Reasoning on BGP Routing Filters Using Tree Automata*

Caroline Battaglia Véronique Bruyère† Olivier Gauwin‡
Bruno Quoitin§

June 2012

Abstract

The Border Gateway Protocol (BGP) is the protocol used to distribute Internet routes between different organizations. BGP routing policies are very important because they enable organizations to enforce their business relationships by controlling route redistribution and route selection. In this paper, we investigate the semantic of BGP policies. We aim to determine whether two policies are equivalent, that is, if given the same set of incoming routes, they will generate the same set of outgoing routes. We show how this problem can be solved using the tree automata theory and describe several optimizations. We also propose a prototype implementing this approach. The experimental results are very promising. They show the efficiency of our approach and the interest of using the tree automata theory in the context of BGP routing policies.

1 Introduction

The *Border Gateway Protocol* (BGP) [12] is the protocol used to distribute Internet routes between different organizations, also called *Autonomous Systems* (AS). In BGP, routing policies are very important because they enable ASs to enforce their business relationships by controlling route redistribution and route selection. This in turns influences how the traffic flows in the Internet. ASs are motivated to control traffic flow as carrying traffic internally is costly and they are billed differently by the different neighboring ASs, with whom they have a business relationship, for sending traffic

*This work was partially funded by FNRS through the European project Gasics
†University of Mons - UMONS
‡Univ. Bordeaux, CNRS, LaBRI, UMR 5800
§University of Mons - UMONS

through them. This billing often relies on the amount of traffic sent to the neighboring AS. For example, an organization A can buy transit service from an Internet provider. In addition, it may connect to another organization B for the sole purpose of exchanging information destined to that organization. BGP policies enable organization A to prevent traffic between the Internet provider and its peering organization B to transit through its network. Network operators may wish to implement a wide variety of policies ranging from limiting the advertisement of some prefixes, to preferring sending traffic to some cheaper neighboring ASs, to influencing the route selection in distant ASs, and to stop a DDoS attack, to name a few.

The configuration of BGP policies is complex and often source of errors [11, 7] The implementation of a single policy is distributed among multiple routers, each configured differently. Usually, some action takes place at the entrance of the AS and a different set of actions takes place at the exit of the AS. Due to this distribution, it is not easy to build a high level view of the BGP policies solely based on the router configuration files. Furthermore, the configuration languages provided by the router vendors are very low level. Each vendor provides a different syntax. Translation from one language to another is complex as the expressiveness of the languages varies greatly. Even using a single language, it is possible to implement a single high level policy in multiple ways.

In this paper, we investigate the semantic of BGP configuration policies. We aim to determine if two BGP configurations have the same semantic. That is, if given the same set of incoming routes, the two policies will generate the same set of outgoing routes. The solution to this problem is important as it is the first step to being able to detect configuration mistakes before committing a configuration change and thus prevent unnecessary traffic disruptions. It enables to push much further the work started by Feamster et al [7] and by Le et al [10]. Identifying if two policies have the same effect enables network operators to check the correctness of router configurations with regard to the high level policies they aim to enforce. We could imagine that operators use a high level policy specification language to express their policies and compare these specifications to the implemented policies. Additionally, such a solution is useful for refactoring old BGP policy configurations upon a change of network equipment, the acquisition of another network, a configuration clean up or the development/deployment of a configuration tool.

The first idea that comes in mind to test if two routing policies have the same semantic is the following one: to enumerate all the possible routes (up to a certain size) and to test if the two given policies generate the same output routes. In this paper we propose to rely on *tree automata* theory [6], a powerful mathematical tool well-known for its applications in XML processing [9]. We model routes as trees, and routing policies as tree automata. We use the tree automata theory to decide whether two routing policies have the same semantics, that is, are equivalent total functions. Therefore contrarily to the previous algorithm which works at the level of routes and tests route after route for the equivalence of policies, we test for equivalence directly at the level of the policies.

Paper organization – The paper is organized as follows. In Section 2, we briefly describe how the BGP routing protocol works and how it implements routing policies with *routing filters*. We then formally define the semantics of routing filters, as total functions operating on routes.

In Section 3, we explain how to model a route as a tree. We recall the notion of tree automaton, present some of their useful properties, and illustrate with some pedagogical examples. We then show progressively how routing filters can be modeled as tree automata. We start with filter predicates used in routing filters to test if a filter can be apply to a route. Such predicates can easily and naturally be modeled by tree automata.

In Section 4, we focus on filter actions. An action is used in a filter to generate a modified output route from a given input route. We show that filter actions can also be modeled with tree automata. To this end, we show that an action can be seen as a binary tree relation and how to model this relation as a tree automaton. This model is again easy and natural. We also show that a routing filter can be modeled as a tree automaton, and that the equivalence of two filters reduces to the equivalence of their related tree automata. Testing the equivalence of two tree automata is a classical operation in tree automata theory.

In Section 5, we propose a prototype implementing this approach and show some experiments. This tool benefits from several optimizations described in Section 6. Especially, we propose a new algorithm for deciding the equivalence of two tree automata encoding routing filters, in linear time. This test is in exponential time on arbitrary tree automata. Our algorithm takes into account that routing filters are total functions. A second optimization avoids another exponential blow-up at the level of predicates used in filters. In a third optimization, we also show how the range of values (ASs, communities...) to be considered by our algorithm can be seriously reduced. The experimental results are very promising. They show the efficiency of our approach and the interest of using the tree automata theory in the context of routing filters.

2 Background

This section briefly describes how the Internet routing architecture is organized. Section 2.1 introduces the *Border Gateway Protocol* (BGP). Section 2.2 defines the structure of BGP routes, the supported path attributes as well as how BGP routes are selected. Section 2.4 defines routing filters.

2.1 The Border Gateway Protocol (BGP)

The Internet is an interconnection of several independent networks called *Autonomous Systems* (AS). An AS is a network under the administration of a single institution. Examples of such institutions are a commercial company, an Internet Service Provider (ISP), a university. Each AS is identified by a unique *AS number* (`ASN`)[1] allocated by a regional Internet Routing Registry[2].

Routing in the Internet is organized in a two level hierarchy. Within each AS, the network operators are free to use the intradomain routing protocol they want. Usually, link-state routing protocols such as IS-IS or OSPF are used. Those protocols allow each router in the AS to build the complete graph of the AS's internal network. Each router can then use a shortest-path computation algorithm to select which path(s) must be used to reach any destination within the AS.

On the opposite, to exchange routing information between multiple ASs and to compute routes that span multiple ASs, a single standard protocol is used. This protocol is the *Border Gateway Protocol* (BGP) [12]. BGP belongs to the family of path-vector protocols, that is protocols that advertise the reachability of a destination along with the path traversed by the routing information. In BGP, a router has one or several neighbors with which it will exchange routing information. Two neighbor routers exchange routing information over a *BGP session*. Routing information is sent over a BGP session in an incremental manner, meaning that a router only sends routing information when this information has changed.

There are two flavours of BGP sessions: *external sessions* used between routers in different ASs (eBGP) and *internal sessions* used between routers within the same AS (iBGP). The rules used to redistribute routing information over eBGP and iBGP sessions are different. A router that receives routing information from an iBGP session cannot redistribute it over an iBGP session.

Fig. 1 illustrates the interconnection of 3 ASs with BGP. AS1 is composed of routers R1, R2 and R3; AS2 of routers R4, R5 and R6 and AS3 is composed of routers R7 and R8. There are two eBGP sessions between AS1 and AS2: one between routers R2 and R4 and another one between routers R3 and

[1] According to the CIDR report (`http://www.cidr-report.org`), there were more than 65000 `ASN`s allocated in the Internet at the time of this writing, 35000 of which were visible in BGP tables.

[2] ARIN (`http://www.arin.net`) and RIPE (`http://www.ripe.net`) are examples of Internet Routing Registries for North America and Europe respectively.

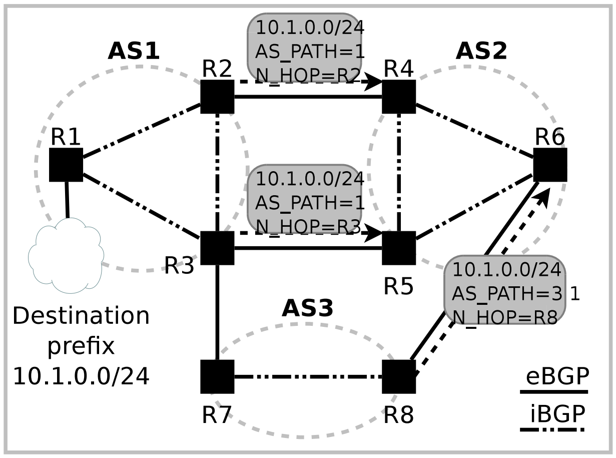


Figure 1: Example of an interconnection of 3 AS with BGP sessions.

R5. There is a single eBGP session between AS1 and AS3 and between AS2 and AS3. Within each AS, each router is connected by iBGP sessions to all the other routers.

2.2 Routing Information

The basic unit of routing information in BGP is a *route*. A route announces the reachability of a remote destination. Although BGP can be used to advertise the reachability of several kinds of address families [2], in this paper we focus on IP addresses. For these address families, destinations are announced in the prefix form. An IP prefix, expressed as a couple (address, mask length) represents a set of contiguous IP addresses that share a common prefix. An example is the IPv4 prefix 192.168.128.0/17 which represents the set of IPv4 addresses between 192.168.128.0 and 192.168.255.255 (32768 different addresses). In a route, we will call `DST_PREFIX` the *attribute* that contains the destination prefix.

A BGP route associates a destination prefix `DST_PREFIX` with several *path attributes*. The list of the most important path attributes is provided in the next paragraphs.

- `AS_PATH`: is a sequence of AS numbers (`ASN`s). The `AS_PATH` contains the `ASN`s of the ASs traversed by the route ordered from the closest to the nearest. When a router sends a route to another AS's router, it adds its own `ASN` at the beginning of the route's `AS_PATH`. An `ASN` can appear multiple times in the `AS_PATH`. The `AS_PATH` attribute is used to rank routes as well as to test against routing loops. Each ASN is represented as a 16 bits or 32 bits unsigned integer

- LOCAL_PREF: is an unsigned integer (32-bits). The LOCAL_PREF is used to give a route a preference that has meaning local to the AS. The LOCAL_PREF attribute is not propagated across eBGP sessions. The LOCAL_PREF has a default value in every network, 100 being the most frequently used.

- ORIGIN: indicates how the route was first learned by BGP. This attribute can only take three different values: IGP if the route was statically configured on the origin router, EGP if the route was learned from another external routing protocol and INCOMPLETE if the route was learned from any other mean (typically redistributed into BGP from another routing protocol).

- NEXT_HOP: is an IP address. The NEXT_HOP identifies the router to which packets must be sent in order to reach the route's destination.

- MULTI_EXIT_DISC: is a 32-bits unsigned integer. The MULTI_EXIT_DISC is used by a neighbor AS to tell which route should be preferred.

- COMMUNITIES [5]: is a set of 32-bits unsigned integer values (called communities). Each community value has a semantic that is usually local to an AS or to an AS and its direct neighbors. All routes that carry the same set of community values will typically undergo the same treatment. Some community values are defined with a global semantic by the standard.

To illustrate some of the above path attributes, let us take the example of Figure 1. There are three different routes shown: one sent from R2 to R4, one sent from R3 to R5 and one sent from R8 to R6. All these routes advertise the reachability of prefix 10.1.0.0/24 originated in AS1 and as such contain this prefix as DST_PREFIX. Routes from R2 and R3 have traversed a single AS (AS1). For this reason their AS_PATH contains a single ASN (1). The route from R8 has traversed AS1, then AS3, and therefore its AS_PATH contains ASN 3 followed by ASN 1.

In addition to the above path attributes a route can contain attributes specific to route aggregation. A router can be configured to aggregate multiple routes towards contiguous prefixes into a single route towards a single prefix. This aggregation reduces the burden on the control plane. The aggregation specific attributes are ATOMIC_AGGREGATE, AGGREGATOR and the ability to put sets of ASNs in the AS_PATH in addition to sequences[3]. We ignore the aggregation mechanism in this paper although the techniques that we will use to

[3]The AS_PATH attribute can contain different path segments: some of them are AS_SEQUENCE and other are AS_SET. RFC 4274 mentions that AS_SET are rarely used in the Internet

model routing filters later in this paper can easily be extended to support the attributes related to aggregation.

To handle the redistribution of BGP routes within a domain, routes can also contain additional attributes. Two techniques are standardized: route reflection [1] and BGP confederations [13]. Each technique requires specific attributes. Route-reflection introduces `ORIGINATOR_ID` and `CLUSTER_LIST` while BGP confederations introduce two new `AS_PATH` segments, namely `AS_CONFED_SEQUENCE` and `AS_CONFED_SET`. Attributes related to route-reflection should not be affected by routing filters. Attributes related to confederations could be affected by routing filters, but their structure and this handling is similar to the above path attributes and the model presented later will be able to handle them at low cost.

2.3 Routing Decisions

This section explains the behavior of a router when it has received multiple routes towards the same destination. In such a situation, a router will pick a single route named its *best route*. It will use that route to forward packets towards the advertised destination. Moreover, the router will not advertise all the received routes to its neighbors but only its best route.

To select its best route towards a given destination, a BGP router relies on a *decision process* that compares available routes together using a sequence of rules. The role of each rule is to keep only the routes that are best according to a given criterion, typically based on one route attribute. The decision process starts by applying the first rule on the set of received routes. If the resulting set contains more than one route, the decision process applies the next criterion on this set. This process is repeated until a single route remains. The last rule is a tie-break that ensures a single route is selected if the other rules where not enough to pinpoint a single route.

Table 1 lists the main rules of the BGP decision process. The first rule keeps only the routes with the highest value of the `LOCAL_PREF` attribute; the second criterion, only the routes with the shortest `AS_PATH`; the third criterion, only the routes with the lowest `ORIGIN` (IGP $<$ EGP $<$ INCOMPLETE); the fourth criterion only routes learned through eBGP sessions if any, the routes learned with iBGP otherwise; and the fifth rule the routes with the lowest value of the `MULTI_EXIT_DISC`.

To illustrate the operation of the decision process, let's again rely on the example of Figure 1. Router R6 will receive three different routes. From R4 and R5, R6 receives an iBGP route with an `AS_PATH` equal to (1). From R8, it receives an eBGP route with an `AS_PATH` equal to $(3, 1)$. R6 applies its decision process. All routes have an equal (default) `LOCAL_PREF`, as a result, the 1st rule has no effect. The second rule is applied. The route

1	Highest `LOCAL_PREF`
2	Shortest `AS_PATH`
3	Lowest `ORIGIN`
4	eBGP routes > iBGP routes
5	Lowest `MULTI_EXIT_DISC`
6	Tie-breaking rule(s)

Table 1: The main BGP decision process rules.

received from R8 has a longer `AS_PATH` than the routes received from R4 and R5, therefore, the route from R8 is discarded. There are still two routes available. Rules 3 to 5 make no decision as the two available routes are considered equal according to their criteria. The decision process then relies on the tie-breaking rule to pick a single route.

2.4 Routing Filters

An essential feature of the BGP protocol is the ability for any router to filter routes received from or sent to neighbors. To filter a route has two different meanings: it can mean to reject the route or to accept it after its attributes have eventually been modified. Filtering routes has several applications [3] from enforcing routing policies (rejecting routes that do not agree with business relationships among domains) to traffic engineering (influence how BGP selects the best route towards a specific destination by changing the route's attributes).

Routing filters in BGP are defined on every single router on a per-session basis. That means that a router can act differently on a route towards the same destination but received from or sent to different neighbors. Routing filters are usually defined by the network operator using the equipment's configuration language. This language is vendor specific; the BGP specification [12] does not specify routing filters. The two most known configuration languages are used on the routing platforms from Cisco Systems and Juniper Networks, but other vendors provide their own language as well.

Generally speaking, a *routing filter* can be described using the following formalism. A routing filter F is composed of a sequence of n *rules* $(R_1, \ldots, R_n)$. Each rule $R = \langle P, A \rangle$ is composed of two parts: a *predicate* P and an *action* A. The predicate determines if the rule applies to a route or not. A predicate is a Boolean combination of *atomic predicates* that each tests a single attribute of the route. The action is a sequence of *atomic actions* that each modifies a single attribute of the route. The action is applied to the route when the predicate matches the route.

Name	Predicate	Description
Community membership	$comm\_in(x)$	True iff the community value x belongs to the COMMUNITIES attribute.
Path membership	$path\_in(x)$	True iff the ASN x belongs to the AS_PATH attribute.
Path origin	$path\_ori(x)$	True iff the ASN x appears at the last position in the AS_PATH. The last ASN in the AS_PATH identifies the AS which originated the route.
Path neighbor	$path\_nei(x)$	True iff the ASN x appears at the first position in the AS_PATH. The first ASN in the AS_PATH identifies the neighbor AS from which the route was received.
Path subsequence	$path\_sub(s)$	True iff the sequence of ASNs, s, is included as is in the AS_PATH attribute.
Next-hop equality	$nh\_is(x)$	True iff the NEXT_HOP equals the IP address x.
Next-hop inclusion	$nh\_in(x)$	True iff the NEXT_HOP is included in the IP prefix x.
Destination equality	$dst\_is(x)$	True iff the route's destination prefix, DST_PREFIX, is equal to the IP prefix x.
Destination inclusion	$dst\_in(x)$	True iff the route's destination prefix, DST_PREFIX, is included into the IP prefix x.

Table 2: List of the most common atomic predicates.

Atomic predicates can test the following path attributes: COMMUNITIES, AS_PATH, NEXT_HOP and the route's destination, DST_PREFIX. Table 2 shows the most common atomic predicates. Note that configuration languages allow the expression of more complex predicates such as regular expressions on AS_PATH or the definition of sets of community values using regular expressions. Most of the time, these predicates can be considered as syntactic sugars for more complex combinations of the above atomic predicates.

Atomic actions can modify the following path attributes: COMMUNITIES, AS_PATH, NEXT_HOP, LOCAL_PREF and MULTI_EXIT_DISC. Special actions can be applied to the route to end its treatment before all the filter rules have been applied. The *accept*() action immediately accepts the route while the *reject*() action immediately rejects the route. Table 3 shows the most common atomic actions.

Algorithm 1 summarizes how a filter is applied to a route. The algorithm

Name	Action	Description
Absolute preference	*pref_set*(x)	Set the LOCAL_PREF attribute to an absolute value x.
Relative preference	*pref_add*(x)	Add a value x to the LOCAL_PREF attribute. In case the new value is larger than $2^{32}-1$, the LOCAL_PREF value is set to $2^{32}-1$.
	pref_sub(x)	Subtract a value x from the LOCAL_PREF attribute. In case the new value is smaller than 0, the LOCAL_PREF value is set to 0 .
Path prepending	*path_prepend*(x)	Prepend the ASN x, *i.e.* add the ASN s at the beginning of, the AS_PATH attribute.
Community membership	*comm_add*(x)	Add a community value x to the COMMUNITIES attribute. If the community value x is already part of the COMMUNITIES attribute, this action has no effect.
	comm_remove(x)	Remove a community value x from the COMMUNITIES attribute. If the community value x is not part of the COMMUNITIES attribute, this action has no effect.
Clear communities	*comm_clear*()	Remove all community values from the COMMUNITIES attribute.
Change next-hop	*nh_set*(x)	Set the route's NEXT_HOP attribute to IP address x.
Absolute MED	*med_set*(x)	Set the MULTI_EXIT_DISC attribute to the absolute value x.
Reject route	*reject*()	Reject the route.
Accept route	*accept*()	Accept the route.

Table 3: List of the most common atomic actions.

returns a modified version of the route and a mode that indicates if the route was accepted (acc) or rejected (rej) by the filter. The algorithm applies each rule in sequence. For each rule, the algorithm tests if the predicate matches or not. If the predicate matches, the algorithm applies the rule actions in sequence. Each action modifies the route. If special *accept*() or *reject*() action is encountered, the algorithm finishes immediately and the current version of the modified route, along with the route's mode are returned.

Algorithm 1 Applies a $filter$ to a $route$

```
mod_route ← route
for all rule in rules(filter) do
  if predicate(rule)(mod_route) then
    for all action in actions(rule) do
      if action = accept then
        return (mod_route, acc)
      else if action = reject then
        return (mod_route, rej)
      else
        mod_route ← action(mod_route)
      end if
    end for
  end if
end for
return (mod_route, acc)
```

2.5 Routing Filter Example

To illustrate the model presented in Section 2.4, this section presents a simple routing filter example. The objective of the filter is to allow a neighbor router to influence the `LOCAL_PREF` attribute set locally by attaching special communities to its routes. If community value 1 is attached, then the `LOCAL_PREF` should be set to 100 while if community value 2 is attached the `LOCAL_PREF` should be set to 200. Given the above rules, it would be ambiguous to attach both community values to a route, hence a route that contains both community values should be rejected. Finally, all routes that conform to the above rules have their `COMMUNITIES` attribute cleared and are accepted.

Using our formalism, the above filter can be specified with the following sequence of rules.

$$\begin{aligned}
R_1 &= \langle comm_in(1) \wedge comm_in(2), (reject()) \rangle \\
R_2 &= \langle comm_in(1), (pref_set(100)) \rangle \\
R_3 &= \langle comm_in(2), (pref_set(200)) \rangle \\
R_4 &= \langle true, (comm_clear(), accept()) \rangle
\end{aligned}$$

Router vendors allow network operators to specify such filters in their own equipments. For this purpose, specific, proprietary languages are provided. We show how our example filter can be expressed in the languages of the Cisco IOS and Juniper JunOS platforms in Figure 2 and 3 respectively.

```
ip community-list 10 permit 1
ip community-list 20 permit 2
route-map EXAMPLE deny 10
  match community 10
  match community 20
route-map EXAMPLE permit 20
  match community 10
  set local-preference 100
  set community none
route-map EXAMPLE permit 30
  match community 20
  set local-preference 200
  set community none
```

Figure 2: Example of a simple routing filter expressed in the Cisco IOS configuration language.

2.6 Problem Statement

The main objective of this paper is to provide a test for the equivalence of two routing filters.

Let $\mathcal{R}$ be the set of possible routes. A routing filter F as defined in Section 2.4 can be seen as a total function $F : \mathcal{R} \to \mathcal{R} \times \{\mathsf{acc}, \mathsf{rej}\} : r \to F(r) = (r', m)$. The result of a filter F applied on a route r is a pair $F(r) = (r', m)$ where r' is a modified version of the route r and m a mode in $\{\mathsf{acc}, \mathsf{rej}\}$ that indicates if the route is accepted or rejected by the filter.

Two routing filters F_1 and F_2 are equivalent if and only if, for all routes, their results are equal, i.e. $F_1 \equiv F_2 \Leftrightarrow \forall r \in \mathcal{R}, F_1(r) = F_2(r)$. Two routes are equal if all their attributes are equal.

The above definition of the equivalence of two routing filters leads to a straightforward, naive test algorithm: enumerate all routes in $\mathcal{R}$, apply the filters to each route and compare the results. If no route was found for which the filters have different results, then the test succeeds. Otherwise, a counterexample is found and the test fails. The complexity of this algorithm is mainly dependent on the size of $\mathcal{R}$.

Testing the equivalence of routing filters with the above naive algorithm is clearly not practical. In this paper, we aim at providing a novel method for testing the equivalence directly at the level of the filters rather than at the level of routes. To achieve this objective, we model routes with trees and routing filters as tree automata. The equivalence of routing filters can therefore be reduced to testing the equivalence of tree automata.

```
policy-statement EXAMPLE {
  community COMM1 members [ 1 ];
  community COMM2 members [ 2 ];
  term R1 {
    from {
      community COMM1;
      community COMM2;
    }
    then reject;
  }
  term R2 {
    from community COMM1;
    then local-preference 100;
  }
  term R3 {
    from community COMM2;
    then local-preference 200;
  }
  term R4 {
    then {
      community set none;
      accept;
    }
  }
}
```

Figure 3: Example of a simple routing filter expressed in the Juniper Networks' JunOS configuration language.

In the remaining of the paper, for clarity reasons, we do not take into account all the path attributes described in Section 2.2. We do not model the `ORIGIN` attribute as it is seldom used in routing filters. We do not model the `MULTI_EXIT_DISC` attribute as the related predicates and actions are similar to that of the `LOCAL_PREF` attribute. We do not model the `NEXT_HOP` attribute as it is seldom tested or modified by routing filters. We only consider 2 bytes ASNs and not 4 bytes ASNs. It should be clear for the reader that this does not decrease the generality of the proposed approach. Support for those attributes could easily be added later.

3 Tree Automata and Predicates

In this section, we recall the notion of tree automaton and illustrate it with some examples. We then describe how (atomic) predicates are modeled by tree automata. The modeling of actions is given in the next section, as well as the modeling of filters.

3.1 Model of a Route

In this section, we recall the notion of ranked tree and show how a route can be modeled by a ranked tree.

Trees – We consider ranked trees, *i.e.* trees where the number of children of a node is fixed by its label. Hence, an *alphabet* Σ is a finite set of *labels*, with a function ar mapping each label $a \in \Sigma$ to a positive integer $\mathsf{ar}(a)$ called its *arity*. For convenience, we write Σ_n for the set of labels of arity n: $\Sigma_n = \{a \in \Sigma \mid \mathsf{ar}(a) = n\}$.

The set T_Σ of *trees* over Σ is the least set containing all finite trees $a(t_1, \ldots, t_n)$ where $a \in \Sigma_n$ and $t_i \in \mathsf{T}_\Sigma$ for all $1 \leq i \leq n$. Note that children of a node are ordered. We identify each node of a tree with a word of integers (representing its position in the tree): The root is the empty word ϵ; the *nodes* of a tree $t = a(t_1, \ldots, t_n)$ are inductively defined by $\mathsf{nodes}(t) = \{\epsilon\} \cup \bigcup_{1 \leq i \leq n} \{i \cdot \pi \mid \pi \in \mathsf{nodes}(t_i)\}$. A node labeled by $a \in \Sigma$ is called an a-node.

Let us illustrate these definitions. Consider the alphabet $\Sigma^{\text{abcd}} = \{a, b, c, d\}$ where $\mathsf{ar}(a) = \mathsf{ar}(b) = 2$, $\mathsf{ar}(c) = 1$ and $\mathsf{ar}(d) = 0$. In other words, $\Sigma^{\text{abcd}}_2 = \{a, b\}$, $\Sigma^{\text{abcd}}_1 = \{c\}$ and $\Sigma^{\text{abcd}}_0 = \{d\}$. The tree $a(d)$ does not belong to $\mathsf{T}_{\Sigma^{\text{abcd}}}$, because $\mathsf{ar}(a) = 2$, so the root node should have two children. The tree $t = b(a(d, c(a(d, d))), d)$ belongs to $\mathsf{T}_{\Sigma^{\text{abcd}}}$. It is depicted in Figure 4. We have: $\mathsf{nodes}(t) = \{\epsilon, 1, 1{\cdot}1, 1{\cdot}2, 1{\cdot}2{\cdot}1, 1{\cdot}2{\cdot}1{\cdot}1, 1{\cdot}2{\cdot}1{\cdot}2, 2\}$, where the unique c-node is $1{\cdot}2$.

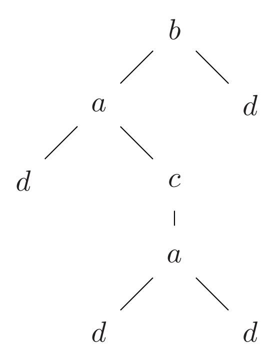


Figure 4: A tree $t \in \mathsf{T}_{\Sigma^{\text{abcd}}}$.

Routes as Trees – A route is composed of the attributes `DST_PREFIX`, `AS_PATH`, `LOCAL_PREF`, `COMMUNITIES` and of a mode indicating if the route is being modified (accepted, rejected, respectively) by the routing filter. As motivated in Section 2.6, the attributes `ORIGIN`, `NEXT_HOP` and `MULTI_EXIT_DISC` are not considered.

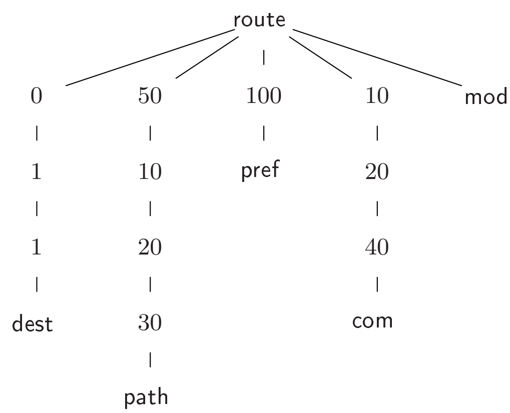


Figure 5: Tree modeling a route.

A route can be modeled as a tree as shown in Figure 5. This tree has a root labeled by label route of arity 5. The first branch models the destination prefix written in binary and ended by leaf dest, the most significant bit being at the bottom. For example, the route modeled on Figure 5 has the 192.0.0.0/3 destination prefix. The second branch models the `AS_PATH`, as a sequence of `ASN`s, such that the first `ASN` is at the bottom of the branch and the last `ASN` is at the top of the branch. This inverse order allows an easy modeling of the action of path prepending (see Section 4.2). This branch is ended with leaf path. The third branch contains attribute `LOCAL_PREF` and is ended with leaf pref. The fourth branch contains the set `COMMUNITIES` as a sorted sequence with the least number at the top of the branch. It is ended with leaf com. The last branch indicates the mode of the route: either mod (being modified), or acc (accepted), or rej (rejected). The first four branches are respectively called dest branch, path branch, pref branch and com branch. The underlying alphabet $\Sigma^{\mathcal{R}}$ is thus decomposed as follows: $\Sigma^{\mathcal{R}}_5 = \{\mathsf{route}\}$, $\Sigma^{\mathcal{R}}_4 = \Sigma^{\mathcal{R}}_3 = \Sigma^{\mathcal{R}}_2 = \emptyset$, $\Sigma^{\mathcal{R}}_1 = \Sigma^{\mathsf{dest}} \cup \Sigma^{\mathsf{path}} \cup \Sigma^{\mathsf{pref}} \cup \Sigma^{\mathsf{com}}$, and $\Sigma^{\mathcal{R}}_0 = \{\mathsf{dest}, \mathsf{path}, \mathsf{pref}, \mathsf{com}, \mathsf{mod}, \mathsf{acc}, \mathsf{rej}\}$. The branches (except the last one) have their specific alphabet of labels of arity 1. The dest branch uses $\Sigma^{\mathsf{dest}} = \{0, 1\}$ to represent `DST_PREFIX` written in binary; the path branch uses $\Sigma^{\mathsf{path}} = [0, 2^{16} - 1]$ (or $\Sigma^{\mathsf{path}} = [0, 2^{32} - 1]$) whose labels represent `ASN`s; the pref branch uses a label of $\Sigma^{\mathsf{pref}} = [0, 2^{32} - 1]$ to represent `LOCAL_PREF`; and the com branch uses $\Sigma^{\mathsf{com}} = [0, 2^{32} - 1]$ whose labels represent communities. In Section 6.2, we show that only parts of the alphabets Σ^{dest}, Σ^{path}, Σ^{pref} and Σ^{com} are to be considered, depending of the routing filters submitted for equivalence. This observation will be important for performance reasons.

3.2 Tree Automata

In this section, we recall the notion of tree automaton and illustrate it with the previous example of alphabet Σ^{abcd}. We also give an example related to routes.

Tree Automata – A *tree automaton* $\mathcal{A}$ over Σ is a tuple (Q, F, Σ, δ) where Q is a finite set of *states*, $F \subseteq Q$ is a set of final states, and δ is a set of *transitions* of the form $(q_1, \ldots, q_n) \xrightarrow{a} q$ with $a \in \Sigma_n$ and $q, q_1, \ldots, q_n \in Q$. The number of states is denoted by $|Q|$ and the number of transitions by $|\delta|$. The size $|\mathcal{A}|$ of $\mathcal{A}$ is equal to $|Q|$.

A *run* of a tree automaton $\mathcal{A}$ on a tree t is a function ρ mapping a state of $\mathcal{A}$ to each node of t, such that for every node π of t, if π is labeled by $a \in \Sigma_n$, then $(\rho(\pi \cdot 1), \ldots, \rho(\pi \cdot n)) \xrightarrow{a} \rho(\pi) \in \delta$. Intuitively, a tree automaton operates in a bottom-up manner on a tree: it assigns a state to each leaf, and then to each internal node, according to the states assigned to its children. A run ρ is *accepting* if the root is labeled by a final state, *i.e.* $\rho(\epsilon) \in F$.

A tree $t \in \mathsf{T}_\Sigma$ is *accepted* by the tree automaton $\mathcal{A}$ if there is an accepting run among all runs of $\mathcal{A}$ on this tree. The *language* of $\mathcal{A}$ is the set of trees accepted by $\mathcal{A}$, and is written $\mathsf{L}(\mathcal{A})$. We say that $\mathcal{A}$ *recognizes* $\mathsf{L}(\mathcal{A})$. A tree language $\mathcal{L} \subseteq \mathsf{T}_\Sigma$ is *recognizable* if there exists a tree automaton $\mathcal{A}$ recognizing it. Two tree automata are *equivalent* if they recognize the same language.

A tree automaton $\mathcal{A}$ is *deterministic* if it has no pair of distinct transitions with the same left-hand side. Formally, whenever $(q_1, \ldots, q_n) \xrightarrow{a} q \in \delta$ and $(q_1, \ldots, q_n) \xrightarrow{a} q' \in \delta$, we must have $q = q'$. Hence, a deterministic tree automaton has at most one run per tree. Every tree automaton can be *determinized*, *i.e.*, one can build an equivalent deterministic tree automaton [6]. However, the determinization procedure is exponential in time, and yields automata of exponential size.

Given a language $\mathcal{L} \subseteq \mathsf{T}_\Sigma$, a tree automaton $\mathcal{A}$ is $\mathcal{L}$-*complete* if there is at least one run of $\mathcal{A}$ on every $t \in \mathcal{L}$. Automaton $\mathcal{A}$ is *complete* if it is T_Σ-complete. For every tree automaton, one can easily change it into an equivalent complete one by adding a (non-final) sink state and transitions going to it, for every missing left-hand side. We name this operation *completion*.

Consider for instance the following automaton $\mathcal{A}_{\mathrm{branch}}$ that accepts all trees in $\mathsf{T}_{\Sigma^{\mathrm{abcd}}}$ having a branch with an even number of a-nodes. An accepting run of $\mathcal{A}_{\mathrm{branch}}$ on the tree $t = b(a(d, c(a(d, d))), d)$ is depicted in Figure 6.

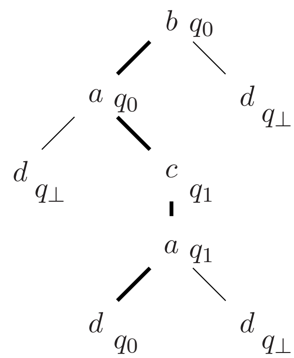


Figure 6: A run of $\mathcal{A}_{\text{branch}}$ on t.

The automaton is non-deterministic: it *guesses* a branch β, and *checks* that β has an even number of a-nodes. Its states $\{q_0, q_1, q_\perp\}$ are used in the following way. On every node π of β, the automaton assigns the state q_0 (resp. q_1) to π when the number of a-nodes in β below (and including) π is even (resp. odd). State $q_\perp$ is used on all nodes outside β. And q_0 is the unique final state. See again Figure 6 and branch β decorated with states q_0 and q_1. The transitions of the automaton are as follows. At the leaves (they are d-nodes), we can either start branch β and go to q_0 (as there is no a-node below), or start a branch different from β and go to $q_\perp$:

$$() \xrightarrow{d} q_0 \qquad () \xrightarrow{d} q_\perp$$

These two transitions indicate that the automaton is non deterministic. Traversing a c-node does not change the state:

$$(q_0) \xrightarrow{c} q_0 \qquad (q_1) \xrightarrow{c} q_1 \qquad (q_\perp) \xrightarrow{c} q_\perp$$

Traversing a b-node in β means that one child is in state either q_0 or q_1, and we do not change it, and the other child is in $q_\perp$. If the node is outside β, both states are $q_\perp$:

$$\begin{array}{lll} (q_0, q_\perp) \xrightarrow{b} q_0 & (q_1, q_\perp) \xrightarrow{b} q_1 & (q_\perp, q_\perp) \xrightarrow{b} q_\perp \\ (q_\perp, q_0) \xrightarrow{b} q_0 & (q_\perp, q_1) \xrightarrow{b} q_1 & \end{array}$$

Traversing an a-node is the same as for a b-node, except that q_0 becomes q_1, and conversely:

$$\begin{array}{lll} (q_0, q_\perp) \xrightarrow{a} q_1 & (q_1, q_\perp) \xrightarrow{a} q_0 & (q_\perp, q_\perp) \xrightarrow{a} q_\perp \\ (q_\perp, q_0) \xrightarrow{a} q_1 & (q_\perp, q_1) \xrightarrow{a} q_0 & \end{array}$$

As mentioned before, this automaton is non-deterministic. Coming back to Figure 6, we notice that there exists another accepting run which guesses the branch $\beta = bacad$, where d is the label of node $1{\cdot}2{\cdot}1{\cdot}2$. The automaton is also non-complete, as there is no transition with left-hand side $(q_0, q_0) \xrightarrow{a}$.

The Language of Routes – The set $\mathcal{R}$ of trees modeling routes is recognized by the following tree automaton $\mathcal{A}_{\mathcal{R}}$ with a unique final state q_f and the transitions

1. $() \xrightarrow{\mathsf{dest}} q_1, \quad (q_1) \xrightarrow{i} q_1, \;\; i \in \Sigma^{\mathsf{dest}},$

2. $() \xrightarrow{\mathsf{path}} q_2, \quad (q_2) \xrightarrow{i} q_2, \;\; i \in \Sigma^{\mathsf{path}},$

3. $() \xrightarrow{\mathsf{pref}} q_3, \quad (q_3) \xrightarrow{i} q_3', \;\; i \in \Sigma^{\mathsf{pref}},$

4. $() \xrightarrow{\mathsf{com}} q_4, \quad (q_4) \xrightarrow{i} q_{4,i}, \;\; i \in \Sigma^{\mathsf{com}},$

   $(q_{4,j}) \xrightarrow{i} q_{4,i}, \;\; i,j \in \Sigma^{\mathsf{com}}$ with $j > i,$

5. $() \xrightarrow{\mathsf{mod}} q_5, \quad () \xrightarrow{\mathsf{acc}} q_5, \quad () \xrightarrow{\mathsf{rej}} q_5,$

6. $(q_1, q_2, q_3', q_{4,i}, q_5) \xrightarrow{\mathsf{route}} q_f, \;\; i \in \Sigma^{\mathsf{com}},$

   $(q_1, q_2, q_3', q_4, q_5) \xrightarrow{\mathsf{route}} q_f.$

In this automaton, transitions 1 describe the dest branch as any sequence of bits ended by leaf dest. To limit its size, this automaton does not check that the dest branch has length at most 32. We will show in Section 3.6 that this has no impact on the filters equivalence test. Transitions 2 describe the path branch as any sequence of labels in Σ^{path} ended by leaf path. Transitions 3 describe the pref branch as one label in Σ^{pref} followed by leaf pref. Transitions 4 describe the com branch as an ordered sequence of labels in Σ^{com} ended by leaf com. The label j just read is stored in the current state $q_{4,j}$ in order to be compared with the label i read just after j, and the transition is applied if $j > i$. Transitions 5 describe the three modes, mod, acc, rej, of the route. Finally transitions 6 are applied at the root of the tree if the structure of each branch has been respected (when `COMMUNITIES` is a non-empty set in the first case, and when it is empty in the second case). Notice that automaton $\mathcal{A}_{\mathcal{R}}$ is deterministic, but non-complete. Moreover, it has a finite number of states, as states $q_{4,i}$ are restricted to $i \in \Sigma^{\mathsf{com}}$.

3.3 Operations on Tree Automata

Recognizable tree languages enjoy closure under all standard Boolean operations. The *complement* of a tree language $T \subseteq \mathsf{T}_\Sigma$ is the tree language $\mathsf{T}_\Sigma \setminus T$, *i.e.* the set of all trees that are not in T. The *intersection* and *union* of two tree languages $T_1, T_2 \subseteq \mathsf{T}_\Sigma$ are respectively $T_1 \cap T_2$ and $T_1 \cup T_2$.

Theorem 3.1. *Recognizable tree languages are closed under complementation, intersection and union.*

In other terms, given automata $\mathcal{A}_1 = (Q_1, F_1, \Sigma, \delta_1)$ and $\mathcal{A}_2 = (Q_2, F_2, \Sigma, \delta_2)$, one can always find automata $\mathcal{A}'_1$, $\mathcal{A}'_2$ and $\mathcal{A}'_3$ recognizing respectively $\mathsf{T}_\Sigma \setminus \mathsf{L}(\mathcal{A}_1)$, $\mathsf{L}(\mathcal{A}_1) \cap \mathsf{L}(\mathcal{A}_2)$, and $\mathsf{L}(\mathcal{A}_1) \cup \mathsf{L}(\mathcal{A}_3)$. This result is folklore [6], we only give some insights. Intersection and union can be obtained by computing the synchronized product of two automata $\mathcal{A}_1$ and $\mathcal{A}_2$. This construction is in time $O(|\delta_1| \cdot |\delta_2|)$ and yields automata of size $O(|\mathcal{A}_1| \cdot |\mathcal{A}_2|)$. Complementation is obtained by determinizing the automaton, completing it (so that each tree has exactly one run on it), and then swapping its final states with its non-final states. The complementation procedure is exponential in time and the obtained automaton is of exponential size. However if the initial automaton is already deterministic and complete, then the resulting automaton is constructed in constant time and has the same size. If the initial automaton is only deterministic, its completion requires to create a new state q^* and new transitions $(q_1, \ldots, q_n) \xrightarrow{a} q^*$ for all lacking transitions $(q_1, \ldots, q_n) \xrightarrow{a} -$. The time complexity of completion thus depends on the number of states and on the size of the alphabet.

Thanks to the Boolean closure properties, it can be checked whether two tree automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent. Indeed, it suffices to check that $\mathsf{L}(\mathcal{A}_1) \subseteq \mathsf{L}(\mathcal{A}_2)$, and conversely. The former inclusion is equivalent to $\mathsf{L}(\mathcal{A}_1) \cap (\mathsf{T}_\Sigma \setminus \mathsf{L}(\mathcal{A}_2)) = \emptyset$. Emptiness of tree automata is decidable, so we get [6]:

Theorem 3.2. *Equivalence of tree automata is decidable.*

This test is in exponential time if automata are non-deterministic [6], and in polynomial time otherwise [4].

3.4 Words and Trees

Tree structures subsume words. A word $a_1 a_2 \cdots a_n$ can be considered as a tree $a_n(a_{n-1}(\ldots a_1(\mathsf{nil})))$. We consider that each label has arity 1, and add a label nil of arity 0, so that a word is mapped to a branch. This reverse ordering is due to the bottom-up processing of tree automata.

This mapping also holds at the automata level. A *word automaton* over Σ is a tuple $(Q, I, F, \Sigma, \delta)$, where Q is a finite set of states, $I, F \subseteq Q$ are sets of initial (resp. final) states, and δ is a set of transitions of the form $q_0 \xrightarrow{a} q_1$. A run starts in an initial state and applies a series of transitions corresponding to labels of the input word. A word is accepted if a run ends in a final state. We refer the reader to [8] for more details. We recall that word automata have the same expressiveness as *regular expressions*: every regular expression can be translated to a word automaton recognizing the same words, and vice-versa. For instance the regular expression $(a|b)^*c$ can be translated to the word automaton in Figure 7 such that $Q = \{q_0, q_1\}$, q_0 (resp. q_1) is the unique initial (resp. final) state, and the transitions are $q_0 \xrightarrow{a} q_0$, $q_0 \xrightarrow{b} q_0$, and $q_0 \xrightarrow{c} q_1$.

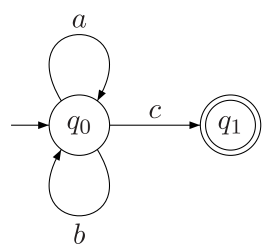


Figure 7: Word automaton recognizing $(a|b)^*c$.

3.5 General Approach for Modeling

Filters seen as Tree Automata – We recall that a routing filter F is composed of a sequence of rules $(R_1, \ldots R_n)$. Each rule $R_i = \langle P_i, A_i \rangle$ is composed of a predicate P_i which is a Boolean combination of atomic predicates, and of an action A_i which is a sequence of atomic actions. The problem studied in this article is the equivalence of two routing filters. This problem can be modeled as the equivalence of two tree automata (one for each filter).

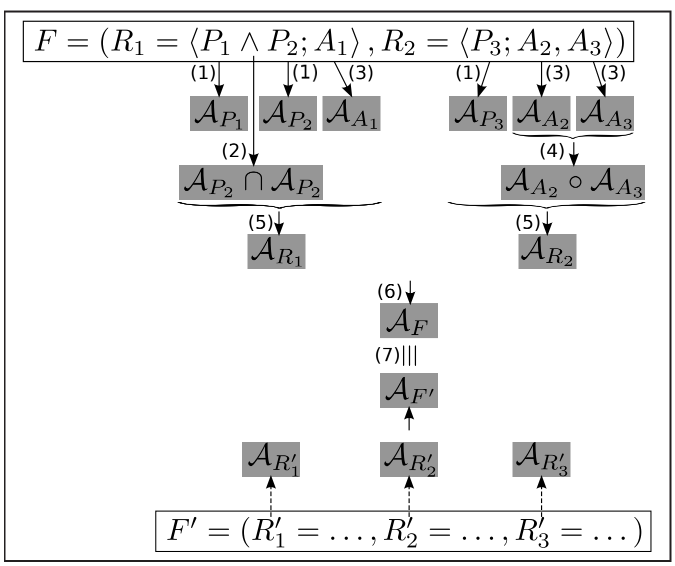


Figure 8: General approach for modeling.

The main ideas of our approach are depicted in Figure 8 and briefly described in the following paragraphs. The numbers between parentheses that appear in Figure 8 refer to the list items below.

1. Each atomic predicate appearing in predicate P_i is modeled by a tree automaton that exactly accepts routes (seen as trees t) satisfying the

atomic predicate.

2. By Theorem 3.1, each predicate P_i can be modeled by a tree automaton (see Section 3.6).

3. Each atomic action appearing in action A_i is modeled by a tree automaton that exactly accepts pairs of routes (t, t') such that t' is the image of t by the atomic action.

4. Each action A_i, as a sequence of atomic actions, is modeled by an automaton obtained by *composition* of the automata of the atomic actions (see Section 4.3).

5. A rule $R_i = \langle P_i, A_i \rangle$ is also modeled by a tree automaton that accepts pairs of routes (t, t') as follows: if t satisfies predicate P_i, then t' is the image of t by action A_i, otherwise $t' = t$. This automaton can be constructed from the automata for P_i and A_i thanks to the composition operation (see Section 4.4).

6. Finally, a routing filter $F = (R_1, \ldots, R_n)$, is modeled by a tree automaton obtained by composing the automata of rules R_i.

7. Two filters F, F' are equivalent if their corresponding automata are equivalent (see Theorem 3.2 and Section 4.4).

Quasi-routes – In order to work with smaller and simpler tree automata for atomic predicates and atomic actions and thus for routing filters, we consider *quasi-routes* instead of routes. A quasi-route is a tree with a root labeled by route, five branches of arbitrary length labeled by elements in $\Sigma_1^{\mathcal{R}} = \Sigma^{\mathsf{dest}} \cup \Sigma^{\mathsf{path}} \cup \Sigma^{\mathsf{pref}} \cup \Sigma^{\mathsf{com}}$ and ended by leaves labeled by elements in $\{\mathsf{dest}, \mathsf{path}, \mathsf{pref}, \mathsf{com}, \mathsf{mod}, \mathsf{acc}, \mathsf{rej}\}$. The deterministic automaton $\mathcal{A}_{quasi\mathcal{R}}$ with one final state q_f and the following transitions exactly accepts all quasi-routes:

1. $() \xrightarrow{\mathsf{dest}} q_0, \quad () \xrightarrow{\mathsf{path}} q_0, \quad () \xrightarrow{\mathsf{pref}} q_0, \quad () \xrightarrow{\mathsf{com}} q_0,$
   $() \xrightarrow{\mathsf{mod}} q_0, \quad () \xrightarrow{\mathsf{acc}} q_0, \quad () \xrightarrow{\mathsf{rej}} q_0,$

2. $(q_0) \xrightarrow{i} q_0, \quad i \in \Sigma_1^{\mathcal{R}}, \quad (q_0, q_0, q_0, q_0, q_0) \xrightarrow{\mathsf{route}} q_f.$

In the next sections, we describe the automata modeling the atomic predicates and the atomic actions such that each predicate operates on a quasi-route instead of a route, and each action modifies a quasi-route instead of a route. We will see in Section 4.4 where the automata $\mathcal{A}, \mathcal{A}'$ are constructed for the two filters F, F', how these automata are restricted to routes before testing for equivalence.

This approach which consists in working with quasi-routes instead of routes, and restrict to routes at the very last step, leads to small tree automata and thus to a more efficient algorithm. Additional optimizations are detailed in Section 6.

3.6 Model of a Predicate

Atomic Predicates – The most important atomic predicates used in routing filters have been described in Table 2. We show that such a predicate can be modeled by a tree automaton that accepts quasi-routes satisfying the predicate. The considered predicates are restricted to community membership, path membership, path origin, path neighbor, path subsequence, destination equality, and destination inclusion, as motivated in Section 2.6.

Let us consider in more details the atomic predicate $P_{\mathsf{pm}} = \mathit{path\_in}(x)$ of path membership, which tests whether an `ASN` x belongs to the `AS_PATH` attribute of a route. The next tree automaton $\mathcal{A}_{\mathsf{pm}}$ accepts all quasi-routes which second branch contains label x. In particular, all routes satisfying P_{pm} are accepted by the automaton, that is $\mathsf{L}(\mathcal{A}_{\mathsf{pm}})_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P_{\mathsf{pm}}\}$. The transitions are the following ones:

1. $() \xrightarrow{\mathsf{dest}} q_0, \quad () \xrightarrow{\mathsf{pref}} q_0, \quad () \xrightarrow{\mathsf{com}} q_0,$
   $() \xrightarrow{\mathsf{mod}} q_0, \quad () \xrightarrow{\mathsf{acc}} q_0, \quad () \xrightarrow{\mathsf{rej}} q_0,$
   $(q_0) \xrightarrow{i} q_0, \ i \in \Sigma_1^{\mathcal{R}},$
2. $() \xrightarrow{\mathsf{path}} q_{\neg x}, \quad (q_{\neg x}) \xrightarrow{x} q_x,$
   $(q_{\neg x}) \xrightarrow{i} q_{\neg x}, \ i \in \Sigma_1^{\mathcal{R}}, i \neq x,$
   $(q_x) \xrightarrow{i} q_x, \ i \in \Sigma_1^{\mathcal{R}},$
3. $(q_0, q_x, q_0, q_0, q_0) \xrightarrow{\mathsf{route}} q_\top,$
   $(q_0, q_{\neg x}, q_0, q_0, q_0) \xrightarrow{\mathsf{route}} q_\bot.$

In this automaton, transitions 2 use two states, $q_x, q_{\neg x}$ to remember if x has been seen or not on the second branch of the tree. Transitions 1 allow the same transitions as in automaton $\mathcal{A}_{quasi\mathcal{R}}$ for the other branches. Transitions 3 indicate that the final state $q_\top$ is reached in the case x has been seen, otherwise the non-final state $q_\bot$ is reached.

We can check that automaton $\mathcal{A}_{\mathsf{pm}}$ is deterministic and $\mathcal{R}$-complete. Indeed, for any route t there is exactly one run that assigns $q_\top$ (resp. $q_\bot$) to the root when t satisfies (resp. does not satisfy) predicate P_{pm}. In the sequel we require this property for each tree automaton associated with an

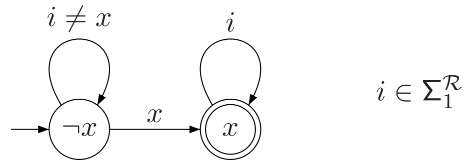


Figure 9: Word automaton recognizing words containing x.

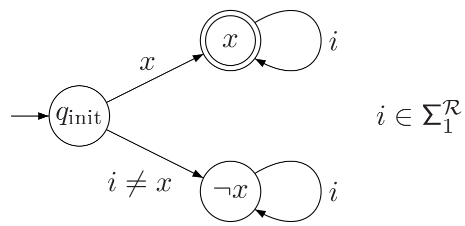


Figure 10: Word automaton recognizing words starting with x.

atomic predicate. This is necessary to get a correct modeling of rules (see Section 4.4) and to optimize the modeling of predicates (see Section 6.3).

As already mentioned in Section 3.4, tree structures subsume words, and tree automata subsume word automata. In the previous automaton, transitions 2 act like in the word automaton $\mathcal{B}$ depicted in Figure 9. This automaton uses alphabet $\Sigma_1^{\mathcal{R}}$, it is deterministic and complete. A word over $\Sigma_1^{\mathcal{R}}$ is accepted by $\mathcal{B}$ if and only if it contains label x.

Let us now consider the other atomic predicates P. The aim is to construct a tree automaton $\mathcal{A}_P$ such that it allows the same transitions as in $\mathcal{A}_{quasi\mathcal{R}}$, except for the branch concerned with predicate P. Moreover $\mathcal{A}_P$ must be deterministic, $\mathcal{R}$-complete and such that $\mathsf{L}(\mathcal{A}_P)_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P\}$.

For predicate of community membership, the treatment of the second branch by $\mathcal{A}_{\mathsf{pm}}$ is simply transposed to the fourth branch. For predicate of path neighbor, the approach is similar as with automaton $\mathcal{A}_{\mathsf{pm}}$, except that automaton $\mathcal{B}$ is modified in order to check that x is the first label of the word, as indicated in Figure 10.

The approach is similar for predicates of path origin, path subsequence, destination equality, and destination inclusion. For the two last predicates, $dst\_is(x)$ and $dst\_in(x)$, the `DST_PREFIX` and the IP prefix x are supposed to be written in binary and of length at most 32. We recall (see Section 3.2) that the automaton $\mathcal{A}_{\mathcal{R}}$ accepts the set $\mathcal{R}$ of trees modeling routes such that the dest branch is any sequence of bits ended by leaf dest, even those longer than 32. This lack of constraint of $\mathcal{A}_{\mathcal{R}}$ on the dest branch is not a problem. Indeed the automaton for predicate $dst\_is(x)$ recognizes all routes with dest branch equal to x. The justification for predicate $dst\_in(x)$ is divided into two cases. First, for routes where the dest branch is limited to 32 bits, the automaton for predicate $dst\_in(x)$ checks that x is a prefix of the branch. Second, a route with a longer dest branch is accepted if and only if it is accepted with its dest branch limited to the first 32 bits.

More generally the approach described in this section also holds for atomic predicates expressed by a regular expression. Given a regular expression imposing a condition on a branch of a route, this expression can be translated to a word automaton $(Q, \{q_{init}\}, F, \Sigma, \delta)$ that is deterministic and complete [8]. Suppose that the predicate is concerned with the path branch and $\Sigma = \Sigma_1^{\mathcal{R}}$, then the corresponding tree automaton has the next transitions:

1. $() \xrightarrow{\mathsf{dest}} q_0, \quad () \xrightarrow{\mathsf{pref}} q_0, \quad () \xrightarrow{\mathsf{com}} q_0,$

   $() \xrightarrow{\mathsf{mod}} q_0, \quad () \xrightarrow{\mathsf{acc}} q_0, \quad () \xrightarrow{\mathsf{rej}} q_0,$

   $(q_0) \xrightarrow{i} q_0, \ i \in \Sigma_1^{\mathcal{R}},$

2. $() \xrightarrow{\mathsf{path}} q_{init},$

   $(p) \xrightarrow{i} q, \quad \text{with transition } p \xrightarrow{i} q \text{ in } \delta,$

3. $(q_0, q, q_0, q_0, q_0) \xrightarrow{\mathsf{route}} q_\top, \quad \text{with } q \in F,$

   $(q_0, q, q_0, q_0, q_0) \xrightarrow{\mathsf{route}} q_\bot, \quad \text{with } q \notin F.$

Therefore, each atomic predicate P can be modeled by a tree automaton $\mathcal{A}_P$ working on quasi-routes, such that $\mathsf{L}(\mathcal{A}_P)_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P\}$. Moreover this automaton is deterministic and $\mathcal{R}$-complete.

Predicates – A predicate P is a Boolean combination of atomic predicates P_i, $1 \leq i \leq n$. Let $\mathcal{A}_i$ be a deterministic and $\mathcal{R}$-complete automaton for P_i as described previously. Then by Theorem 3.1, one can construct an automaton $\mathcal{A}_P$ modeling predicate P, in the sense that $\mathsf{L}(\mathcal{A}_P)_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P\}$. In the worst case, an exponential blow-up appears with each complementation appearing in the Boolean combination. Fortunately, since each automaton $\mathcal{A}_{P_i}$ is deterministic and $\mathcal{R}$-complete, the construction of automaton $\mathcal{A}_P$ can be done in polynomial time. Moreover, $\mathcal{A}_P$ is still deterministic and $\mathcal{R}$-complete.

Proposition 3.3. *Let P_1 and P_2 be two predicates, and $\mathcal{A}_1 = (Q_1, F_1, \Sigma^{\mathcal{R}}, \delta_1)$ and $\mathcal{A}_2 = (Q_2, F_2, \Sigma^{\mathcal{R}}, \delta_2)$ their respective deterministic and $\mathcal{R}$-complete automata. Then one can construct an automaton for $\neg P_1$ in constant time and with the same size as $\mathcal{A}_1$. One can also construct an automaton for $P_1 \wedge P_2$ (resp. $P_1 \vee P_2$) in time $O(|\delta_1| \cdot |\delta_2|)$ and with size $O(|\mathcal{A}_1| \cdot |\mathcal{A}_2|)$. In each case, the resulting automaton is still deterministic and $\mathcal{R}$-complete.*

In case of $P_1 \wedge P_2$ and $P_1 \vee P_2$, the previous proposition states that the resulting automaton is built in time $O(|\delta_1| \cdot |\delta_2|)$. In fact it can be built in time $O(|\mathcal{A}_1|^k \cdot |\mathcal{A}_2|^k \cdot |\Sigma|)$ where k is the maximal arity of the alphabet $\Sigma^{\mathcal{R}}$. Recall that except for route of arity 5, the other labels of $\Sigma^{\mathcal{R}}$ have arity 0 or 1. So in practice this k exponent does not play a major role. The proof of Proposition 3.3 is delayed in Section 6.3.

4 Tree Relations, Actions and Filters

Up to now, we used tree automata to describe routes satisfying a predicate. In routing filters, rules are made of predicates and actions. An action consists in transforming each route t_1 to another route t_2. Hence, we can consider an action (resp. a filter) as a binary relation $\mathsf{R} \subseteq \mathsf{T}_\Sigma \times \mathsf{T}_\Sigma$ containing such pairs (t_1, t_2).

4.1 Binary Tree Relations

In this section, we explain how tree automata can recognize such binary relations, not just tree languages. This is based on an operation mapping each pair (t_1, t_2) to a new tree.

The *overlay* of two trees $t_1, t_2 \in \mathsf{T}_\Sigma$ is the tree $t_1 \circledast t_2$. This tree is obtained by overlapping t_1 and t_2, in the following top-down way, as illustrated in Figure 11. Intuitively, labels of $t_1 \circledast t_2$ are pairs of labels of t_1 and t_2, and a fresh label $\diamond$ is used to fill the gaps. If roots of t_1 and t_2 are labelled by a and b respectively, then the root of $t_1 \circledast t_2$ is labelled by (a, b). The arities of a and b may differ, and in this case we use label $\diamond$. Let us name $a_1, \dots, a_n$ the children of the a-root in t_1, and $b_1, \dots, b_p$ the children of the b-root in t_2, and let us assume that $n > p$. Then (a, b) have n children equal, from left to right, to $(a_1, b_1), \dots, (a_p, b_p), (a_{p+1}, \diamond), \dots, (a_n, \diamond)$. The process is then repeated inductively on these children. We write $\Sigma_\diamond$ for the corresponding alphabet: it contains all labels $(a, b) \in \Sigma \times \Sigma$ with arity $\max\{\mathsf{ar}(a), \mathsf{ar}(b)\}$, and also all labels $(a, \diamond)$ and $(\diamond, a)$, for $a \in \Sigma$, with arity $\mathsf{ar}(a)$. We refer the reader to [6] for a formal definition.

A *binary tree relation* R over Σ is a subset of $\mathsf{T}_\Sigma \times \mathsf{T}_\Sigma$, *i.e.* a set of pairs (t_1, t_2) with $t_1, t_2 \in \mathsf{T}_\Sigma$. We say that R is *recognizable* if the tree language $\{t_1 \circledast t_2 \mid (t_1, t_2) \in \mathsf{R}\}$ is recognizable.

4.2 Modeling Atomic Actions

The most important atomic actions used in routing filters have been described in Table 3. We show that each atomic action A, seen as a binary

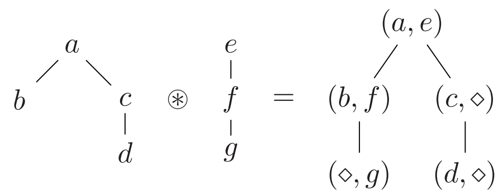


Figure 11: Example of overlay.

relation, is recognizable. In other words, there exists a tree automaton that accepts the overlays of routes $t \circledast t'$ such that t is transformed into t' by the action, *i.e.*, $(t, t') \in A$. Among the atomic actions described in Table 3, we consider the actions of absolute preference, relative preference, path prepending, community membership, clear communities, reject route and accept route.

We recall that a route t is modeled as a tree such that the last branch indicates the mode, mod, acc or rej, of the route. If t is in mod mode, then it is modified by the atomic action into a route t' with the same mode. Otherwise, it must remain unchanged.

As for predicates (see Section 3.6), in order to have small automata, we are going to construct automata for atomic actions that transform quasi-routes instead of routes. Contrarily to predicates, these automata are non-deterministic in order to guess in a bottom-up manner if the quasi-route has to be modified or not (depending on the mode). This simplifies the automata to build, and we will see that, in our context, determinism is not required for testing equivalence efficiently.

Relative preference – We begin with the atomic action $A_{\mathsf{rp}} = \mathit{pref\_add}(x)$ of relative preference: adding a value x to the `LOCAL_PREF`, such that the new value is set to $c = 2^{32} - 1$ when it is larger than c. Let t, t' be such that A_{rp} transforms t to t'. The two trees have the same shape and no label $\diamond$ is needed for the overlay $t \circledast t'$. The corresponding automaton $\mathcal{A}_{\mathsf{rp}}$ has one final state q_f and the following transitions:

1. $() \xrightarrow{(\mathsf{dest},\mathsf{dest})} q_0, \quad () \xrightarrow{(\mathsf{path},\mathsf{path})} q_0, \quad () \xrightarrow{(\mathsf{pref},\mathsf{pref})} q_0, \quad () \xrightarrow{(\mathsf{com},\mathsf{com})} q_0,$
   $(q_0) \xrightarrow{(i,i)} q_0, \ i \in \Sigma_1^{\mathcal{R}},$
2. $() \xrightarrow{(\mathsf{mod},\mathsf{mod})} q_{mod}, \quad () \xrightarrow{(\mathsf{acc},\mathsf{acc})} q_{fix}, \quad () \xrightarrow{(\mathsf{rej},\mathsf{rej})} q_{fix},$
3. $() \xrightarrow{(\mathsf{pref},\mathsf{pref})} q_1,$
   $(q_1) \xrightarrow{(i,i+x)} q_{+x}, \ i \in \Sigma^{\mathsf{pref}}, i + x \leq c,$

$$(q_1) \xrightarrow{(i,c)} q_{+x},\ i \in \Sigma^{\mathsf{pref}}, i + x > c,$$

4. $(q_0, q_0, q_{+x}, q_0, q_{mod}) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f,$

$$(q_0, q_0, q_0, q_0, q_{fix}) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f.$$

Automaton $\mathcal{A}_{\mathsf{rp}}$ checks the identity relation on the first, second and fourth branches, as indicated by transitions 1. Concerning the third branch, non-determinism appears among transitions 1 and 3: either the identity relation is checked on the pref branch with transitions 1, or the action of relative preference is performed (with states q_1 and q_{+x}) with transitions 3. Notice that one state q_0 is enough to check identity with transitions 1, as done for the automaton $\mathcal{A}_{quasi\mathcal{R}}$ accepting the quasi-routes. Transitions 2 memorize in state q_{mod} (resp. q_{fix}) whether the mode of trees t, t' is mod (resp. acc, rej). Finally, depending on the mode, transitions 4 lead to the final state q_f, either with the action A_{rp} performed on the third branch, or with the identity relation on this branch.

Automaton $\mathcal{A}_{\mathsf{rp}}$ deals with quasi-routes instead of routes. Let $\mathcal{R} \circledast \mathcal{R}$ be the set $\{t \circledast t' \mid t, t' \in \mathcal{R}\}$. Then we have $\mathsf{L}(\mathcal{A}_{\mathsf{rp}})_{|\mathcal{R}\circledast\mathcal{R}} = \{t \circledast t' \mid t, t' \in \mathcal{R} \text{ and } (t, t') \in A_{\mathsf{rp}}\}$

Community membership – Let us now proceed with the atomic action $A_{\mathsf{cm}} = comm\_add(x)$ of community membership: adding a community value x to the sorted sequence of communities. If x is already present, it is not added. Let t, t' be such that A_{cm} transforms t to t'. If x is not in the com branch of t, then t and t' have the same shape except on the com branch: the com branch of t' has one additional community (x) that has been correctly inserted in the com branch of t. The corresponding automaton $\mathcal{A}_{\mathsf{cm}}$ is similar to the previous automaton $\mathcal{A}_{\mathsf{rp}}$. It checks the identity relation on the first, second and third branches (see transitions 1 below). It checks the correct insertion of x in the fourth branch if the mode of t, t' is mod (see transitions 3 below). Non-determinism appears on the level of the com branch, depending on the current mode. Automaton $\mathcal{A}_{\mathsf{cm}}$ has one final state q_f and the following transitions:

1. $() \xrightarrow{(\mathsf{dest},\mathsf{dest})} q_0, \quad () \xrightarrow{(\mathsf{path},\mathsf{path})} q_0, \quad () \xrightarrow{(\mathsf{pref},\mathsf{pref})} q_0, \quad () \xrightarrow{(\mathsf{com},\mathsf{com})} q_0$

$$(q_0) \xrightarrow{(i,i)} q_0,\ i \in \Sigma_1^{\mathcal{R}},$$

2. $() \xrightarrow{(\mathsf{mod},\mathsf{mod})} q_{mod}, \quad () \xrightarrow{(\mathsf{acc},\mathsf{acc})} q_{fix}, \quad () \xrightarrow{(\mathsf{rej},\mathsf{rej})} q_{fix},$

3. $() \xrightarrow{(\diamond,\mathsf{com})} q_1, \quad (q_1) \xrightarrow{(\mathsf{com},x)} q_{ins},$

$$(q_{ins}) \xrightarrow{(i,i)} q_{ins},\ i \in \Sigma^{\mathsf{com}}, x > i, \qquad (q_1) \xrightarrow{(\mathsf{com},i)} q_{1,i},\ i \in \Sigma^{\mathsf{com}}, i > x,$$
$$(q_{1,j}) \xrightarrow{(j,i)} q_{1,i},\ i, j \in \Sigma^{\mathsf{com}}, j > i > x,$$
$$(q_{1,j}) \xrightarrow{(j,x)} q_{ins},\ j \in \Sigma^{\mathsf{com}}, j > x,$$
$$() \xrightarrow{(\mathsf{com},\mathsf{com})} q_2, \quad (q_2) \xrightarrow{(i,i)} q_2,\ i \in \Sigma^{\mathsf{com}}, i > x, \quad (q_2) \xrightarrow{(x,x)} q_{ins},$$

4. $(q_0, q_0, q_0, q_{ins}, q_{mod}) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f,$

   $(q_0, q_0, q_0, q_0, q_{fix}) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f.$

Transitions 3 need some explanations. State q_{ins} indicates that x has been inserted. There are three cases of insertion: (1) x is larger than all labels and it is inserted at the bottom of the branch, (2) x is not the largest value and it is properly inserted (state $q_{1,j}$ remembers the last seen value j), (3) x is already present and is therefore not inserted.

As for automaton $\mathcal{A}_{\mathsf{rp}}$, we have $\mathsf{L}(\mathcal{A}_{\mathsf{cm}})_{|\mathcal{R}\circledast\mathcal{R}} = \{t \circledast t' \mid t, t' \in \mathcal{R} \text{ and } (t, t') \in A_{\mathsf{cm}}\}$.

Route accepting – We now consider the atomic action $A_{\mathsf{ra}} = \mathit{accept}()$ of route accepting. Except for the mode, the corresponding automaton $\mathcal{A}_{\mathsf{ra}}$ checks for identity between t and t'. In case of mod mode for t, it checks for acc mode for t'. In case of acc or rej mode for t, it checks that the mode is left unchanged for t'. Automaton $\mathcal{A}_{\mathsf{ra}}$ has one final state q_f and the following transitions:

1. $() \xrightarrow{(\mathsf{dest},\mathsf{dest})} q_0, \quad () \xrightarrow{(\mathsf{path},\mathsf{path})} q_0, \quad () \xrightarrow{(\mathsf{pref},\mathsf{pref})} q_0, \quad () \xrightarrow{(\mathsf{com},\mathsf{com})} q_0,$

   $(q_0) \xrightarrow{(i,i)} q_0,\ i \in \Sigma_1^{\mathcal{R}},$

2. $() \xrightarrow{(\mathsf{mod},\mathsf{acc})} q_1, \quad () \xrightarrow{(\mathsf{acc},\mathsf{acc})} q_1, \quad () \xrightarrow{(\mathsf{rej},\mathsf{rej})} q_1,$

3. $(q_0, q_0, q_0, q_0, q_1) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f.$

The approach is similar for the other atomic actions. Thus, each atomic action A can be modeled by a (non-deterministic) tree automaton $\mathcal{A}_A$ working on quasi-routes, such that $\mathsf{L}(\mathcal{A}_A)_{|\mathcal{R}\circledast\mathcal{R}} = \{t \circledast t' \mid t, t' \in \mathcal{R} \text{ and } (t, t') \in A\}$.

4.3 Modeling Actions

An action A is a sequence $(A_1, \ldots, A_n)$ of atomic actions, for each of which we can build an automaton $\mathcal{A}_{A_i}$, as explained above. In this section, we show how to compute an automaton $\mathcal{A}_A$ for A from the automata $\mathcal{A}_{A_i}$, by using the *composition* operation denoted $\circ$. We start by composing $\mathcal{A}_{A_1}$ and $\mathcal{A}_{A_2}$, which gives us the automaton $\mathcal{A}'_2$, then compose $\mathcal{A}'_2$ with $\mathcal{A}_{A_3}$, and so on until we compose $\mathcal{A}'_{n-1}$ with $\mathcal{A}_{A_n}$, which gives us $\mathcal{A}_A = \mathcal{A}'_n$. In other words:

$$\mathcal{A}_A = (((\mathcal{A}_{A_1} \circ \mathcal{A}_{A_2}) \circ \mathcal{A}_{A_3}) \circ \cdots \circ \mathcal{A}_{A_n})$$

Let us detail how a composition of actions should operate. Given two actions A, A' and their related automata $\mathcal{A}_A$ and $\mathcal{A}_{A'}$, the tree $t \circledast t'$ accepted by $\mathcal{A}_A \circ \mathcal{A}_{A'}$ must be those for which there exists a tree t'' such that $t \circledast t''$ is accepted by $\mathcal{A}_A$ and $t'' \circledast t'$ is accepted by $\mathcal{A}_{A'}$. The construction of $\mathcal{A}_A \circ \mathcal{A}_{A'}$ works as follows. States of $\mathcal{A}_A \circ \mathcal{A}_{A'}$ are pairs (q, q') with q (resp. q') state of $\mathcal{A}_A$ (resp $\mathcal{A}_{A'}$). For labels of arity 1, a transition $((p, p')) \xrightarrow{(a,b)} (q, q')$ is a transition of $\mathcal{A}_A \circ \mathcal{A}_{A'}$ if and only if there is a label c and:

- a transition $(p) \xrightarrow{(a,c)} q$ in $\mathcal{A}_A$ and
- a transition $(p') \xrightarrow{(c,b)} q'$ in $\mathcal{A}_{A'}$.

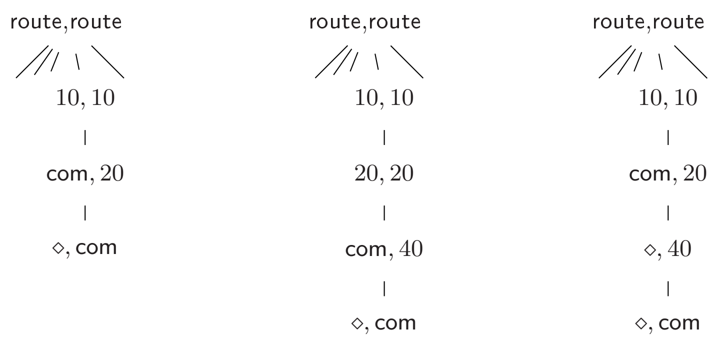
The construction is similar for labels of other arity.

Due to the possible different shapes of the trees involved in the composition, the previous procedure is incomplete. Let us explain on an example. Consider for instance the actions A and A' that respectively insert 20 and 40 in the com branch. Let us assume that we start with a route with only 10 in the com branch, *i.e.* 10(com). This branch is transformed into 10(20(com)) by A, and then into 10(20(40(com))) by A'. Figure 12a and 12b depict the corresponding two overlapped trees. If we compose the automata $\mathcal{A}_A$ and $\mathcal{A}_{A'}$ as described above, we should obtain the automaton $\mathcal{A}_A \circ \mathcal{A}_{A'}$ that accepts the overlapped tree of Figure 12c. We see that this is not the case since no transition $() \xrightarrow{(\diamond,\mathsf{com})} (q, q')$ has been constructed for $\mathcal{A}_A \circ \mathcal{A}_{A'}$. Indeed to get such a transition, we should have a transition $() \xrightarrow{(\diamond,\diamond)} q$ in $\mathcal{A}_A$ (that does not exist) and a transition $() \xrightarrow{(\diamond,\mathsf{com})} q'$ in $\mathcal{A}_{A'}$.

To avoid the problem illustrated by this example, before constructing the automaton $\mathcal{A}_A \circ \mathcal{A}_{A'}$ as explained above, we first slightly modify[4] automata $\mathcal{A}_A$ and $\mathcal{A}_{A'}$, so that they accept trees with an *arbitrary* number of labels $(\diamond, \diamond)$ at the bottom of branches path and com. We say that we $\diamond$-*fill* these automata. In this way, the branches path and com (the length of which may vary with the applied actions) now have the same shape thanks to the added labels $(\diamond, \diamond)$, and can thus be properly composed.

After the construction of the automaton $\mathcal{A}_A \circ \mathcal{A}_{A'}$, we must again slightly modify it such that it accepts trees with *no* label $(\diamond, \diamond)$ at the bottom of

[4] This modification is rather simple, and detailed in the proof of Theorem 4.1.



(a) $t \circledast t'' \in \mathsf{L}(\mathcal{A}_A)$ (b) $t'' \circledast t' \in \mathsf{L}(\mathcal{A}_{A'})$ (c) $t \circledast t' \in \mathsf{L}(\mathcal{A}_A \circ \mathcal{A}_{A'})$

Figure 12: Composition in com branch.

branches path and com. We say that we $\diamond$-*clean* this automaton. In this way, the language of the resulting automaton, restricted to $\mathcal{R} \circledast \mathcal{R}$, is equal to $\{t \circledast t' \mid t, t' \in \mathcal{R} \text{ and } (t, t'') \in A, (t'', t') \in A' \text{ for some } t'' \in \mathcal{R}\}$.

To summarize, the full composition procedure of two automata $\mathcal{A}_A$, $\mathcal{A}_{A'}$ is done in three steps: first we $\diamond$-fill these automata, then we construct $\mathcal{A}_A \circ \mathcal{A}_{A'}$, and finally we $\diamond$-clean the constructed automaton. We again denote by $\circ$ this operation of full composition.

Hence, given an action $A = (A_1, \ldots, A_n)$, and automata $\mathcal{A}_{A_i}$ for each atomic action A_i, we construct the automaton $\mathcal{A}_A = ((\mathcal{A}_{A_1} \circ \mathcal{A}_{A_2}) \circ \cdots \circ \mathcal{A}_{A_n})$ such that $\mathsf{L}(\mathcal{A}_A))_{|\mathcal{R} \circledast \mathcal{R}} = \{t \circledast t' \mid t, t' \in \mathcal{R} \text{ and } (t, t') \in A\}$.

4.4 Model of a Routing Filter

Rules – A routing filter F is a sequence of rules, and a rule $R = \langle P, A \rangle$ is composed of a predicate P and an action A. We have explained in Sections 3.6 and 4.3 how to build a tree automaton $\mathcal{A}_P$ (resp. $\mathcal{A}_A$) for predicate P (resp. action A). We recall that $\mathsf{L}(\mathcal{A}_P)_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P\}$, and $\mathsf{L}(\mathcal{A}_A)_{|\mathcal{R} \circledast \mathcal{R}} = \{t \circledast t' \mid t, t' \in \mathcal{R} \text{ and } (t, t') \in A\}$. In this section, we show how to build an automaton for rule R from the automata $\mathcal{A}_P$ and $\mathcal{A}_A$. We use the composition operation (presented in the previous section), however with some care since $\mathcal{A}_P$ accepts trees and $\mathcal{A}_A$ accepts overlays of trees.

We consider a new label $\overline{\mathsf{route}}$. Given a quasi-route t, we denote by $\overline{t}$ the tree obtained from t by replacing the root label route by $\overline{\mathsf{route}}$. We denote by $\mathcal{R}'$ the set $\mathcal{R} \cup \{\overline{t} \mid t \in \mathcal{R}\}$. Let us give the main ideas to build a tree automaton $\mathcal{A}_R$ for rule R. First we derive from $\mathcal{A}_P$ a new automaton $\mathcal{B}_P$ such that $\mathsf{L}(\mathcal{B}_P)_{|\mathcal{R} \circledast \mathcal{R}'} = \{t \circledast t \mid t \text{ satisfies } P\} \cup \{t \circledast \overline{t} \mid t \text{ does not satisfy } P\}$. Second we derive from $\mathcal{A}_A$ a new automaton $\mathcal{B}_A$ such that $\mathsf{L}(\mathcal{B}_A)_{|\mathcal{R}' \circledast \mathcal{R}} =$

$\{t \circledast t' \mid t, t' \in \mathcal{R} \text{ and } (t,t') \in A\} \cup \{\overline{t} \circledast t \mid t \in \mathcal{R}\}$. Finally we compute $\mathcal{B}_P \circ \mathcal{B}_A$. By definition of the composition and thanks to the trees $\overline{t}$, the resulting automaton $\mathcal{A}_R$ accepts overlays $t \circledast t'$ of trees (when restricted to $\mathcal{R} \circledast \mathcal{R}$) such that either t satisfies P and thus is transformed in t' by A, or t does not satisfy P and thus is left unchanged by A.

Let us now explain in more details how to compute automata $\mathcal{B}_P$ and $\mathcal{B}_A$. Suppose that $\mathcal{A}_P = (Q, F, \Sigma^{\mathcal{R}}, \delta)$. Recall that $\mathcal{A}_P$ is deterministic and $\mathcal{R}$-complete (see Section 3.6). Then we build $\mathcal{B}_P = (Q, \{q_f\}, \Sigma^{\mathcal{R}} \times \Sigma^{\mathcal{R}}, \delta')$ with

- $() \xrightarrow{(a,a)} q \in \delta', \quad \text{if } () \xrightarrow{a} q \in \delta,$
- $(p) \xrightarrow{(a,a)} q \in \delta', \quad \text{if } (p) \xrightarrow{a} q \in \delta,$
- $(p_1,p_2,p_3,p_4,p_5) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f \in \delta', \quad \text{if } (p_1,p_2,p_3,p_4,p_5) \xrightarrow{\mathsf{route}} q \in \delta$ with $q \in F$,

  $(p_1,p_2,p_3,p_4,p_5) \xrightarrow{(\mathsf{route},\overline{\mathsf{route}})} q_f \in \delta', \quad \text{if } (p_1,p_2,p_3,p_4,p_5) \xrightarrow{\mathsf{route}} q \in \delta$ with $q \notin F$.

This construction works because, given a route t, there is exactly one run for t in $\mathcal{A}_P$, and this run is accepting if and only if t satisfies P. Thus, the labels of transitions in $\mathcal{A}_P$ are duplicated in transitions of $\mathcal{B}_P$ except for label route which is replaced by $(\mathsf{route},\mathsf{route})$ (resp. $(\mathsf{route},\overline{\mathsf{route}})$) if the run is (resp. is not) accepting.

Concerning action A, we construct $\mathcal{B}_A$ from $\mathcal{A}_A$ by adding to the later automaton two new states q_0 and q_f such that q_f is final, and the following transitions[5]:

1. $() \xrightarrow{(\mathsf{dest},\mathsf{dest})} q_0, \quad () \xrightarrow{(\mathsf{path},\mathsf{path})} q_0, \quad () \xrightarrow{(\mathsf{pref},\mathsf{pref})} q_0, \quad () \xrightarrow{(\mathsf{com},\mathsf{com})} q_0,$
   $() \xrightarrow{(\mathsf{mod},\mathsf{mod})} q_0, \quad () \xrightarrow{(\mathsf{acc},\mathsf{acc})} q_0, \quad () \xrightarrow{(\mathsf{rej},\mathsf{rej})} q_0,$
2. $(q_0) \xrightarrow{(i,i)} q_0 \quad i \in \Sigma_1^{\mathcal{R}}, \quad (q_0,q_0,q_0,q_0,q_0) \xrightarrow{(\overline{\mathsf{route}},\mathsf{route})} q_f.$

Given a rule $R = \langle P, A \rangle$, we are thus able to compute a tree automaton $\mathcal{A}_R$ such that $\mathsf{L}(\mathcal{A}_R)_{|\mathcal{R}\circledast\mathcal{R}} = \{t \circledast t' \mid (t \text{ satisfies } P \text{ and } (t,t') \in A) \text{ or } (t \text{ does not satisfy } P \text{ and } t' = t)\ \}$.

[5]These transitions are similar to the transitions of automaton $\mathcal{A}_{quasi\mathcal{R}}$ accepting quasi-routes (see Section 3.5).

Filters – A routing filter F is a sequence $(R_1, \ldots, R_n)$ of rules, for each of which we can build an automaton $\mathcal{A}_{R_i}$ as explained above. A tree automaton $\mathcal{A}_F$ for F is simply obtained by composing the automata $\mathcal{A}_{R_1}, \ldots, \mathcal{A}_{R_n}$. Recall that this automaton treats quasi-routes instead of routes, and that $\mathsf{L}(\mathcal{A}_F)_{|\mathcal{R}\circledast\mathcal{R}} = \{t \circledast t' \mid t, t' \in \mathcal{R} \text{ and } (t, t') \in F\}$.

It remains to explain how to test the equivalence of two filters. In this aim, it is necessary to modify automaton $\mathcal{A}_F$ into $\mathcal{B}_F$ such that automaton $\mathcal{B}_F$ now treats routes (and no longer quasi-routes). Thanks to the composition operation, it is easy to construct $\mathcal{B}_F$ such that $\mathsf{L}(\mathcal{B}_F) = \mathsf{L}(\mathcal{A}_F)_{|\mathcal{R}\circledast\mathcal{R}}$. We construct a tree automaton $\mathcal{A}_{id}$ such that $\mathsf{L}(\mathcal{A}_{id}) = \{t \circledast t \mid t \in \mathcal{R}\}$. This automaton is easily built from automaton $\mathcal{A}_{\mathcal{R}}$ recognizing the set of routes. Then we have $\mathcal{B}_F = \mathcal{A}_{id} \circ \mathcal{A}_F$.

Given two filters F, F' and their corresponding automata $\mathcal{B}_F, \mathcal{B}_{F'}$, testing if F, F' are equivalent amounts to test if the automata $\mathcal{B}_F, \mathcal{B}_{F'}$ are equivalent. This test is decidable in exponential time (see Theorem 3.2). However this exponential blow-up can be avoided in our context because the relations involved in filters are *functional*. Indeed, every action inside a filter transforms each route into a unique route.

Theorem 4.1. *Let F and F' be two filters and their respective automata $\mathcal{B}_F$ and $\mathcal{B}_{F'}$. Then it can be decided in time $O(|\mathcal{B}_F| \cdot |\mathcal{B}_{F'}|)$ whether F and F' are equivalent.*

Moreover, in case of non equivalence, a tree t is constructed such that $(t, t_1) \in F$, $(t, t_2) \in F'$ with $t_1 \neq t_2$.

The proof of this theorem is given in Section 6.3. Notice that the result also holds for any pair of tree relations (instead of filters) as soon as they are total functions.

Summary – Let us recall the whole process to model routing filters and its related test of equivalence by tree automata.

- Recall that a routing filter F is a sequence of rules, and a rule $R = \langle P, A \rangle$ is composed of a predicate P and an action A.

- Given a filter F, a corresponding automaton $\mathcal{A}_F$ is constructed by induction on the structure of the filter. For efficiency reasons, all the intermediate automata as well as $\mathcal{A}_F$ operate on quasi-routes instead of routes in a way to limit the size of automata. Before testing whether two filters F and F' are equivalent, the automata $\mathcal{A}_F$ and $\mathcal{A}_{F'}$ are modified into $\mathcal{B}_F, \mathcal{B}_{F'}$ respectively in a way to treat routes and are then tested for automata equivalence.

- For each atomic predicate and each atomic action, we have constructed a corresponding tree automaton (accepting trees in the first case and overlays of two trees in the second case). For an atomic predicate P, the related automaton $\mathcal{A}_P$ is deterministic and $\mathcal{R}$-complete. In this way, for any route t there exists exactly one run that assigns a final (resp. non final) to the root of t if t satisfies (resp. does not satisfy) P. The property is imposed to $\mathcal{A}_P$ for efficiency reasons. For an atomic relation A, the related automaton $\mathcal{A}_A$ is in general non-deterministic. Moreover for each route t, there is exactly one route t' such that $t \circledast t'$ is accepted to $\mathcal{A}_A$, i.e. $(t, t') \in A$ (the relation is functional).

- Each predicate P is a Boolean combination of atomic predicates $P_i, 1 \leq i \leq n$. The associated automaton $\mathcal{A}_P$ can be built from the automata $\mathcal{A}_{P_i}$ thanks to Theorem 3.1. As each $\mathcal{A}_{P_i}$ is deterministic and $\mathcal{R}$-complete, the exponential blow-up that could appear at each complementation operation is avoided (see Proposition 3.3)

- The composition operation $\circ$ on two automata (on overlays of trees) is used at several places of the process: when dealing with (1) sequences of atomic actions, (2) rules, (3) sequences of rules, and (4) when limiting an automaton $\mathcal{A}_F$ for a filter F to work only for routes. For technical reasons due to symbol $\diamond$ used in the overlay of trees, the two automata are first $\diamond$-filled, then composed, and finally $\diamond$-cleaned.

- Each action is a sequence $(A_1, \ldots, A_n)$ of atomic actions. The automaton $\mathcal{A}_A$ for A is built from the automata $\mathcal{A}_{A_i}$, $1 \leq i \leq n$, as $((\mathcal{A}_{A_1} \circ \mathcal{A}_{A_2}) \circ \cdots \circ \mathcal{A}_{A_n})$. A rule $R = \langle P, A \rangle$ is composed of a predicate P and an action A. The automata $\mathcal{A}_P$ and $\mathcal{A}_A$ are modified in a way such that their composition results in an automaton $\mathcal{A}_R$ that accepts overlays $t \circledast t'$ of trees (when restricted to $\mathcal{R} \circledast \mathcal{R}$) where either t satisfies P and thus is transformed in t' by A, or t does not satisfy P and is left unchanged by A. Each filter F is a sequence $(R_1, \ldots, R_n)$ of rules. The automaton $\mathcal{A}_F$ for F is built from the automata $\mathcal{A}_{R_i}$, $1 \leq i \leq n$, as $((\mathcal{A}_{R_1} \circ \mathcal{A}_{R_2}) \circ \cdots \circ \mathcal{A}_{R_n})$.

- Given two filters F, F', we test for their equivalence by testing equivalence of the automata $\mathcal{B}_F$ and $\mathcal{B}_{F'}$ (see Theorem 3.2). The exponential blow-up of this test can be avoided in our context because the relations involved in filters are functional (see Theorem 4.1).

Complexity of the algorithm – We conclude this section with the overall complexity of our algorithm. The size $|F|$ of a filter F is the sum of the sizes of its rules. The size $|R|$ of a rule $R = \langle P, A \rangle$ is defined as $n_P + n_A + 1$ where n_P is the number of atomic predicates of P and n_A is the number of atomic actions of A. We add the constant 1 to reflect the cost of composition. We make the hypothesis that the automata for the atomic predicates and actions have their sizes all bounded by p.

Theorem 4.2. *For the proposed algorithm, the automaton constructed for a filter F has size $O(p^{|F|})$ and can be constructed in time $O(p^{|F|})$; testing the equivalence of two filters F, F' is done in time $O(p^{|F|+|F'|})$.*

Proof. First, when constructing the automaton for a predicate P from the automata for the atomic predicates of P, we are able to avoid the exponential blow-up due to the complementation of non-deterministic automata (see Theorem 3.1 and Proposition 3.3). Therefore the most costly operations are the union and intersection based on the synchronized product of two automata. Second, when constructing the automaton for an action A from the automata for the atomic actions of A, the needed composition operation also requires some product of two automata (see Section 4.3). This composition operation is also used for a rule and for a sequence of rules. At the end of the process, we get an automaton $\mathcal{B}_F$ for the given filter F, such that its size is in $O(p^{|F|})$ and the time to build it is in $O(p^{|F|})$. Finally, given two automata $\mathcal{B}_F, \mathcal{B}_{F'}$ for two filters F, F', we can avoid a second exponential blow-up and test in time linear in the size of these automata whether the two filters are equivalent (see Theorems 3.2 and 4.1), leading to the announced time complexity. $\square$

5 Evaluation

5.1 Implementation

We have implemented a prototype in Java, publicly available under the GPLv2 licence. It implements all atomic predicates and actions as presented in Section 2.6. Filters implementation is based on the model presented in this paper. The construction of the automata follows the inductive process described in this paper, including the optimizations given in Section 6. We used a homemade parser to convert Cisco IOS configuration files into Java source code. Tree automata objects and standard operations are implemented inside a separate library, also publicly available.

5.2 Experiments

We used the prototype described in Section 5.1 to perform several experiments, the results of which we present in this section. Our experiments show the link between filter size, tree automata size and running-time of the equivalence test. All tests were performed on a computer running Linux 3.2

with an Intel Core2 Duo CPU and 4GB of RAM. We used Java 1.6 through OpenJDK (IcedTea6).

Instances – We tested our algorithm on five families of filters. The first family of filters, called *cisco* in our figures, has been generated from the BGP routing filters defined in the Cisco IOS configuration of a router from a large European transit network. The configuration contained 48 *route-maps* each of them describing a single filter. We focused on 12 filters. Each of them has then been tested against its subfilters, where the n-th subfilter is obtained from the original one by keeping its first n rules. Usually, such filters are not equivalent to their subfilters, except for the full subfilter containing all rules of the original filter. This way we get both equivalent and non-equivalent pairs of filters.

The remaining four families were built by hand. Family *eq_ com* is the set of filters with n rules containing only action $comm\_add(1)$. Increasing values of n yielded filters of increasing sizes. Family *eq_ path_ com* is the same as *eq_ com*, but each rule also contains a predicate $path\_in(1)$. Family *path_ com* is composed of filters with ith rule made of a predicate $path\_in(i)$ and an action $comm\_add(i)$. Finally, family *path_ com_ acc* is similar to *path_ com*, except that the action is the composition of $comm\_add(i)$ with $accept()$. For each family, we generated 10 modified versions of each filter by performing a random permutation of its rules. We then tested the equivalence between a filter and its modified versions. Each equivalence test was positive, except for the family *path_ com_ acc*.

Results – We show the results of our experiments in figures 13 to 16. Note that the y axis is in logarithmic scale for all figures in this section.

Figure 13 shows how the size of the resulting automata varies with the filter size. In this Figure, there is a point for each filter in the families described earlier in the section as well as for the shuffled versions. We observe a linear alignment of the points, showing that the size of the automaton is exponential in the size of the filter, as stated in Theorem 4.2.

Figure 14 shows how the time to build the automaton also follows this complexity, as described in Theorem 4.2. We can observe that for very small filters, automata are built in a few milliseconds. It takes about one second for a filter of size 25. As we will see, this is negligible compared to the time needed for testing equivalence.

In Figures 15 and 16, positive instances, i.e. those for which equivalence holds, are denoted by +, and negative instances by **x**. Figure 15 shows how the execution time of the equivalence test varies with the product of the automata sizes. Note that the x axis is shown on a logarithmic scale. This

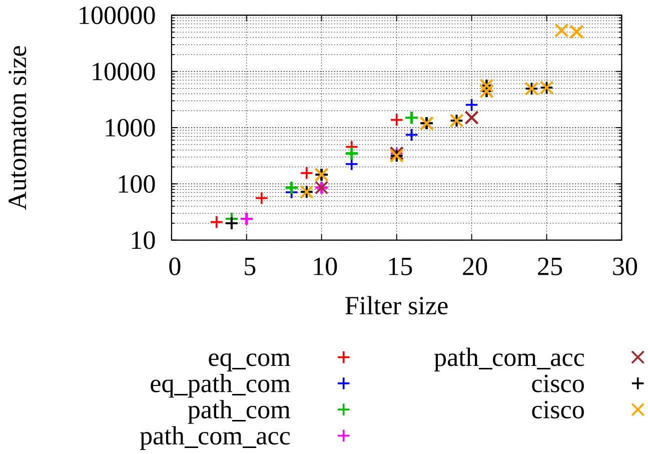


Figure 13: Size of automata.

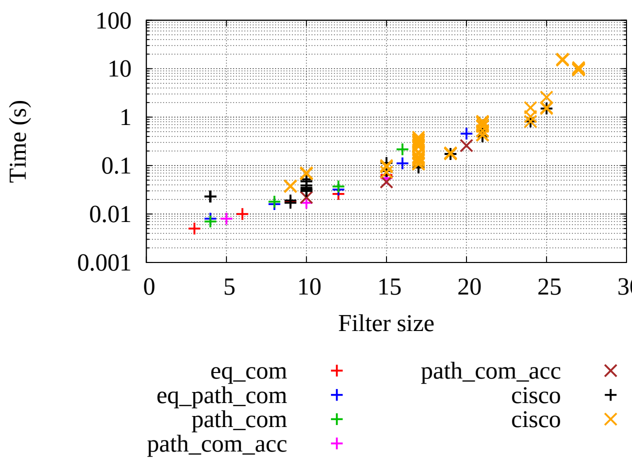


Figure 14: Time for building automata.

figure suggests that there is a linear relationship between the product of the automata sizes and the running time of the algorithm. We state formally in Theorem 4.1 that testing the equivalence of two filters F and F' is in time $O(|\mathcal{B}_F| \cdot |\mathcal{B}_{F'}|)$ when the input is two automata $\mathcal{B}_F$ and $\mathcal{B}_{F'}$. The overall complexity of testing equivalence between two filters F and F' is a single exponential in the size of the filters, i.e. $O(p^{|F|+|F'|})$ for a fixed p, see Theorem 4.2. Our experiments confirm this complexity, as depicted in Figure 16.

We note that real-world filters, like those obtained from the Cisco configuration file, are generally more efficiently processed by our algorithm than the synthetic filters. One explanation for this behaviour is the existence, in every rule of the *cisco* filters, of an *accept*() or *reject*() action that prevents further processing of the routes. Moreover, equivalence is generally faster on negative instances, as explained in the proof of Theorem 4.1.

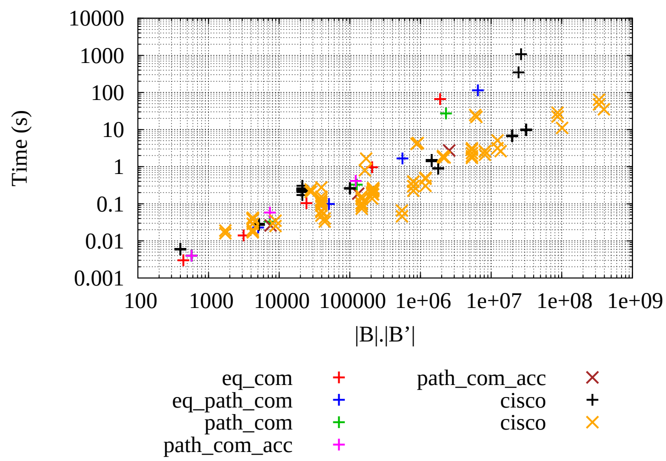


Figure 15: Testing equivalence from automata.

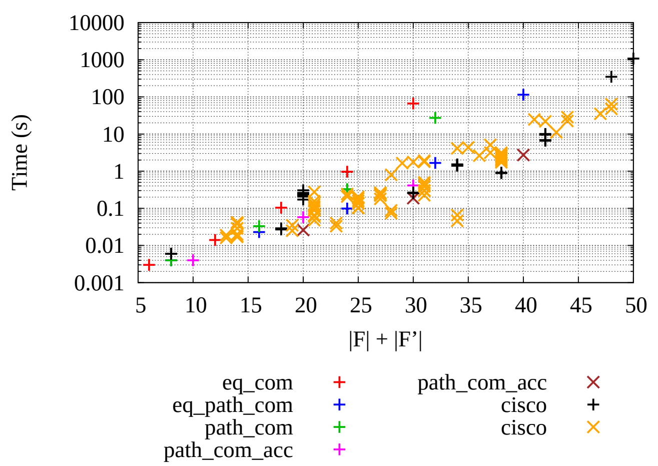


Figure 16: Testing equivalence from filters.

We also tried to compare our efficient equivalence test proposed in Theorem 4.1 with the usual equivalence test based on Boolean operations on tree automata (see Theorem 3.2). For very small filters made of only one rule with one action, our algorithm takes 1 millisecond, while the usual equivalence test based on Boolean operations takes 28 minutes.

6 Optimizations

In this section, we propose several optimizations to get a more efficient algorithm for testing the equivalence of routing filters.

6.1 Preprocessing Actions

When considering an action $A = (A_1, \dots, A_n)$, two trivial optimizations can be applied to reduce the number n of atomic actions, while keeping an action equivalent to A.

The first one is obtained by removing all atomic actions following an atomic action *accept*() (resp. *reject*()). Indeed, if $A_i = accept()$, then all atomic actions A_j with $j > i$ will not modify any route, as all of them will be in acc mode. Hence A is equivalent to $(A_1, \dots, A_i)$.

The second optimization applies to the pref branch. Three atomic actions relate to this branch: absolute preference $pref\_set(x)$, and relative preference $pref\_add(x)$ and $pref\_sub(x)$. Assume that $A_i = pref\_set(a)$. Then all atomic actions A_j of absolute and relative preference with $j < i$ can be removed, as their effects will be replaced by the effect of $pref\_set(a)$. Hence, one can remove all relative and absolute preference atomic actions preceding the last absolute preference atomic action.

6.2 Reducing Alphabet Size

In this section, we show that it is possible to optimize the proposed modeling of routing filters, by reducing the sizes of the built tree automata, especially by reducing the sizes of their underlying alphabet.

Let F_1 and F_2 be two routing filters that we want to test for equivalence. We show below that we can restrict routes to consider to those having only labels appearing in atomic predicates and actions of F_1 and F_2 (with some refinement in the path branch).

Consider for instance the com branch. The atomic predicates possibly used by F_1 and F_2 are community membership (see $comm\_in(x)$ in Table 2) and the atomic actions are community membership and clear communities (see $comm\_add(x)$, $comm\_remove(x)$ and $comm\_clear()$ in Table 3). Let us denote by $\Sigma_{\mathsf{F}}{}^{\mathsf{com}}$ the set of labels a in Σ^{com} such that a appears in an atomic predicate/action of F_1 or F_2, as $comm\_in(a)$, $comm\_add(a)$, or $comm\_remove(a)$.

If a is a community of route t such that $a \notin \Sigma_{\mathsf{F}}{}^{\mathsf{com}}$, then there is no need to store it in the com branch of t. Indeed, all atomic predicates and actions relative to the com branch behave the same on t and on the route t without

a. Therefore, for all routes t, we only store their communities $a \in \Sigma_\mathsf{F}{}^\mathsf{com}$ in the com branch. Moreover the alphabet used by the automata on the com branch is reduced to $\Sigma_\mathsf{F}{}^\mathsf{com}$ (instead of Σ^com).

The same kind of argument can be repeated for the path branch. In this case, we define by $\Sigma_\mathsf{F}{}^\mathsf{path}$ the set of labels a in Σ^path such that a appears in an atomic predicate/action of F_1 or F_2, as $path\_in(a)$, $path\_ori(a)$, $path\_nei(a)$, $path\_sub(s)$ with a a label in word s, and $path\_prepend(a)$. We also add to $\Sigma_\mathsf{F}{}^\mathsf{path}$ a new label denoted by $\star$. This label is used at the place of each $a \in \Sigma^\mathsf{path} \setminus \Sigma_\mathsf{F}{}^\mathsf{path}$ in the path branch. Contrarily to the com branch, we cannot forget any symbol not in $\Sigma_\mathsf{F}{}^\mathsf{path}$, due to predicate $path\_sub(s)$. Let us illustrate this with an example. Assume that F_1 uses predicate $P = path\_sub(ab)$, and route t has a path branch $b(c(d(a(\mathsf{path}))))$ with $c, d \notin \Sigma_\mathsf{F}{}^\mathsf{path}$. If we forget c, d, then the path branch is replaced by $b(a(\mathsf{path}))$ (instead of $b(\star(\star(a(\mathsf{path})))))$. Predicate P is then satisfied, which is not correct. In this way, the alphabet used by the automata on the path branch is reduced to $\Sigma_\mathsf{F}{}^\mathsf{path}$.

Similarly, on the dest branch, if no predicate $dst\_is(x)$ or $dst\_in(x)$ appears in F_1 nor F_2, we can take $\Sigma_\mathsf{F}{}^\mathsf{dest} = \emptyset$.

On the pref branch, we can also avoid to consider the whole range $[0, 2^{32} - 1]$ for alphabet Σ^pref. Indeed, when entering a routing filter, `LOCAL_PREF` is set to a fixed value for all incoming routes. Recall that the default for this value is 100. Moreover, each action has a unique effect on the value of `LOCAL_PREF`: given an input value, it generates a unique possible output value. Hence, for each filter rule $R_i = \langle P_i, A_i \rangle$, each input value p of `LOCAL_PREF` can yield two output value: p if predicate P_i is false, and the result of applying A_i on p otherwise. This gives at most 2^n values to consider, for a filter with n rules, therefore reducing the size of Σ^pref when n is small.

6.3 Efficient Automata Operations

In this section, we come back to the automata operations used for testing equivalence of filters. In Theorems 3.1 and 3.2, the prohibitive (exponential) operation is the complementation of a non-deterministic automaton $\mathcal{A}$. In our context, these two theorems have more efficient counterparts, as stated in Proposition 3.3 and Theorem 4.1. We here give the proof of these results.

We recall that complementation of $\mathcal{A}$ is obtained by determinizing it, completing it and then swapping its final states with its non-final states. For convenience we write $\mathcal{A}^{swap} = (Q, Q \setminus F, \Sigma, \delta)$ for the automaton obtained from $\mathcal{A} = (Q, F, \Sigma, \delta)$ by setting its final states to $Q \setminus F$.

Boolean Combination of Atomic Predicates – Let us begin with the proof of Proposition 3.3.

Proof of Proposition 3.3. Let P be a predicate and $\mathcal{A}_P$ its related deterministic and $\mathcal{R}$-complete automaton such that $\mathsf{L}(\mathcal{A}_P)_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P\}$ (see Section 3.6). Thus for each route t there exists a unique run that is accepting if and only if t satisfies P. Therefore, an automaton for the negation $\neg P$ of predicate P is simply obtained from $\mathcal{A}_P$ as the tree automaton $\mathcal{A}_P^{swap}$. Indeed we have $\mathsf{L}(\mathcal{A}_P^{swap})_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } \neg P\}$. The resulting automaton is again deterministic and $\mathcal{R}$-complete.

Suppose now that for two predicates P' and P'', we have two deterministic and $\mathcal{R}$-complete automata $\mathcal{A}_{P'}$ and $\mathcal{A}_{P''}$ such that $\mathsf{L}(\mathcal{A}_{P'})_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P'\}$ and $\mathsf{L}(\mathcal{A}_{P''})_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P''\}$." If the synchronized product of the automata $\mathcal{A}_{P'}$ and $\mathcal{A}_{P''}$ is used for performing the union operation, then the resulting automaton $\mathcal{A}_{P' \vee P''}$ is deterministic and $\mathcal{R}$-complete and $\mathsf{L}(\mathcal{A}_{P' \vee P''})_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P' \vee P''\}$. Similarly for the intersection operation.

The announced complexities follow. □

Equivalence Test of Routing Filters – We now proceed with the proof of Theorem 4.1.

Proof of Theorem 4.1. Let $\mathcal{L}$ be the set of trees $t_1 \circledast t_2$ such that

$$\exists t,\ (t, t_1) \in F_1 \text{ and } (t, t_2) \in F_2 \text{ and } t_1 \neq t_2.$$

The relations F_1 and F_2 are total functions on the set $\mathcal{R}$ of routes: for every $t \in \mathcal{R}$, there is a unique t' such that $(t, t') \in F_i$. Thus we have that $\mathcal{L} = \emptyset$ if and only if F_1 and F_2 are equivalent.

From the definition of $\mathcal{B}_{F_1}$ and $\mathcal{B}_{F_2}$, we have $\mathcal{L} = \{t_1 \circledast t_2 \mid \exists t \in \mathcal{R},\ t \circledast t_1 \in \mathsf{L}(\mathcal{B}_{F_1}) \wedge t \circledast t_2 \in \mathsf{L}(\mathcal{B}_{F_2}) \wedge t_1 \neq t_2\}$. We now build an automaton $\mathcal{A}$ recognizing $\mathcal{L}$ in time (and of size) $O(|\mathcal{B}_{F_1}| \cdot |\mathcal{B}_{F_2}|)$. As emptiness of tree automata is decidable in linear time, and a counterexample is constructed in case of non emptiness [6], this will prove the result. More precisely, the counterexample would give $t_1 \circledast t_2$ instead of t, but we easily adapt the algorithm to also store parts of t in the states.

The way we construct automaton $\mathcal{A}$ is in the same vein as for the composition operation (see Section 4.3).

First we $\diamond$-fill $\mathcal{B}_{F_i}$, for each $i \in \{1, 2\}$, *i.e.* we transform $\mathcal{B}_{F_i}$ to $\mathcal{B}_{F_i}^{\diamond}$ such that $\mathcal{B}_{F_i}$ accepts a tree t if and only if $\mathcal{B}_{F_i}^{\diamond}$ accepts it with an arbitrary number of labels $(\diamond, \diamond)$ at the bottom of branches path and com. This can be done in time $O(|\mathcal{B}_{F_1}| + |\mathcal{B}_{F_2}|)$ and increases the automata sizes by a constant factor of 2. Indeed, we only need one more state $q_\diamond$, and the following transitions:

- $() \xrightarrow{(\diamond,\diamond)} q_\diamond$

- $(q_\diamond) \xrightarrow{(\diamond,\diamond)_1} q_\diamond$ [6]

- $(q_\diamond) \xrightarrow{(a,b)_1} q$,

  for each transition $() \xrightarrow{(a,b)} q$ in $\mathcal{B}_{F_i}$ with $a, b \in \{\mathsf{path}, \mathsf{com}, \diamond\}$.

Then we build $\mathcal{A}$ from $\mathcal{B}^\diamond_{F_1}$ and $\mathcal{B}^\diamond_{F_2}$. For each run of $\mathcal{B}^\diamond_{F_1}$ on $t \circledast t_1$ and each run of $\mathcal{B}^\diamond_{F_2}$ on $t \circledast t_2$, with the same t, $\mathcal{A}$ has a run on $t_1 \circledast t_2$. A state of $\mathcal{A}$ is thus a pair $(q_1, q_2)_\checkmark$ (resp. $(q_1, q_2)_\perp$) where q_i is the state of the corresponding run of $\mathcal{B}^\diamond_{F_i}$, and $\checkmark$ (resp. $\perp$) indicates whether $t_1 = t_2$ (resp. $t_1 \neq t_2$). The final states of $\mathcal{A}$ are pairs $(q_1, q_2)_\perp$ such that each q_i is a final state in $\mathcal{B}^\diamond_{F_i}$. Let us illustrate transitions for labels of arity 1. Assume for instance that:

$$(p_1) \xrightarrow{(a,b)} q_1 \quad \text{and} \quad (p_2) \xrightarrow{(a,c)} q_2$$

are transitions in $\mathcal{B}^\diamond_{F_1}$ and $\mathcal{B}^\diamond_{F_2}$ respectively. Then, if $b = c$, we add the transition:

$$((p_1, p_2)_\checkmark) \xrightarrow{(b,c)} (q_1, q_2)_\checkmark$$

while, if $b \neq c$, we add the transition:

$$((p_1, p_2)_\checkmark) \xrightarrow{(b,c)} (q_1, q_2)_\perp$$

In both cases, we also propagate a detected difference:

$$((p_1, p_2)_\perp) \xrightarrow{(b,c)} (q_1, q_2)_\perp$$

Hence we only have to consider pairs of transitions, and this procedure runs in time (and yields an automaton $\mathcal{A}$ of size) $O(|\mathcal{B}^\diamond_{F_1}| \cdot |\mathcal{B}^\diamond_{F_2}|)$.

Notice that $\mathcal{A}$ does not exactly recognizes $\mathcal{L}$ since the automata $\mathcal{B}_{F_1}$ and $\mathcal{B}_{F_2}$ have been $\diamond$-filled. It could be $\diamond$-cleaned as explained in Section 4.3. However in this context, this is not necessary, because $\mathsf{L}(\mathcal{A}) = \emptyset$ iff $\mathcal{L} = \emptyset$. □

Note that the worst-case complexity of Theorem 4.1 can be avoided when equivalence fails. Indeed, the rule generation can be limited to accessible states, starting from leaf-rules. Hence, once a state $(q_1, q_2)_\perp$ is generated, with each q_i final state of $\mathcal{B}_{F_i}$, we know that filters are not equivalent.

7 Conclusion

In this paper, we have investigated the semantic of BGP routing filters, with the aim of determining whether two given filters are equivalent or not. We

[6]Given label (a, b) with arity 0, $(a, b)_1$ is a fresh label with arity 1.

have shown how this problem could be solved using tree automata theory. Our approach was as follows: routes were modeled as trees, and routing filters as tree automata. Testing the equivalence of two filters was then reduced to testing if their corresponding tree automata are equivalent. This is a classical operation in tree automata theory. Using this approach has the additional benefit that when two filters are not equivalent, the test generates a counterexample.

We have implemented our model in a fully-functional prototype. This tool takes as input BGP routing filters expressed in the Cisco IOS configuration language, generates corresponding tree automata and tests their equivalence. To make the tool of practical use, we had to enhance it with several optimizations. Most optimizations were brought to the model so as to reduce the size of automata and the running time of the equivalence test. The first optimization avoids an exponential blow-up at the level of predicates used in filters, by translating them into deterministic automata. The second optimization avoids another exponential blow-up by taking into account that routing filters are total functions. With a third optimization, we have tried to reduce the size of the constructed automata by using quasi-routes instead of routes and by limiting the ranges of values (ASNs, community values...) to be considered.

We used our prototype tool to conduct several experiments to assess the practical feasibility of our approach. We conducted our measurements on two different datasets. The first dataset was composed of routing filters coming from routers of a large European transit network. The second dataset contained synthetic filters that we generated to stress-test the scalability of our approach. The experimental results are promising. They show the efficiency of our approach and the interest of using tree automata theory in the context of routing filters.

References

[1] T. Bates, E. Chen, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). Internet Engineering Task Force (IETF), RFC 4456, 2006.

[2] T. Bates, Y. Rekhter, R. Chandra, and D. Katz. Multiprotocol extensions for bgp-4. Internet Engineering Task Force (IETF), RFC 2858, 2000.

[3] M. Caesar and J. Rexford. BGP routing policies in ISP networks. *IEEE Network*, 19:5–11, November 2005.

[4] Jérôme Champavère, Rémi Gilleron, Aurélien Lemay, and Joachim Niehren. Efficient inclusion checking for deterministic tree automata

and XML schemas. *Information and Computation*, 207(11):1181–1208, 2009.

[5] R. Chandra, P. Traina, and T. Li. BGP Communities Attribute. Internet Engineering Task Force (IETF), RFC 1997, 1996.

[6] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available online since 1997: `http://tata.gforge.inria.fr`, 2007. Revised October, 12th 2007.

[7] Nick Feamster and Hari Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *2nd Symp. on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[8] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[9] Haruo Hosoya. *Foundations of XML Processing: The Tree-Automata Approach*. Cambridge Press, 2010.

[10] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. Detecting Network-wide and Router-specific Misconfigurations through Data Mining, February 2009.

[11] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *In Proceedings of the ACM SIGCOMM Conference*, pages 3–16, 2002.

[12] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). Internet Engineering Task Force (IETF), RFC 4271, 2006.

[13] P. Traina, D. McPherson, and J. Scudder. Autonomous System Confederations for BGP. Internet Engineering Task Force (IETF), RFC 3065, 2001.