence.

Route Oscillation Detection While Maude commands `frew/rew` explore a small portion of possible runs of the instance, the `search` command allows us to exhaustively explore the entire execution space. To exhaustively search BGP execution for route oscillation, we only need to first input the BGP instance's network topology and routing policy to generate the corresponding initial state, as described in Section 6.1.2; and then use the `search` command to automatically search for oscillation. For example, for Disagree, we run:

```
search [1] gadget =>+ X such that detectCycle(X) = true .
```

Here, `gadget` is Disagree's initial state, and `=>+ X` tells Maude to search for any reachable network state `X` such that at that state, the logger `pa` contains recurring path assignment (`detectCycle(X)=true`). `search` command exhaustively explores Disagree runs and returns with the first Disagree state that exhibits oscillation:

```
{pa : Logger | history: ({[n1 n2 n0] [n2 n0]}
                        {[n1 n2 n0] [n2 n1 n0]}
                        {[n1 n2 n0] [n2 n0]}
                        {[n1 n0] [n2 n0]})}
[n1 : router |...] [n2 : router |...] ...
```

Here, the resulting path assignment content in `pa` exhibits an oscillation (line 1, line 3).

In general, Maude allows us to exhaustively search for violation of a safety property `P` by running the following command:

```
search initialNetwork =>+ X:Configuration such that P(X) == false.
```

which tells Maude to exhaustively search for a network state `X` that violates `P` along all possible execution traces from the initial state `initialNetwork`. If Maude returns with `No solution`, we can conclude property `P` holds for all execution traces.

6.2 Verifying Declarative Networks

To prove the correct of declarative implementation, and analyzing alternative routing mechanisms such as HLP, *FVR* further provides formal analysis support to verify declarative network. This chapter presents *FVR*'s ability to maps declarative network programs [76] automatically into logical *axioms* that can be directly used in existing theorem provers to validate protocol correctness. *FVR* is a significant improvement compared to existing use case of theorem proving which typically require several man-months to construct the system specifications.

FVR takes as input *NDlog* program specifications of the routing mechanisms protocol. Since most theorem provers leverage type information, *FVR* further includes a *Type Schema* with the *NDlog* program specifications. This is not unlike a database-like schema storing the attribute types of all network state being used. In order to carry out the formal verification process, the *NDlog* programs and schema information are automatically compiled into formal specifications recognizable by a standard theorem prover (e.g. PVS [62], Coq [3]) using the *axiom generator*. At the same time, the protocol designer specifies high-level invariant properties of the protocol to be checked via two mechanisms: invariants can be written directly as theorems into the theorem prover, or expressed as *NDlog* rules which are then automatically translated into theorems using the axiom generator. The first approach increases the expressiveness of invariant properties, where one can reason with invariants that can be only expressible in higher order logic. The second approach has restricted expressiveness based on *NDlog*'s use of Datalog, but has the added advantage that the same properties expressed in *NDlog* can be verified by both theorem prover and at run-time.

To illustrate the verification process, we step through the path-vector protocol example, used in BGP networks. For ease of exposition, we defer the treatment

of soft-state derivations and events to Section 6.2.3, focusing instead on traditional *hard-state* data (with infinite lifetimes) that are valid until explicitly deleted.

6.2.1 Path-vector Protocol in Declarative Network

We present an example *NDlog* program that implements the *path-vector* protocol [57], a standard textbook route protocol used for computing paths between any two nodes in the network.

```
p1 path(@S,D,P,C):- link(@S,D,C),P=f_init(S,D).
p2 path(@S,D,P,C):- link(@S,Z,C1), path(@Z,D,P2,C2),C=C1+C2,
                    P=f_concatPath(Z,P2), f_inPath(P2,S)=false.
p3 bestPathCost(@S,D,min<C>):-path(@S,D,P,C).
p4 bestPath(@S,D,P,C):- bestPathCost(@S,D,C), path(@S,D,P,C).
Query bestPath(@S,D,P,C).
```

The program takes as input `link(@S,D,C)` tuples, where each tuple corresponds to a copy of an entry in the neighbor table, and represents an edge from the node itself (`S`) to one of its neighbors (`D`) of cost `C`. *NDlog* supports a *location specifier* in each predicate, expressed with `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, `link` tuples are stored based on the value of the `S` field.

Rules `p1-p2` recursively derive `path(@S,D,P,C)` tuples, where each tuple represents the fact that the path from `S` to `D` is via the path `P` with a cost of `C`. Rule `p1` computes one-hop reachability trivially given the neighbor set of `S` stored in `link(@S,D,C)`. Rule `P2` computes transitive reachability as follows: if there exists a link from `S` to `Z` with cost `C1`, and `Z` knows about a shortest path `P2` to `D` with cost `C2`, then transitively, `S` can reach `D` via the path `f_concatPath(Z,P2)` with cost `C1+C2`. Note that `p1-p2` also utilizes two list manipulation functions to maintain path vector `p`: `f_init(S,D)` initializes a path vector with two elements `S` and `D`, while `f_concatPath(Z,P2)` pre-pends `Z` to path vector `P2`.

Rules `p3-p4` take as input `hop` tuples generated by rules `p1-p2`, and then derive the hop along the path with the minimal cost for each source/destination pair. The output of the program is the set of `bestPathHop(@S,D,Z,C)` tuples, where each tuple stores the next hop `Z` along the shortest path from `S` to `D`. To prevent computing paths with cycles, an extra predicate $f\_inPath(P,S) = false$ is used in rule `p2`, where the function $f\_inPath(P,S)$ returns true if node *S* is in the path vector *P*.

The execution model of declarative networks is based on a distributed variant of the standard evaluation technique for Datalog programs that is commonly known as *semi-naïve* (SN) evaluation [37], with modifications to enable pipelined asynchronous evaluation suited to a distributed setting. Reference [37] provides details on the implementation and execution model of declarative networking.

For the purposes of formal verification, we do not consider the location specifiers within the proof. This does not affect the program in terms of the set of eventual facts being generated but does affect the notion of data distribution. Our extended technical report [75] elaborate this issue in greater detail.

Axiom Generation: From *NDlog* rules to PVS Axioms The first step in *FVR* declarative network verification involves the *automatic* generation of PVS *formalization* (or axioms) directly from *NDlog* rules. Based on the proof-theoretic semantics of Datalog [65], there is a natural and automatic mapping from *NDlog* rules to PVS axioms.[3] Before showing the actual PVS encoding for the path-vector protocol, it is informative to understand the proof-theoretic semantics of `p1` and `p2` as inference rules used in proof system:

The inference rule `p1` expresses the logical statement $\forall(S,D,P,C).link(S,D,C) \wedge P = f_{init}(S,D) \implies path(S,D,P,C)$

[3]The equivalence of *NDlog*'s proof-theoretic semantics and operational semantics guarantees that *FVR* is sound in the sense that, the correctness property established by *FVR* corresponds precisely to the operational semantics of *NDlog* execution.

Rule `p2` is slightly more complex as some attribute variables do not appear in the resulting head. The general technique to express these variables is in terms of existential quantification. Accordingly, rule `p2` expresses the logical statement that $\forall(S, D, P, C).\exists(C_1, C_2, Z, P_2).link(S, Z, C_1) \wedge bestPath(Z, D, P_2, C_2) \wedge C = C_1 + C_2 \wedge P = f_{concatPath}(Z, P_2) \implies path(S, D, P, C)$

From the above logical statements, *FVR* generates the following axioms:

```
path_generate: AXIOM
FORALL (S,D,Z:Node)(C:Metric)(P:Path):(link(S,D,C) AND P=f_init(S,D)) OR
  ((EXISTS (P2:Path)(C1,C2:Metric):(link(S,Z,C1) AND bestPath(Z,D,P2,C2)
    AND C=C1+C2 AND P=f_concatPath(Z,P2))) =>path(S,D,P,C)
path_close: AXIOM
 FORALL (S,D,Z:Node)(C:Metric)(P):path(S,D,P,C)=>
   ((link(S,D,C) AND P=f_init(S,D)) OR (EXISTS (Z:Node)(P,P2:Path)
     (C1,C2:Metric): (link(S,Z,C1) AND bestPath(Z,D,P2,C2) AND C=C1+C2
      AND P=f_concatPath(Z,P2))))
```

The first `path_generate` axiom is generated in a straightforward manner from rules `p1` and `p2`, where the logical `OR` indicates that `path` facts can be generated from either rule. The `path_close` axiom indicates that the `path` tuple is the smallest set derived by the two rules, ensuring that these axioms automatically generated in *FVR* correctly reflected the minimal model of *NDlog* semantics. The list manipulation functions `f_concatPath` and `f_init` are predefined from PVS primitive types.

PVS provides *inductive definitions* that allows the two axioms above to be written in a more concise and logically equivalent form:

```
path(S,D,(P: Path),C): INDUCTIVE bool =
  (link(S,D,C) AND P=f_init(S,D) AND Z=D) OR (EXISTS (C1,C2:Metric)
  (Z2:Node) (P2:Path): link(S,Z,C1) AND path(Z,D,P2,C2) AND
  C=C1+C2 AND P=f_concatPath(S,P2) AND f_inPath(S,P2)=FALSE)
```

The universal quantifiers over the attributes to `path` (i.e. `S,D,Z...`) are implicitly embedding and existential quantifiers such as `C1` and `C2` are explicitly stated. *FVR* axiom generator always produces this inductive definition, and employs the axiom form only in the presence of mutual dependencies among the head predicates which makes PVS inductive definition impossible. Also note that the use of `f_inPath(S,P2)=FALSE` constraint prevents loops in `path`.

Accordingly, *NDlog* rules `p3-p4` are automatically compiled into PVS formalization in a similar way:

```
bestPathCost(S,D,min_C): INDUCTIVE bool =
  (EXISTS (P:Path): path(S,D,P,min_C)) AND (FORALL (C2:Metric):
    (EXISTS (P2:Path): path(S,D,P2,C2)) => min_C<=C2)
bestPath(S,D,P,C):INDUCTIVE bool =
  bestPathCost(S,D,C) AND path(S,D,P,C)
```

In addition to the above PVS encoding for *NDlog* rules, type definitions are produced automatically from the database schema information. For instance, given a database schema definition for `link(src:string, dst:string, metric:integer)` the corresponding PVS type declaration is `link:[Node,Node,Metric -> bool]` where `Node` is declared as a string type and `Metric` as an integer type.

6.2.2 Verifying Path-Vector Protocol

The next step involves proving actual properties in PVS. Properties are expressed as PVS *theorems* and serve as *starting points* (or *goals*) in the proof construction process. We illustrate this process by verifying the *route optimality* property in the path-vector protocol expressed in the following PVS `bestPathStrong` theorem:

```
bestPathStrong: THEOREM
  FORALL (S,D:Node) (C:Metric) (P:Path): bestPath(S,D,P,C) =>
    NOT (EXISTS (C2:Metric) (P2:Path): path(S,D,P2,C2) AND C2<C)
```

The above theorem specifies that for a given `bestPath(S,D,P,C)` from `S` to `D,P` is the *optimal* path, i.e. there does not exist another path `P2` from `S` to `D` with lower cost `C2`.

Given the above theorem, one can then utilize PVS to carry out the proof process. PVS performs the proof in a *goal-directed* fashion, in this case, starting from the `bestPathStrong` goal, and then recursively reducing it to sub-goals until all sub-goals are trivially true. PVS has approximately 100 built-in proof strategies, of which 20 are usually sufficient to automate a majority of the proof effort. We display the straw-man proof process that does not utilize any user-defined proof strategies specific to declarative network beyond PVS's built-in proof commands:

```
("" (skosimp*) (expand bestPath) (prop) (expand bestPathCost)
(prop) (skosimp*) (inst -2 C2!1) (grind))
```

The proof script reflects the interactive proof process in PVS directed by the user, where PVS takes care of all low level proof details and allows the user to concentrate on high-level proof strategies. Without going into details of each PVS command, we provide a high-level intuition of each step. The first command `skosimp*` performs repeated skolemization that removes universal quantifiers `S,D,C` and `P` in the theorem. Skolemization is generally the first proof step to try in proving any universal quantified theorems. The subsequent two `expand` commands are used to unfold the earlier inductive definition shown in 6.2.1, each followed by `prop` that performs proportional simplification. Then `skosimp*` is employed to remove universal quantifiers and `inst` to instantiate the existential quantified variable with proper instance (`C2!1`) . The rest of the proof is complete by using PVS's `grind` command which performs skolemization, heuristic instantiation, propositional simplification and decision procedures for linear arithmetic and equality.

Once the above proof script is supplied, PVS requires only fraction of a second to carry out the actual proof. When the proof is completed, it covers *all* instances

of the network. This is in contrast to model checking, which explores only specific network instances. In addition to proving the route optimality property of the declarative path-vector protocol, we have proved properties such as the potential cycles in the protocol if the cycle check (enforced using the `f_inPath` function) is removed.

The straw-man proof process here is restricted to PVS's built-in proof commands, and does not utilize any user-defined proof strategies that exploits domain-specific information. As a result, the proof requires an expert in declarative network and theorem proving. Given that our target users are network designers, the proof process should ideally be automated. In reference [75], we discuss the potential of using domain-specific PVS strategies tailored to declarative networking to support the proof construction.

6.2.3 Soft-state, Events and Network Dynamics

Up to this point, we have limited our verification to a subset of the complete *NDlog* language by omitting the treatment of *soft-state tuples (i.e. predicates)*. This simplification enables us to generate axioms recognizable by a theorem prover directly from *NDlog* programs without having to worry about the semantics of time and data expiration. In practice, soft-state data and events are central in network protocols, and adopted in many declarative network implementations. In the rest of this section, we will introduce the soft-state model in declarative networking, describe how rules with soft-state predicates (referred as *soft-state rules*) can be verified in a similar fashion, by first rewriting soft-state rules into logically equivalent rules with only hard-state predicates (i.e.*hard-state rules*).

Soft-state Model in Declarative Networking Declarative networking incorporates support for *soft-state* [58] derivations commonly used in networks. In the soft state storage model, all data (input and derivations) has an explicit "time to live"

(TTL) or lifetime, and all expired tuples must be explicitly reinserted with their latest values and a new TTL or they are deleted.

To support soft-state, an additional language feature is added to the *NDlog* language, in the form of a `materialize` declaration at the beginning of each *NDlog* program that specifies the TTL of predicates. For example, the expression `materialized(link,10,keys(1,2))` specifies that the `link` tuple is stored at a table with primary key set to the first and second attributes (denoted by `keys(1,2)` and that each `link` tuple has a lifetime of 10 seconds[4]. If the TTL is set to infinity, the predicate will be treated as *hard-state*.

The soft-state storage semantics are as follows. When a tuple is derived, if there exists another tuple with the same primary key but differs on other attributes, an *update* occurs, in which the new tuple replaces the previous one. On the other hand, if the two tuples are identical, a *refresh* occurs, in which the existing tuple is extended by its TTL.

For a given predicate, in the absence of any `materialize` declaration, it is treated as an *event* predicate with lifetime set to zero. Since events are not stored, they are primarily used to trigger other rules or in response to network events. Reference [37] provides more details on how soft-state storage model and events are implemented within a declarative networking engine.

Soft-state to Hard-state Rewrite in *FVR* The rule rewrite consists of two steps. First, all soft-state predicates of the form `p(...)` where "`...`" refer to predicate arguments, are translated into an equivalent hard-state predicate of the form `p(...,Tc,Tl)`, where the additional attributes `Tc` and `Tl` denote the creation time and lifetime of each tuple `p` respectively. This initial rewrite step makes explicit the creation time and lifetime by adopting `Tc`, `Tl` in each soft-state predicate. Event

[4]Following the conventions of the *P2* declarative networking system, attribute 0 is reserved for the predicate name.

predicates are rewritten in a similar fashion. However, `Tl` is omitted since events have zero lifetime by definition.

After the first step, additional constraints reflecting soft-state semantics are added to ensure that all soft-state facts only process with other facts valid within the same window period of time, as expressed in terms of constraints over `Tc` and `Tl`. Consider soft-state rules of the form, $e :- e_1, s_1, s_2, ..., s_n$. This rule triggered by input event e_1 with creation time Tc_{e1}, takes as input both the triggering event and several soft-state predicates $s_1, s_2, ..., s_n$, and generates a event. The rewritten equivalent hard-state rules is of the form:

$$e(..., Tc_{e1}) :- e_1(..., Tc_{e1}), s_1(..., Tc_{s1}, Tl_{s1}), s_2(..., Tc_{s2}, Tl_{s2}), ..., s_n(..., Tc_{sn}, Tl_{sn}),$$
$$Tc_{s1} < Tc_{e1} \leq Tc_{s1} + tl_{s1}, ..., Tc_{sn} < Tc_{e1} \leq Tc_{sn} + Tl_{sn}.$$

Since the event e_1 directly triggers the derivation of e, the creation time of the derived event e is set to be the same as that of the input e_1 (i.e. Tc_{e1}). An additional n constraints $Tc_{si} < Tc_{e1} \leq Tc_{si} + Tl_{si}$ are added to ensure that only soft-states s_i with valid time interval $[Tc_{si}, Tc_{si} + Tl_{si}]$ that always overlaps with Tc_{e1} are used to generate e.

Another possible class of soft-state rules are of the form, $e :- s_1, s_2, ..., s_n$, where an event e is generated by sets of soft-states. The main difference compared to the previous soft-state rule is the lack of a triggering event. The rewritten hard-state rule is of the form:

$$e(..., Tc) :- s_1(..., Tc_{s1}, Tl_{s1}), s_2(..., Tc_{s2}, Tl_{s2}), ..., s_n(..., Tc_{sn}, Tl_{sn}), Tc = max < Tc_{s1},$$
$$Tc_{s2}, ..., Tc_{sn} >, Tc_{s1} < Tc \leq Tc_{s1} + tl_{s1}, ..., Tc_{sn} < Tc \leq Tc_{sn} + Tl_{sn}.$$

Note that Tc is set to the max of all possible creation times of the input soft-state predicates (since the derived fact is true only when all the input facts are present).

The same rewrite process applies to rules that derive soft-state predicates s instead of events e. The main difference is an additional `Tl` attribute to s in the rewritten rule. This `Tl` attribute is set to the to the declared lifetime in corresponding table for s (indicated in the `materialize` statement).

6.2.4 Alternative Routing Mechanisms: Distance-Vector

In this section, we illustrate the capability of *FVR* in reasoning about eventual semantics of protocols in dynamic networks. We base our illustration on the verification of the *distance-vector* protocol, commonly used for computing shortest routes in a network.

Distance Vector Protocol Specification in *NDlog* The following soft-state *NDlog* program implements the distance-vector protocol, computing best paths with least cost:

```
materialize(hop,10,keys(1,2,3)).
materialize(bestHop,10,keys(1,2)).
materialize(bestHopCost,10,keys(1,2)).
dv1 hop(@S,D,D,C) :- link(@S,D,C).
dv2 hop(@S,D,Z,C) :- hopMsg(@S,D,Z,C).
dv3 bestHopCost(@S,D,min<C>) :- hop(@S,D,Z,C).
dv4 bestHop(@S,D,Z,C) :- bestHopCost(@S,D,C), hop(@S,D,Z,C).
dv5 hopMsg(@N,D,S,C1+C2):-periodic(@S,5),bestHop(@S,D,Z,C1),link(@S,N,C2).
Query bestHop(@S,D,Z,C)
```

This program derives soft-state predicates `hop`, `bestHop`, and `bestHopCost` with TTL of 10 seconds, and an event predicate `hopMsg`, and takes as input `link` tuples which represents dynamic network topology and is implemented by some periodic neighbor maintenance mechanism.

First, rules `dv1-dv2` derive `hop(@S,D,Z,C)` tuples, where `Z` denotes the next hop (instead of the entire path) along the path from `S` to `D`. Second, the protocol is driven by the periodic generation of `hopMsg(@S,D,Z,C)` in rule `dv5`, where each node `S` advertises its knowledge of all possible best hops table (`bestHop`) to all its neighbors. Note that bidirectional connectivity and cost is assumed. Each node receives the advertisements as `hopMsg` events (rule `dv2`) which it then stores

locally in its hop table. Finally, Rules dv3-dv4 compute the best hop for each source/destination pair in a similar fashion as the earlier path-vector protocol.

Unlike the path-vector protocol presented in Section 6.2.1, the distance-vector protocol computes only the *next hop* along the best path, and hence does not store the entire path between any two nodes.

Soft-state Rewrite and Automated Axiom Generation The following *NDlog* rules dv1-dv6 shows the equivalent hard-state rules after applying the soft-state rewrite process described in Section 6.2.3.

```
dv1 hop(@S,D,D,C,Tc,10) :- link(@S,D,C,Tc,10).
dv2 hop(@S,D,Z,C,Tc,10) :- hopMsg(@Z,D,W,C2,Tc2), Tc=Tc2+5, C=C2+1.
dv3 bestHopCost(@S,D,min<C>,Tc,10) :- hop(@S,D,D,C,Tc,10).
dv4 bestHop(@S,D,Z,C,Tc,10) :- bestHopCost(@S,D,C,Tc,10),
                               hop(@S,D,Z,C,Tc1,10), Tc1<Tc<=Tc1+10.
dv5 hopMsg(@N,D,Z,C,Tc) :- periodic_dv(@S,5,Tc), bestHop(@S,D,Z,C,Tc1,10),
                      link(@S,N,C,Tc2,10), Tc2<Tc<=Tc2+10, Tc1<Tc<=Tc1+10.
dv6 periodic_dv(@S,5,Tc) :- periodic_dv(@S,5,Tc2), Tc=Tc2+5
Query bestHop(@S,D,Z,C,Tc,Tl)
```

Rules dv1-dv5 are the corresponding hard-state rewrites, and dv6 emulates the behavior of periodic streams employed in dv5, as described in Section 6.2.3. We introduce an extra constraint Tc=Tc2+5 in rule dv2. This condition is required so that causality of rule execution is preserved within one interval: resulting hopMsg events generated within one periodic interval derives hop facts used in the next period internal and not vice versa. We note that this addition constraint is automatically added: required only in cases when rules depend on each other in a cyclical fashion (e.g. hop derived in dv1-dv2, hopMsg in dv5, and bestHop in dv4), a dependency that can be detected via static check.

Based on this rewritten program, the automatically generated PVS axioms are as follows:

```
hopMsg(S,D,Z,C,Tc): INDUCTIVE bool =
  (EXISTS (Tc2,T3:Time): bestHop (S,D,Z,C,Tc2,10) AND periodic(S,5,Tc)
    AND link(S,D,Tc3,10) AND Tc2<Tc<=Tc2+10 AND Tc3<Tc<=Tc3+10 AND C=1)
hop(S,D,Z,C,Tc,Tl): INDUCTIVE bool =
  (link(S,D,Tc,10) AND Z=D AND Tl=10 AND C=1) OR (EXISTS (C2:Metric)
    hopMsg(S,D,Z,C2,Tc2) AND C=C2+1 AND Tl=10 AND Tc=Tc2+5)
bestHopCost(S,D,MIN_C,Tc,Tl): INDUCTIVE bool =
  (EXISTS (Z:Node): hop(S,D,Z,MIN_C,Tc) AND Tl=10) AND
    (FORALL (C:Metric): (EXISTS (Z:Node): hop(S,D,Z,C,Tc,10))=>MIN_C<=C)
bestHop_refresh: AXIOM
  FORALL (S,D,Z:Node) (C:Metric) (Tc:Time): bestHopCost(S,D,C,Tc,10)
    AND hop(S,D,Z,C,Tc,10)=>bestHop(S,D,Z,C,Tc,10)
bestHop_close: AXIOM
  FORALL (S,D,Z:Node) (C:Metric) (Tc:Time): bestHop(S,D,Z,C,Tc,10)
    => (bestHopCost(S,D,C,Tc,10) AND hop(S,D,Z,C,Tc,10))
periodic_dv(S,I,Tc): INDUCTIVE bool =
  EXISTS (Tc2:Time): periodic_dv(S,I,Tc2) AND Tc=Tc2+5 AND I=5
```

Recall automatic axiom generation process in Section 6.2.1, PVS axioms would be explicitly used in face of mutual dependencies between rules (that derive `bestHop`, `hop`, and `hopMsg`). To break the dependency, we therefore specify `dv4` with two axioms `bestHop_refresh` and `bestHop_close`.

Eventual Convergence Proof in a Stable Network The lack of knowledge of the entire path in the distance-vector protocol comes at the expense of potential update loops in the presence of link updates. This is a well-known limitation of the distance-vector protocol, commonly known as the *count-to-infinity* problem.

Our verification is performed on a 4-node network instance as shown in Figure 6.2. Note that this instance represents a loop consisting of three nodes (`a`, `b`, and `c`) that can reach the rest part of the network via a fourth node `d`, and the results of this verification apply to any *arbitrary* network that contains such a loop. For ease

of exposition we do not consider computation of `link` tuple here and supply this network instance using the following PVS inductive definition, where each clause connected by logical operator `OR` represents a link between two nodes:

```
link(S,D,C,Tc,Tl): INDUCTIVE bool =
((S=a AND D=b AND C=1 AND Tl=10 AND (EXISTS (i:posnat): Tc=5*i)) OR
((S=b AND D=c AND C=1 AND Tl=10 AND (EXISTS (i:posnat): Tc=5*i)) OR
...
((S=a AND D=d AND C=1 AND Tl=10 AND (EXISTS (i:posnat): Tc=5*i))
```

Network convergence is expressed using the following theorem:

```
bestHopCost_converge: THEOREM
EXISTS (j:posnat): FORALL (S,D:Node)(C:Metric)(i:posnat): (i>j)
  => bestHopCost(S,D,C,5*i,10) = bestHopCost(S,D,C,5*j,10)
```

Given an input network, the distance-vector protocol requires a number of rounds of communication among all nodes, for route advertisements (in the form of `hopMsg`) to be propagated in the network. In the above theorem, the existential quantified variable `j` denotes the initial number of periodic intervals (set to be at least the network diameter) required to propagate all route advertisements. The theorem states that for any arbitrary time `i` after `j`, the value of `bestHopCost` always converges (i.e. no longer changes).

Count-to-Infinity Analysis in a Dynamic Network In the final *FVR* example, we demonstrate the capability of *FVR* to prove the presence of the *count-to-infinity* problem in the distance-vector protocol. This is a well-studied limitation where the protocol potentially *diverges* (i.e. not reach steady state) in the presence of link failures.

Before showing the actual proofs, we provide a textbook example [57] that clearly demonstrates the problem intuitively. Revisiting the network in Figure 6.2,

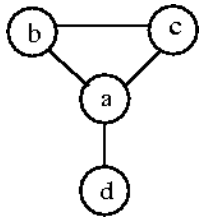

Figure 6.2: *Network Dynamics*

when `link(a,d)` fails, node `a` would advertises this information to its immediate neighbors `b` and `c`. However, despite the fact that `d` is no longer reachable from either `a` `b` or `c`, based on information that `c` can reach `d` in two hops, `b` would conclude that it can reach `d` in three hops. Node `c` makes a similar conclusion. In the next round of updates, node `a` learns that `b` and `c` can reach `d` in three hops, and updates its distance to `d` as four accordingly. This cycle continues indefinitely, resulting in the count-to-infinity problem.

The proof requires a network scenario that results in a count-to-infinity problem. Using the example described above, we supply this network dynamics using the following PVS inductive definition:

```
link (S,D,C,Tc): INDUCTIVE bool =
((S=a AND D=b AND C=1 AND (EXISTS (i:posnat): Tc=5*i) AND Tc<100)) OR
((S=b AND D=a AND C=1 AND (EXISTS (i:posnat): Tc=5*i) AND Tc<100)) OR
...
((S=a AND D=d AND C=1 AND (EXISTS (i:posnat): Tc=5*i) AND Tc<100)) OR
((S=d AND D=a AND C=1 AND (EXISTS (i:posnat): Tc=5*i) AND Tc<100)) OR
((S=a AND D=b AND C=1 AND (EXISTS (i:posnat): Tc=5*i) AND Tc>=100)) OR
((S=b AND D=a AND C=1 AND (EXISTS (i:posnat): Tc=5*i) AND Tc>=100)) OR
...
((S=c AND D=b AND C=1 AND (EXISTS (i:posnat): Tc=5*i) AND Tc>=100)) OR
((S=b AND D=c AND C=1 AND (EXISTS (i:posnat): Tc=5*i) AND Tc>=100))
```

The definition indicates that the `link(a,d)` and `link(d,a)` facts are only present

before time `100`, denoting that a link failure between nodes `a` and `d` happens at time `100`. The count-to-infinity theorem is expressed as follows:

```
bestHop_increase_to_infinity: THEOREM
FORALL (a,b,d:Node)(t:Time)(c:Metric):(t>100 AND bestHop(a,d,b,c,t,10))=>
(EXISTS (t':Time)(c':Metric):(t'>t AND c'>c AND bestHop(a,d,b,c',t',10)))
```

The theorem above states that if the distance vector protocol diverges, the best hop from `a` to `d` will increase indefinitely over time, a symptom of the count-to-infinity problem. In reference [10], we have the complete proof of this theorem, as well as addition theorems that further verify the presence of the count-to-infinity problem. Interestingly, we have been able to prove a *stronger* PVS theorem specific to a three-node network cycle: $\forall b, d, a, c, t.(\exists i.t = i \times 5 \wedge t > 100) \implies (bestHop(b, d, a, c, t, 10) \implies bestHop(b, d, a, c + 2, t + 10, 10))$, which expresses the precise pattern that the value of `cost` argument increases by 2 at every two update intervals of 10 seconds.

We further verify that a well-known solution to this problem, known as the *split-horizon* solution can avoid any two-node cycle, and show that this solution is insufficient for fixing the count-to-infinity problem in a three-node cycle. Refer to our extended technical report [75] for more details.

6.3 Evaluation

We have analyzed well-known eBGP instances, including good gadget, bad gadget, disagree [24]. In addition, we analyze two iBGP configuration instances: a 9-node iBGP gadget [16] that is known to oscillate, and a 25-node configuration randomly extracted from the Rocketfuel [68] dataset. Rocketfuel is a well-known dataset on actual iBGP configurations that are made available to the networking community. Given that an ISP has complete knowledge of its internal router con-

	Disagree	**Bad**	**Good**	**9-node iBGP**	**25-node iBGP**
Simulation	2	NA	4	20	31
Exhaustive	2,10	181,641	10997,37692	20063,52264	723827,177483
Safe?	No	No	Yes	No	No

Table 6.2: Summary of BGP analysis in Maude. In the first row, each entry shows the simulation time in milliseconds. In the second row, for each entry, the first value denotes exhaustive search time in milliseconds, the second denotes number of states explored, and the third on whether our tool determines the instances to be safe ("Yes") or unsafe ("No").

figurations, the Rocketfuel experiment presents a practical use case for using our tool to check an actual BGP configuration instance for safety.

For each BGP instance, we simulate its possible executions using rewriting commands (*Simulation*), and check for route oscillation using exhaustive search (*Exhaustive*). We summarize our analysis results are as follows:

We have carried out these analysis on a laptop with 1.9 GB memory and 2.40GHz dual-cores running Debian 5.0.6. The version of Maude is Maude 2.4. While route oscillation detection explores the entire state space of the instance execution, the analysis time for rewriting based execution are measured for only one possible terminating execution (that converges to a stable path assignment).

Here we summarize findings from our case studies. Single-trace simulation is helpful in finding permanent routing oscillation. When simulating the execute trace that diverges, Maude does not terminate (e.g., in executing Bad gadget [5]). However, simulation can miss temporary oscillations which are only manifested on a particular executing trace. When Maude terminates, single-trace simulation time increases when network size grows. On the other hand, exhaustive search always provides a solid safety proof. For instances of similar network size, the search time for a safe instance (`good`) is considerably longer than that of an unsafe instance (`bad`). For instances of different sizes, as network size grows, exhaustive

[5]Bad gadget always diverges and does not have any stable path assignment, therefore, when we simulate bad gadget with rewriting, Maude does not terminate, and we do not record the statistics.

search time grows exponentially. Nevertheless, even for the 25-node scenario, exhaustive search can be completed in 12 minutes. As future work, we are going to scale our analysis technique to larger networks.

6.4 Summary

In this chapter, we present *FVR*'s ability to verify actual BGP systems. We first developed a Maude library for detecting routing anomalies in a policy configuration. To use the library, users provide specifications of the network topology and routing policies. The dynamic behavior of the resulting BGP system can then be analyzed automatically by Maude. To validate this approach, we performed safety analysis of well-known BGP instances and actual routing configurations.

Second, we use theorem prover PVS to verify BGP routing mechanism. We take as input declarative networking specifications written in the Network Datalog (*NDlog*) query language, and maps that automatically into logical *axioms* that can be directly used in PVS to validate correctness. *FVR* is a significant improvement compared to existing use case of theorem proving which typically require several man-months to construct the system specifications. To validate the use of PVS, we present case studies of verifying various properties, including routing optimality and eventual properties in dynamic settings.

Chapter 7

Scalability Techniques for Analysis

In Chapter 6, we have shown *FVR* automatically detects anomalies in policy configurations via Maude's exhaustive search capability. Unfortunately, this approach works only for small instances due to the state explosion problem. To address this limitation, this chapter presents a novel scalability technique called network reduction that transform policy configurations into smaller and simpler forms while preserving safety property (arc 7), as shown in Figure 7.1.

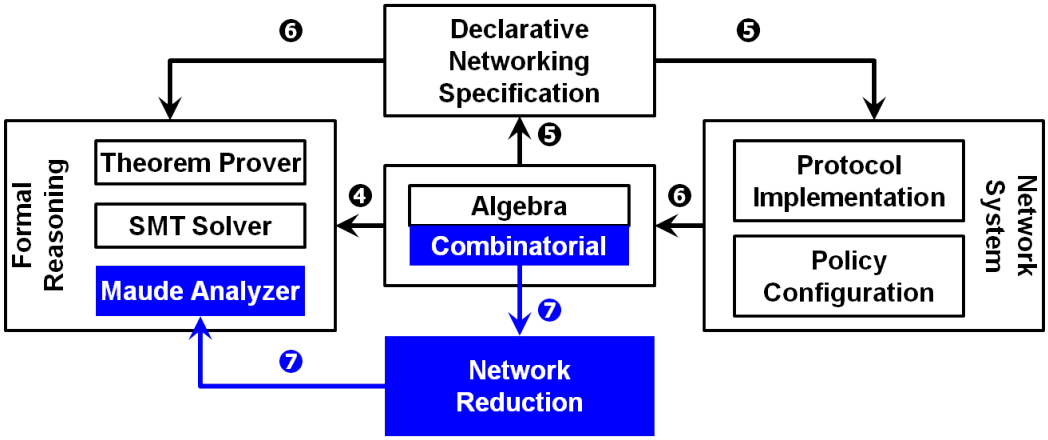

Figure 7.1: *FVR* Architecture: Network Reduction.

In network reduction, we provide two types of reduction rules that transform policy configurations by merging duplicate and complementary router configura-

tions to simplify analysis. We show that the reductions are sound, dual of each other and are locally complete. The reductions are also computationally attractive, requiring only local configuration information and modification.

We have developed a prototype of network reduction and demonstrated that it enables significant savings in analysis time due the the use of reduction. We evaluated the effectiveness of the reduction technique analysis on routing configurations ranging from synthetic networks to sampled CAIDA and Rocketfuel dataset, using the Maude-based safety analysis (Chapter 6) as reasoning engine. Our experimental results show that network reduction enables us to perform safety analysis efficiently, often completing the analysis on large networks that would otherwise not be possible to study within reasonable time.

In addition to making possible safety analysis on large networks that would otherwise not complete within reasonable time, network reduction is also a useful tool for discovering possible redundancies in BGP systems.

7.1 Network Reduction

7.1.1 Formal Model for Reduction

We first present the formal model used for performing reduction and analysis. The central data structure, called the *Enhanced Path Digraph* (EPD), is a compact representation of two configuration aspects of a BGP system: the *topology* of how routers are connected, and for each router, the export, import, and route preference *policies*.

The policy configuration problem can be understood independently of the means for calculating routes—the BGP path-vector mechanism as implemented by the various router vendors. Policy conflicts exist independently from the details of

how messages are exchanged and local data structures are updated. So the EPD model abstracts away the mechanism, and focuses on the policy itself.

Enhanced path digraph (EPD) is an extension of the path digraph structure [67], tuned to enable us to conveniently express and perform reduction. As we will see in Section 7.2, through this modification, EPD allows us to prove the correctness of reduction in a much more intuitive and concise way reasoning directly with path digraph in our prior work [84].

Definition 5 (Path digraph)**.** *Let (V, E) be a directed graph and let d be a designated 'destination' node in V. A* path digraph *instance on (V, E, d) is given by (P, E_v, E_p), where P is a set of paths in (V, E) terminating at d, and E_v and E_p are binary relations on P fulfilling the following properties:*

C1. *(p, q) is in E_v if and only if p is a prefix of q.*

C2. *If p and q have different origin nodes, then (p, q) is not in E_p.*

C3. *The restriction of E_p to any set of paths having the same origin node is a strict linear order.*

The relations E_v and E_p are called the transmission *and* preference *relations respectively.*

We may also write E_p as '$\prec$', where $p \prec q$ means that p is strictly preferred to q. The 'path digraph' structure, then, is the derived graph where the nodes are the elements of P and the arcs are $E_v \cup E_p$. For example, Figure 7.2 shows a network of three nodes `0`, `1` and `2`, where `0` is the particular destination. The paths are shown alongside their origin nodes, in preference order (so `1` prefers path `120` over `10`); any path not shown is not permitted. The path digraph is shown in Figure 7.3, where dashed arcs correspond to prefers arcs, and solid arcs for transmission arcs.

We can define a notion of 'stable solution' corresponding to the endpoint of the route computation process.

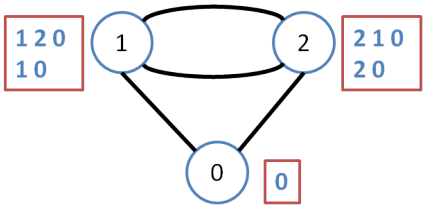

Figure 7.2: The network configuration of *Disagee*

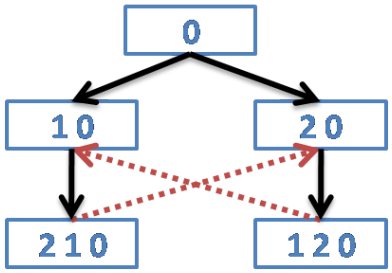

Figure 7.3: The path digraph for *Disagee*

Definition 6. *A* stable solution *to a path digraph* (P, E_v, E_p) *is a subset* S *of* P *that contains the empty path* ϵ_d *from* d *to itself, and such that any other path* q *is in* S *if and only if*

C1. *there is some* p *in* S *such that* (p, q) *is in* E_v*, and*

C2. *there is no* p' *in* S *such that* (p', q) *is in* E_p*.*

In general, there may be zero, one, or many stable solutions. If there is no stable solution then the routing protocol will certainly oscillate; if there are more than one, then it might oscillate (depending on details of the path-vector mechanism). If, however, there is exactly one stable solution, then the protocol will necessarily converge to it [67, 29]. A sufficient condition for this is that the path digraph be *acyclic*.

Theorem 7. *If a path digraph has no cycle (that is, the transitive closure of* $E_v \cup E_p$ *is irreflexive) then it has a unique stable solution.*

Proof. See [31]. □

The property of a path digraph having a unique stable solution implies that the configuration is both *safe* and *robust* [19, 14, 22, 64]. Informally, a routing configuration is safe if any fair execution sequence for the path-vector protocol must eventually result in convergence of the routing state. It is robust if it remains safe even after removing some subset of the nodes and arcs in the graph.

The transmission relation E_v forms an arborescence rooted at ϵ_d. It therefore contains, implicitly, data about the connectivity of the original graph. The extended structure, which we now define, makes that information more explicit.

Definition 8 (Extended path digraph (EPD))**.** *If (P, E_v, E_p) is a path digraph on (V, E, d) then the* extended path digraph *is (P, E_v, E_p, s), where s is the function from $P \setminus \{\epsilon_d\}$ to V that maps each path to its origin node.*

An EPD may be represented diagramatically by grouping the paths in P according to s, as in Figure 7.4. Here, and in the rest of this thesis, paths in P are represented by square boxes, nodes in V by ovals, preference arcs by dashed arrows and transmission arcs by solid arrows.

We will also use the following notation, for an EPD instance (P, E_v, E_p, s) on a graph (V, E, d):

- P_u is the set $\{p \in P \mid s(p) = u\}$.
- Concatenation of arcs and paths is denoted by '$\circ$'.
- $\Gamma^+(u) = \{v \in V \mid (u, v) \in E\}$ and $\Gamma^-(u) = \{v \in V \mid (v, u) \in E\}$.

We call elements of $\Gamma^+(u)$ the downstream neighbors of u, and elements of $\Gamma^-(u)$ the upstream neighbors of u. We will also use the notation $\Gamma^+(u, v)$ to mean the union of $\Gamma^+(u)$ and $\Gamma^+(v)$, and similarly for $\Gamma^-(u, v)$.

In general, a cycle in an EPD must involve more than one node in V, since the preference arcs alone do not contain cycles. That is, this paper does not consider inconsistent preference relations within a single node. More details are in Appendix A.1. Moreover, since the transmission arcs form a tree, the EPD cannot be cyclic unless there is a cycle in (V, E). That is, regardless of the routing preferences, convergence is guaranteed for any network that has no cycles.

7.1.2 Network Reduction

This section presents a rewriting calculus for policy-based routing systems, based on the idea of reducing a given network to one which is smaller, but has the same safety property. We define two specific rules, called *duplicate* and *complementary* reduction, for merging two router nodes in an EPD. Both of these are purely *local*, meaning that the operations only require examination of the relevant nodes and their immediate neighbors. In the following, assume we are working with a given EPD instance $G = (P, E_v, E_p, s)$ on a graph (V, E, d).

Definitions To ease the discussion of these reductions, we first introduce three auxiliary notions: 'consistent node', 'node rewrite', and 'strongly adjacent'. We say two nodes are consistent if their configurations do not form a cycle in the EPD representation, formalized as follows:

Definition 9 (Consistent nodes)**.** *Given a policy configuration's EPD (P, E_v, E_p, s) on the network graph (V, E, d), nodes u and v in V are* consistent *if there is no cycle in the EPD which consists only of paths p for which $s(p) \in \{u, v\}$.*

An example of nodes violating consistency is in Figure 7.4: nodes 1 and 2 are not consistent since there is a cycle of paths (`120`, `10`, `210` and `20`) that only involves these two nodes. This conflicting configuration causes route oscillation behavior. Such examples of consistency violation, where two nodes have a policy conflict,

should not be eliminated during reduction, in order for the problem to be detected in the final analysis. A consistency check is hence a precondition for reduction.

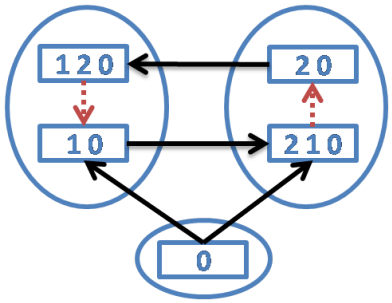

Figure 7.4: The EPD notation for the *Disagree* configuration.

Definition 10 (Strongly adjacent)**.** *Two nodes u and v in V are* strongly adjacent *if for every path in P_u, either v does not appear, or it appears as the next node following u; and likewise for P_v.*

Strong adjacency implies that two nodes are either immediate neighbors, or one does not route to destination through the other. It is also a precondition for reduction.

Definition 11 (Node rewrite)**.** *The procedure to rewrite node u to v is as follows: Rewrite the path $p \in P_u$ in u to $p' \in P_v$ in v by: If $p = u \circ w \circ r$, and $w \neq v$, then rewrite p to $w \circ r$; If $p = u \circ v \circ r$, then rewrite p to r; For all other cases, abort rewrite. Rewrite the preference among P_u to that among P_v by: Rewrite preference arc (p, q) to (p', q'), where p rewrites to p' and q to q'.*

The global rewrite on the EPD is straightforward, once the two nodes have been merged. All paths in P are rewritten according to the procedure of Definition 11, as is the transmission relation E_v. Write $p[u, v \mapsto w]$ for a path p where all occurrences of u or v are replaced with w, eliding any 'ww' subpath. The preferences at the new node w are determined by the specific reduction procedure; for any *other* node t,

the new path preferences on P_t are obtained as follows. For a path p, let $\hat{p}$ denote the path that is minimal according to E_p among $\{q \in P_t \mid q[u, v \mapsto w] = p\}$. Then the written E_p on t is $\{(\hat{p}, \hat{q}) \mid (p, q) \in E_p\}$: that is, a path 'inherits' the relative preference of the highest ranked path in its preimage.

Duplicate Reduction

Definition 12 (Duplicate condition)**.** *Suppose that u and v are two consistent and strongly adjacent nodes. Then v is a* duplicate *of u, if after rewriting v to u, the following conditions hold: (1) v's path P'_v is equivalent to (or a subset of) the u's paths P_u; (2) For every preference arc (p, q) in v, there exists (p', q') in u.*

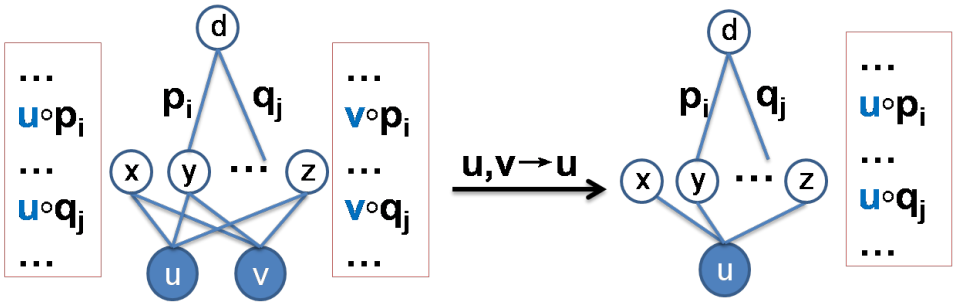

Figure 7.5: Nodes u, v are merged by duplicate reduction if they agree on how to route to destination d through their neighbors $x, y, \ldots, z$: For any path p_i, q_j, u, v agree on their preference.

The duplicate precondition ensures two nodes agree on the paths and their preference to reach the destination. The duplicate 'redundancy' of u and b is characterized in terms of the neighbors through which they route to destination d. The general duplicate reduction process is shown in Figure 7.5, where u and v are merged into one node u. One can view the local change as merging the paths of v into u, and this operation can be done consistently since they have the same routing preference.

Complementary Reduction In contrast to duplicate reduction, complementary reduction captures the redundancy observed from the point of the view of the neighbors of u and v. The intuition is that if all the neighbors route to destinations through u and v in a consistent way, then u and v can be combined into one node without changing the routing behavior of these neighbors. This is formalized as follows.

Definition 13 (Complementary Condition)**.** *Two consistent and strongly adjacent nodes u and v are* complementary *if, for any paths p and q in $P_u \cup P_v$, and any nodes x and y which are immediately downstream from u and v, the preference $(x \circ p, x \circ q)$ is in E_p if and only if $(y \circ p, y \circ q)$ is in E_p.*

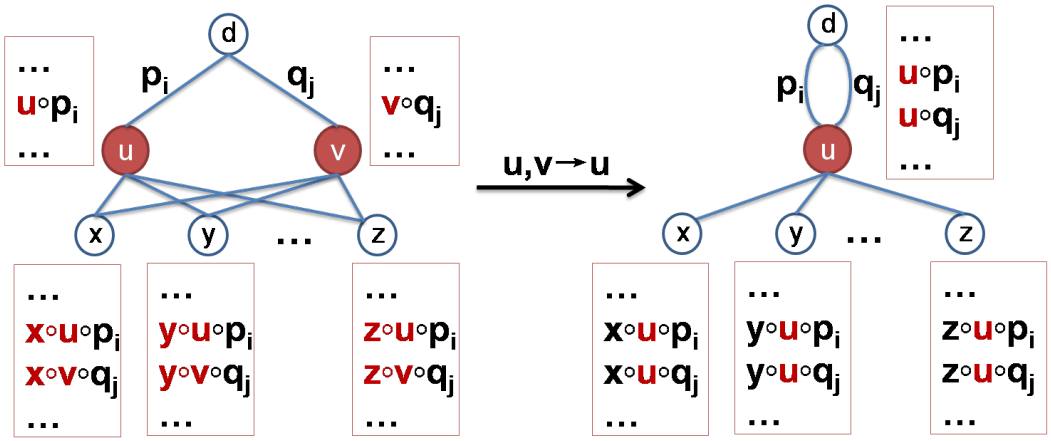

Figure 7.6: Nodes u, v are merged by complementary reduction if their neighbors $x, y, \ldots, z$ agree on how to route to destination d through them: After merging, the route preference for any path p_i, q_j are set according to the consensus among $x, y, \ldots, z$.

The general complementary reduction process is illustrated in Figure 7.6 where nodes u and v, whose neighbors $x, y, \ldots, z$ agree upon routes through them, are merged into one node. The merging is more subtle than the duplicate one: (1) The merged node w's paths *combine* those from u and v, i.e. P_w is the union of P_u and P_v; (2) The route preferences for the new node w are partly determined by the

consensus of preferences of their neighbors (in cases where the preference could not be derived from either u or v). That is, if u has path p (and not q) and v has path q (and not p) then we set p to be preferred over q in w if and only if all upstream neighbors x agree that $x \circ p$ is preferred over $x \circ q$.

Example Consider the configuration in Figure 7.7, Here, there is an AS with three border routers (3, 7 and 4) and two internal routers (4 and 6), as well as three external router nodes (1, 2, and the destination 0). The nodes 3 and 4 are complementary because their downstream neighbors (the internal nodes 5 and 6) agree on the preference among paths to 0. After merging them, the new node is again complementary to node 7. Following a second complementary reduction step, the two internal nodes are both duplicate, and can also be eliminated.

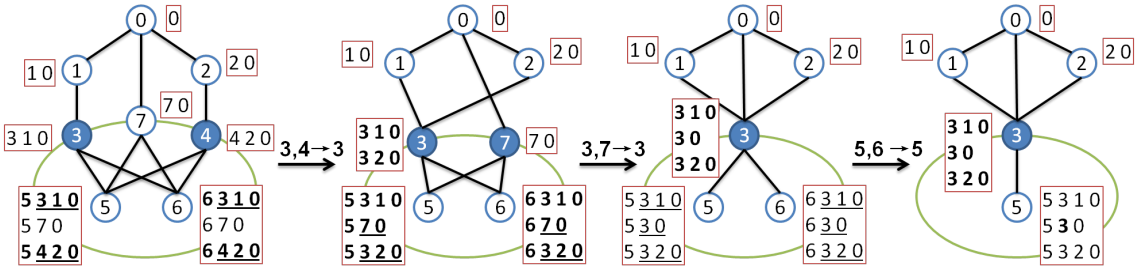

Figure 7.7: Application of complementary and duplicate reduction to border and internal routers, respectively.

Another example configuration is in Figure 7.8 (left), it consists of an AS consists of 5 nodes 3, 7, 4 (border gateway routers) and 5, 6 (internal routers), two external nodes 1, 2, and the destination 0. Nodes 3, 4 are complementary because their downstream neighbors (i.e. internal nodes 5, 6) agree upon the preference among paths to 0. Therefore 3, 4 can be merged. On the other hand, node 7 is not complementary with 3 nor 4 because 5, 6 have different preferences: while 5 prefers paths from 7 over that from 4, 6 prefers otherwise.

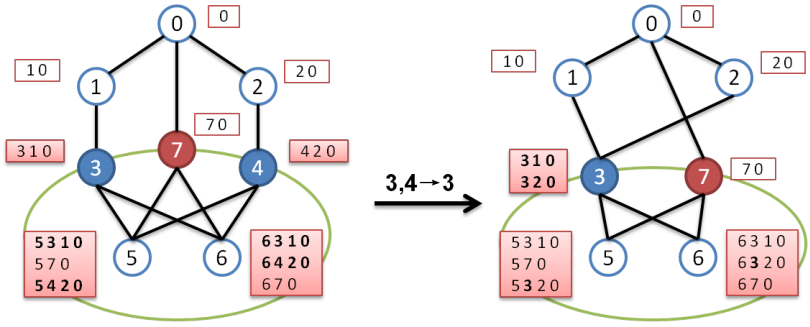

Figure 7.8: One-step reduction of complementary nodes $3, 4$

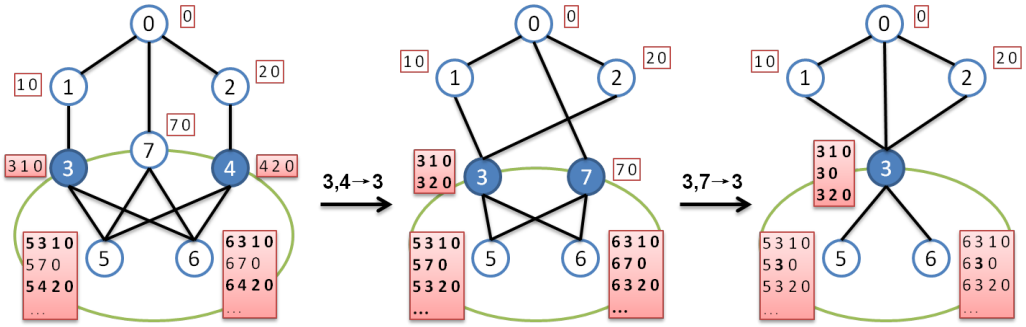

Figure 7.9: Two-steps reduction: when internal nodes have consensus on how to rank paths from border gateway nodes, all the border gateway nodes can be reduced into one.

In a configuration with same topology but different route preference, as shown in Figure 7.9 (left), all `3,4,7` are complementary nodes and can be reduced to one in two steps, because `5,6` agree upon the same route preference. Note that, after merging all border gateway routers, we are left with two internal routers that are duplicate. This is due to the dual nature of duplicate and complementary reduction, which we will revisit in details in Section 7.2.

7.2 Properties of Network Reduction

This section establishes the three properties of network reduction: (1) The *duality* property reveals that duplicate and complementary reduction are symmetric; (2) The *soundness* property enables us to use the reduced configuration to study the original one; (3) The *local completeness* property shows that reduction can be done efficiently just by examining only "local configuration" — the two nodes and their immediate neighbors, and that duplicate and complementary reductions form a complete repertoire of such "local" methods; and (4) The *confluence* property reveals the role of merging order in reduction. In this section we present the definitions and proof sketches. Our complete proofs are available in Appendix A.2.

Assume in the rest of the section that we are working with a given EPD instance $G = (P, E_v, E_p, s)$ on a graph (V, E, d), where u and v are two reducible (duplicate or complementary) nodes. Write N_{up} for the upstream neighbors of u and v, through which they route to d, and N_{down} for the downstream neighbors which route to the destination through u and v. Let G' be an instance to which G reduces by duplicate or complementary reductions.

7.2.1 Duality: Relating Duplicate and Complementary Reduction

Theorem 14. ***a.*** *If all the nodes in N_{from} can be merged into one node by (multiple steps of) complementary reductions, then u and v must be duplicate.* ***b.*** *If all the nodes in N_{to} can be merged into one node by (multiple steps of) duplicate reductions, then u and v must be complementary.*

Proof. Proof by definitions. The complete proof and its graph illustration are in Appendix A.2. □

The duality theorem reveals the symmetry between duplicate and complemen-

tary reduction, as prefigured in Section 7.1.2 (where the border routers were complementary but the internal, downstream routers were duplicate). It also implies that if two nodes' upstream (or downstream) neighbors can be reduced into one node in our calculus, then these two nodes themselves can be further merged into one.

7.2.2 Soundness

The main soundness result is that the reduced policy configuration has the same safety and robustness properties as the original one, and so we can use the reduced one to analyze the original.

Theorem 15. *If G' is safe then G is safe; and if G' experiences route oscillation, then in running G, there exists at least one execution trace that exhibits route oscillation.*

According to Theorem 2, we only need to prove that the rewriting process preserves the presence or absence of cycles in the configuration's EPD representation, as follows:

Lemma 1. *The path digraph of G is acyclic if and only if the path digraph of G' is acyclic.*

Proof. For duplicate reduction, we prove rewriting preserves cyclicity by constructing a cycle in G' for any cycle c in G. The duplicate rewrite from G to G' is defined by merging duplicate nodes u and v, and the proof proceeds by case analysis of whether any of the paths originating from u or v are on c. We prove rewriting preserves acyclicity via the contrapositive: if G' is cyclic then G is cyclic, which is also proved by construction.

For complementary reduction, the proof is similar thanks to the EPD formalization and the dual nature of the two rules. □

We only provide a proof sketch here, the complete proof and its graph illustration are in Appendix A.2.2.

7.2.3 Local Completeness

We first formalize the notion of "local reduction" and "local safety", and then prove that duplicate and complementary reductions are locally complete with regard to preserving the presence or absence of EPD cycles. Intuitively, a reduction rule applied to nodes u and v is "local", if it only requires information from u, v and their immediate neighbors ($\Gamma^-(u,v)$ and $\Gamma^+(u,v)$) in order to test the reduction precondition, and generate the configuration of the merged node.

Let N_{rest} stand for the nodes in V which are not within one hop of u or v. We introduce a binary relation $\sim_{u,v}$ on EPDs, capturing the idea that they only differ on the configuration of N_{rest}, by $G \sim_{u,v} G'$ if and only if the following hold:

1. G and G' are on graphs having the same set of nodes.
2. They have the same path configuration for u and v: so $P_u = P'_u$, $P_v = P'_v$, with the same preference arcs; and they have the same set of transmission arcs to and from u or v.
3. A preference arc $(y \circ p, y \circ q)$ is in E_p if and only if it is in E'_p, for any y in $\Gamma^+(u,v)$, and any p and q in $P_u \cup P_v$.

Definition 16 (Local Safety)**.** *A network reduction rule on G by merging u and v is locally safe, if it also preserves safety for any G' with $G' \sim_{u,v} G$.*

Theorem 17 (Local Completeness)**.** *If a network reduction rule that rewrites G by merging u and v is locally safe, then it must be either duplicate or complementary reduction.*

Proof. We use proof by contradiction to establish that if u and v are neither duplicate nor complementary, then the reduction rule that merges them is not locally safe. That is, there is some acyclic EPD G, where application of the merge results in G' being cyclic, but $G \sim_{u,v} G'$.

By assumption, G is acyclic, so in particular u and v are not on a cycle (see the left

of Figure 7.10). Consider an EPD where there is a series of transmission arcs from a downstream neighbor of v to an upstream neighbor of u (illustrated from y_2 to x_1). Merging u and v creates a cycle, shown in the right of Figure 7.10. □

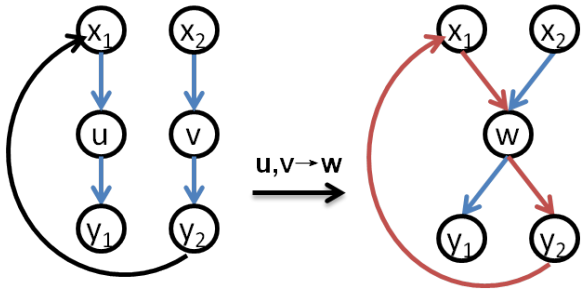

Figure 7.10: If u and v are neither duplicate nor complementary, merging them can create a cycle.

Note that, while duplicate and complementary reduction are locally complete, we do not exclude the existence of other safety preserving reduction that requires checking policy configuration beyond u, v and their neighbors. That is, we do not exclude less efficient algorithms for simplifying networks.

7.2.4 Confluence

This section discusses confluence properties of the reductions: we first prove duplicate reduction is confluent but complementary reduction is not.

Theorem 18. *[Duplicate reduction is confluent] If, for a set of nodes V, any pair of nodes u and v in V are duplicate, then V can be merged into one single node by multiple steps of duplicate reduction, regardless of the reduction order.*

Proof. By induction on the size of V.

The base case. $|V| = 2$ is trivial.

The induction step. For $|V| = k + 1 > 2$. Consider two nodes u and v in V, which by assumption are duplicate. By merging them into a new node z, we can rewrite

V to $V' = W \cup \{z\}$ where $W = V \setminus \{u, v\}$. By the induction hypothesis that any k pair-wise duplicate nodes can be merged into one node, it is sufficient to prove that V reduces to one node by showing that V' is pair-wise duplicate, since $|V'| = k$. By definition, in V', the subset W is pair-wise duplicate, so we only need to show that z is duplicate with any w in W. Since u and v are duplicate with w, it must be the case that z and w satisfy at least the duplicate conditions. Since $P_z = P_u \cup P_v$, and by the pair-wise duplicate definition we know that paths in P_u and P_w, in P_v and P_w, and in P_u and P_v always form a unique total ordering. That is, for any three paths $p \in P_u$, $q \in P_v$, and $r \in P_w$, we know how to set the preferences between any two of them. Then there must be be a unique ordering between the three of them, and so all paths from $P_u \cup P_v \cup P_w$ are totally ordered. □

On the other hand, complementary reductions are not confluent, counter-example is shown Figure 7.11(a). Nodes `u`, `v` and `w` have the same set of downstream neighbors. For example, node `u` has two paths p_2 and p_3, and there is some downstream preference $p_2 \prec p_3$. All downstream neighbors have consensus on preference among paths from `u` and `v` ($p_2 \prec p_1 \prec p_3$), and among paths from `v` and `w` ($p_2 \prec p_1 \prec p_4$). However, there is no consistent ranking for paths from `u` and `w`, since some nodes prefer p_3 over p_4, and others prefer the reverse. While complementary reduction can be applied to either `u` and `v` (as in Figure 7.11(b)), or `u` and `w` (as in Figure 7.11(c)), a further reduction step is not possible.

Finally, we show that duplicate reduction does not commute with complementary reduction, by exhibiting a counterexample. Consider the EPD in Figure 7.12, where nodes u and v are duplicate, and v and w are complementary. If u and v are merged into z through duplicate reduction, then this z is not reducible with w, due to the lack of consensus on paths p_3 and p_4 among downstream neighbors.

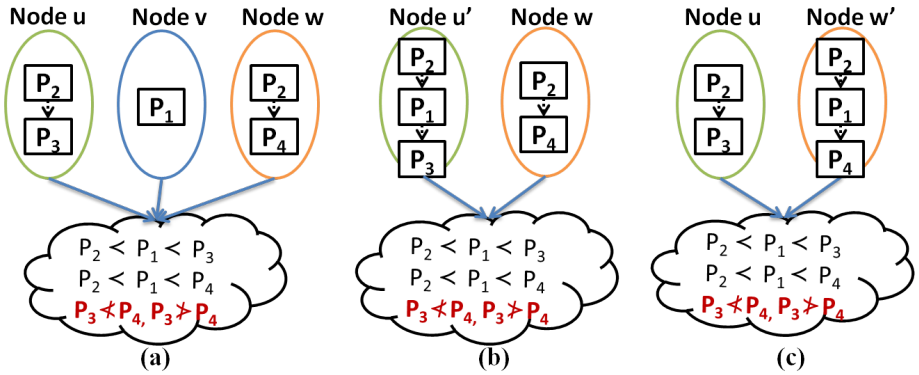

Figure 7.11: The EPD in (a) either rewrites to (b) or (c) depending on the order of two complementary reductions (u, v or v, w)

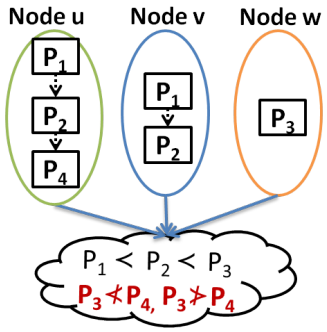

Figure 7.12: Duplicate/complementary reductions do not commute

7.3 Evaluation

We have implemented a prototype of network reduction using Maude. With the prototype, we demonstrate that network reduction is applicable on various networks, can be done efficiently at low overhead, and enables analysis of BGP configurations that cannot otherwise be completed. Moreover, by comparing BGP systems before and after reduction, we not only validate our reduction theory, but also gain insights into redundancy and conflicts in network configurations. We primarily selected Maude due to its existing libraries [85, 84] for modeling BGP systems and performing safety analysis [85].

7.3.1 Network Generation

We present evaluation on a variety of networks ranging from synthetic networks including configurations of Cisco guidelines [87], and random network topologies generated using GT-ITM, to actual network topologies including CAIDA inter-AS level topologies [4], and Rocketfuel router-level ISP topologies [68]. All experiments are carried out on an Intel Xeon 2.33GHz CPU with 4GB memory, running Linux 2.6.

Reduction on Synthetic Networks We evaluate network configurations that span multiple ASes, consisting of both iBGP and eBGP configurations. We first develop a model of a BGP system [60] in Maude, which consists of several ASes and routers running the path-vector protocol, and exchanging routes based on their import, export, and route selection policies. In particular, both the *Cisco-Synthetic* and *GT-ITM* network policies are realized by the *local preference* and *AS path* attributes for route selection, and import/export filtering for route exchange. In addition, we develop Maude functions that generates the EPD model from a BGP system in terms of topology and configuration attributes [60].

Cisco-Synthetic Network. To evaluate network reduction on well-designed, highly structural policy configurations proposed by Cisco, we construct various synthetic topologies combining full-mesh and reflection configurations according to these guidelines [87].

An *full mesh* topology is simply a complete graph of the routers. Our reduction theory will collapse all of these routers into a single node, so long as they implement the same policy. In *route reflector* configuration, the network graph is partitioned into a set of clusters. Inter-cluster communication is done by special routers configured as 'reflectors'; other routers within a cluster are clients of the local reflector(s). As depicted in Figure 7.13, the reflectors form a full-mesh core graph, while the clients are only connected to their reflectors. However, the clus-

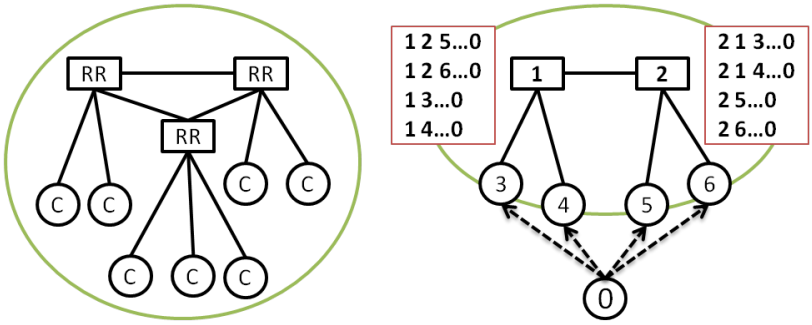

Figure 7.13: Route reflector example: clients are border routers

ters can be interconnected by either a reflector or a client, and our experiment includes both. Our experiments also include configurations with multiple redundant reflectors in a single cluster, as shown in Figure 7.14.

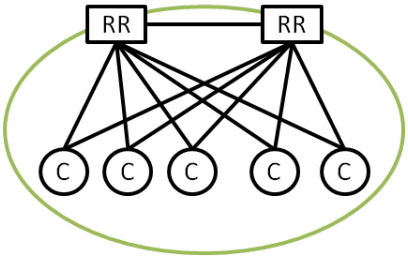

Figure 7.14: Route reflector with POP

To understand how reduction helps in detecting route oscillation due to policy misconfigurations, we embed in the network three small substructures or *gadgets* [26]: namely the *Good*, *Bad* and *Disagree* gadgets that correspond to safe, permanent, and transient oscillation behaviors. These gadgets are embedded within the transit ASes by configuration of the local preference attributes. There are also several stub ASes, set up with full-mesh or reflection topologies (described below), and employing a policy that prefers paths with fewer AS hops. (To break tie, an older route is preferred over a newly generated one.)

GT-ITM networks. As an alternative dataset, we generate transit-stub topologies using the GT-ITM topology generator [47]. Each transit-stub topology is param-

eterized by the number of transit domains, nodes within a transit domain, stubs per transit nodes, and finally, nodes per stub. We increase the network size by increasing all of these parameters. We configure routing policies as follows: transit ASes are willing to carry all traffic, while each stub AS carries traffic only for itself. Given the randomness of GT-ITM topology generation, this dataset are less structured compared to the earlier Cisco-Synthetic topologies, resulting in increased variance in our results.

Reduction on Actual Topologies . We evaluate the effectiveness of our reduction techniques on actual Internet topologies, obtained from the CAIDA Inter-AS level topologies [4] and the Rocketfuel router-level ISP topologies [68]. In the CAIDA and Rocketfuel dataset, we sample[1] the dataset to derive network of sizes up to 185 and 128 respectively. For all the topology samples, we insert the same policy configurations as our earlier Cisco-Synthetic and GT-ITM setups. We observe that the reduction rate was high, achieving a rate of 75% and 69% on average respectively. This suggests that in practice, there is significant configuration redundancy in actual configurations, observable even for a sample of the network.

7.3.2 Reduction Performance

Table 7.1 summarizes the performance overhead of network reduction and analysis on the two classes of input topologies for various network sizes. *Cisco-Good-22* refers to a 22-node Cisco-Synthetic topology embedded with *Good Gadgets*. The columns shown refer to:

- **EPD Generation.** Time to generate a EPD model from the input BGP configuration.

[1]Our experimental dataset was limited by the physical memory constraints of storing the entire EPD in memory. As future work, we plan to explore out-of-core implementations or the use of multiple machines for executing a single reduction.

Input Topology	EPD Time (ms)	Reduction Time (ms)	Reduction Time (ms, Dup)	Reduction Rate	Reduction Rate (Dup)	Analysis Time (ms)
Cisco-Good-22	3	74	22	68%	63%	429043
Cisco-Good-48	113	863	124	85%	84%	429043
Cisco-Good-87	5299	5665	649	92%	92%	429043
Cisco-Good-104	26567	10341	1814	93%	93%	429043
Cisco-Good-140	983300	32562	1814	95%	94%	429043
Cisco-Bad-22	5	96	23	69%	68%	80224
Cisco-Bad-49	112	935	119	86%	86%	80224
Cisco-Bad-87	5204	6075	465	92%	92%	80224
Cisco-Bad-104	25449	11258	725	93%	93%	80224
Cisco-Bad-121	177421	19741	1111	94%	94%	80224
Cisco-Disagree-23	2	30	14	78%	80%	184
Cisco-Disagree-53	40	352	73	90%	90%	184
Cisco-Disagree-70	182	901	164	93%	92%	184
Cisco-Disagree-103	3951	3641	469	95%	95%	184
Cisco-Disagree-122	20792	6430	810	96%	96%	184
GT-ITM-12	1	6	2	82%	81%	1
GT-ITM-38	7	24	9	94%	94%	1
GT-ITM-77	57	2279	68	95%	95%	1
GT-ITM-80	71	5241	84	90%	90%	2
GT-ITM-118	350	583143	455	86%	91%	2

Table 7.1: Summary of results across various input topologies. Averages across multiple runs are presented.

- **Reduction Time.** Reduction time required to generate the reduced EPD from the corresponding input EPD. Both reduction rules are applied, duplicate followed by complementary.

- **Reduction Time (Dup).** Same as above, except that complementary reduction is not applied. The difference allows us to compare the marginal overhead of applying complementary reduction.

- **Reduction Rate.** Percentage of redundant nodes that are reduced. For example, *68%* for *Cisco-Good-22* means that the reduced EPD is only *1-68% = 32%* of the original network size.

- **Reduction Rate (Dup).** Rate of reduction achieved by only merging duplicate nodes.

- **Reduced Analysis.** Time required to run the safety analysis, using our Maude analyzer (Section 6) on the reduced EPD after reduction.

EPD Generation and Reduction. The overhead of reduction includes the time required to generate the EPD representation of the policy configuration, and the overhead of doing the reduction itself. Due to space constraints, we will show performance graphs (derived from Table 7.1) for the the Cisco-Synthetic networks, but discuss conclusions drawn from both input topology classes.

Figure 7.16 shows the EPD generation time (left) and reduction time (right) as the number of nodes increases. We observe that the execution times are polynomial (cubic/quadratic) with respect to network size. While the complexity bounds are not ideal for scaling up, we note that the absolute numbers are easily within the realm of practicality. For instance, on a single commodity PC, EPD and reduction using our unoptimized Maude code requires only 16 minutes and 32 seconds (or 18 seconds with duplicate only reduction) respectively, for a network of 140 nodes (*Cisco-Good-140*). While the EPD generation time dominates, this cost is amortized across both reduction and analysis, since the subsequent analysis essentially uses the same EPD representation.

In Cisco-Synthetic networks, the reduction overhead is dominated by the EPD generation time. Note however that EPD generation is amortized across both reduction and analysis, since the subsequent analysis essentially uses the same EPD representation. However, in GT-ITM networks, we observe that the actual reduction dominates over EPD generation, suggesting that a nosier (more randomize) configuration increases reduction overhead. Among Cisco-Synthetic networks, we observe that reduction times are increased on denser topologies with full meshes within an AS, as compared to ASes that use route reflectors internally.

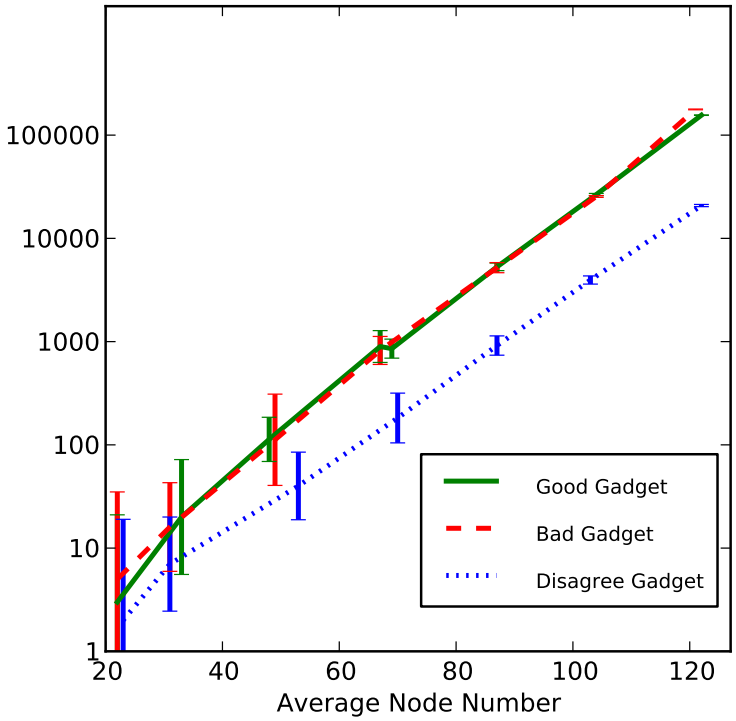

Figure 7.15: EPD Generation time as number of nodes increases for the Cisco-Synthetic topologies

Reduction rate. Table 7.1 shows that reduction is very effective at reducing the size of the EPDs. In some cases, as the network sizes increases, the reduction can reduce the original EPD by 95%. Figure 7.17 shows the reduction rates on the Cisco-Synthetic networks. For networks beyond 40 nodes, the reduction rate is above 80% and relatively stable. The effectiveness of reduction can be attributed to the highly structured natures of these topologies, where the resulting reduced EPD is often identical to the original embedded gadgets themselves. Another source of irreducibility is if the BGP decision procedure falls through to attributes we do not analyze.

The trends observed in GT-ITM are largely similar, though we note that since these topologies are randomly generated, the reduction times and rates have higher

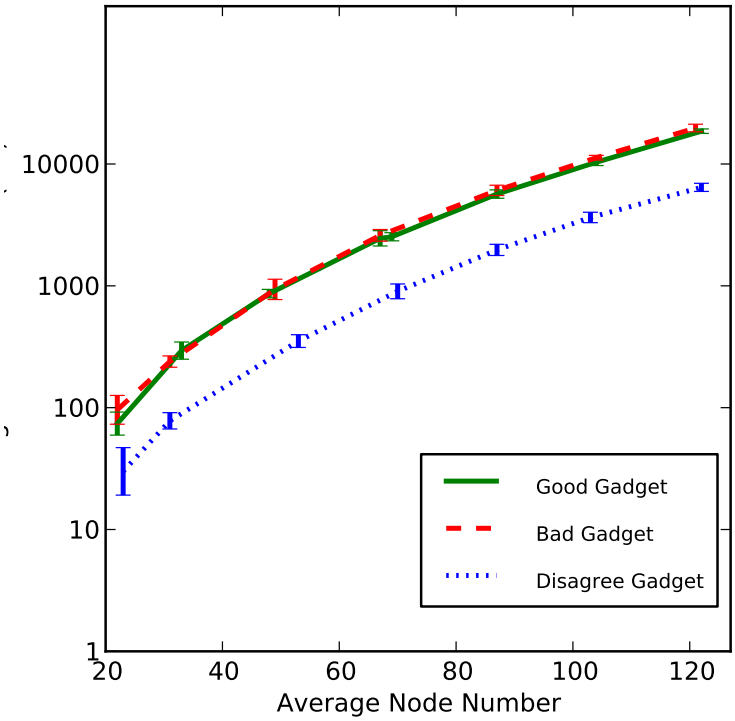

Figure 7.16: Reduction time as number of nodes increases for the Cisco-Synthetic topologies

variance across experimental runs. In Cisco-Synthetic networks, the reduction rate exhibits smaller variance due to its regular structure. In general, when a network becomes more hierarchical, (from GT-ITM to Cisco, from full-mesh to reflection), reduction rate improves due to increased redundancies. Moreover, the reduction overhead is relatively smaller (compared with the growth of network size). All in all, our results imply that a well structured hierarchical network configuration is easier to analyze in terms of reduction times. They are also more likely to result in safer configurations that do not oscillate.

Duplicate vs Complementary. As we noted in Section 7.1.2, the complementary condition is more complex. Our experimental results summarized in Table 7.1

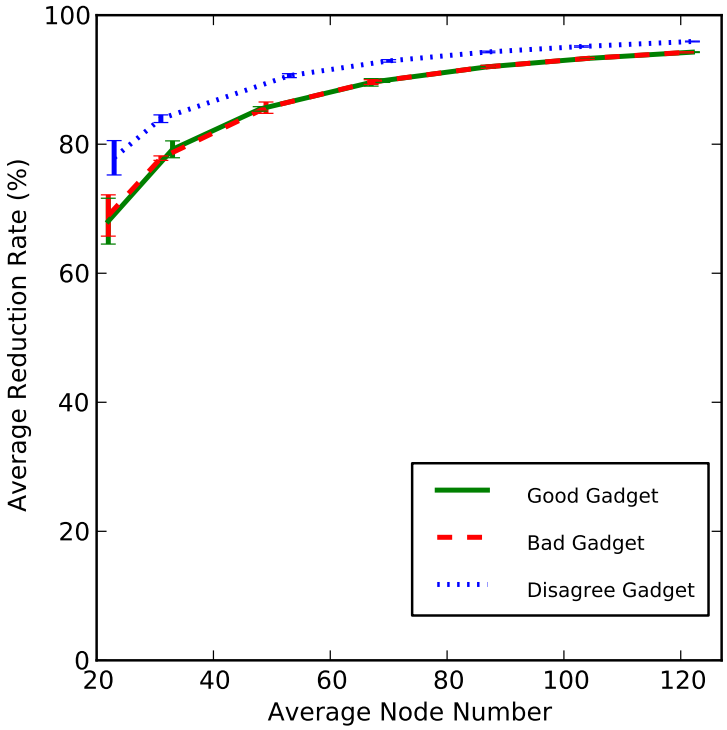

Figure 7.17: Reduction rate as number of nodes increases for the Cisco-Synthetic topologies.

validate that the overall reduction time tends to be dominated by complementary reduction. While duplicate reduction only requires two nodes to agree upon what they learned from their neighbors, complementary requires all the neighbors of the two nodes to agree upon what are learnt from them. In addition, the marginal benefit of performing complementary reduction on top of duplicate reduction is often small. For instance, *Cisco-Good-22* results in a 63% reduction compared to the original EPD when only duplicate reduction is used, and 68% (i.e. an additional 5%) with both forms. While complementary reduction is less effective, we note that in almost all cases, the EPD is further reduced by the reduction. Moreover, as noted in Section 7.3.3, both forms of reduction allow us to shed light into the policy configurations themselves.

Analysis time. To understand the benefits of performing safety analysis on the reduced EPDs, we run analysis on the original EPD, as well as the reduced EPD. The analysis is achieved by exhaustive search over all executions of the BGP system using Maude [85]. Oscillation is detected if the same best route is selected multiple times during protocol execution. Overall, we observe that after reduction, we are able to detect the same route oscillation pattern found in the original network. While the original pre-reduction EPDs did not terminate within minutes, all the post-reduction EPDs completed successfully, while requiring significantly less time and state exploration. The Cisco-Synthetic topologies with *Good Gadget* requires the longest analysis time, since these are safe instances, and hence require enumerating all possible states before completing the analysis. In the case of unsafe instances, the analysis time was quicker and terminated whenever an unsafe execution trace was obtained.

In the GT-ITM networks, since they were less structured, not all of input topologies reduced to small gadgets (like the Cisco-Synthetic examples do) that can be analyzed quickly. For instance, in *GT-ITM-118*, only 25% of the reduced EPDs were analyzable, since the other reduced instances themselves still contain 20 nodes. We note however that when these instances are analyzable, they are typically reduced to small EPDs which can be analyzed quickly.

Overall, we observe that reduction is very effective at reducing the size of the EPDs. In some cases, as the network sizes increases, the reduction can reduce the original EPD by 95%. After reduction, we are able to detect the same route oscillation pattern found in the original network. While the original pre-reduction EPDs did not terminate within minutes, all the post-reduction EPDs completed successfully, while requiring significantly less time and state exploration.

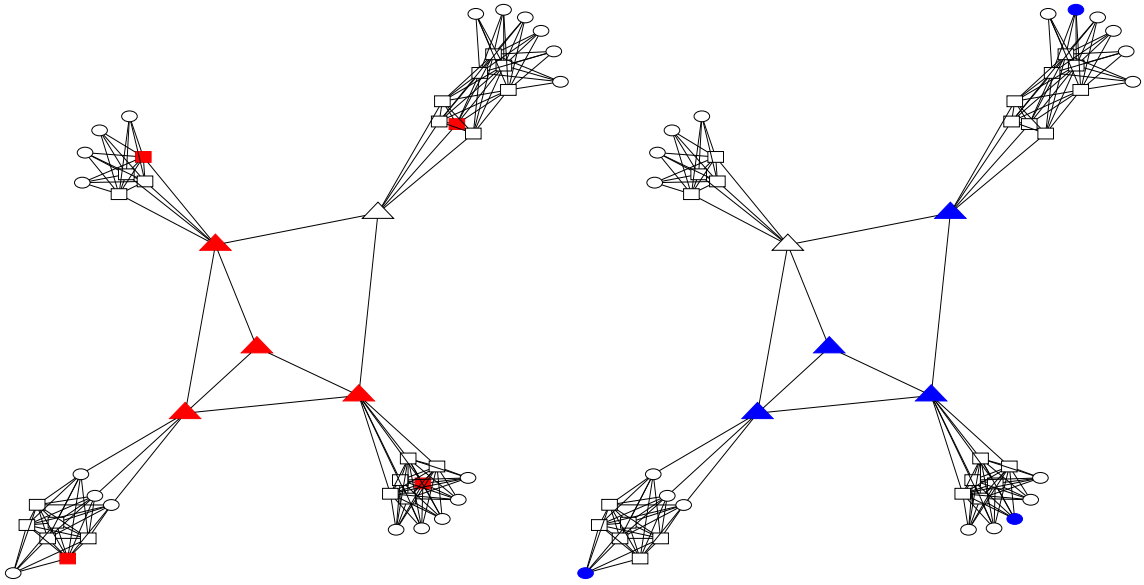

Figure 7.18: In a Cisco-Synthetic network, duplicate reduction (left) merges core (triangles), internal routers (ovals) and retains the border gateway nodes (highlighted squares) post-reduction; Complementary reduction (right) merges core, border gateway routers and retains internal nodes (highlighted ovals).

7.3.3 Observations and Implications

In addition to performance benefits of feasible analysis, the process of reduction allows us to gain new insights into policy configurations.

In Section 7.2, we prove that duplicate and complementary are dual concepts. Figure 7.18 illustrates these effects, by comparing the EPDs before and after reduction, while only applying duplicate and complementary reductions respectively. In these figures, triangles denote core network (transit) ASes (which includes the embedded gadgets) in the *Cisco-Synthetic* networks, and transit AS nodes in GT-ITM. Squares refer to reflectors in stub ASes, and all other nodes are drawn as ovals. Nodes that remain after reduction are highlighted.

The duplicate reduction (left) removes some of the core network nodes, as well as some internal nodes in stub networks (those which are not border routers) In

contrast, complementary reduction (right) removes an opposite family of nodes, namely the border routers that connect each stub network to the core.

This duality reveals deeper insights into the role of redundancy in networks. For core and iBGP internal nodes, duplicates arise because they are likely configured to agree upon how to route to their neighboring BGP routers for a given destination node. Such redundancies are typically eliminated by duplicate reduction. On the other hand, redundancies among border routers may be caused by configuring one router as a backup for another, so that the internal nodes that route through them view them in the same way. This falls into the definition of complementary nodes.

7.4 Summary

We present network reduction, *FVR*'s scalability technique for efficiently analyzing BGP configurations in a sound and automated fashion. Based on a unified EPD (extended path digraph) model of policy configurations for both specification and analysis, we develop two reduction rules that transform a policy configuration to a smaller one by only local inspection. We proved that the two reduction rules (duplicate and complementary) are dual to each other, and are sound and locally complete with respect to the safety property. Our evaluation results not only show the benefits of reduction by scaling up safety analysis of BGP systems, but also allow us to gain insights into structural redundancies among Internet routing policy configurations.

Chapter 8

Conclusion

8.1 Summary

This thesis centers around the application of formal methods and programming languages techniques that enable us to create network systems that are functionally correct, scalable, and easy to manage. In the setting of Border Gateway Protocol (BGP), the single de-facto inter-domain routing system, the control logic and the "brain" that inter-connects the Internet, this thesis develops *FVR*, to bridge the formal reasoning and the actual configurations.

FVR's key enabling components and techniques are as follows.

- **Verifying formal network models.** To free the network operators from the manual reasoning process, *FVR* automates analysis on network models, by using existing formal tools.
- **Generating faithful implementations from verified models.** To enforce the verified correctness properties in the actual implementations, *FVR* compiles the verified model into property-preserving distributed implementations, by leveraging recent advances in network programming framework.
- **Verifying actual network systems.** To go beyond network models, *FVR*

automatically analyzes real-world network systems, by combining formal methods and network domain knowledge.

- **Scalability techniques for analysis.** To scale up formal analysis, *FVR* includes a novel scalability techniques – network reduction, by exploring the networking problem space.

8.2 Future Directions

Building on my research experience in formal verification of today's inter-domain routing systems, an interesting future direction of research is to ease network management in software-defined networks (SDN) and virtual networks through the use of formal synthesis.

In order to lower the barriers to innovation in the network, and to accommodate multi-tenants and the incremental deployments of new techniques, new network techniques such as SDN and network virtualization are emerging. I envision that, these promising techniques, while still early in their stages, can benefit from formal synthesis — a systematic way that shortens the development cycles of new solutions, and guarantees formal correctness. In both scenarios for software-defined network and virtual networks, a driving synthesis task is network migration.

8.2.1 Software-Defined Network

SDN aims for new solutions for future network architecture, leading to network with flexibility (e.g. programmable), ease of management (e.g. reconfiguration, migration), rapid development (e.g. network virtualization), and multi-tenants (e.g. network virtualization). The key enabling technique of SDN is to decouple control

decision and actual forwarding — the large number of switches that perform forwarding actions are configured by one (or multiple) separate logically centralized controller through communication specified by the openflow protocol [55].

The killer application for SDN is network virtualization, a collection of promising techniques ranging from resource allocation to network slice/isolation, proposed and deployed at different network layers. A common theme in network virtualization is to allocate and map the physical network infrastructure (network equipments) into virtual slices for end-to-end logical services. Hence, network virtualization can be viewed as an effective way to accommodate multi-tenants, granting them individual virtual slices on one single physical infrastructure.

8.2.2 Formal Synthesis for Software-Defined Networks

The new trends in software-defined networks, while directly impacting current networking practice and research, also promise new opportunities for adopting formal methods. The logical controller, which maintains a global view of the entire network, is more amenable to formal methods in a distributed setting. Moreover, the openflow protocol, used by the controller and switches to communicate, serves as a general abstraction for management and new applications deployment. I envision that this "general abstraction layer" also provides the foundation for a unified formal synthesis framework. I plan to develop a general formal synthesis framework, and as proof of concept, to synthesize the control configurations that realize a given high-level policy.

On the other, I envision that the multi-tenancy in virtual networks maps naturally into compositional synthesis. The core of the methodology is to decompose the multi-tenancy virtual network allocation/management task under "global" infrastructure constraints into (potentially overlapping) a collection of individual synthesis tasks. Each individual synthesis task seeks to manage the virtual slice

for a particular tenant subjecting to the tenant's "local" slice policy. These individual synthesis tasks, while solved in isolation with one another, when put together, should also satisfy the global infrastructure constraints. Moreover, the modularity nature of compositional synthesis also makes its application more flexible. For example, the network operators may perform compositional synthesis for a particular tenant on a fraction of the network without affecting the rest of the tenants or infrastructure.

Driving Task: Network Migration While targeting network management in software-defined virtual networks, I plan to use network migration as the common driving management task. Example network migration task includes moving a collection of virtual node in virtual networks while conforming to the global infrastructure constraints (e.g. substrate node capacity and link bandwidth), and updating a set of configuration rules in software-defined networks while enforcing the high-level security policy.

I choose network migration as the driving synthesis goal for two reasons: First, network migration is an critical management to accommodate planned or unexpected changes; Second, there is a common theme in network migration that is amenable to synthesis — figure out a "proper" ordering to perform a series of atomic network updates. The ordering is crucial to maintain or satisfy network constraints during all transient states in the migration. While software-defined virtual networks ease each atomic updates, e.g. one move of virtual node, or update of a single control rule, it is still an open question regarding the updating ordering. To this end, I propose a formal synthesis approach that formulates migration as a reachability problem from the initial network state to the target final one.

As a proof of concept, for configuration migration in software-defined networks, I will use model checkers to logically synthesize an update ordering that preserves the high-level policy in all transient states. Moving forward, I will de-

velop a compositional synthesis framework for migration in virtual networks. The framework will enable network operators to manage various migration tasks including (1) migrations that are restricted to a single network slice that does not overlap with other slices, (2) migrations that occur in a slice that overlaps with others, and (3) migrations that occur across network slices.

Appendix A

Appendix

A.1 Inconsistent Policy Configuration

Inconsistent policy preference within a node can occur, of the form $p_1 \prec p_2, \cdots p_{k-1} \prec p_k, p_i \prec p_1$. Inconsistent policy preference at a single node can cause route oscillation and make the BGP system unsafe.

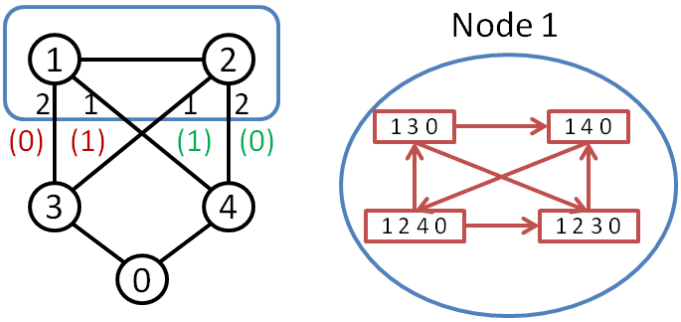

Figure A.1: A policy configuration, known to suffer oscillation due to inconsistent configuration (cyclic preference arcs) within a single node.

An example unsafe configuration is shown on the left of Figure A.1, originally presented in [21]. The right hand shows that its EPD contains a cycle, which is illustrates the problematic preference configuration at node 1.

The inconsistent policy preference is un-desirable and can be avoided by performing local check within a node. Therefore, this paper focuses on unsafety caused by conflicting policies across nodes and assumes the inconsistent policies have been prevented.

A.2 Properties of Network Reduction

A.2.1 Duality

Theorem 19. ***a.*** *If all the nodes in N_{from} can be merged into one node by (multiple steps of) complementary reductions, then u and v must be duplicate.* ***b.*** *If all the nodes in N_{to} can be merged into one node by (multiple steps of) duplicate reductions, then u and v must be complementary.*

Proof. Part **a** may be proved as shown in Figure A.2 (a). After all nodes in N_{from} (left-most EPD) are merged to x (the middle EPD), nodes u and v are duplicate, since they satisfy the criteria of Definition 12. Part **b** can be proved in the same way. □

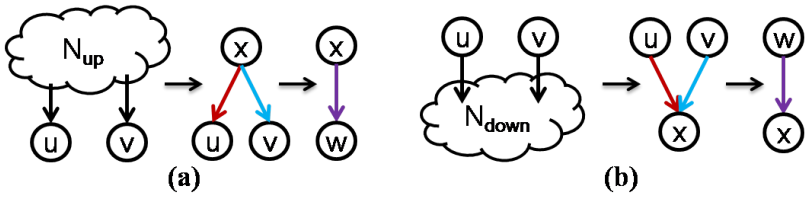

Figure A.2: Relate duplicate and complementary reduction

A.2.2 Soundness

Our main soundness result is that the reductions preserve the presence or absence of cycles in the EPD. From Theorem 7, this means that the reduced EPD has the

same safety and robustness properties as the original. In the following, let G be the original EPD, containing nodes u and v which are merged (by duplicate or complementary reduction) to a single node w in the reduced EPD G'.

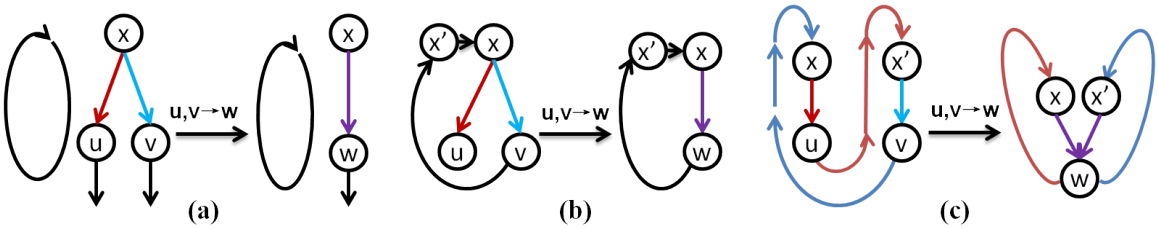

Figure A.3: Lemma 2: **Case (a.2) and (b)** None of N_{from} are on a cycle; **Case (c.1)** Some of N_{from} and u, v are in the cycle, and at least one of those in N_{from} is in $\Gamma^-(u)$ and $\Gamma^-(v)$; **Case(c.2)** Same as **(c.1)** except that no nodes in N_{from} are both in $\Gamma^-(u)$ and $\Gamma^-(v)$.

Duplicate Reduction Preserves Cyclicity

Lemma 2. *If G rewrites to G' by duplicate reduction, then* ***(1)*** *G is cyclic implies G' is cyclic, and* ***(2)*** *G is acyclic implies G' is acyclic.*

Proof. For **(1)**, we construct a cycle c' in G' for any cycle c in G. The duplicate rewrite from G to G' is defined on u,v and N_{from}, and the proof proceeds by case analysis of whether any of these nodes are on c.

Case (a) None of the nodes in N_{from} are on c. Consider two sub-cases: **(a.1)** $N_{\text{from}} = \emptyset$ when u and v have no common upstream neighbor. Regardless of whether u or v is on c, a cycle c' in G' is constructed from c by the global rewrite $c[u, v \mapsto w]$. **(a.2)** $N_{\text{from}} \neq \emptyset$. We know u and v cannot be on c either, as then one of the upstream nodes from N_{from} would be on c too. Merging u and v will not affect c, and c' is obtained by $c[u, v \mapsto w]$ (Figure A.3 (a)).

Case (b) Some of the nodes in N_{from}, but neither of u and v, are on c. As in case **(a.1)**, c' can be constructed from c by $c[u, v \mapsto w]$.

Case (c) A subset of N_{from} (call it X), and u and v, are on c. Consider two sub-cases: **(c.1)** Some of N_{from}, u and v are on c, and at least one of those in N_{from} is an upstream neighbor of both u and v. On the cycle, x must be the last element in X, shown in Figure A.3 (b). After merging u and v, they are replaced by w in c'. The rest of the changes in c' are obtained by $c[u, v \mapsto w]$. Note that the presence of arcs between u and v will not affect the result, represented by the line between u and v in the figure. **(c.2)** Some of N_{from}, u and v are on c, and none of those in N_{from} are upstream neighbors of both u and v. There must exist at least two nodes x and x' in c, shown in Figure A.3 (c). After merging u and v, c is broken into two cycles. Pick either for c'.

Part **(2)** is proved via the contrapositive: if G' is cyclic then G is cyclic. The proof is similar to that of **(1)**: for any cycle c in G', we construct a cycle c' in G. Consider the two cases:

Case (a) None of the nodes in N_{from} are on c. There are two sub-cases: **(a.1)** $N_{\text{from}} = \emptyset$. Regardless of whether w is on c, in the preimage, a cycle c' must exist in G where w replaces either u or v. **(a.2)** $N_{\text{from}} \neq \emptyset$. In this case, w cannot be on c. In the preimage G, where w is split into u and v, the cycle is still present. (Note that we only depict one possible splitting here.)

Case (b) Some x in N_{from}, but not w, is on c. The proof is similar to that of case **(a.1)**.

Case (c) At least one x in N_{from}, and w, are on c. In the preimage, there are two cycles, one through u and the other through v (but which are otherwise identical).

□

Complementary Reduction Preserves Cyclicity

Lemma 3. *If G rewrites to G' by complementary reduction, then G is cyclic implies G' is cyclic.*

Proof. Proof by case analysis, for any cycle c in G, consider whether u, v are on it.

Case (1) Neither u nor v is in the cycle. Merging u, v does not affect cycle, shown on the left of Figure A.4.

Case (2) Either u or v is in the cycle. Then according to complementary reduction definition, consider two sub-cases: **(2.1)** shown on the right of Figure A.4, if a common downstream neighbor x of u, v is also on c, then after merging u, v, c transforms to a new cycle c' where u is replaced by w. **(2.2)** If none of u, v's common neighbor is on c, c' can still be constructed similarly.

Case (3) Both u and v is in the cycle, as shown in Figure A.5. Regardless whether some of u, v's common downstream neighbor (the figure depicts the case where such a common neighbor is x), after merging u, v, cycle c transforms to two cycles in G'. □

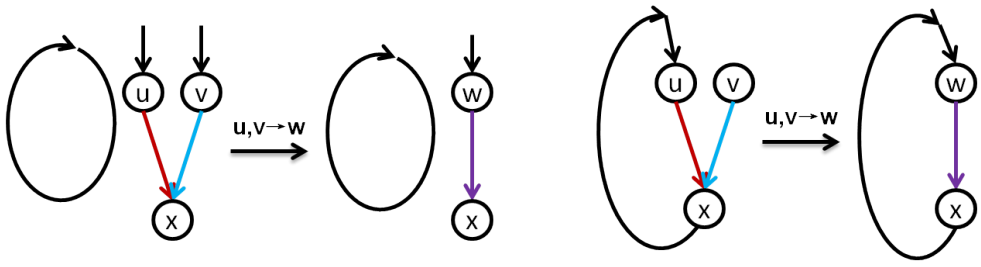

Figure A.4: Proof sketch of Lemma 3: Case 1 (left) and Case 2.1 (right).

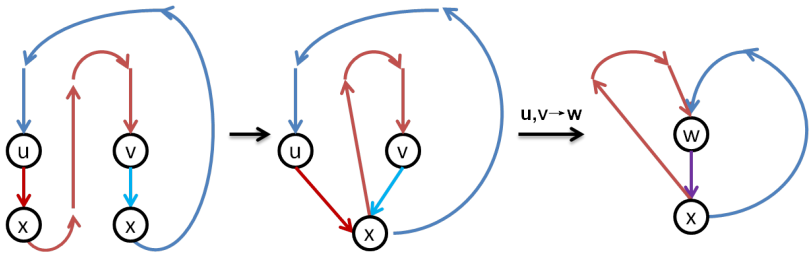

Figure A.5: Proof sketch of Lemma 3: Case 3.

Lemma 4. *If G rewrites to G' by complementary reduction, then G is acyclic implies G' is acyclic.*

Proof. Prove the dual statement: If G' is cyclic, then G is also cyclic. Proof by case analysis. For any c in G', assume G' is obtained from G by merging complementary nodes u, v into w.

Case (1) If w is not on c, as shown in the left of Figure A.6. Obviously, reversing the reduction by splitting w into u, v does not affect the cycle.

Case (2) If w is on the cycle of G', as shown in Figure A.6. Then at least one of the downstream neighbor x is also on c. Reversing the reduction by splitting u, v will split c into two cycles in the original G. □

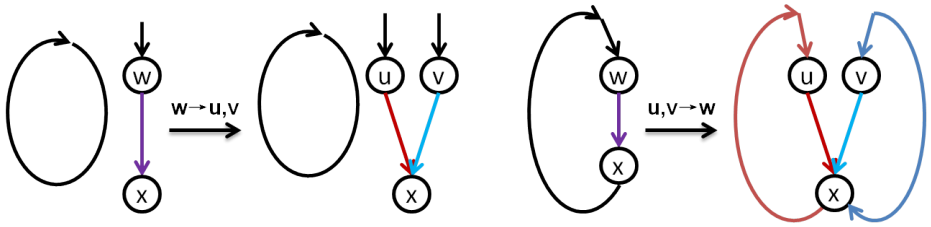

Figure A.6: Proof sketch of Lemma 4, Case 1 (left), and Case 2 (right).

Bibliography

[1] FVR (Formally Verifiable Routing). `http://netdb.cis.upenn.edu/fvr/`.

[2] RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation. `http://netdb.cis.upenn.edu/rapidnet/`.

[3] The Coq Proof Assistant. `http://coq.inria.fr`.

[4] The IPv4 Routed /24 Topology Dataset. `http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml`.

[5] T. Bates, E. Chen, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (iBGP), RFC 4456, 2006.

[6] Matthew Caesar and Jennifer Rexford. BGP Routing Policies in ISP Networks. In *IEEE Network Magazine special issue on Interdomain Routing*, 2005.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*. Springer, 2007.

[8] G. Denker, J. Meseguer, and C. Talcott. Formal Specification and Analysis of Active Networks and Communication Protocols: The Maude Experience. *DARPA Information Survivability Conference and Exposition,*, 2000.

[9] Xenofontas Dimitropoulos, Dmitri Krioukov, Marina Fomenkov, Bradley Huffaker, Young Hyun, kc claffy, and George Riley. AS relationships: inference and validation. *ACM SIGCOMM Computer Communication Review*, 2007.

[10] DNV use cases for protocol verification. `http://www.seas.upenn.edu/~anduo/dnv.html`.

[11] Dawson Engler and Madanlal Musuvathi. Model-checking large network protocol implementations. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[12] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, pages 1–50, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Nick Feamster and Hari Balakrishnan. Towards a logic for wide-area Internet routing. In *Future Directions in Network Architecture (FDNA)*. ACM, 2003.

[14] Nick Feamster, Ramesh Johari, and Hari Balakrishnan. Implications of Autonomy for the Expressiveness of Policy Routing. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2005.

[15] Anja Feldmann, Olaf Maennel, Z. Morley Mao, Arthur Berger, and Bruce Maggs. Locating Internet routing instabilities. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2004.

[16] Ashley Flavel and Matthew Roughan. Stable and flexible iBGP. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2009.

[17] Ashley Flavel, Matthew Roughan, Nigel Bean, and Aman Shaikh. Where's Waldo? Practical Searches for Stability in iBGP. In *Proc. International Conference on Network Protocols (ICNP)*, October 2008.

[18] Lixin Gao, Timothy G. Griffin, and Jennifer Rexford. Inherently Safe Backup Routing with BGP. In *Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2001.

[19] Lixin Gao and Jennifer Rexford. Stable Internet routing without global coordination. In *ACM SIGMETRICS*, 2000.

[20] T. G. Griffin and G. Wilfong. A Safe Path Vector Protocol. In *Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2000.

[21] Timothy Griffin and Gordon T. Wilfong. Analysis of the MED Oscillation Problem in BGP. In *IEEE International Conference on Network Protocols (ICNP)*, 2002.

[22] Timothy G. Griffin. The Stratified Shortest-Paths Problem. In *COMSNETS*, 2010.

[23] Timothy G. Griffin, Aaron Jaggard, and Vijay Ramachandran. Design Principles of Policy Languages for Path Vector Protocols. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2003.

[24] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Transactions on Networking (TON)*, 10:232–243, 2002.

[25] Timothy G. Griffin and Joao Luis Sobrinho. Metarouting. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2005.

[26] Timothy G. Griffin and Gordon Wilfong. An Analysis of BGP Convergence Properties. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 1999.

[27] Timothy G. Griffin and Gordon Wilfong. On the Correctness of iBGP Configuration. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2002.

[28] Alexander Gurney and Timothy G. Griffin. Neighbor-specific BGP: An algebraic exploration. In *IEEE International Conference on Network Protocols (ICNP)*, 2010.

[29] Alexander J. T. Gurney. *Construction and Verification of Routing Algebras*. PhD thesis. University of Cambridge, 2009.

[30] Alexander J. T. Gurney and Timothy G. Griffin. Lexicographic products in metarouting. *Network Protocols, IEEE International Conference on*, 0:113–122, 2007.

[31] Alexander J. T. Gurney, Limin Jia, Anduo Wang, and Boon Thau Loo. Partial Specification of Routing Configurations. In *Workshop on Rigorous Protocol Engineering (WRiPE)*, 2011.

[32] Andreas Haeberlen, Ioannis Avramopoulos, Jennifer Rexford, and Peter Druschel. NetReview: Detecting when interdomain routing goes wrong. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[33] Aaron D. Jaggard and Vijay Ramachandran. Relating two formal models of path-vector routing. In *Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2005.

[34] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[35] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM TOCS*, 18(3):263–297, 2000.

[36] Craig Labovitz, G.Robert Malan, and Farnam Jahanian. Internet Routing Instability. *IEEE/ACM Transactions on Networking (TON)*, 1998.

[37] Boon Thau Loo. The Design and Implementation of Declarative Networks (Ph.D. Dissertation). Technical Report UCB/EECS-2006-177, UC Berkeley, 2006.

[38] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. ACM, 2006.

[39] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking. In *Communications of the ACM*, 2009.

[40] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *ACM Symposium on Operating Systems Principles*, 2005.

[41] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.

[42] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP misconfiguration. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2002.

[43] Maude. `http://maude.cs.uiuc.edu/`.

[44] D. McPherson, V. Gill, D. Walton, and A. Retana. Border Gateway Protocol (BGP) Persistent Route Oscillation Condition, RFC 3345, 2002.

[45] Mengmeng Liu, Nicholas Taylor, Wenchao Zhou, Zachary Ives, and Boon Thau Loo. Recursive Computation of Regions and Connectivity in Networks. In *Proceedings of IEEE Conference on Data Engineering (ICDE)*, 2009.

[46] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[47] Modeling Topology of Large Internetworks. `http://www.cc.gatech.edu/projects/gtitm/`.

[48] Shivkumar C. Muthukumar, Xiaozhou Li, Changbin Liu, Joseph B. Kopena, Mihai Oprea, Ricardo Correa, Boon Thau Loo, and Prithwish Basu. RapidMesh: Declarative Toolkit for Rapid Experimentation of Wireless Mesh Network. In *WiNTECH*, 2009.

[49] Shivkumar C. Muthukumar, Xiaozhou Li, Changbin Liu, Joseph B. Kopena, Mihai Oprea, and Boon Thau Loo. Declarative Toolkit for Rapid Network Protocol Simulation and Experimentation. In *ACM SIGCOMM Conference on Data Communication, Demonstration*, 2009.

[50] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[51] Network Simulator 3. `http://www.nsnam.org/`.

[52] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. Maintaining Distributed Logic Programs Incrementally. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2011.

[53] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andrew Scedrov. Maintaining Distributed Recursive Views Incremenetally. Technical Report MS-CIS-10-26, CIS Dept. University of Pennsylvania, 2010.

[54] Peter Csaba Ölveczky and José Meseguer. Real-Time Maude: A Tool for Simulating and Analyzing Real-Time and Hybrid Systems. *Electr. Notes Theor. Comput. Sci.*, 36, 2000.

[55] OpenFlow. `http://www.openflowswitch.org/`.

[56] J. A. Navarro Perez, A. Rybalchenko, and A. Singh. Cardinality Abstraction for Declarative Networking Applications. In *Computer Aided Verification (CAV)*, 2009.

[57] Larry Peterson and Bruce Davie. *Computer Networks: A Systems Approach, Fourth Edition*. Morgan-KaufMann, 2007.

[58] Suchitra Raman and Steven McCanne. A Model, Analysis, and Protocol Framework for Soft State-Based Communication. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, pages 15–25, 1999.

[59] R. Raszuk, R. Fernando, K. Patel, D. McPherson, and K. Kumaki. Distribution of diverse BGP paths. Internet-Draft, January 4, 2011.

[60] Y. Rekhter., T. Li., and S. Hares. A Border Gateway Protocol 4 (BGP-4), RFC 4271, 2006.

[61] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[62] Sam Owre and S. Rajan and John M. Rushby and Natarajan Shankar and Mandayam K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *Computer Aided Verification (CAV)*, 1996.

[63] Rahul Sami, Michael Schapira, and Aviv Zohar. Searching for Stability in Interdomain Routing. In *Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2009.

[64] Michael Schapira, Yaping Zhu, and Jennifer Rexford. Putting BGP on the Right Path: A Case for Next-Hop Routing. In *ACM SIGCOMM Hot Topics in Networks*, October 2010.

[65] Serge Abiteboul, et.al. *Foundations of Databases*. Addison-Wesley, 1995.

[66] João Luís Sobrinho. An Algebraic Theory of Dynamic Network Routing. *IEEE/ACM Transactions on Networking (TON)*, 13, October 2005.

[67] JoaoLuis Sobrinho. Network routing with path vector protocols: theory and applications. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2003.

[68] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP topologies with Rocketfuel. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2002.

[69] Lakshminarayanan Subramanian, Matthew Caesar, Cheng Tien Ee, Mark Handley, Morley Mao, Scott Shenker, and Ion Stoica. HLP: A Next-generation Interdomain Routing Protocol. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2005.

[70] Philip Taylor and Timothy Griffin. A Model of Configuration Languages for Routing Protocols. In *Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2009.

[71] P. Traina, D. McPherson, and J. Scudder. Autonomous System Confederations for BGP, RFC 5065, 2007.

[72] J. Uttaro, V. Van den Schrieck, P. Francois, R. Fragassi, A. Simpson, and P. Mohapatra. Add paths guidelines. Internet-Draft, May 25, 2011.

[73] Mythili Vutukuru, Paul Valiant, Swastik Kopparty, and Hari Balakrishnan. How to Construct a Correct and Scalable iBGP Configuration. In *Annual IEEE International Conference on Computer Communications (INFOCOM)*, Barcelona, Spain, April 2006.

[74] D. Walton, D. Cook, A. Retana, and J. Scudder. BGP Persistent Route Oscillation Solution. Internet-Draft, March 14, 2011.

[75] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative Network Verification. University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-34, 2008.

[76] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative Network Verification. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2009.

[77] Anduo Wang, Alexander J. T. Gurney, Carolyn Talcott, Boon Thau Loo, and Andre Scedrov. A Calculus of Policy-Based Routing Systems. In *31st Annual ACM SIGACT-SIGOPS Symposium on. Principles of Distributed Computing.*, 2012. Brief announcement.

[78] Anduo Wang, Alexander J.T. Gurney, Xianglong Han, Jinyan Cao, Carolyn Talcot, Boon Thau Loo, and Andre Scedrov. Reduction-based analysis of BGP systems with BGPVerif. In *ACM SIGCOMM Conference on Data Communication, Demonstration*, 2012.

[79] Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally Verifiable Networking. In *ACM Workshop on Hot Topics in Networks (HotNets-VIII), 2009*.

[80] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. FSR: Formal Analysis and Implementation Toolkit for Safe Inter-domain Routing. *IEEE/ACM Transactions on Networking*, 2012.

[81] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau, Loo Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. FSR: Formal Analysis and Implementation Toolkit for Safe Inter-domain Routing. In *ACM SIGCOMM Conference on Data Communication, Demonstration*, 2011.

[82] Anduo Wang and Boon Thau Loo. Formalizing Metarouting in PVS. In *Automated Formal Methods workshop (AFM), co-located with 21st International Conference on Computer Aided Verification CAV, France, 2009*.

[83] Anduo Wang, Boon Thau Loo, Changbin Liu, Oleg Sokolsky, and Prithwish Basu. A Theorem Proving Approach Towards Declarative Networking. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs) emerging trends proceedings*, 2009.

[84] Anduo Wang, Carolyn Talcott, Alexander J. T. Gurney, Boon Thau Loo, and Andre Scedrov. Reduction-based Formal Analysis of BGP Instances. In *18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2012.

[85] Anduo Wang, Carolyn Talcott, Limin Jia, Boon Thau Loo, and Andre Scedrov. Analyzing BGP Instances in Maude. In *IFIP International Conference on Formal Techniques for Distributed Systems joint international conference 13th Formal*

Methods for Open Object-Based Distributed Systems 31nd Formal Techniques for Networked and Distributed Systems (FMOODS/FORTE), 2011.

[86] Yices. `http://yices.csl.sri.com/`.

[87] Randy Zhang and Micah Bartell. *BGP Design and Implementation*. Cisco Press, 2003.