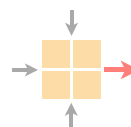




Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Networked Systems
ETH Zürich — seit 2015

Automated BGP Policy Analysis

Semester Thesis

Author: Yu Chen

Tutor: Rüdiger Birkner, Rui Yang

Supervisor: Prof. Dr. Laurent Vanbever

February 2021 to May 2021

Abstract

Autonomous Systems (ASes) use the Border Gateway Protocol (BGP) to exchange route information. However, BGP routers are challenging to configure in reality and misconfiguration can cause serious outages. This thesis presents a full geometric model for the BGP control plane. With this model, one can efficiently verify common BGP control plane policies such as transit, route preferences and consistent tagging, which are difficult or even impossible for prior verification work. A big advantage of our geometric model is that we are able to inject symbolic routes and therefore detect the entire set of announcements that lead to a policy violation instead of a single counter-example.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Project Goal	3
1.3	Main contribution	4
1.4	Overview	4
2	Background and Related Work	5
2.1	Background	5
2.1.1	BGP sessions	5
2.1.2	BGP announcements	7
2.1.3	BGP route map	8
2.1.4	BGP processing	8
2.1.5	BGP policies	9
2.2	Related Work	10
2.2.1	Data plane verification	10
2.2.2	Configuration verification	11
3	Symbolic Announcement	12
3.1	Header Space Analysis	12
3.2	Equivalence Class Computation	14
3.2.1	LOCAL-PREF and MED	14
3.2.2	Community	15
3.2.3	NEXT-HOP	16
3.2.4	IP Prefix	19
3.2.5	AS-PATH	23
3.3	Symbolic Announcement Creation	29
4	Geometric Model	31
4.1	Block Diagram	31
4.2	Parser	31
4.3	Route Map Filtering	32
4.3.1	Single matching	32
4.3.2	Multiple matching	36
4.4	Verifier	38
4.4.1	No transit from A to B	39
4.4.2	Prefer A over B	39
4.4.3	All neighbors are tagged	40

4.4.4	Want X prefer R1 over R2	40
4.5	Initial Network Reduction	40
5	Evaluation	41
5.1	Efficiency Test	41
5.1.1	Test 1: multiple matching test	41
5.1.2	Test 2: multiple route map items test	42
5.1.3	Test 3: equivalence class computation test	43
5.2	Case Study	44
6	Discussion	49
6.1	Limitations	49
6.2	Outlook	50
7	Summary	51
	References	52

List of Figures

1.1	A configuration typo	2
2.1	An example of BGP routing	6
2.2	The propagation rules on different BGP sessions	6
2.3	An example of a route map	8
2.4	BGP announcement processing pipeline	9
3.1	An example of the HSA framework	13
3.2	The HSA framework with header partition	14
3.3	An example of matching LOCAL-PREF and MED	15
3.4	An example of matching the community	15
3.5	An example of matching NEXT-HOP	16
3.6	An example output of the NEXT-HOP partition algorithm	18
3.7	An example of NEXT-HOP label distribution	19
3.8	The syntax of a prefix list item	19
3.9	An example of a prefix list item with <i>ge</i> and <i>le</i>	20
3.10	A subset of IP prefixes contained in PLIST2	20
3.11	An example output of the IP Prefix partition algorithm	21
3.12	An example of IP Prefix label distribution	21
3.13	An example of matching AS-PATH	23
3.14	An example of AS-PATH regex-tree	24
3.15	An example of AS-PATH label distribution	26
3.16	An example of the true graph and false list	27
3.17	The flow chart of the function <i>CheckConflict</i>	28
3.18	An configuration after applying attribute partition	30
3.19	The symbolic announcement created from Figure 3.18	30
4.1	The block diagram of the geometric model	32
4.2	An example of single matching (MED)	33
4.3	The filtering process inside an attribute list	34
4.4	An example of single matching (AS-PATH)	34
4.5	An example of the multiple matching	36
4.6	An example output of multiple matching algorithm	38
4.7	An example of the fake egress point implementation	39
5.1	A small subset of each attribute value pool	42
5.2	The result of the multiple matching test	43
5.3	The result of the multiple route map items test	44

5.4	The result of the equivalence class computation test	45
5.5	The topology of the case-study network	46
5.6	The verifier’s answer when all BGP properties hold	47
5.7	The verifier’s answer when the policy <i>business relationship</i> is violated	47
5.8	The verifier’s answer when the policy <i>no transit</i> is violated	48
5.9	The verifier’s answer when the policy <i>community on all routes</i> is violated	48

List of Tables

2.1	BGP decision process (partial)	9
3.1	The most common regex constraints for AS-PATH	23

List of Algorithms

1	Equivalence class computation algorithm for NEXT-HOP	17
2	Label distribution algorithm for NEXT-HOP	18
3	Equivalence class computation algorithm for IP Prefix	22
4	Equivalence class computation algorithm for AS-PATH	25
5	Label distribution algorithm for AS-PATH	26
6	Single matching algorithm for the prefix list and the AS-PATH list	35
7	Single matching algorithm for the community list	35
8	Multiple matching algorithm	37

Chapter 1

Introduction

Safely managing a network is extremely complex. On the one hand, configuring a network correctly is difficult as a network’s behavior depends on the complex interactions among different routing protocols. On the other hand, typical networks consist of hundreds of routers serving many different roles. Even today, configuration still happens manually. Operators manually try to bridge the gap between the high-level behavior and low-level configurations for many devices. Naturally, misconfigurations happen, which can have extreme consequences.

The Border Gateway Protocol (BGP) is the only inter-domain routing protocol in use today. Misconfiguration in BGP can lead to serious consequences as many incidents show: For example, In 2017, a Google operator accidentally announced wrong IP prefixes to neighbors. The false announcements were picked up by an Autonomous System (AS) in Japan, the latter started to send local Japanese traffic to Google, and Google then sent it to a black hole. Although the outage only lasted for few hours, it was still severe [20]. More recently, in April 2021, an Autonomous System in India mistakenly announced 30 000 IP prefixes, which caused a severe BGP hijacking accident, even though it only lasted for a few minutes [7]. Outages due to BGP problems are not uncommon as these examples show. A survey conducted in 2015 showed that 89% of network operators are never sure that their configuration changes are bug-free [16].

Before pushing configuration changes to production, one should check them. Best would be do that automatically: that is why verification enters the scene. However, lots of research in the past years has shown that building a sound BGP verification tool is difficult for two reasons:

- **Control plane coverage:** BGP uses route maps to control import and export traffic. Each route map is a filter that consists of arbitrary filtering rules. It takes effort to consider all possible route map behavior.
- **Data plane coverage:** Data plane determines the actual forwarding behavior. A control plane dynamically generates different data planes, based on the link status and the announcements it has received. A sound BGP verifier needs to consider all data planes.

State-of-art BGP verification can be categorized into two groups: **data plane verification** and **configuration verification**. The data plane verification analyzes the configuration based on a snapshot of the currently installed data plane [17, 14, 15] instead of looking at the configuration itself. A huge disadvantage of this approach is that it is difficult to obtain the dynamic forwarding states from the network as it changes. The configuration verification, on the contrary, takes as input the configuration file and the environment constraints [3, 18, 21]. Generally speaking, the configuration verification outperforms data plane verification because the configuration normally does not change that often so that one can derive the forwarding state outside of the real network.

1.1 Motivation

To illustrate how easy it is to make a mistake while configuring, even a small-scale network, consider the really simple network given in Figure 1.1. The network has three border routers R1, R2, R3, which connect to a provider, peer and customer neighbor, respectively. Typically, the network obeys the standard business relationship, where the announcements received from its providers or peers are only sent to its customers. In practice, this outbound control is often realized by tagging the advertisements at the ingress with different community values, and filtering the announcements at the egress based on the assigned community values [5]. However, in Figure 1.1, the network operator wrongly tags the provider's announcements with the community value prepared for its customer due to a configuration typo in R1's import route map. The consequence is that now the network starts announcing its provider's announcement to its peer.

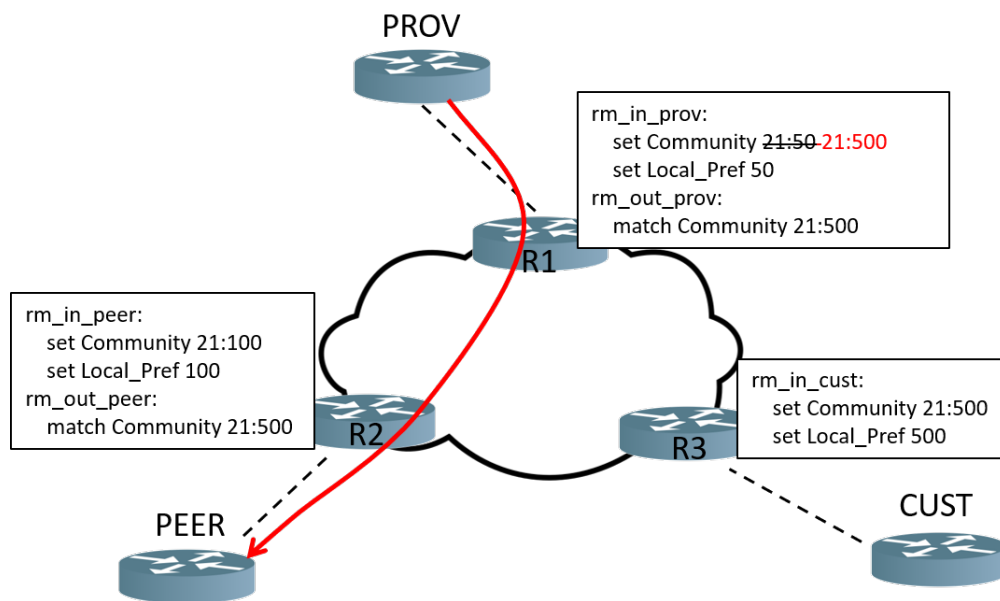


Figure 1.1: A configuration typo. The network accidentally tags its provider's announcements with the community value prepared for its customer. The typo causes the network to propagate its provider's announcements to its peer

Such a typo can cause serious economic loss or even worse lead to a DoS situation as the network is attracting more and more traffic. However, debugging such misconfiguration via manual inspection can be torturous even for an experienced network operator. That's why we need an automatic BGP verification tool.

In the recently, lot of progress has been made in this field and a glimpse of them can be found in § 2.2. However, state-of-art verification still has limitations.

First, current data plane verifiers such as HSA [14] and Veriflow [17] can only analyze the one forwarding state which is currently deployed. Therefore they can capture the configuration bug only if the current data plane reveals it. With every link failure, the forwarding state changes, and properties which might have held in the previously analyzed forwarding state, might now be violated.

The emergence of control plane verifiers solves this problem to a large extent. With the help of SMT solvers and model checkers, they are able to analyze all (or many) data planes at once [3, 18,

21]. However, a new issue arises with this method. Either a SMT solver [8] or a model checker [11] requires the translation from low-level configuration to logical formulas. The complex translation adds to the workload of the whole verification process and is also vulnerable to bugs. Moreover, these verification tools are designed for a wide range of software and are therefore not tailored for BGP verification.

One notable issue is that current verifiers are made for verifying data plane properties: reachability and security properties. However, at a higher level, they fail to reveal properties in the control plane, such as neighbor preference and consistent tagging, which are also very important.

Another common problem for the current BGP verifiers is that they only output one counter-example when a property does not hold. This is because of the nature of SMT solvers they use as finding more counter-examples requires more iterations in a SMT solver.

Moreover, to guarantee a high data plane coverage, state-of-art control plane verifiers need to go through the propagation process for all possible announcements. Plankton [18] speeds up this process with packet partition. This method mitigates the issue, but the iteration is still inevitable depending on the number of partitions.

In this project, we would like to develop an approach that largely speeds up the BGP control plane verification process while still guaranteeing a high data plane coverage. The key to the solution is to reduce the verification complexity. The insights and challenges we observe for this goal are listed as follows:

- Compared to the number of possible input BGP announcements, the number of equivalence classes for each BGP announcement attribute is much fewer. Therefore we can pre-compute the equivalence classes for each attribute before analyzing the network. The biggest challenge for this is that the computation needs to cover the entire huge attribute space to ensure a high data plane coverage.
- After computing the equivalence classes for each BGP announcement attribute, we can use a single symbolic announcement to represent all possible input announcements. This symbolic announcement carries the equivalence class labels for each attribute. We can then build a geometric model to process the symbolic announcement. The difficulty with this is that the geometric model needs to guarantee a high control plane coverage such as considering all possible route map behavior.

1.2 Project Goal

The ultimate goal of this project is that, given the network-wide BGP configurations and if any, a set of external BGP routing announcements, we can efficiently verify BGP control plane properties such as:

- no transit from neighbor X to neighbor Y
- internally prefers the routes from neighbor U to neighbor V
- want neighbor AS to prefer the routes from the internal router A over B
- has a consistent community tagging at all ingress points

In addition to soundly answer whether or not a BGP control plane property holds, the verifier should also be able to tell if any property is violated, what are all possible announcements and

their traces that lead to the violation. For example, if the property "no transit from neighbor X to neighbor Y " does not hold, the verifier will return all possible announcements that could actually be sent from X to Y and how they are transmitted through the network.

1.3 Main contribution

Our main contribution in this project is that we build a BGP control plane verification that can soundly and efficiently verify common BGP control plane properties (e.g., no transit). More specifically, our contribution can be divided into the following points:

1. We build a parser to transform the low-level configuration file into high-level objects.
2. We compute the equivalence classes for all BGP announcement attributes and define the symbolic announcement.
3. We build the geometric model to correctly model the control plane behavior and to propagate the symbolic announcement.
4. We build the control plane verifier.

1.4 Overview

In this thesis, we first provide a preliminary background on the BGP and the related work in Chapter 2. Then we explain the idea of the equivalence class computation and the data structures we have designed for it in Chapter 3. In Chapter 4, we introduce the geometric model and the entire verification procedure. We then evaluate our model in Chapter 5 and discuss the limitation of our work and potential future steps in Chapter 6. We give the final summary in Chapter 7.

Chapter 2

Background and Related Work

In the early days, the Internet was of a handful, centrally managed devices. Shortest-path routing was sufficient at that stage. As the Internet grew and consequently became a network of networks, such basic routing policies were no longer satisfactory as they could not express their business interests. Internet service providers (ISP) and enterprise networks were interested in being able to specify their own, custom routing policies for economic and political reasons. The Border Gateway Protocol (BGP) was born out of this need [5].

BGP is the only inter-domain routing protocol. Its primary function is to exchange network reachability information between different Autonomous Systems (ASes) [19]. This information can then be used to construct AS connectivity graphs, prune routing loops as well as enforce routing decisions. BGP itself is a relatively simple path-vector protocol, most of its complexity is in the decision process and the policies an AS uses to influence this process [5]. Having a solid understanding of its policies and decision process is therefore necessary before tackling BGP configuration verification.

2.1 Background¹

In this section, we first introduce the fundamentals of BGP, including what a BGP session is, what a BGP announcement consists of and how a BGP announcement is propagated. We then introduce the most typical BGP policies that have been widely applied. As well as a brief explanation of how BGP allows to implement the policies.

2.1.1 BGP sessions

BGP sessions come in two flavors: **external BGP (eBGP)** and **internal BGP (iBGP)** session. An eBGP session is established between border routers of different ASes to exchange routes to external destinations. An iBGP session is established between routers inside the same network to disseminate the external routing information. Once a border router receives a new route on an eBGP session, it propagates this route internally via all iBGP sessions. Usually, each internal router must have an iBGP session with all other internal routers, which is called an **iBGP full mesh**. This is because an internal router only propagates a route on its iBGP sessions when the route is received on an eBGP session. The propagation rules on different BGP sessions can be summarized as in Figure 2.2. When a router receives a new route on an eBGP session, it will

¹This section is inspired from Communication Networks: https://comm-net.ethz.ch/pdfs/slides/03d_internet_bgp_policies.pdf

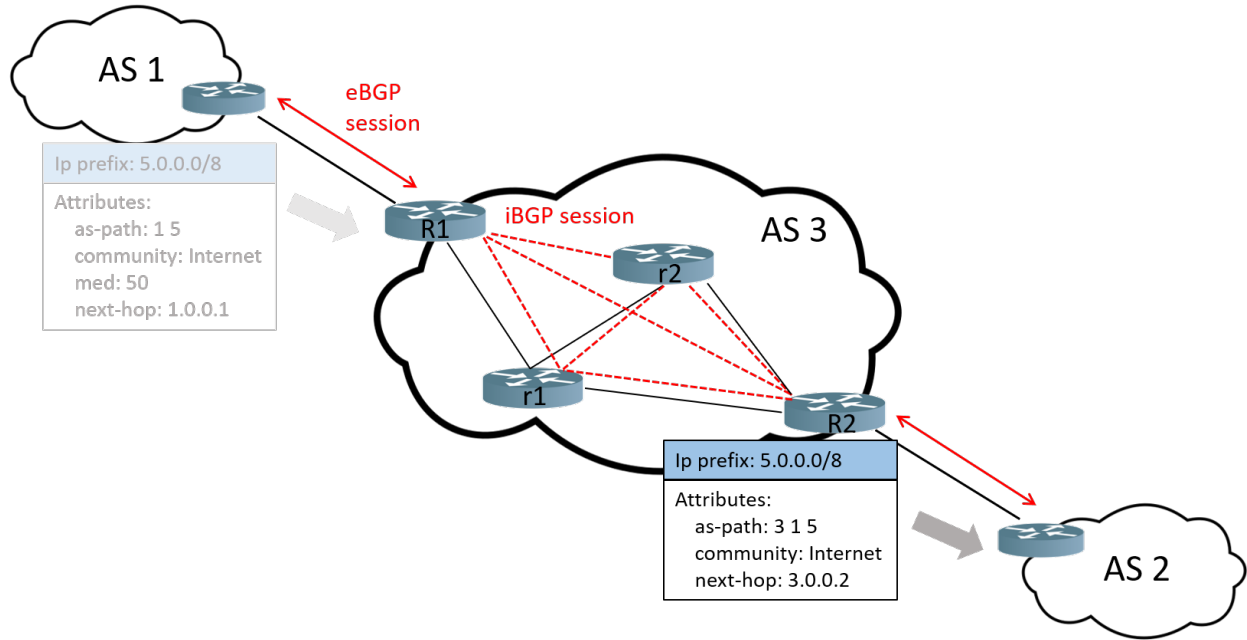


Figure 2.1: An example of BGP routing. AS3 has 2 border routers R1, R2 and 2 non-border routers r1, r2. Each border router establishes an eBGP session with a neighbor router from another AS. Every 2 routers inside AS3 are connected with an iBGP session. AS3 received a BGP announcement from AS1 via R1 and sends it to AS2 via R2.

propagate it on every eBGP and iBGP sessions. When a router receives a new route on an iBGP session, it will only propagate the route on its eBGP sessions. Therefore if a border router A does not have an iBGP session with another internal router B, B has no chance to know the routes the router A has learned.

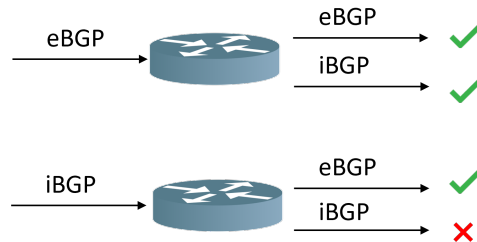


Figure 2.2: The propagation rules on different BGP sessions. When a router receives a new route on an eBGP session, it will propagate it on every eBGP and iBGP sessions. When a router receives a new route on an iBGP session, it will only propagate the route on its eBGP sessions.

In Figure 2.1, there are two border routers R1, R2 and 2 non-border routers r1, r2. The black solid lines represent physical links between two routers, the red solid lines represent eBGP sessions, and the red dashed lines represent iBGP sessions. Note that a full mesh of physical connections is not required because routers can use the IGP, for example OSPF, to send iBGP messages.

The full mesh of iBGP sessions adds to the configuration complexity and does not scale well

because every time a new router is deployed, the network operator has to set its iBGP sessions with all the other internal routers. Network operators therefor often use **route reflector** to alleviate this issue. The route reflector acts as a relay point in the network: Each internal router only sends the new route to the route reflector it connects to, and the route reflector propagates this route to all other internal routers that also connect to it [2].

2.1.2 BGP announcements

There are 4 types of BGP messages: **OPEN**, **NOTIFICATION**, **UPDATE**, **KEEPALIVE** [19]. The **OPEN** message is used to set up new BGP sessions. The **NOTIFICATION** and **KEEPALIVE** messages are used for session maintenance. The **UPDATE** message informs the neighbor of route information, such as a new best route, or the best route withdrawal.

When a BGP **UPDATE** message carries new route information, we also call this message a **BGP announcement**. A BGP announcement carries an **IP Prefix** and several attributes. The IP Prefix is an aggregation of IP addresses (e.g., 10.0.0.0/8) and the attributes describe the route properties, which are used in the BGP decision process. We focus on the most common attributes in the rest of this subsection.

LOCAL-PREF is a local attribute² and shall always be included when propagating an announcement to other internal peers via iBGP. Each **LOCAL-PREF** value is a 32-bit integer which indicates the network's preference for this announcement based on the local policy.

MULTI-EXIT-DISC (MED) is a global non-transitive attribute. Each MED value is a 32-bit integer and is intended to be used on eBGP sessions to discriminate multiple egress points to the same neighbor AS. The MED attribute received from a neighbor AS must not be propagated to other ASes.

NEXT-HOP is a global mandatory attribute. Its value is an IP address which stands for the next hop when sending packets to the destination (i.e., IP Prefix).

AS-PATH is a global mandatory attribute. It carries the AS numbers of the ASes through which this announcement has passed and the length is variable. When an AS propagates an announcement to another AS, it has to prepend its own AS number to the end (i.e., leftmost) of the **AS-PATH** attribute.

Community specifies a group of destinations which share some common properties [6]. The standard format of a community value is *ASN:value* where the 16-bit ASN represents the current AS number and the 16-bit value represents the specific value assigned for a group of destinations. An AS can define custom communities based on its local policies. There are also well-known communities that should be recognized by any community-aware AS such as Internet (advertise the prefix to all BGP neighbors) and No-Export (do not advertise the prefix to any eBGP neighbors). An announcement can contain multiple community values in this attribute.

In Figure 2.1, R1 from AS3 received a BGP announcement to reach the IP prefix 5.0.0.0/8 from AS1. The announcement was also attached with several attributes. R1 then propagated this announcement inside AS3 and finally output the announcement to AS2 via R2. Before outputting

²local attributes are only seen on iBGP sessions

the announcement, AS3 prepended its AS number to AS-PATH, rewrote the NEXT-HOP with R2's IP address and removed AS1's MED attribute.

2.1.3 BGP route map

Instead of simply forwarding the announcement intact, a router utilizes **route maps** to filter the announcements. Each route map has one of the two directions: *IN* or *OUT*. When a route map's direction is *IN*, then it filters the incoming announcements received by the router, otherwise it filters the outgoing announcements. A route map can have multiple *route map items* with different sequence numbers. Each route map item consists of a series of *match* and *set* statements and a route map type *permit|deny*.

Consider the route map given in Figure 2.3. The route map RM_IN consists of two route map items. The first item with sequence number 5 has a higher priority and the second item with sequence number 10 has a lower priority. The first item consists of two *match* statements and two *set* statements. The *match* statements specify the requirements an incoming announcement has to match. The route map type *permit* before the sequence number declares that the matching announcements are accepted by the router. If the route map type is *deny*, then the router will drop the matching announcement. The two *set* statements specify the attribute modification on the matching announcement. If an announcement does not match the requirements in the first route map item, then it goes to the next item with a lower priority.

Depending on the attribute type to be matched in a *match* statement, the match patterns are different. When the attribute to be matched is LOCAL-PREF or MED, the match pattern is a single value. When the attribute is the community, then the pattern is a community list. When the attribute is NEXT-HOP or IP Prefix, a prefix list is to be matched. When the attribute is AS-PATH, then it matches an AS-PATH list. We go into details of different match patterns in the next chapter.

```
route-map RM_IN permit 5
  match MED 50
  match ip next-hop prefix-list PLIST
  set community 21:50
  set LOCAL-PREF 50

route-map RM_IN permit 10
  set LOCAL-PREF 100
```

Figure 2.3: An example of a route map. The route map RM_IN consists of two route map items. The item with a smaller sequence number has a higher priority. Each route map item consists of a series of *match* and *set* statements and a route map type *permit—deny*.

2.1.4 BGP processing

The route map implementation is only a part of the entire BGP processing flow. Each router utilizes a specific pipeline to process the announcements it has received. As shown in Figure 2.4, the pipeline consists of 3 stages.

In the first stage, the router filters incoming announcements from each neighbor with the IN route maps and stores all routes which are accepted. An announcement is only accepted when it

satisfies all the requirements specified in the *match* statement. If this is the case, this announcement will be forwarded to the *set* statement, where its attributes will be modified accordingly.

In the second stage, all acceptable routes go through the **BGP decision process**, where the best route for each destination is selected. In the decision process, each router uses the selection preference given in Table 2.1. For all alternative routes that can reach the same destination, the router goes down the table and compares each attribute. If two routes have the same value for an attribute, then the router moves on to compare the next attribute. In the third stage, the best route is advertised to all neighboring router, but only after going through the OUT route maps.

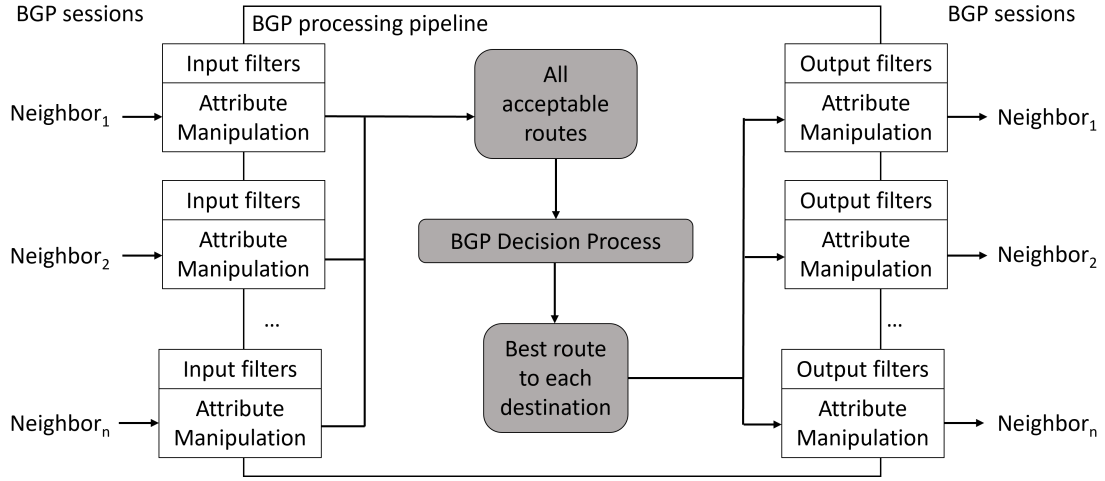


Figure 2.4: BGP announcement processing pipeline³. The pipeline consists of 3 stages. In the 1st stage, the router filters each incoming announcement, and stores all acceptable routes. In the 2nd stage, it selects the best route for each destination. In the 3rd stage, the best route is announced to the neighbors, but only after passing the export filter.

Step	Attribute
1	highest LOCAL-PREF
2	shortest AS-PATH length
3	lowest MED
4	eBGP-learned over iBGP-learned
5	smallest NEXT-HOP (tie-break)

Table 2.1: BGP decision process (partial)

2.1.5 BGP policies

The diversity of BGP attributes allows each AS to implement custom **BGP policies**. We can divide most BGP policies into 3 categories: (i) business relationship policies, (ii) traffic engineering policies, (iii) security policies.

³Source: https://comm-net.ethz.ch/pdfs/slides/03d_internet_bgp_policies.pdf

Business relationship policies There are two common business relationships in the Internet: *customer-provider* and *peer-peer*. A customer AS pays the provider AS for all the traffic between them. Two ASes that peer do not pay each other for exchanging traffic. Therefore, two policies are usually desired:

1. never transit traffic between its providers or its peers, or between its peer and its provider.
2. always prefer a customer-learned route (get paid for exchanging traffic), then a peer-learned route (exchange traffic at no cost), and a provider-learned route at last (pay to exchange traffic)

In practice, the first policy is usually implemented by tagging announcements with different community values and only allowing the announcements tagged with customer's communities to pass the output filter. The second policy can be achieved by assigning the highest LOCAL-PREF value to customers' announcements, the second-highest to peers' announcements and the lowest LOCAL-PREF value to the providers'.

Traffic engineering policies There are 2 kinds of traffic engineering: **outbound traffic control** and **inbound traffic control**. An example for outbound traffic control is *load balancing*. For example, an AS can change the LOCAL-PREF values of 2 equally preferred routes to shift the traffic from one route to another. Inbound traffic control is more difficult because it requires an AS to influence the route selection in another AS. One way to do this is via MED. The AS can set a lower MED value on the announcement that it wants the neighbor AS to accept. The other way is to prepend the AS number more times on the announcement it does not want the neighbor to use as the first choice. However, there is no guarantee that the neighbor AS will accept the route with the lower MED or shorter AS prepending if its local policy prefers another route.

Security policies ASes are vulnerable to invalid announcements. It is therefore necessary to detect and discard them at the import filter. One example security policy is *No-Martians*. *No-Martians* declares a set of invalid announcements that the AS should discard (e.g., announcements with private IP prefixes). An AS can also protect its internal resources by not exporting certain routes to other ASes.

2.2 Related Work

Network verification is the process of proving whether an *abstraction* of the network satisfies intended network-wide intents [10]. This abstraction could be drawn either from the data plane level (e.g., forwarding tables) or the control plane level (e.g., configuration files). Based on the abstraction level, current research on network verification can be divided into data plane verification and configuration verification.

2.2.1 Data plane verification

The initial progress on the data plane verification is built upon the static analysis of a network state snapshot. Xie et al. [22] propose an algorithm to compute the network reachable sets by mapping packet filters, routing information and packet transformation to a unified model. Anteater [17] translates the high-level network specification into boolean satisfiability problems and verifies them against the network model (the model is developed from Xie et al.'s algorithm) with an SAT solver.

HSA [14] builds a geometric model which can statically check the packet forwarding behavior and identify many problems including reachability, loops and isolation. HSA is later optimized by ddNF [4], which speeds up HSA computation by pre-computing the header equivalence partition before run-time. Our approach in this project is also built upon the HSA and ddNF, but in the control plane instead of data plane, and is also more complicated due to the complexity of the control plane.

Based on the static analysis, verification tools such as Veriflow [15], NetPlumber [13] and DeltaNet [12] are also able to perform real-time, incremental verification. One significant advantage of data plane verification is its **control plane coverage** as it can support arbitrary network protocols. However, most data plane verifiers have the limitation that they can only analyze the single data plane that has been deployed in the network.

2.2.2 Configuration verification

Depending on the network failure models, one network configuration can generate different data planes. To guarantee a higher **data plane coverage**, configuration verification comes into play. Batfish [9] is a datalog-based verifier. Given a network configuration, a concrete network environment, a correctness specification and a set of given announcements, Batfish is able to check a given announcement against arbitrary correctness specification in a SMT solver. Bagpipe [21], on the contrary, can check arbitrary announcement against a restricted set of policies with a SMT solver. MineSweeper [3] improves Bagpipe in terms of control plane coverage by modelling the network as a combinational circuit. Plankton [18] further improves the scalability of MineSweeper by computing packet equivalence classes (PECs). Each PEC represents a set of packets that behave identically in the configuration. A PEC is then associated with the configuration information related to that PEC.

One common point for these prior work on configuration verification is that their analysis still partly relies on the routing table generated from the configuration. This feature makes them easy to verify diverse forwarding behavior, but not for the control plane properties. In our project, we aim at exploring the possibility of filling this gap.

Chapter 3

Symbolic Announcement

One challenge for the traditional network verification (e.g., HSA, Minesweeper) is to guarantee enough data plane coverage, meaning the verification result should consider diverse data planes generated by the network, as introduced in § 1. On the one hand, to check whether a policy holds, the verifier needs to consider all possible external inputs (i.e., the entire packet header space). On the other hand, the verifier should consider all possible concrete environments (i.e., link failures).

A traditional BGP verifier with a naive approach would solve this by iterating over all possible announcements and concrete environments, one at a time. Since a BGP announcement consists of multiple attributes, this iteration can be very time-consuming. For example, if only considering the MED, we have to consider 2^{32} different options. However, an important observation is that one does not need to consider every single announcement on its own as many announcements are handled the same in the network. Based on this observation, we make a two-step improvement on the BGP verification in our project. First, we find equivalence classes of announcements and only consider them. Second, instead of considering each equivalence class separately, we build a **symbolic announcement** to represent all equivalence classes and only process the symbolic announcement in the geometric model.

In the rest of this chapter, we first introduce the intuition behind the equivalence class computation, we then explain how we compute the equivalence class for each BGP announcement attribute and create the symbolic announcement.

3.1 Header Space Analysis

Header Space Analysis (HSA) [14] is a data plane verification framework that is able to statically check the data plane and identify a complete set of failures given a network specification. To achieve its goal, HSA uses a compact bit representation to compute all reachable IP packets for a destination. Each packet is represented as a point in $\{0, 1\}^L$ space where L is the packet length. Each forwarding rule in the router transforms a packet from one point to another. HSA starts with the full wild-card symbolic packet $\{*\}^L$ which symbolically represents the entire $\{0, 1\}^L$ space. This symbolic packet then gets filtered by each forwarding rule as it passes through the different routers in the network. The final symbolic packet that reaches the destination stands for the set of all reachable packets.

A simplified example of the HSA framework is shown in Figure 3.1¹. The simple network consists of four routers R1-R4 and assume the header space is only 2-bit long. R2 has two forwarding entries:

¹This example is inspired from ddNF paper: <https://www.microsoft.com/en-us/research/uploads/prod/2015/11/An-Efficient-Data-Structure-for-Header-Spaces.pdf>

one with higher priority that sends all packets whose first bit is set to 1 out of port B; and the one with lower priority that sends all remaining packets to port C. To check the reachability from R1 to R4, R1 dumps the wild-card symbolic packet $**$ (representing the IP packets destined for 00, 01, 10, 11) into all connected links (here only one link is connected with R2). According to R2's forwarding rule, the packets that match 1^* are sent to port B, and the remainder is sent to port C. The remainder is computed as the difference of $**$ and 1^* , in other words $** - 1^* = 0^*$. In the end, packets whose bit representation falls into 0^* will reach R4.

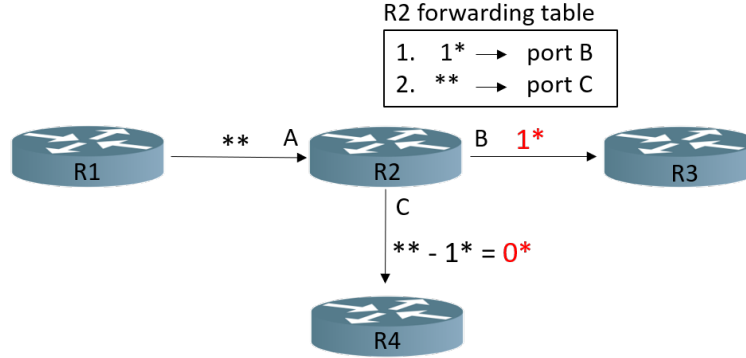


Figure 3.1: An example of the HSA framework. To verify the reachability from R1 to R4, HSA starts with 2-bit symbolic packet $**$. When it passes R2, the filtered symbolic packet 1^* goes to port B, the remaining symbolic packet 0^* goes to port C. Therefore 0^* represents all packets that can reach R4 from R1 (assuming R1-R2-R4 is the only path between R1 and R4).

A limitation of the HSA framework is that it needs to compute the difference of symbolic packets at the run time. Yang and Lam [23] observe that most headers are treated the same given the network forwarding rules. It is therefore more efficient to find the relatively small set of **header equivalence classes** at the compile time, and perform the verification based on this header partition. In this way, the complicated header space calculation is simplified to the simple set difference calculation. With this optimization, the reachability problem in Figure 3.1 can now be simplified to Figure 3.2. Based on the R2's forwarding table, the entire header space can be partitioned into two equivalence classes: 1^* and 0^* . Because the full wild-card expression $**$ is exactly the union of 1^* and 0^* , 2 equivalence classes are enough in this case.

Yang and Lam use a BDD algorithm to pre-compute the header partition. ddNF [4] proposes another more efficient algorithm for this. In our project, we are inspired by both algorithms and come up with our custom algorithms for BGP announcement partition.

In our project, we use similar ideas as presented above for BGP control plane verification. There is a clear analogy between the forwarding of data packets in the data plane and the processing of BGP announcements in the control plane. A BGP announcement in the control plane is analogous of an IP packet in the data plane. The route maps can be seen as the forwarding rules of the control plane. However, we cannot apply the ideas from HSA and ddNF one-to-one as BGP announcements and their attributes are more complex than IP packets. The same holds for route maps. On the one hand, each field in the IP packet header has a fixed length (without IP options), but BGP announcements have variable lengths. On the other hand, it is also more difficult to model the behavior of a BGP route map compared to the IP forwarding table due to the complex logic of a route map (e.g., multiple matching, attribute modification, dropping, etc.).

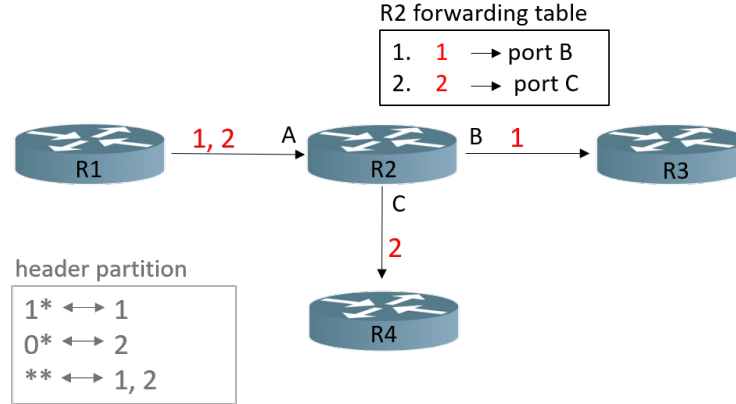


Figure 3.2: The HSA framework with header partition. Based on R2's forwarding table, the entire header space can be partitioned into 2 equivalence classes: 1^* and 0^* . We use label 1 for 1^* and label 2 for 0^* .

3.2 Equivalence Class Computation

Similar to IP packet equivalence classes, we aim at computing announcement equivalence classes in our project. The challenge is that on the one hand, BGP announcements have several different attributes as introduced in § 2.1.2 and we have to take them all into account. On the other hand, attributes have different semantics so that we cannot just use the same approach for every single attribute. Therefore, we separate the equivalence class computation for each BGP announcement attribute. Similar to data plane HSA, we first extract all the values for each attribute that appear in the *match* segments in any route map item. We then compute the equivalence classes for each attribute based on the values collected from the route maps. Finally, we assign a unique label for each class, representing a set of attributes that are handled the same in the network. In addition to the five attributes we list in § 2.1.2, we treat the IP Prefix as another attribute because it also appears in the *match* statement of a route map item. We explain our algorithms and data structures used for each attribute partition in the rest of this section.

3.2.1 LOCAL-PREF and MED

We first compute the equivalence classes for LOCAL-PREF and MED. Since LOCAL-PREF and MED are similar in their structure (they are both 32-bit integers), we can use the same approach to handle the two attributes².

Consider the two route maps shown in Figure 3.3 and assume they are the only route maps in the network that match LOCAL-PREF and MED. After automatically extracting their values from the route maps, we will have values $\{500\}$ for LOCAL-PREF and $\{50\}$ for MED. Based on the extracted values we can compute the equivalence classes. Since a route map always matches LOCAL-PREF or MED by a single integer as in Figure 3.3, each value that appears in the route map is already an equivalence class, meaning for BGP announcements with $\text{LOCAL-PREF} = 500$ or $\text{MED} = 50$, they are handled the same in the network. For the label assignment, we can directly use their values as the labels.

²Although LOCAL-PREF is a local attribute, we still compute its attribute partition for completeness.

Therefore, for LOCAL-PREF and MED, each value that appears in any route map is a label standing for an equivalence class. To make sure the union of equivalence classes of each attribute covers the entire attribute space, we use another label **-1** for each attribute to represent all the other values that do not appear in any route map and are then treated in the same default way by the network. Therefore, in Figure 3.3, the labels for LOCAL-PREF and MED are $\{500, -1\}$ and $\{50, -1\}$, respectively (assuming no other route map items).

```
route-map RM_IN permit 5
  match MED 50

route-map RM_OUT permit 10
  match LOCAL-PREF 500
```

Figure 3.3: An example of matching LOCAL-PREF and the MED. Route map RM_IN has a route map item matching announcements with MED = 50. Route map RM_OUT has a route map item matching announcements with LOCAL-PREF = 500.

3.2.2 Community

As introduced in § 2.1.2, the standard format of a community value is *ASN:value* where ASN and value are both 16-bit long. The partition for the community is a little more difficult than LOCAL-PREF and MED because instead of matching a single integer, the *match* statement always matches a **community list** which can be transformed into DNF (disjunctive normal form, i.e., a disjunction of conjunctions).

Consider the configuration in Figure 3.4. CLIST is a community list consists of two community list items. Each item is a disjunctive part of the DNF, and each community value in an item is a conjunctive part of that item. Therefore, CLIST can be transformed to the DNF: 21:300 OR (21:200 AND 21:500). A route map item in the RM_IN matches this community list, meaning an announcement is matched as long as it contains both community values 21:200 and 21:500, or it contains 21:300.

```
route-map RM_IN permit 5
  match community CLIST

ip community-list standard CLIST permit 21:300
ip community-list standard CLIST permit 21:500 21:200
```

Figure 3.4: An example of matching the community. CLIST is a community list consists of two disjunctive community list items and the first item consists of two conjunctive community values. CLIST can be transformed to the DNF: 21:300 or (21:500 AND 21:200). A route map item in the RM_IN then matches this community list.

Based on the community list construction, we can define each community equivalence class to be each disjunctive part of a community list. For example, in Figure 3.4, $\{21:300\}$ is an equivalence class which stands for all community values that contain 21:300. $\{21:500, 21:200\}$ is another equivalence class that stands for all community values containing both 21:500 and 21:200 (in order). We also use -1 to represent all other possible community values that are handled in the default way

in the network. Therefore, if CLIST is the only community list in the configuration, then there are three partition labels for the community: $\{\{21:300\}, \{21:500, 21:200\}, -1\}$.

3.2.3 NEXT-HOP

The attribute NEXT-HOP specifies the next hop address for an announcement, therefore its equivalence class computation is similar to what one would expect in HSA or ddNF. A significant difference between NEXT-HOP and the attributes covered before is that NEXT-HOP matching can match a continuous range of IP addresses, instead of matching several discrete points. The IP address to be matched in the route map item is defined in the **prefix list**.

An introduction of a prefix list Figure 3.5 gives an example of matching NEXT-HOP. The prefix list PLIST consists of two **prefix list items** with different sequence numbers: 0.0.0.10/31 and 0.0.0.0/16. Similar to the route map item processing, the prefix list item with a lower sequence number has a higher priority and is processed first. There are also two list types: *permit* or *deny*. If the list type is *permit*, the announcement whose NEXT-HOP address belongs to that prefix list item will be matched and other announcements that do not match will go to the next list item with a lower priority. If the list type is *deny*, the announcement that has a matched NEXT-HOP will be filtered out and other announcements go to the next list item. For the equivalence class computation, the different list types do not make a difference since we only care about the attribute value. We talk more about different list types in the next chapter.

Absolute value interval Back to Figure 3.5, a route map item in the RM_IN matches NEXT-HOP with PLIST, meaning it matches any announcement whose NEXT-HOP address belongs to 0.0.0.0/16 but not 0.0.0.10/31. Therefore the acceptable NEXT-HOP addresses range from 0.0.0.0 to 0.0.0.9 or from 0.0.0.12 to 0.0.255.255. If we transform the 32-bit IP address to the **absolute value interval**, the acceptable interval is $[0, 10) \cup [12, 65536)^3$.

```
route-map RM_IN permit 5
  match ip next-hop prefix-list PLIST

ip prefix-list PLIST seq 5 deny 0.0.0.10/31
ip prefix-list PLIST seq 10 permit 0.0.0.0/16
```

Figure 3.5: An example of matching NEXT-HOP. PLIST is a prefix list consisting of two prefix list items 0.0.0.10/31 and 0.0.0.0/16. A route map item in the RM_IN matches the NEXT-HOP with PLIST. This means any announcement whose absolute value of its NEXT-HOP belongs to $[0, 10) \cup [12, 65536)$ will be matched by the RM_IN.

Attribute partition algorithm for NEXT-HOP We now compute the attribute partition for NEXT-HOP. Our goal is to find as few equivalence classes as possible while still covers the entire attribute space, and make sure all NEXT-HOP addresses covered in each class are treated the same. Take the PLIST for example, the two prefix list items lead to four equivalence classes: $[0, 10)$, $[10, 12)$, $[12, 65536)$ and $[65536, 2^{32})$.

³From now on, we will use the absolute value interval to represent a IP prefix without further explanation.

We use a tree algorithm Alg 1 to implement this. The algorithm gradually constructs a tree into which we insert each IP prefix list item in the form of its absolute value interval. In the end, it makes sure that each leaf node of the tree stands for an equivalence class. All equivalence classes are disjoint with each other and their union equals the entire 32-bit IPv4 address space.

Initially, the tree only has a root prefix $[0, 2^{32})$ which means that all IP addresses are treated the same if the configuration does not contain any prefix list. The tree then inserts each new IP prefix recursively based on the intersection between the new IP prefix and the IP prefixes that have been inserted before.

Algorithm 1 Equivalence class computation algorithm for NEXT-HOP

```

1: Input: prefixes: a list of IP prefixes in the form of their absolute value intervals
2: Output: root: the root of a tree, each leaf node is an equivalence class
3: Initialize: root  $\leftarrow [0, 2^{32})$   $\triangleright$  root.left = 0, root.right =  $2^{32}$ 

4: for prefix in prefixes do
5:   INSERT(root, prefix)
6: end for
7: return root

8: function INSERT(parent, prefix)
9:   if parent == prefix then
10:    return
11:   end if
12:   intersection = parent  $\cap$  prefix  $\triangleright$  i.e.,  $[0, 10) \cap [5, 15) = [5, 10)$ 
13:   if not intersection then
14:    return
15:   else if intersection == prefix then
16:     if not parent.child then
17:       parent.child.append([parent.left, prefix.left])  $\triangleright$  If parent.left == prefix.left, skip
18:       parent.child.append([prefix.left, prefix.right])
19:       parent.child.append([prefix.right, parent.right])  $\triangleright$  If prefix.right == parent.right, skip
20:     else
21:       for child in parent.child do
22:         INSERT(parent.child, prefix)
23:       end for
24:     end if
25:   else
26:     INSERT(parent, intersection)
27:   end if
28: end function

```

Figure 3.6 shows the algorithm output after inserting the two IP prefixes in PLIST. There are four equivalence classes in Figure 3.6, which are all consistent with the theoretical computation before.

Label assignment algorithm for NEXT-HOP We now assign a label for each equivalence class. If a class has an intersection with any input IP prefix, then we assign a positive label for this equivalence class. For all equivalence classes that do not have an intersection with any input IP prefix, we assign the label -1 collectively for them. To simplify the route map filtering process afterwards, we attach each NEXT-HOP label to the prefix list item it belongs to with Alg 2. In general, each prefix list shares the same copy of the labels, the prefix list item with a smaller sequence number first picks up the labels it belongs to, then the item with a higher sequence number picks up from the remaining labels.

After applying Alg 2, the PLIST in Figure 3.5 is like Figure 3.7. The first prefix list item $[10, 12)$ picks up the label 2 because only the absolute value interval with label 2 is a subset of $[10, 12)$. The second item $[0, 65536)$ picks up the label 1, 3. Although label 2 is also a subset of $[0, 65536)$ but it has been picked up by $[10, 12)$ which has a higher priority.

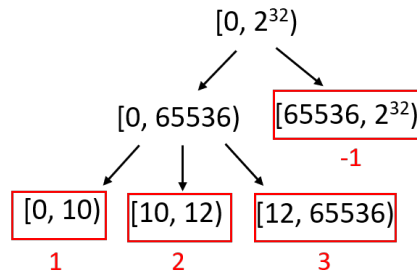


Figure 3.6: An example output of the NEXT-HOP partition algorithm. After inserting $[0, 65536)$ and $[10, 12)$, the algorithm calculates four equivalence classes labeled with 1, 2, 3, -1, where -1 stands for all equivalence classes that do not have an intersection with any input IP prefix.

Algorithm 2 Label distribution algorithm for NEXT-HOP

```

1: Input:
   prefix_list: a prefix list consisting of several prefix list items
   labels: a dictionary that maps each label to NEXT-HOP equivalence classes
2: Output: prefix_list: the prefix list with each prefix list item attached with NEXT-HOP labels

3: for item in SORTED(prefix_list) do           ▷ sort in the order of seq from smallest to largest
4:   for label in labels do
5:     if labels[label]  $\subseteq$  item.prefix then
6:       item.append(label)
7:       labels.remove(label)
8:     end if
9:   end for
10: end for
11: return prefix_list

```

```

route-map RM_IN permit 5
  match ip next-hop prefix-list PLIST

ip prefix-list PLIST seq 5 deny 0.0.0.10/31 NH: 2
ip prefix-list PLIST seq 10 permit 0.0.0.0/16 permit NH: 1, 3

```

Figure 3.7: An example of NEXT-HOP label distribution. $[10, 12)$ picks up the label 2, and $[0, 65536)$ picks up the label 1, 3. Although label 2 is also a subset of $[0, 65536)$ but it has been picked up by $[10, 12)$ which has a higher priority.

3.2.4 IP Prefix

The IP Prefix attribute is in the form of A.B.C.D/length, specifying the range of IP addresses announced by an announcement. IP Prefix is also matched by a prefix list, but with more matching rules compared to NEXT-HOP.

An additional introduction of a prefix list When we explain the equivalence computation for NEXT-HOP, we deliberately omit some parameters when defining a prefix list. The complete syntax of a prefix list item is shown in Figure 3.8. In a prefix list item, *eq*, *le*, *ge* are three optional parameters to specify the prefix length to match. For any matching announcement, the prefix in its IP Prefix (i.e., A.B.C.D) must be a subset of the prefix declared in the prefix list item. If the prefix list item is constrained with *eq*, it requires the IP prefix to have exactly the same prefix length (i.e., /length) as in the prefix list item. If the prefix list item is constrained with *ge* and *le*, then the prefix length *l* in the matching announcement must satisfies $ge \leq l \leq le^4$. For example in Figure 3.9, only the first two IP Prefixes fall into PLIST2 because their prefixes (i.e., A.B.C.D) is a subset of 0.0.0.0/16 and their prefix length is between *ge* and *le*.

```

ip prefix-list name [seq number] {permit|deny} prefix [eq length|ge length] [le length]

```

Figure 3.8: The syntax of a prefix list item. *eq*, *le*, *ge* are three optional parameters that specify the prefix length to match in IP Prefix. The prefix in any matching announcement's IP Prefix must be a subset of the prefix declared in the prefix list item. The *eq*, *le*, *ge* constrain the prefix length in the IP Prefix.

We do not cover *ge*, *le*, *eq* in the NEXT-HOP partition, because a NEXT-HOP address will be matched by a prefix list item as long as the address falls into the IP prefix (i.e., 0.0.0.0/16). However, when matching an IP Prefix, the prefix length will be examined by the prefix length constraint. Therefore we also need to take *ge*, *eq*, *le* into consideration during the IP Prefix partition.

The coverage of a prefix list Before we introduce the algorithm used for IP Prefix partition, we first look at what IP Prefixes are contained in a given prefix list with length constraints. We still take PLIST2 as an example. Figure 3.10 shows a subset of IP prefixes contained in PLIST2. When the prefix length is /24, any IP prefix whose absolute value interval is of length 2^8 and is a subset of $[0, 65536)$ is covered by PLIST2. When the prefix length is /25, any IP prefix whose absolute value interval is of length 2^7 and is a subset of $[0, 65536)$ is also covered by PLIST2.

⁴In a Cisco configuration, a valid prefix list item should have the prefix length constraint $ge \leq length \leq le$.

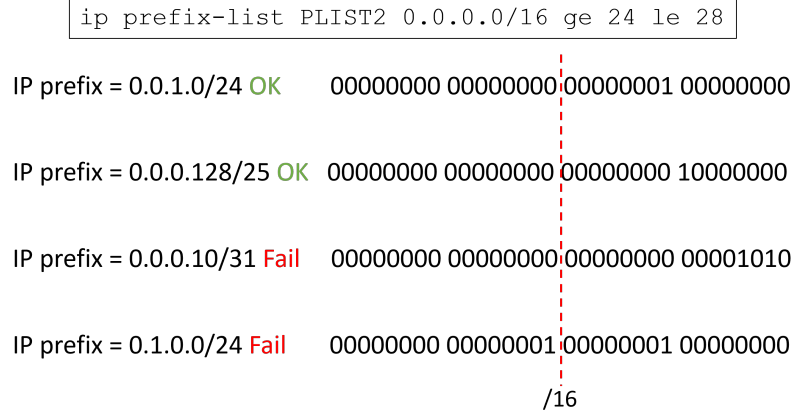


Figure 3.9: An example of a prefix list item with *ge* and *le*. For any IP Prefix A.B.C.D/length, it can be matched by PLIST2 as long as its prefix (A.B.C.D) is a subset of 0.0.0.0/16 and its prefix length (/length) is between *ge* and *le*.

With this observation, it is easy to conclude that an IP prefix is contained in a prefix list item $[a, b)$ *ge c le d* if its absolute value interval $[x, y)$ satisfies Eq 3.1 and Eq 3.2. If the prefix list item is constrained with *eq*, then only the IP prefix with the same prefix/length as in the item will be matched.

$$a \leq x \leq y \leq b \quad (3.1)$$

$$2^l = y - x, \quad \text{where } c \leq 32 - l \leq d \quad (3.2)$$

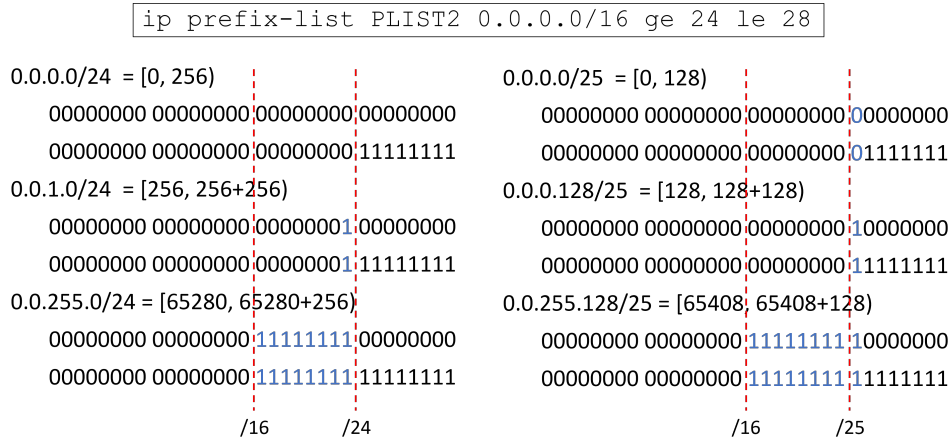


Figure 3.10: A subset of IP prefixes contained in PLIST2. When the prefix length is 24/25, any IP prefix whose absolute value interval is of length $2^8/2^7$ and is a subset of $[0, 65536)$ is covered in the PLIST2.

A 4-tuple representation of a prefix list item We can then define a 4-tuple to represent each prefix list item $[a, b)$ *ge c le d*: $(a, b, 32 - d, 32 - c)$ where *a* and *b* are the left (closed) and right

(open) boundaries of all IP prefixes (in the form of absolute value interval) covered by the item, 2^{32-d} and 2^{33-c} are the minimum (closed) and maximum (open) interval length of any covered IP prefix. For example, the PLIST2 is now represented as (0, 65536, 4, 9).

Attribute partition algorithm for IP Prefix Compared TO the NEXT-HOP partition algorithm Alg 1 which only needs to calculate the prefix intersection during the insertion, IP Prefix requires another calculation of the prefix length intersection. For example, (0, 2^{32} , 0, 33) stands for the entire prefix list item space, if we insert (0, 65536, 4, 9) under (0, 2^{32} , 0, 33), we should get the following equivalence classes: (0, 65536, 0, 4), (0, 65536, 4, 9), (0, 65536, 9, 33) and (65536, 2^{32} , 0, 33).

To achieve this, we can reuse Alg 1 by attaching an inner tree to each prefix node and gradually building both the inner and outer tree when recursively inserting a new 4-tuple node. The inner tree insertion process is the same as described in Alg 1 except that we initialize the root to be [0,33) instead of [0, 2^{32}) and we input the prefix length intervals instead of prefix intervals. The IP Prefix partition algorithm is given in Alg 3. Figure 3.11 shows the algorithm output after inserting the tuple (0, 65536, 4, 9). The outer tree structure is colored in black and the inner tree attached to each node is colored in blue.

Label assignment algorithm for IP Prefix We then distribute IP Prefix labels to each prefix list item in a similar way as in Alg 2 with a minor change in line 5. Instead of only comparing the prefix interval, we attach an IP Prefix equivalence class (a, b, c, d) (if available) to a prefix list item (x, y, u, v) when $x \leq a \leq b \leq y$ and $u \leq c \leq d \leq v$. After the label distribution, PLIST2 will be like Figure 3.12, where the PLIST2 is attached with its corresponding IP label.

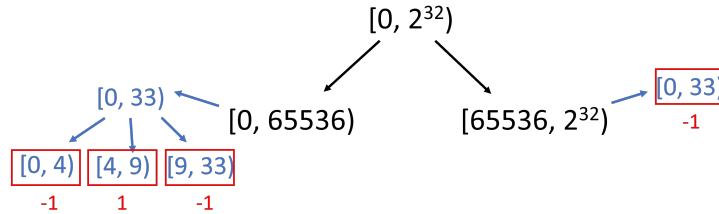


Figure 3.11: An example output of the IP Prefix partition algorithm. The outer tree and the inner tree are distinguished by different colors. After inserting (0, 65536, 4, 9), the algorithm calculates four equivalence classes with 2 labels 1, -1. This is because only label 1 is a subset of (0, 65536, 4, 9).

```
route-map RM_IN permit 5
  match ip address prefix-list PLIST2

ip prefix-list PLIST2 seq 5 permit 0.0.0.0/16 le 24 le 28 IP: 1
```

Figure 3.12: An example output of IP Prefix label distribution. The PLIST2 picks up IP Prefix label 1.

Algorithm 3 Equivalence class computation algorithm for IP Prefix

```

1: Input: plist_items: a list of prefix list items in the form of 4-tuple
2: Output: root: the root of a two-layer tree, each leaf node of the inner tree attached to a
   leaf node of the outer tree is an equivalence class
3: Initialize:  $root \leftarrow (0, 2^{32})$ ,  $root.length \leftarrow (0, 33)$ 
    $\triangleright a$  stands for the prefix interval (i.e, first 2 values in the 4-tuple),  $a.length$  stands for the
   length interval (i.e., the last 2 values).  $a.inner$  points to the root of  $a$ 's inner tree.

4: for item in plist_list do
5:   INSERTIP(root, item)
6: end for
7: return root

8: function INSERTIP(parent, item)
9:   if item == parent then
10:    if parent.inner exists then
11:      INSERT(parent.inner, item.length)
    $\triangleright$  call the modified Alg 1 ( $root \leftarrow [0, 33)$ ) to insert item.length into the inner tree
12:    else
13:      for child in parent.child do
14:        INSERTIP(parent.child, item)
15:      end for
16:    end if
17:  else  $\triangleright$  similar to Alg 1 line 11-25
18:     $intersection = parent \cap item$ 
19:    if not intersection then
20:      return
21:    else if intersection == item.prefix then
22:      if not parent.child then:
23:        parent.child.append([parent.left, prefix.left])
24:        parent.child.append([prefix.left, prefix.right])
25:        parent.child.append([prefix.right, parent.right])
26:      for child in parent.child do
27:        parent.child.inner = parent.inner
28:      end for
29:      parent.inner = None
    $\triangleright$  clear parent's inner tree once it has been inherited by all child nodes
30:    else
31:      for child in parent.child do
32:        INSERTIP(parent.child, item)
33:      end for
34:    end if
35:  else
36:    INSERTIP(parent, intersection)
37:  end if
38: end if
39: end function

```

3.2.5 AS-PATH

The AS-PATH attribute records all ASes the announcement has passed, as introduced in § 2.1.2. An AS-PATH value is a long string of ASNs separated by spaces (e.g., 400 900 1800).

An introduction of AS-PATH list Unlike other attributes, AS-PATH cannot be matched by an exact value because the path length and the ASNs can be arbitrary. Instead, in the route map, an AS-PATH is matched by a regular expression defined in the **AS-PATH list**. Similar to other attribute lists, each AS-PATH list can consist of multiple AS-PATH list items. Each list item has a list type *permit|deny* and a regular expression to specify the AS-PATH this item should match. The usage of *permit|deny* is the same as in a prefix list. For example, Figure 3.13 defines an AS-PATH list ALIST that matches any announcement that is not originated by AS1800 and pass AS100 or originated by AS900⁵. Table 3.1 lists the most common regular expressions in an AS-PATH list.

```
route-map RM_IN permit 5
  match as-path ALIST

ip as-path access-list ALIST deny _1800$
ip as-path access-list ALIST permit _100_
ip as-path access-list ALIST permit _900$
```

Figure 3.13: An example of matching AS-PATH. ALIST is an AS-PATH list consisting of three as-path list items, it matches an announcement if its AS-PATH is not originated by AS1800 and pass AS100 or originated by AS900.

Regex	Indication
_1800\$	originated by AS1800
^1800_	received from AS1800
1800	via AS1800
_790_1800_	via AS1800 and AS790 in order
^[0-9]+	matches AS path length of 1

Table 3.1: The most common regex constraints for AS-PATH

Attribute algorithm for AS-PATH Similar to the prefix list partition, we compute the equivalence classes for AS-PATH based on all AS-PATH list items. However, the regular expressions do not directly indicate how they intersect with each other. Therefore, we need a new algorithm to compute the intersection between two regex constraints before the partition.

One way is to use an existing regex library [1]. However, since only a small subset of regexes are used in the AS-PATH list item (as shown in Table 3.1), we can also design a more specialized way to realize this. Our solution is to use a binary decision tree called **regex-tree** to calculate all **possible regex paths** given a set of AS-PATH list items. Each possible regex path is constrained by a set of regexes placed on this path which do not have any conflict with each other (i.e., can be satisfied at the same time).

⁵In AS-PATH, ASNs are listed in the order of appearance from right to left, therefore an ASN of the AS that originates an announcement appears in the rightmost position.

Each regex on a path can be seen as a property of this path, and a property can have two states: hold or not hold. For example, we use $\hat{1800}_- = T$ to represent that the property $\hat{1800}_-$ holds on this path, and use $\hat{1800}_- = F$ to represent that it does not hold. Therefore, it is easy to find that a possible regex path can have both $\hat{1800}_- = T$ and $_{100}_- = T$, but it cannot have both $\hat{1800}_- = T$ and $\hat{900}_- = T$.

Alg 4 gives the general building flow of the regex-tree (we leave out the function *CheckConflict* for later discussion). Figure 3.14 visualizes the structure of the regex-tree with ALIST. Initially, the tree only has a root regex \cdot^* which stands for no constraint on the AS-PATH values. After inserting $\hat{1800}_-$, $_{100}_-$ and $\hat{900}_-$, there are six possible regex paths in the tree. Each path also maintains a **regex-list** that records if a regex property holds or not on this path. In Figure 3.14, all regex properties listed in the regex-list are on the corresponding path. We will see that this is not always the case when we talk about the regex conflict resolution later.

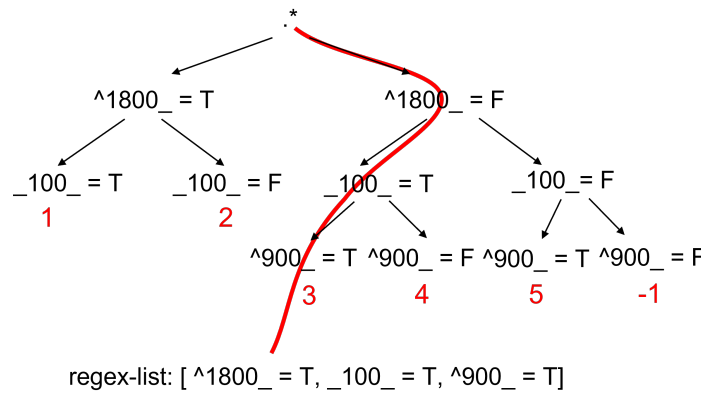


Figure 3.14: An example of AS-PATH regex-tree. After inserting $\hat{1800}_-$, $_{100}_-$ and $\hat{900}_-$, there are six possible regex paths.

Label assignment algorithm for AS-PATH As can be inferred from Figure 3.14, each regex path/regex-list also stands for a disjoint equivalence class for AS-PATH. We give each regex-list a positive label if its regex-list contains any property that holds (i.e., some regex = T), and assigns -1 for all equivalence classes that only contain properties that do not hold (i.e., some regex = F). We then distribute the path labels to each AS-PATH list item by looking at each path regex-list. The label distribution algorithm is given in Alg 5. Although AS-PATH lists do not have sequence numbers to order its items, we can assume the item that appears early in the list has a higher priority and assign a lower sequence number for it. The AS-PATH list after applying Alg 5 is shown in Figure 3.15, where each AS-PATH list item is attached with its corresponding AS-PATH labels.

Conflict resolution We now discuss the **conflict resolution**, which is the most challenging task in the AS-PATH partition algorithm. In general, there are two types of conflict: **mutual conflict** and **non-mutual conflict**. Mutual conflict means the conflict can be detected when doing the pairwise comparison during the insertion. Some examples of mutual conflict are listed as follows. For example, In the first bullet point, $\hat{1800}_-$ and $\hat{900}_-$ are two regexes that match on AS paths that have different last receivers, therefore they can be satisfied at the same time. In the second bullet point, $\hat{1800}_-$ match on AS paths that are received by AS1800, $_{900}_-$ match on AS

Algorithm 4 Equivalence class computation algorithm for AS-PATH

```

1: Input: regexes: a list of regexes appear in any AS-PATH list item
2: Output: root: the root of a regex-tree, each path of the tree is an equivalence class
3: Initialize: root  $\leftarrow$   $. * = T$ 

4: for regex in regexes do                                 $\triangleright$  here regex is without state (i.e., hold/not hold)
5:   INSERTAS(root, regex)
6: end for
7: return root

8: function INSERTAS(parent, regex)
9:   if parent == regex then                                 $\triangleright$  only compare the regex, not the state
10:    return
11:   end if
12:   result = CHECKCONFLICT(regex, parent.path)               $\triangleright$  parent.path points to the current path the parent is on
13:   if result = Hold then                                     $\triangleright$  regex = F has conflict with the path
14:     parent.path.regex_list.append(regex = T)
15:   else if result = NotHold then                             $\triangleright$  regex = T has conflict with the path
16:     return
17:   else
18:     if not parent.child then
19:       parent.child_1 = (regex = T)
20:       parent.child_1.path.regex_list = parent.path.regex_list
21:       parent.child_1.path.regex_list.append(regex = T)
22:        $\triangleright$  update the regex-list with the new property
23:       parent.child_2 = (regex = F)
24:       parent.child_2.path.regex_list = parent.path.regex_list
25:       parent.child_2.path.regex_list.append(regex = F)
26:     else
27:       for child in parent.child do
28:         INSERTAS(parent.child, regex)
29:       end for
30:     end if
31:   end if
32: end function

```

Algorithm 5 Label distribution algorithm for AS-PATH

```

1: Input:
   as_list: an as-path list consisting of several as-path list items.
2: labels: a dictionary that maps each label to AS-PATH regex-lists.
3: Output: as_list: the as-path list with each as-path list item attached with AS-PATH labels.

4: for item in SORTED(as_list) do
5:   for label in labels do
6:     if item.regex = T in labels[label] then
7:       item.append(label)
8:       labels.remove(label)
9:     end if
10:  end for
11: end for
12: return as_list

```

```

route-map RM_IN permit 5
  match as-path ALIST

ip as-path access-list ALIST deny _1800$ AS: 1, 2
ip as-path access-list ALIST permit _100_ AS: 1, 3, 4
ip as-path access-list ALIST permit _900$ AS: 3, 5

```

Figure 3.15: An example of AS-PATH label distribution. Each AS-PATH list item picks up the labels it belongs to, the item that appears early in the list has a higher priority.

paths that first pass AS1800 then AS900. Since a valid AS path cannot have duplicate ASNs, these two regexes cannot be satisfied at the same time.

- $\text{_1800_} = T$ and $\text{_900_} = T$
- $\text{_1800_} = T$ and $\text{_900_1800_} = T$
- $\text{_100_200_300_} = T$ and $\text{_300_500_100_} = T$
- $\text{_100_200_} = T$ and $\text{_100_} = F$
- $\text{_100_300_} = T$ and $\text{\^{}[0-9]\+} = T$

Handling mutual conflicts is not difficult. Since we always insert a new regex from the tree root, we can check the mutual conflict before we perform the insertion.

Resolving non-mutual conflicts is much more challenging. A non-mutual conflict means the conflict cannot be detected in the pairwise comparison and can only be found when knowing the global information of the path. Some examples of non-mutual conflicts are listed as follows. For example, in the first bullet point, when $\text{_100_200_} = T$ and $\text{_200_300_} = T$ already appear on the same path, then this path can never match _300_100_ , otherwise a loop is created in the AS path. In the second bullet point, if $\text{_100_300_500_} = F$ and $\text{_100_300_} = T$ are in the same path, then this path can no longer match _300_500_ , otherwise the property _100_300_500_ begins to hold.

- $_100_200_ = T, _200_300_ = T \rightarrow _300_100_ = T$
- $_100_300_500_ = F, _100_300_ = T \rightarrow _300_500_ = T$
- $_100_200_ = T, _200_300_ = T \rightarrow _100_300_ = T$
- $_100_200_ = T, _100_300_ = T, _200_300_ = F \rightarrow _300_200_ = F$
- $_100_ = T, _200_ = T \rightarrow \wedge[0-9]^+ = T$

Unlike mutual conflicts, it is not easy to detect the non-mutual conflict for a set of waypoint regexes at a first glance. It is easier to find the conflict by combining a **true graph** and **false list** to record these waypoint regexes. A true graph is a directed graph that stores the waypoint regexes which hold in the path. A false list stores the waypoint regexes which do not hold in the path. Take the second non-mutual conflict example in the bullet list above, the true graph and the false list before inserting $_300_500_ = T$ is shown in Figure 3.16 (black). When we try to add $_300_500_ = T$ to the regex path, the updated true graph (red) will then have the connection $100 \rightarrow 300 \rightarrow 500$, which violates the false list. Therefore the property $_300_500_ = T$ conflicts with the path.

$$_100_300_500_ = F \quad _100_300_ = T \quad \rightarrow \quad _300_500_ = T$$

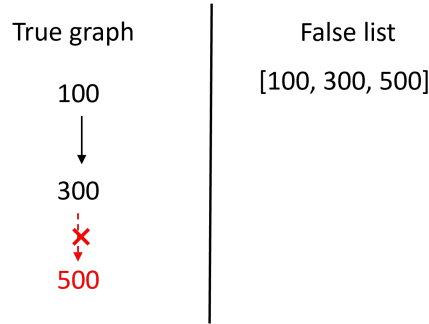
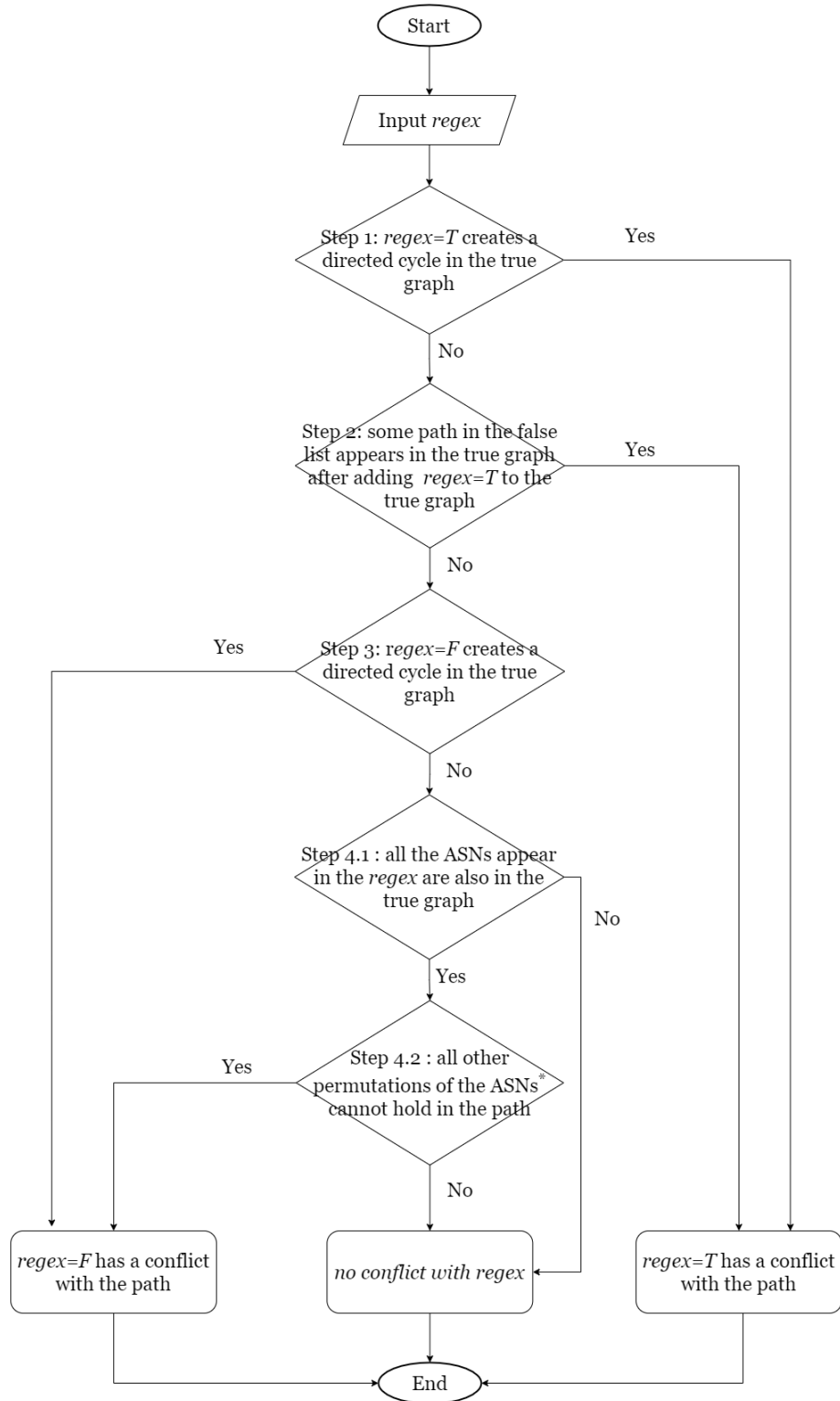


Figure 3.16: An example of the true graph and false list. Initially, the path has the property $_100_300_500_ = F$ and $_100_300_ = T$. When we try to add $_300_500_ = T$ to the regex path, the updated true graph will then have the connection $100 \rightarrow 300 \rightarrow 500$, which violates the false list.

Aside from the example given above, there are other conditions when a set of waypoint regexes create a conflict (i.e., create a directed cycle in the true graph). In general We use a four-step procedure as shown in Figure 3.17 to detect any non-mutual waypoint conflict⁶:

With the above four steps, we are now able to detect all non-mutual waypoint conflicts we can come up with. When we detect that $regex = T$ has a conflict with the regex path, we stop the insertion and return (see Alg 4 line 15-16). When we detect that $regex = F$ has a conflict (i.e., $regex = T$ already holds in the path), we only add the property $regex = F$ to the path regex-list without inserting any child into the path (see Alg 4 line 13-14). When there is no conflict and we have reached the end of a path, we are safe to append both $regex = T$ and $regex = F$ to the regex path, otherwise we move to the next hop on the path.

⁶We do not cover conflicts regarding regexes like $\wedge 1800_$ and $_900\$$ here because such conflicts can be detected with some simple if-else checking.



* if $regex = _100_300_500_$, then all other permutations are: $_100_500_300_$, $_300_100_500_$, etc

Figure 3.17: The flow chart of the function *CheckConflict*.

For the non-mutual conflict regarding the regex $^{[0-9]}+$, we can deal with it by inserting it after all other regexes and counting the number of different ASNs appear on each regex path. One thing to note is that when a path has the property $^{1800}_=T$ and $_{1800}\$ = T$, then $^{[0-9]}+$ always holds.

A limitation of our AS-PATH partition is that it only supports the preliminary regexes that are listed in Table 3.1 so far. However, for other more complex regexes, we can first disassemble it into a set of preliminary regexes that form the logical AND/OR relationship. For example, the regex $^{6553}[0-1]_$ can be disassembled to $^{6553}0_$ OR $^{6553}1_$.

3.3 Symbolic Announcement Creation

So far, we have calculated the equivalence classes for the most common attributes in an announcement as well as for IP Prefix. We have also attached the equivalence class labels to the route map for NEXT-HOP, IP Prefix and AS-PATH as shown in Figure 3.7, 3.12 and 3.15. We now create the symbolic announcement.

As defined at the beginning of this chapter, a symbolic announcement should be able to represent the entire BGP announcement space. Recall that for each attribute, we compute the equivalence classes whose union covers the entire attribute space, we can fill each attribute in the symbolic announcement with the set of equivalence classes for that attribute. With the symbolic announcement, any concrete BGP announcement will now be represented by a combination of equivalence classes from each attribute. For example, if we have the configuration shown in Figure 3.18, the symbolic announcement will be like Figure 3.19.

```

route-map RM_IN1 permit 5
  match MED 50

route-map RM_IN2 permit 10
  match community CLIST

route-map RM_IN3 permit 15
  match ip next-hop prefix-list PLIST

route-map RM_IN4 permit 20
  match ip address prefix-list PLIST2

route-map RM_IN5 permit 25
  match as-path ALIST

route-map RM_OUT permit 5
  match LOCAL-PREF 500

ip community-list standard CLIST permit 21:300
ip community-list standard CLIST permit 21:500 21:200

ip prefix-list PLIST seq 5 deny 0.0.0.10/31 NH: 2
ip prefix-list PLIST seq 10 permit 0.0.0.0/16 NH: 1, 3

ip prefix-list PLIST2 seq 5 permit 0.0.0.0/16 ge 24 le 28 IP: 1

ip as-path access-list ALIST deny _1800$ AS: 1, 2
ip as-path access-list ALIST permit _100_ AS: 1, 3, 4
ip as-path access-list ALIST permit _900$ AS: 3, 5

```

Figure 3.18: An configuration after applying attribute partition. Red values are the equivalence classes extracted from the configuration (without -1). For prefix lists and AS-PATH lists, the equivalence classes are represented by symbolic labels.

```

Symbolic announcement: {
  LOCAL-PREF: 500, -1
  MED: 50, -1
  Community: {21:300}, {21:200, 21:500}, -1
  NEXT-HOP: 1, 2, 3, -1
  IP Prefix: 1, -1
  AS-PATH: 1, 2, 3, 4, 5, -1
}

```

Figure 3.19: The symbolic announcement created from Figure 3.18. Each attribute in the symbolic announcement is the set of equivalence classes of that attribute. The symbolic announcement is then able to represent the entire BGP announcement space.

Chapter 4

Geometric Model

In the last chapter, we introduce the algorithms and data structures we use to create the symbolic announcement and replace each attribute value with the equivalence classes it belongs to. With this preparation, we can now build the geometric model to process the symbolic announcement and to verify different BGP control plane properties. In this chapter, we first provide a general block diagram of the entire model, then we go into the details of each module in the diagram.

4.1 Block Diagram

Figure 4.1 shows the block diagram design for our geometric model. Overall, the model can be divided into three stages. In the first stage, we use a parser to extract the network topology and route map information from each router configuration file. In the second stage, we compute the attribute partition and create the symbolic announcement with the methods we have explained in the last chapter. In the third stage, we input the symbolic announcement, the network topology and a BGP property query into the verifier, which then starts to process the symbolic announcement filtering inside the network and finally returns the answer to the query based on the filtering result.

4.2 Parser

Before we can use the geometric model to propagate the symbolic announcement and verify properties, we need to extract the network information from the real device configurations. The challenge is that the configurations usually consist of hundreds or even thousands of lines of low-level directives. Therefore, we need a parser to automatically capture those lines that contain the network topology and BGP control plane information.

The parser we design translates the low-level configuration to the high-level Python objects. It is able to recognize the BGP session settings each router declares in the configuration and compute the complete network topology graph. It also extracts the complete route map corresponding to each BGP session and gathers all attribute values that appear in any configuration for later attribute partition.

After parsing all configurations, the parser passes the route map information (e.g., prefix-list matches) to the attribute partitioner, which then computes the equivalence classes for each attribute and creates the symbolic announcement as we have covered in the last chapter.

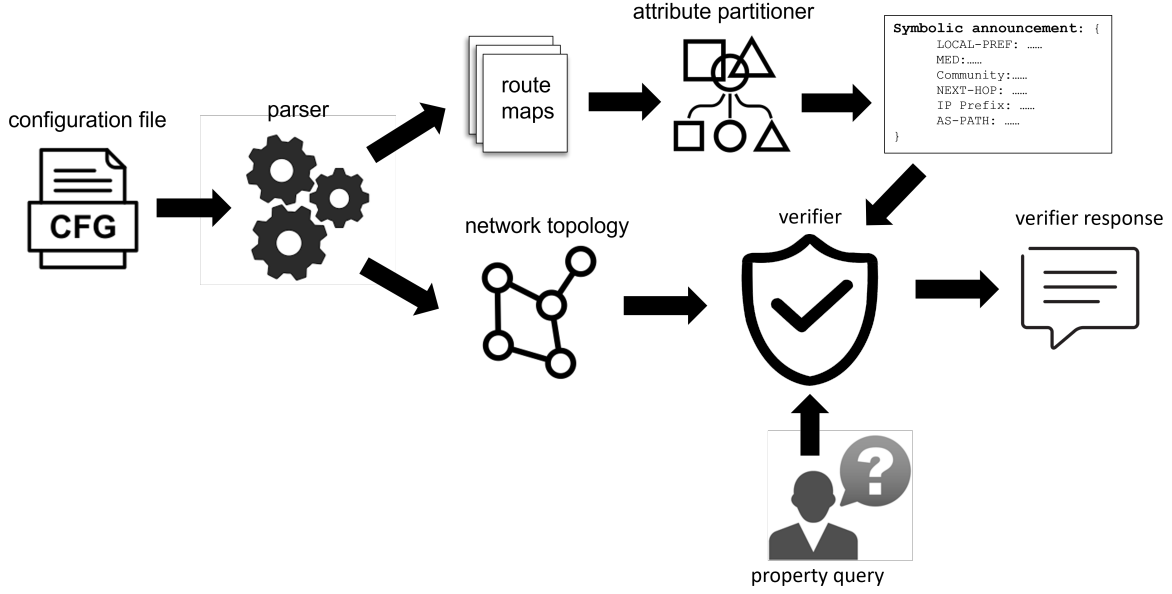


Figure 4.1: The block diagram of the geometric model. The overall work flow can be divided into three stages: 1. configuration parsing; 2. symbolic announcement creation; 3. property verification.

4.3 Route Map Filtering

When a symbolic announcement is input into the network from an ingress point, it represents all the possible BGP announcements that could be received from an external neighbor having an eBGP session with that ingress point. This symbolic announcement will then be filtered by each router it passes through. When a symbolic announcement finally leaves the network from different egress points, the remaining content of the symbolic announcement stands for all possible BGP announcements that could reach that specific egress point. The network verifier can then check if a BGP property holds based on the content in each export announcement. Therefore, before we show how our verifier answers each query, we first introduce how the symbolic announcement is filtered by a route map.

4.3.1 Single matching

We first look at the case when the symbolic announcement is filtered by a route map item with a single *match* statement. Consider the example in Figure 4.2. The route map item matches any announcement with $MED = 50$, then it sets the LOCAL-PREF of all matched announcements to 100. When the symbolic announcement passes this route map item, the label 50 in MED is picked up, and all other attribute labels do not change. This means that as long as an announcement has $MED = 50$, it will be matched no matter what the values of other attributes are. The remaining symbolic announcement will then be filtered by the next route map item. In this example, the remaining symbolic announcement represents all possible BGP announcements that do not have $MED = 50$.

The route map item that matches a particular attribute list needs few more steps to compute the matching symbolic announcement because of the list type *permit|deny*. For clarification, we first compare the effect of *permit|deny* when it appears in an attribute list and in a route map

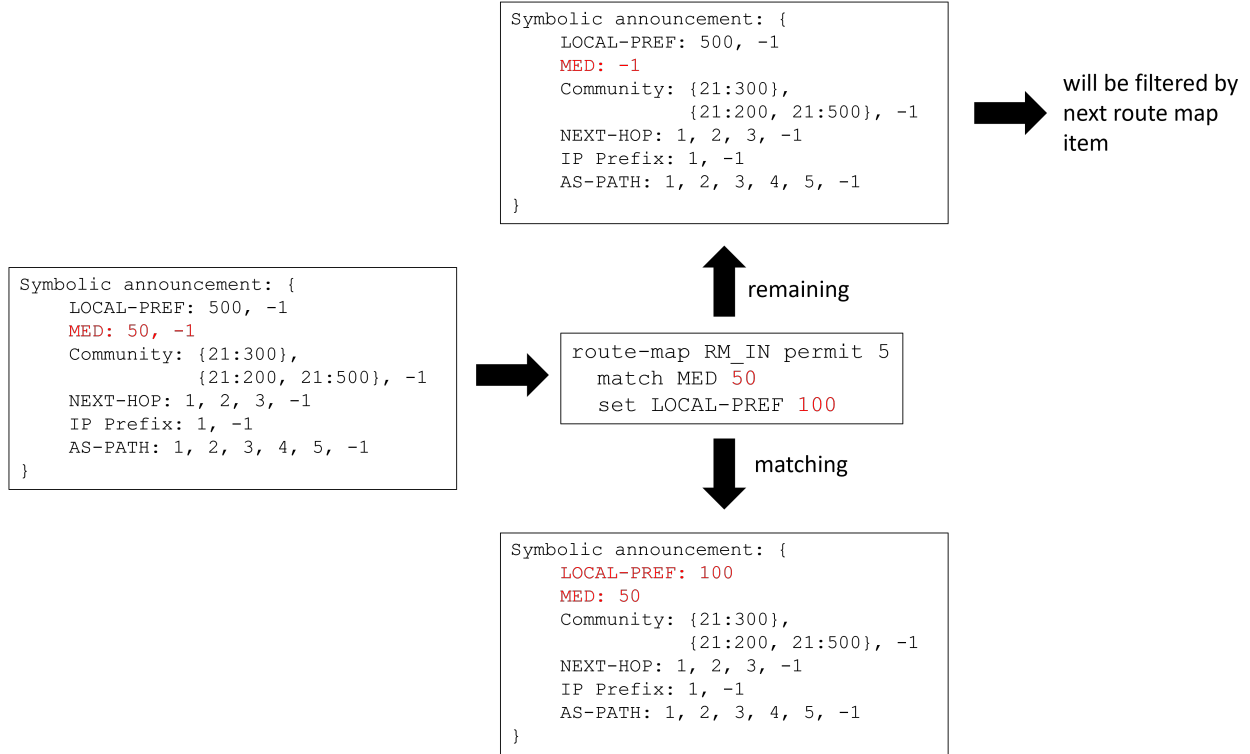


Figure 4.2: An example of single matching (MED). The symbolic announcement with MED label = 50 is matched by the route map item and the LOCAL-PREF of the matching symbolic announcement is set to 100. The remaining symbolic announcement (with MED = -1) passes through the next route map item.

item. As shown in Figure 4.3, a list item of list type *deny* filters out the label(s) from the input symbolic announcement, the remaining announcement passes through the next list item which has a lower priority, if the next item is of type *permit*, then the label(s) declared in that item will be matched. In the end, the matching symbolic announcement will contain the union of labels that are listed in each item of *permit* type. And the remaining announcement will be the difference between the original input announcement and the matching announcement. On the contrary, when a route map item has the route map type *deny*, it will drop the matching announcement matched by the list, otherwise, the matching announcement is accepted.

We now consider the route map filtering in Figure 4.4. The route map item matches the AS-PATH list ALIST, which denies AS-PATH labels 1, 2 and permits label 3. Therefore, the matching announcement will only have label 3 in the AS-PATH attribute, and all the other AS-PATH labels appear in the remaining announcement. The same filtering process also applies to the prefix list, which can be generalized in Alg 6. However it is a little different for the community list.

Recall our partition method for the community attribute, we define each set of community values that appears in a community list item as a community equivalence class. Therefore, there could be overlaps between different community equivalence classes. For example, there can be two equivalence classes: {21:200} and {21:200, 21:500}. It is apparent that for any community values in an announcement, if it belongs to the class {21:200, 21:500}, it also belongs to the class {21:200}. Therefore, we need to take care of this when processing the community value filtering. In practice,

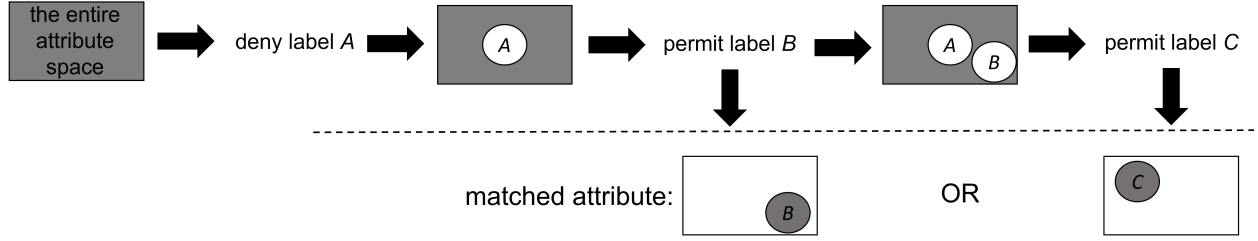


Figure 4.3: The filtering process inside an attribute list. A symbolic announcement passes through each list item in the order of their priority. An item with list type *deny* filters out the label(s) in the symbolic announcement, and the list item with *permit* matches the label(s).

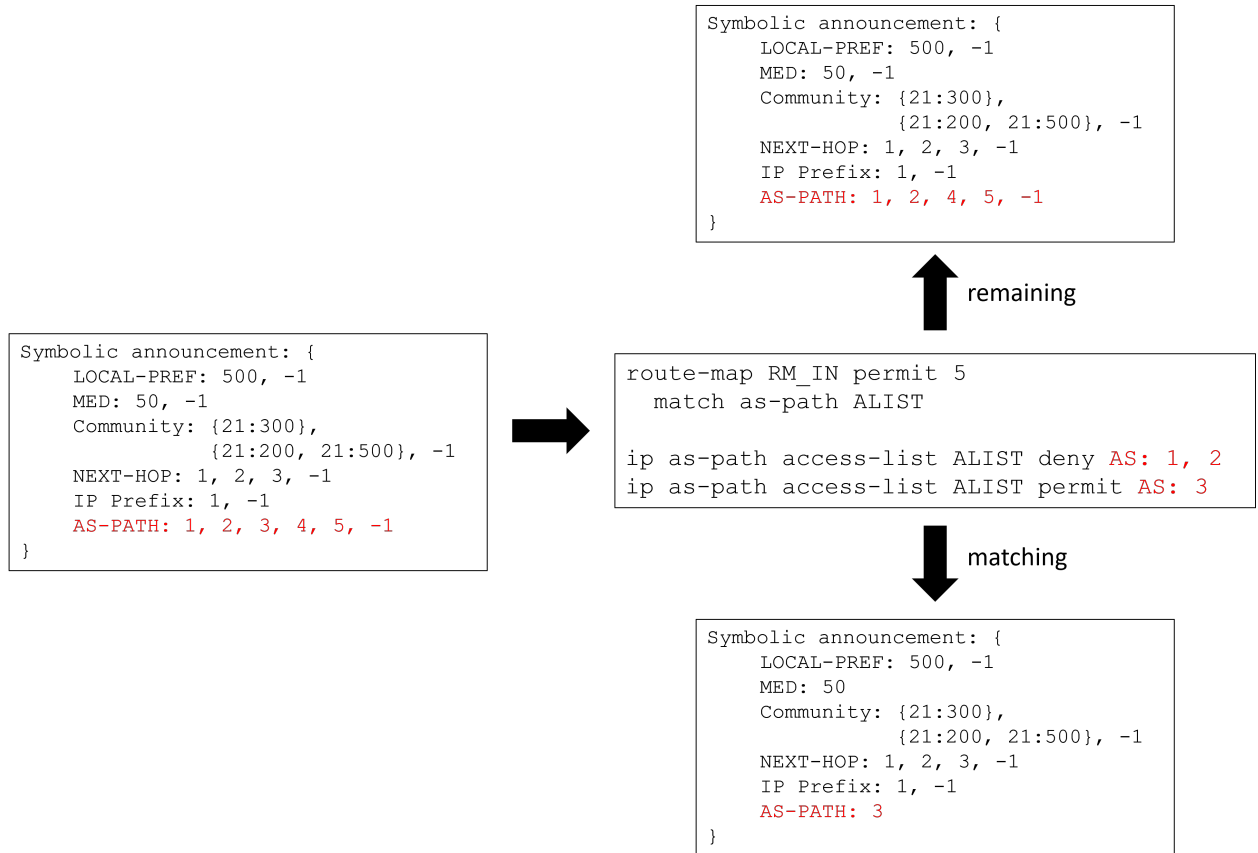


Figure 4.4: An example of single matching (AS-PATH). The symbolic announcement with AS-PATH label = 3 is matched by the route map item. The remaining symbolic announcement containing the remaining AS-PATH labels pass through the next route map item.

we use Alg 7 to deal with the community list filtering. With Alg 7, we can make sure that labels {21:200} and {21:500, 21:200} in a symbolic announcement will be both matched by the community list item that permits {21:200}.

Algorithm 6 Single matching algorithm for the prefix list and the AS-PATH list

```

1: Input:
   labels: a set of NEXT-HOP or IP Prefix or AS-PATH labels in the input symbolic announce-
   ment
2: list: the prefix list or the AS-PATH list that is going to match the labels
3: Output:
   matching: the matching attribute labels
   remaining: the remaining attribute labels that are not matched by the list

4: matching = set()
5: for item in SORTED(list) do           ▷ sort the list items in the order of their sequence numbers
6:   if item.type == permit then
7:     intersection = list[item].labels ∩ labels
8:     matching.include(intersection)
9:   end if
10:  remaining = labels.difference(intersection)
11: end for
12: return matching, remaining

```

Algorithm 7 Single matching algorithm for the community list

```

1: Input:
   labels: a set of community labels in the input symbolic announcement
2: comm.list: the community list that is going to match the labels
3: Output:
   matching: the matching community labels
   remaining: the remaining community labels that are not matched by the comm.list

4: matching = set()
5: for ann_label in labels do           ▷ each label is a set of community values (i.e., {21:200, 21:500})
6:   for list_label in comm.list do
7:     if list_label.values ⊆ ann_label.values then
8:       if list_label.type = permit then
9:         matching.include(ann_label)
10:      break
11:    end if
12:  end if
13: end for
14: end for
15: return matching, remaining

```

4.3.2 Multiple matching

A route map item can also have multiple *match* statements. In this case, only the announcement that satisfies **all** the *match* statements will be matched. Figure 4.5 gives an example of the multiple matching. In general, the creation of the matching announcement is similar to the single matching. If the symbolic announcement has the required labels in all attributes to be matched in the route map item, then a matching announcement will be created with the matched labels for the specified attributes and arbitrary labels for the others. The creation of the remaining announcements is however different because multiple combinations of the attributes appear in the remaining announcements.

As shown in Figure 4.5, unlike the single matching where only one remaining announcement passes through the next route map item, three disjoint remaining announcements are generated when the symbolic announcement passes the route map item. Alg 8 explains how they are generated. In general, the input symbolic announcement is matched by one *match* at one time, the local matching and remaining announcement generated by the current *match* both go to the next *match*. This process is repeated until all *match* statements are traversed. Then, only the announcement that is matched by all *match* statements will be treated as the output matching announcement, all the other announcements generated by the last *match* are treated as the remaining announcements. If there is no matching announcement, then the original input symbolic announcement passes through the next route map item (not shown in Alg 8).

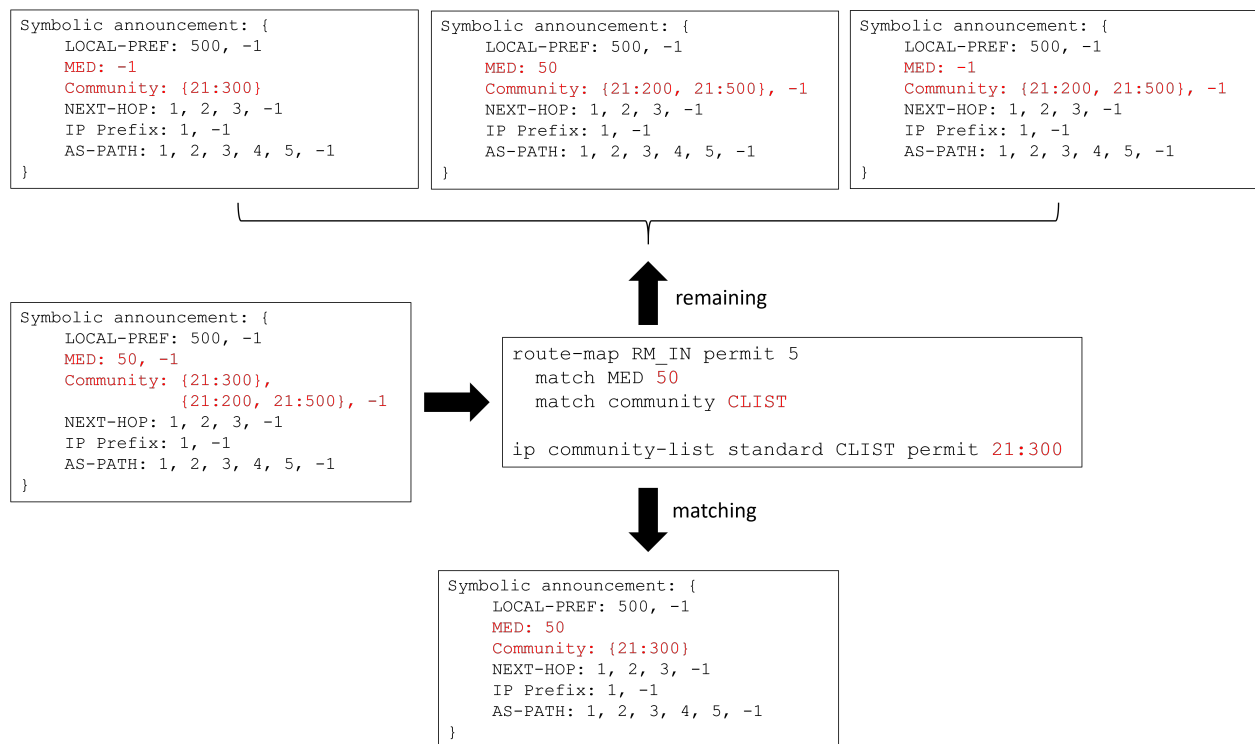


Figure 4.5: An example of the multiple matching. The route map item matches the announcement that has MED = 50 and satisfies community list CLIST. Three remaining announcements generated by the last *match* pass through the next route map item.

Algorithm 8 Multiple matching algorithm

```

1: Input:
   sym_ann: the symbolic announcement to be matched
   rm_item: the route map item with multiple match statements
2: Output:
   matching: the matching announcement
   remaining: the remaining announcement

3: matching = COPY(sym_ann)
4: remain_stack = list()
   ▷ store all the remaining announcements generated from last match
5: tmp_remain_stack = list()
   ▷ store the remaining announcements generated from the current match
6: for match in rm_item.matches do
7:   while len(remain_stack) do                                ▷ filter each remaining announcement
8:     remain_1, remain_2 = match.filter(remain_stack).pop()
9:     tmp_remain_stack.push(remain_1, remain_2)
10:  end while
11:  matching, remain_0 = match.filter(matching)                ▷ update the matching announcement
12:  tmp_remain_stack.push(remain_0)
13:  remain_stack = tmp_remain_stack
14:  tmp_remain_stack.clear()
15: end for
16: remaining = remain_stack
17: return matching, remaining

```

After applying Alg 8, the matching and remaining announcements are structured as in Figure 4.6. In general, if a route map item has n *match* statements, it will generate 1 matching announcement and $2^n - 1$ remaining announcements in total.

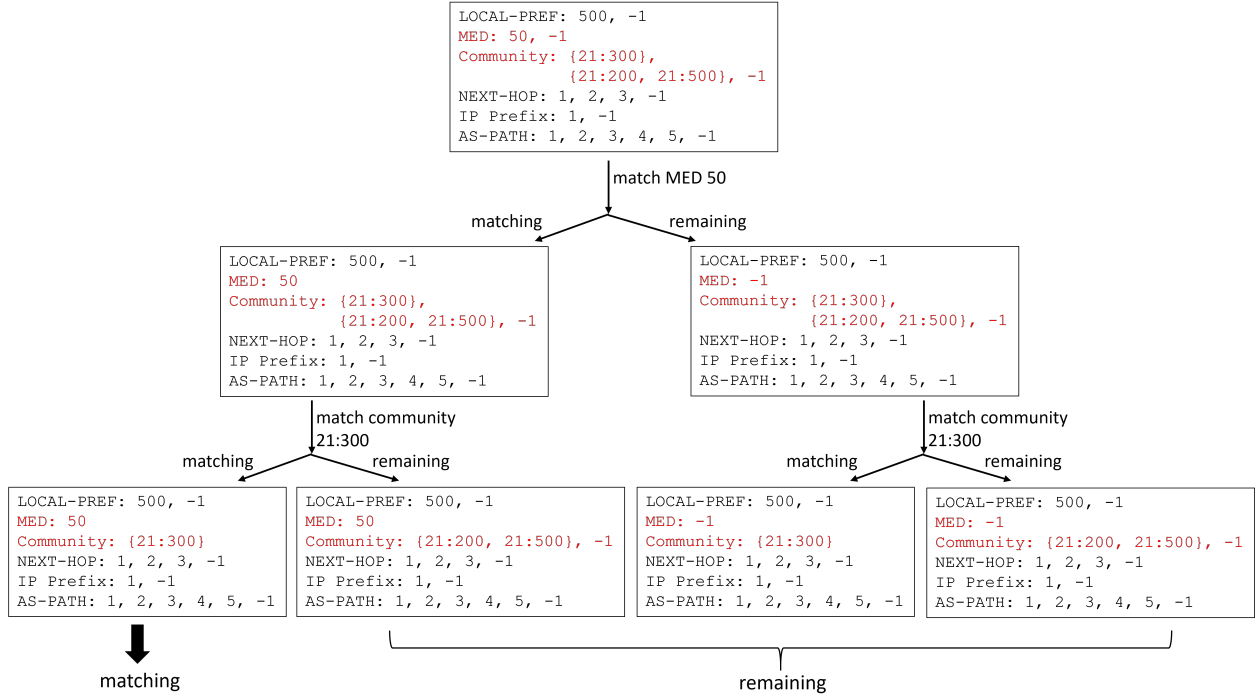


Figure 4.6: An example output of multiple matching algorithm. The input symbolic announcement is filtered by each *match* in order. For a route map item with n *match* statements, there are 1 matching announcement and $2^n - 1$ remaining announcements in total.

4.4 Verifier

When a symbolic announcement is filtered by each route map item, only the matching symbolic announcement will be propagated to other routers. The matching symbolic announcement contains **all** possible BGP announcements that will be propagated to the next router. Therefore, we can easily check many BGP control plane properties by inserting the symbolic announcement at each ingress point and observe how this symbolic announcement is propagated inside the network until it leaves the network.

Our geometric model supports two propagation architectures: iBGP full mesh and route reflection. As introduced in § 2.1.1, in the iBGP full mesh, a router will propagate the matching announcement received from an eBGP peer to all other internal routers. In the route reflection, a router only propagates the matching announcement to the route reflector it connects to. In both cases, an input BGP announcement from an eBGP neighbor may be propagated to any other eBGP neighbor.

The verifier provides APIs for different property queries. For example, it allows the user to check the reachability between any two eBGP neighbors. One advantage of our verifier over others mentioned in the related work is that in addition to returning a boolean answer to a property query and a single counter-example if the property does not hold, it outputs all possible announcements

that violate the property and their traces considering any link failure scenario (i.e., as long as the ingress and egress links are on).

In the rest of this section, we will present some main properties that our verifier is able to verify and the verification criteria it is based on.

4.4.1 No transit from A to B

The business relationship policy specifies that a peer or a provider should not reach another peer or a provider. To verify this property, the verifier allows the user to specify the two neighbors A and B to check if there is no transit from A to B. This is implemented by inserting the full symbolic announcement into the ingress point that connects to A, and checking whether any remaining symbolic announcement is exported from the egress point that connects to B. Because we assume each symbolic announcement is fully propagated inside the network via iBGP full mesh or route reflection, the export announcement considers all possibilities that would violate the policy. To output the original symbolic announcement that violates the policy, the verifier always maintains a copy of the original announcement when modifying the attributes according to the route map.

4.4.2 Prefer A over B

In the business relationship and outbound traffic control policies, the network usually needs to specify the preference order for different eBGP neighbors. A common practice is to set a higher LOCAL-PREF value for the more preferred neighbor. To check if the network prefers neighbor A over B inside the network, the verifier adds a fake egress point with no route map to the network. As shown in Figure 4.7, A and B are both connected to the network via R1 and R2. The fake egress point R_f has an iBGP session with both internal routers, since R_f does not set any route map, it will directly output the announcements that are originated from A and B and processed by R1 and R2. Therefore, the verifier can check the LOCAL-PREF values received from R1 and R2. If all LOCAL-PREF values from R1 are larger than those from R2, then the verifier can guarantee that the network prefers A over B, otherwise it is not clear.

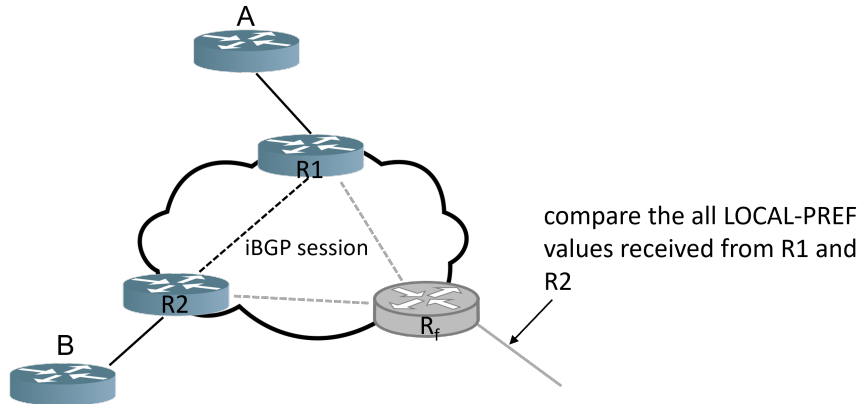


Figure 4.7: An example of the fake egress point implementation. The fake egress point R_f does not have any route map and has iBGP sessions with R1 and R2. It will directly output the announcements that are originated from A and B and processed by R1 and R2. The verifier can then check the fake egress point to compare the LOCAL-PREF values set for A and B.

4.4.3 All neighbors are tagged

Many BGP policies are implemented based on the assigning different groups for the eBGP neighbors. It is therefore important to check whether all external announcements are tagged with a specific community value. This can also be realized with the fake egress point R_f . Since the fake egress point R_f has an iBGP session with all other ingress points and has no route map. It will output the announcements processed by each ingress point as they are. Therefore, by checking the export announcements, the verifier can know if all eBGP neighbors are tagged and their specific community values.

4.4.4 Want X prefer R1 over R2

Another set of important BGP policies the verifier can verify are the network inbound traffic control policies. For example, the verifier can check if the network wants a neighbor to prefer ingress point R1 over R2. This is done by first comparing the AS prepending times in any output announcement exported from R1 and R2. If R1 prepends the AS number more times than R2 in all output announcements, then the verifier considers R1 to be more preferred. If there is no specific priority in terms of AS prepending between R1 and R2, the verifier then compares the MED values in their output announcements and considers the ingress point that has a lower MED in all output announcements as the ore preferred one.

4.5 Initial Network Reduction

So far, we have introduced the entire geometric model for the BGP control plane verification. The verification result considers the full announcement coverage and link failure coverage. There is one last piece left for our verifier to satisfy the full data plane coverage, which is the full network state coverage.

In a BGP announcement processing pipeline, each router will select the best route for each destination and only propagate the best routes to the next router. In different network states, each router may select and propagate the different best routes. Normally, it is necessary for a verifier to consider the verification result in different network states. The initial network reduction (INR) theory proved by Bagpipe [21] shows that it is enough to only consider the initial network state because if the best route will be propagated in any network state, it will also be propagated in the initial state where no former best route has been selected. We use the same idea here and therefore do not need to consider other network states.

Chapter 5

Evaluation

The evaluation of the geometric model is focused on two dimensions: efficiency and correctness. To evaluate the efficiency of our model, we calculate the time it takes for its major processes. To evaluate the correctness, we build a small case-study network and analyze the verifier behavior when we manipulate the network configuration.

5.1 Efficiency Test

We design three tests to measure the efficiency of the geometric model. In Test 1, we test the time required for the multiple matching. In Test 2, we test the time it takes for a symbolic announcement to go through different numbers of route map items. In Test 3, we calculate the time it takes to assign labels for a set of prefix lists and AS-PATH lists, respectively. Each evaluation is performed on a 4-core 8-thread CPU (clocked at 2.1GHz) with a 16G of memory.

5.1.1 Test 1: multiple matching test

In this test, we analyze the time consumption for filtering the symbolic announcement with a single route map item with different number of attributes. Since there are at most six attribute types that can appear in a route map item *match* statement (IP Prefix included), a route map item can have at most six *match* statements.

Test setup To construct a random route map item, we first build an attribute value pool for each attribute that contains enough arbitrary values. A very small subset of the attribute pools is displayed in Figure 5.1 (NEXT-HOP and IP Prefix share the same set of IP prefixes). We then randomly create 5 prefix lists, 5 AS-PATH lists and 15 community sets (i.e., {21:300} and {21:500, 21:200} are two community sets) by randomly picking from the attribute pool. Each prefix list consists of 2 prefix list items, the IP prefix in each list item is randomly picked from the IP prefix pool, and the parameters *ge*, *le*, *eq* are randomly chosen based on the prefix length. Each AS-PATH list also consists of 2 AS-PATH list items whose values are randomly picked from the AS-PATH value pool. Similarly, each community set consists of 1 or 2 community values from the community value pool. The list type *permit|deny* is also randomly picked for all list items.

We then add the specified number of attributes to the route map item. We randomly pick the attribute to be matched. If the community value is chosen, then a community list consisting of 2 community list items is created. The values in each list item are randomly chosen from the 15 community sets. If the NEXT-HOP or IP Prefix is picked up, then a prefix list is randomly selected

from the 5 prefix lists. Similarly, if the attribute AS-PATH is picked up, then a random AS-PATH list is selected from 5 AS-PATH lists. When the LOCAL-PREF or MED is picked up, then the value to be matched is randomly picked from its corresponding attribute pool.

```

LOCAL-PREF: 50, 100, 150, 200, 250 .....
MED: 10, 20, 30, 40, 50, 60 .....
Community: 10:10, 10:20, 10:30, 10:40, 10:50 .....
IP prefix: 0.0.0.0/0, 1.0.0.0/8, 1.1.0.0/16, 1.1.1.0/24 .....
AS-PATH regex: _(10|20)$, ^(10|20)_, _10_20_, ^[0-9]+$ .....

```

Figure 5.1: A small subset of each attribute value pool. When constructing a route map item or an attribute list, each attribute value is randomly picked from the pool. NEXT-HOP and IP Prefix share the same IP prefixes pool because they are matched by the prefix list.

After constructing the route map item, we create the symbolic announcement. For the attributes LOCAL-PREF and MED, the labels are all the values in their attribute pools, with -1 added to each attribute. For the attribute community, the labels are the 15 community sets and -1. For the attributes NEXT-HOP and IP Prefix, we calculate the equivalence class labels based on the 5 prefix lists constructed before (i.e., each prefix list is tagged with both NEXT-HOP labels and IP Prefix labels). For the attribute AS-PATH, we calculate the AS-PATH equivalence class labels based on the 5 AS-PATH lists constructed before.

Test procedure With all test setups prepared, we let the symbolic announcement go through the route map item of different numbers of attributes and start the evaluation. For the evaluation, we calculate the time for computing the equivalence classes before the announcement filtering and for the route map item filtering separately. For each possible number of attributes in a route map item, we repeat the entire procedure from creating random attribute lists to filtering the symbolic unchanged). Figure 5.2 shows the average result of Test 1.

Test result As can be found in Figure 5.2, the green and red lines stand for the time for computing equivalence classes and for running a route map item filtering, respectively. Since the number of prefix lists and AS-PATH lists are always fixed for the equivalence class computation (although the list items are different every time), the green line does not have significant fluctuation. The time used for filtering the symbolic announcement slowly increases with the number of attributes. This is reasonable because according to the multiple matching algorithm Alg 8, more filtering steps are required when the number of attributes in the route map items grows.

Compared to the time used for running a symbolic announcement analysis, it takes much more time to compute the equivalence classes. However, since we only need to compute them in the compile-time once before filtering any symbolic announcement, this time will not affect the run-time analysis.

5.1.2 Test 2: multiple route map items test

In this test, we analyze the time consumption for filtering a symbolic announcement by different numbers of route map items.

Test setup The test setup is similar to Test 1, except for two points. First, instead of only constructing a single route map item, we construct different numbers of route map items. Second,

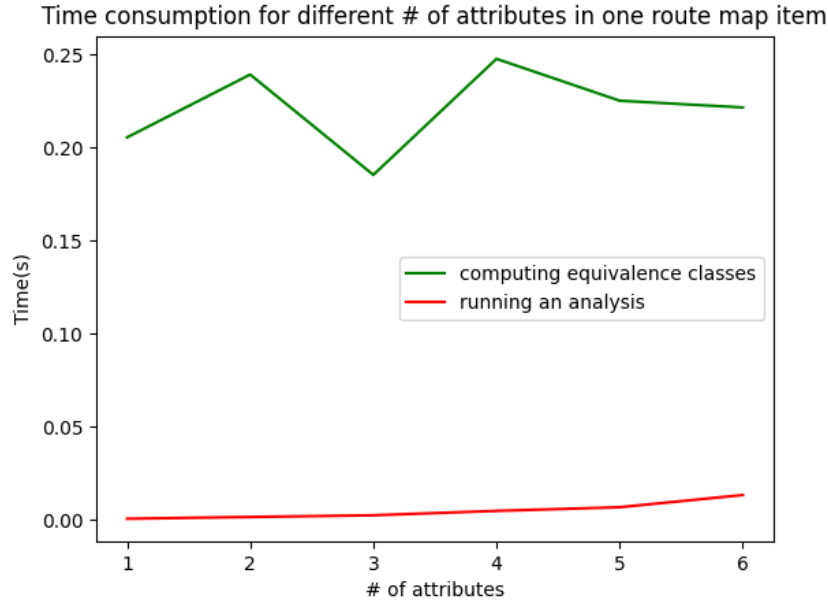


Figure 5.2: The result of the multiple matching test. The green line that standing for the time for equivalence classes computation is relatively stable across different numbers of attributes to be matched. The red line standing for the announcement filtering time slowly increases with the number of attributes.

we fix the number of attributes in each route map item to be 2, the attribute itself is still randomly selected as in Test 1.

Test procedure The test procedure is also similar to Test 1. We first calculate the time for equivalence class computation, then the time for the announcement filtering. For each number of route map items in a route map, we repeat the entire procedure 20 times. The average result is shown in Figure 5.3.

Test result Similar to Figure 5.2, the green line in Figure 5.3 is also relatively stable due to the fixed number of attribute lists in the equivalence class computation. One significant difference between Figure 5.2 and Figure 5.3 is that the red line grows faster in Figure 5.3. This is also predictable because when there are multiple route map items, all the remaining announcements of the last item need to go through the next item. According to Alg 8, the number of remaining announcements increases exponentially during the multiple matching. In a real configuration, the number of route map items in each route map is usually limited, therefore the time consumption is also under control.

5.1.3 Test 3: equivalence class computation test

In this test, we analyzes the efficiency of our equivalence class computation algorithms for the prefix list and the AS-PATH list.

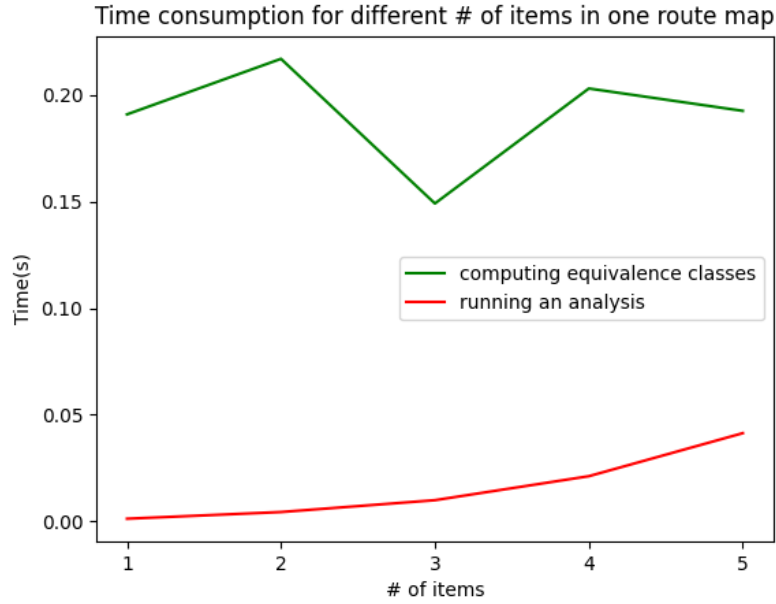


Figure 5.3: The result of the multiple route map items test. The green line is still relatively stable across different numbers of route map items. Compared to Figure 5.2, the red line grows faster with the number of route map items in a route map.

Test setup The construction of either list is the same as in the previous tests and each list consists of 2 list items. Unlike previous tests, we only need to construct specific numbers of lists for each list type and do not need to create the route map or the symbolic announcement.

Test procedure For each list type, we evaluate the time it takes to compute the equivalence classes for different numbers of lists. When we compute the time consumption for one list type, we set the number of another list type to be 0. The test result is shown in Figure 5.4. Each computation result is also an average of 20 repetitions.

Test result The test result shows that, there is a significant difference between the algorithm time complexity in computing equivalence classes for the prefix list and the AS-PATH list. As shown in Figure 5.4, when the number of AS-PATH lists increases, the time consumption grows exponentially. Compared to the time spent on the AS-PATH list, the time used for the prefix list is negligible. The most important reason for this difference is that the algorithm for the AS-PATH partition is much more complicate (see Alg 4 and Figure 3.17) especially when the regex tree grows deeply.

5.2 Case Study

In this section, we show how the geometric model behaves in a small case-study network. The network topology and the route maps are given in Figure 5.5. The topology consists of four internal routers R1-R4 forming an iBGP full mesh and four external routers that have different business relationships with the central network. The route map configuration for each internal router is

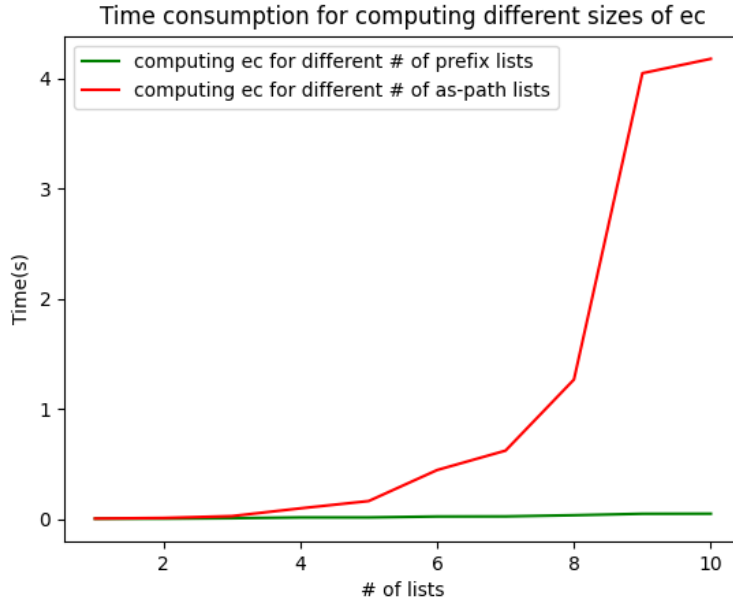


Figure 5.4: The result of the equivalence class computation test. The time consumption for AS-PATH equivalence class computation grows exponentially when the number of AS-PATH lists increases. Compared to the AS-PATH list, the time consumption for computing prefix list partition is negligible.

included alongside. According to Figure 5.5, several desired BGP policies can be inferred from the route maps:

1. prefer CUST1 > CUST2 > PEER > PROV
2. no transit between neighbor PEER and PROV
3. want AS100 to prefer R1 over R4
4. announcements from CUST1 and CUST2 are tagged with the community value 21:1000, announcements from PEER are tagged with 21:100 and announcements from PROV are tagged with 21:10

We then introduce three debugging options in the network. As highlighted in blue in Figure 5.5, when we turn off the option *community on all routes*, the statement "set Community 21:10" in R2's route map INMAP-PROV will be missing. When we turn off the option *no transit*, the statement "match Community 21:1000" will be missing. When the option "business relationship" is off, the highlighted statement "set LOCAL-PREF 100" will be changed to "set LOCAL-PREF 10". By introducing these debugging options, we simulate the situation where some BGP properties do not hold due to configuration errors. We then analyze the verifier behavior when different options are set.

We first look at the case where all BGP properties hold (i.e., all options are on). When we query the verifier about the policies listed above, the verifier returns the answer as in Figure 5.6. It is obvious that the verifier gives the correct answer to all queries.

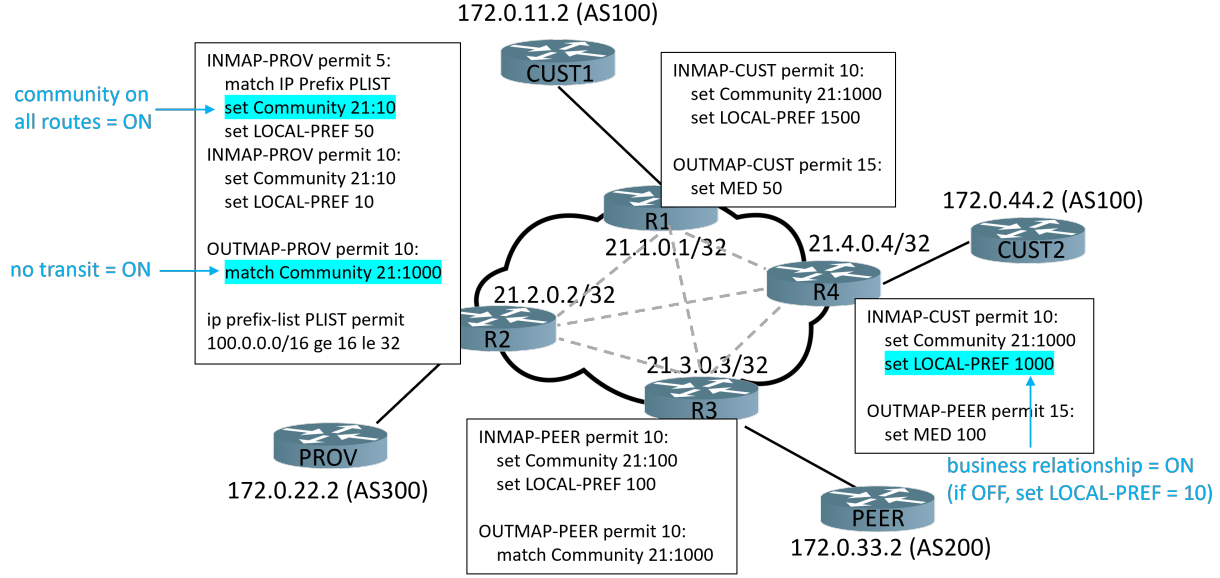


Figure 5.5: The topology of the case-study network. The topology consists of four internal routers forming an iBGP full mesh and four external neighbors that have different business relationships with AS21. Three debugging options are provided as highlighted in blue. When any option is turn off, the declared property beside it no longer holds.

We now turn off the option *business relationship*. In this case, the LOCAL-PREF for CUST2 will be incorrectly set to 10 instead of 1000 and the business relationship policy is violated. The answer given by the verifier has also changed accordingly as in Figure 5.7.

Figure 5.8 shows the results when we turn off the option *no transit*. As output by the verifier, since no *match* is specified in R1's route map INMAP-PEER, any announcements from PEER will be received by PROV. There are two sets of attribute values displayed in Figure 5.8. One is *original*, which shows the original attribute values received from PROV. The other is *export*, when it appears in the output symbolic announcement, it means the attribute value has been modified after it is processed by the network and propagated to another external neighbor, otherwise the original value is propagated. The output announcement also lists the announcement trace information, which is included in the fields "via routers" and "via route map items".

At last, we turn off the option "community on all routes". The verifier's answer is shown in Figure 5.9. It shows that all announcements from PROV with the IP Prefix label 2 (which stands for *100.0.0.0/16 ge 16 le 32*) are not tagged with a specific community value. The fake IP address *###.###.###.###* for the fake egress point appears in the next hop because the community values are examined via the fake egress point (see § 4.4). *fake_i* and *fake_n* are two fake routers at the fake egress point.

```

SETUP: business relationship = True, no transit = True, community on all routes = True

Is there any transit from PROV to PEER?
    No transit from PROV to PEER

Is there any transit from PEER to PROV?
    No transit from PEER to PROV

What's the preference the network wants AS100 to have between R1 and R4?
    The network wants neighbor AS 100 to prefer R1 over R4

What's the internal preference order in the network?
    The network prefers: PROV < PEER < CUST2 < CUST1

What are all the communities the network has tagged on each neighbor?
    Each peer has the following tagged Communities:
        CUST1:[{'21:1000'}]
        PROV:[{'21:10'}]
        PEER:[{'21:100'}]
        CUST2:[{'21:1000'}]

    All peers have a specific Communities

```

Figure 5.6: The verifier's answer when all BGP properties hold. Obviously, the verifier answers correctly to all property queries.

```

SETUP: business relationship = False, no transit = True, community on all routes = True

What's the internal preference order in the network?
    The network prefers: CUST2 < PROV < PEER < CUST1

```

Figure 5.7: The verifier's answer when the policy *business relationship* is violated. The CUST2 now has the lowest preference among all external neighbors due to an incorrect LOCAL-PREF setting.

```

SETUP: business relationship = True, no transit = False, community on all routes = True

Is there any transit from PEER to PROV?
  The following announcements will be received by PROV from PEER:
    (1): RouteAnnouncement:
      Local Preference(original): {-1}
      Local Preference(export): {100}
      Multi Exit Disc(original): {-1}
      Communities(original): [{'21:1000'}, {-1}]
      Communities(export): [{'21:100'}]
      IP Prefix(labels, original): {2, -1}
      Next Hop(labels, original): {-1}
      AS-PATH(labels): {-1}
      AS prepend: None
      set next hop: 21.2.0.1/32
      via routers: PEER -> R3 -> R2 -> PROV
      via route map items:INMAP-PEER(seq10) -> OUTMAP-PROV(seq10)

```

Figure 5.8: The verifier’s answer when the policy *no transit* is violated. All announcements from PEER can now reach PROV. The two sets of attributes: *original* and *export* specify the attribute value when it is received and when it is propagated to another external neighbor, respectively.

```

SETUP: business relationship = True, no transit = True, community on all routes = False

What are all the communities the network has tagged on each neighbor?
  The following announcement received neighbor PROV doesn't have a specific community:
    RouteAnnouncement:
      Local Preference(original): {-1}
      Local Preference(export): {50}
      Multi Exit Disc(original): {-1}
      Communities(original): [{'21:1000'}, {-1}]
      IP Prefix(labels, original): {2}
      Next Hop(labels, original): {-1}
      AS-PATH(labels): {-1}
      AS prepend: None
      set next hop: #.#.#.#
      via routers: PROV -> R2 -> fake_i -> fake_n
      via route map items:INMAP-PROV(seq5)

```

Figure 5.9: The verifier’s answer when the policy *community on all routes* is violated. The verifier shows that all announcements from PROV with the IP Prefix label 2 are not tagged with a specific community value. *fake_i* and *fake_n* are two fake routers at the fake egress point and *#.#.#.#* is the fake IP for the fake egress point.

Chapter 6

Discussion

So far, we have covered the essential technical details of the geometric model for the BGP control plane verification. Compared with the related work, our model can verify the control plane properties more efficiently. However, there are still limitations of the model due to the complication of simulating the BGP control plane behavior. In this chapter, we list some possible optimization we consider for the model as well as some next steps we could look forward to.

6.1 Limitations

When building the geometric model, we mostly focus on the functional integrity and therefore leave much room for further optimization. We see the improvement possibilities in the following perspectives:

Model extension Although the geometric model has modeled the filtering for all route attributes and allows propagation for different architectures (i.e., iBGP full mesh, route reflection), we make some restrictions when applying the model. For example, we do not model the BGP decision process and can only inject routes from one ingress point at a time. In this case, we need to carefully reason about the limitations which might be introduced due to the omission of the interaction between different routes.

AS-PATH partitioner Among all attribute partition algorithm designs, AS-PATH is the most challenging one because we need to deal with conflict resolution when inserting a new regex into the regex tree. Our AS-PATH partition algorithm now recognizes the most preliminary regexes (see Table 3.1), for other more complicated regexes, we can first disassemble it into a series of preliminary regexes. During our evaluation, we find that the time complexity of computing AS-PATH equivalence classes is much more significant than other attributes. In another informal test, we test the AS-PATH partitioner performance when importing hundreds of regexes, and the test result indicates that the algorithm also uses a lot of memory to process the recursion. In summary, the performance of the AS-PATH partitioner is the bottleneck of the performance of the entire geometric model. We already see some possible directions for the optimization and we expect to apply it in the next step.

Community overlap For all the other attributes, we pay much attention to make sure there is no overlap between different equivalence classes. However, we simplify this process in the community. We realize that this could lead to false positives in some verification results. In all the

test performed, we have not observed any false positives. However, theoretically, it could happen: for example, given two equivalence classes: $\{21:300\}$ and $\{21:500\}$. When a route map filters out $\{21:300\}$, the class $\{21:500\}$ is still in the announcement. This means any real community value that contains both 21:300 and 21:500 may still be matched later. To handle this kind of false positive, we think about using a similar but much easier data structure as the AS-PATH regex tree.

Evaluation In the evaluation, we test the model performance with different sizes of route map items and different numbers of attribute lists. However, we leave out the behavior when multiple route maps are applied during the filtering. One reason is that the performance of multiple route maps is highly dependent on the route map contents. Unlike in the multiple route map items test where each route map item filters the remaining announcements of the last route map item, only the matching announcement flows down to the next route map. Since our evaluation is carried out in a fully random way, we estimate that the result of the multiple route maps test can have a large variance for different random setups.

Parser The configuration parser is now able to parse the entire network topology and all the attributes that our model supports. We expect to improve the parser capability by supporting more settings in a real large-scale network configuration, such as neighbor groups and well-known communities.

6.2 Outlook

Apart from the possible optimizations we have listed above, we also look forward to the transition of our focus from the BGP verification to the specification mining. We have already covered some of them in the current verifier such as the reachability map, the internal preference order and the community tagging. We expect to support more BGP specifications that are applied in the real world.

Chapter 7

Summary

In this project, we build a full geometric model for the BGP control plane. With our geometric model, we can validate common control plane policies such as transit, route preferences and consistent tagging, which are difficult or impossible to verify for prior verification work that partly relies on data plane analysis. A big advantage of our geometric model is that we are able to inject symbolic routes and therefore not only get a single counter-example to prove that a policy does not hold, but the entire set of announcements that lead to the policy violation.

With the equivalence class insight from ddNF [4], we amortize the time overhead for the run-time verification since we can reuse the equivalence classes for all analyzes over the same network. Based on our current achievements on the control plane verification, we look forward to the transition to the BGP specification mining in future work.

Bibliography

- [1] Fsm/regex conversion library. <https://github.com/qntm/greenery.git>, 2021.
- [2] BATES, T., CHANDRA, R., AND CHEN, E. Bgp route reflection-an alternative to full mesh ibgp.
- [3] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. A general approach to network configuration verification. In *ACM SIGCOMM* (Los Angeles, CA, USA, 2017).
- [4] BJØRNER, N., JUNIWAL, G., MAHAJAN, R., SESHIA, S. A., AND VARGHESE, G. ddnf: An efficient data structure for header spaces. In *HVC* (Haifa, Isreal, 2016).
- [5] CAESAR, M., AND REXFORD, J. Bgp routing policies in isp networks. *IEEE network* 19, 6 (2005), 5–11.
- [6] CHANDRA, R., TRAINA, P., AND LI, T. Bgp communities attribute. Tech. rep., RFC 1997, August, 1996.
- [7] CHIRGWIN, R. Google routing blunder sent japan’s internet dark on friday. https://www.theregister.com/2017/08/27/google_routing_blunder_sent_japans_internet_dark/. Accessed: 20-21-05-09.
- [8] DE MOURA, L., AND BJØRNER, N. Satisfiability modulo theories: introduction and applications. *Communications of the ACM* 54, 9 (2011), 69–77.
- [9] FOGEL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. A general approach to network configuration analysis. In *USENIX NSDI* (OAKLAND, CA, USA, 2015).
- [10] GODFREY, P. B. Network verification: from algorithms to deployment. <http://pbg.cs.illinois.edu/outreach/2017.11.03-NetworkVerificationTutorial-ASE.pdf>. Accessed: 20-21-05-13.
- [11] HOLZMANN, G. J. The model checker spin. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.
- [12] HORN, A., KHERADMAND, A., AND PRASAD, M. Delta-net: Real-time network verification using atoms. In *USENIX NSDI* (BOSTON, MA, USA, 2017).
- [13] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *USENIX NSDI* (LOMBARD, IL, USA, 2013).

- [14] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *USENIX NSDI* (SAN JOSE, CA, USA, 2012).
- [15] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. Veriflow: Verifying network-wide invariants in real time. In *USENIX NSDI* (LOMBARD, IL, USA, 2013).
- [16] KIM, H., REICH, J., GUPTA, A., SHAHBAB, M., FEAMSTER, N., AND CLARK, R. Kinetic: Verifiable dynamic network control. In *USENIX NSDI* (OAKLAND, CA, USA, 2015).
- [17] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 290–301.
- [18] PRABHU, S., CHOU, K. Y., KHERADMAND, A., GODFREY, B., AND CAESAR, M. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI* (SANTA CLARA, CA, USA, 2020).
- [19] REKHTER, Y., LI, T., HARES, S., ET AL. A border gateway protocol 4 (bgp-4), 1994.
- [20] SIDDIQUI, A. A major bgp route leak by as55410. <https://blog.apnic.net/2021/04/26/a-major-bgp-route-leak-by-as55410/>. Accessed: 20-21-05-09.
- [21] WEITZ, K., WOOS, D., TORLAK, E., ERNST, M. D., KRISHNAMURTHY, A., AND TATLOCK, Z. Scalable verification of border gateway protocol configurations with an smt solver. In *ACM SIGPLAN* (Amsterdam, Netherlands, 2016).
- [22] XIE, G. G., ZHAN, J., MALTZ, D. A., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On static reachability analysis of ip networks. In *IEEE Infocom* (Miami, FL, USA, 2005).
- [23] YANG, H., AND LAM, S. S. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking* 24, 2 (2015), 887–900.