Reasoning on BGP Routing Filters using Tree Automata

Caroline Battaglia[a], Véronique Bruyère[a], Olivier Gauwin[b], Cristel Pelsser[c], Bruno Quoitin[a,*]

[a] *Computer Science Dept., UMONS, Place du Parc 20, B-7000 Mons,Belgium*
[b] *Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France*
[c] *Internet Initiative Japan (IIJ), Innovation Institute, Tokyo, Japan*

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.

Keywords: BGP, routing protocols, routing policies, tree automata

1. Introduction

The *Border Gateway Protocol* (BGP) [RLH06] 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 ASes to enforce their business relationships by controlling route redistribution and route selection. This in turns influences how the traffic flows in the Internet. ASes are motivated to control traffic flow as carrying traffic internally is costly and they are billed differently by the different neighboring ASes, with whom they have a business relationship, for sending traffic 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 ASes, to influencing the route selection in distant ASes, and to stop a DDoS attack, to name a few.

The configuration of BGP policies is complex and often source of errors [MWA02, FB05]. The implementation of a single policy is distributed among filters defined on 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

*Corresponding author. Tel.: +32 65 373448, Fax.: +32 65 373318

Email address: bruno.quoitin@umons.ac.be (Bruno Quoitin)

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. Several attempts have been made at providing tools to manipulate or generate correct BGP policy configurations [CGG+04, Int97, BFM+05, VH09].

In this paper, we investigate the semantic of BGP routing filters. We aim to determine if two BGP routing filters have the same semantic. That is, if given the same set of incoming routes, the two filters 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 routing filters configuration mistakes before committing a configuration change and thus prevent unnecessary traffic disruptions. It enables to push much further the work started by Griffin et al [GJR03], Feamster et al [FB05], by Le et al [LLW+09] and more recently by Perouli et al [PGM+12]. Identifying if two policies have the same effect enables network operators to check the correctness of routing filter configurations with regard to the high level policies they aim to enforce. Additionally, such a solution is useful for refactoring old BGP routing filter 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 [CDG+07], a powerful mathematical tool well-known for its applications in XML processing [Hos10], and program verification [FGVTT04]. 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.

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 applied to a route. Such predicates can easily and naturally be modeled by tree automata.

In Section 5, 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 6, we propose a prototype implementing this approach. We demonstrate the equivalence test on example Cisco IOS route-maps then we discuss additional routing filter verifications that could be provided by our tool in the long-term.

In Section 7, we describe multiple cases where our approach could be applied by network operators to perform sanity checks when deploying or updating routing filters distributed on multiple routers. We show the benefits of reasoning at the level of filters rather than at the level of routes.

In Section 8, we describe several experiments we performed with the prototype implementation. We present performance measurements as well as a study of the algorithmic complexity. Several

optimizations are brought to the prototype implementation to reduce its time and space complexity. Those optimizations are described in Section 9. 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. BGP Routing Policies

The Internet is an interconnection of several independent networks called *Autonomous Systems* (AS), each being uniquely identified by an *AS number* (`ASN`). The *Border Gateway Protocol* (BGP) is the de facto standard protocol used for routing among ASes.

To compute paths across the Internet, BGP routers need to exchange routing information. The basic unit of routing information in BGP is a *route* and its purpose is to announce the reachability of a remote destination. Although BGP can be used to advertise the reachability of several kinds of address families [BRCK00], in this paper we focus on IPv4 addresses. For this address family, destinations are announced in the prefix form. An IP prefix, expressed as a couple (address / prefix length) represents a set of contiguous IP addresses that share a common prefix. An example is `192.168.128.0/17` which represents the set of addresses that share their 17 most significant bits with `192.168.128.0`. In a route, we call `DST_PREFIX` the *attribute* that contains the destination prefix.

A BGP route associates a destination prefix `DST_PREFIX` with several *path attributes*. The most important path attributes are described in the following paragraphs.

- `AS_PATH`: records the `ASN`s of the ASes traversed by the route, ordered from the closest to the nearest. The `AS_PATH` attribute is used for loop detection as well as for ranking routes.
- `LOCAL_PREF`: used to give a route a preference that has a meaning local to the AS. The `LOCAL_PREF` attribute has a default value in every network. In the remaining of this paper, this default value is assumed to be 100.
- `NEXT_HOP`: identifies the router to which packets must be sent in order to follow this route.
- `MULTI_EXIT_DISC` (or `MED`): used by a neighbor AS to suggest which route should be preferred.
- `COMMUNITIES` [CTL96] : used to tag the route as being part of a group of routes that must undergo the same treatment. Each tag, named a *community value*, has a semantic that is usually local to an AS or to an AS and its direct neighbors. Some community values are defined with a global semantic by the standard.

Each attribute has a specific type which mandates how the attribute values are encoded in a route. The type of the above attributes are listed in Table 1.

Other attributes are defined by the BGP standard. We do not list them in Table 1 as they cannot be used in routing filters. Those attributes are `ORIGINATOR_ID`, `CLUSTER_LIST` used in conjunction with route-reflectors [BCC06], `ATOMIC_AGGREGATE` and `AGGREGATOR` used for route aggregation purposes. Moreover, the definition of sets (`AS_SET`) in the `AS_PATH` is also ignored as it is being deprecated by the IETF [KS11]. The attributes listed in this paragraph are ignored in the remaining of this paper. However, should those attribute appear in routing filters in the future, our model could easily be extended to support them.

2.1. 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 either to reject the route or to accept it after its attributes have possibly been modified. Filtering routes has several applications [CR05] from enforcing routing policies (rejecting routes that do not agree with

Attribute	Type
DST_PREFIX	Sequence of up to 32 bits (IPv4)
AS_PATH	Sequence of 16-/32-bits unsigned integers
LOCAL_PREF	Unsigned integer (32-bits)
NEXT_HOP	IPv4 address
MULTI_EXIT_DISC	Unsigned integer (32 bits)
COMMUNITIES	Set of 32-bits unsigned integers

Table 1: Type of BGP path attributes.

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 [RLH06] 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* $(R_1, \ldots, R_n)$ that are applied one after the other. Each rule $R = \langle P, A \rangle$ is composed of two parts: a *predicate* P and an *action* A. The predicate determines if the action applies to a route or not. A predicate is a Boolean combination of *atomic predicates* where each tests a single attribute of the route. The action is a sequence of *atomic actions* where each modifies a single attribute of the route. The action is applied to the route when the predicate matches the route.

An atomic predicate tests a single path attribute. 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. These predicates are syntactic sugars for more complex combinations of the above atomic predicates.

An atomic action modifies a single path attribute. Table 3 shows the most common atomic actions. Special actions can be used in a filter to accept or reject a route. When such action is used, the filter processing stops and the remaining filter rules are not applied.

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

We show in Figure 1 a short BGP routing filter expressed in the syntax of Cisco IOS along with an

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 `ASN`s, 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 `DST_PREFIX` is equal to the IP prefix x.
Destination inclusion	*dst_in*(x)	True iff `DST_PREFIX` is included into the IP prefix x.

Table 2: List of the most common atomic predicates.

Name	Action	Description
Absolute preference	*pref_set*(x)	Set `LOCAL_PREF` value to x.
Relative preference	*pref_add*(x)	Add x to the `LOCAL_PREF` value. If the new value is larger than $2^{32}-1$, the `LOCAL_PREF` value is set to $2^{32}-1$.
	pref_sub(x)	Subtract x from the `LOCAL_PREF` value. If the new value is smaller than 0, the `LOCAL_PREF` value is set to 0 .
Path prepending	*path_prepend*(x)	Add the `ASN` x at the beginning of the `AS_PATH`.
Community membership	*comm_add*(x)	Add a *community value* x to the `COMMUNITIES`. If x is already part of the `COMMUNITIES`, this action has no effect.
	comm_remove(x)	Remove a *community value* x from the `COMMUNITIES`. If the *community value* x is not part of the `COMMUNITIES`, this action has no effect.
	comm_clear()	Empty the `COMMUNITIES`.
Next-hop update	*nh_set*(x)	Set `NEXT_HOP` value to IP address x.
Absolute MED	*med_set*(x)	Set the `MULTI_EXIT_DISC` value to value x.
Acceptance	*accept*()	Accept the route.
Rejection	*reject*()	Reject the route.

Table 3: List of the most common atomic actions.

example Java code that expresses the same filter in our prototype tool.

2.2. 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 2.1 can be seen as a total function associating with each route $r \in \mathcal{R}$ another route $r' \in \mathcal{R}$, together with a mode in $\{\mathsf{acc}, \mathsf{rej}\}$ that indicates if the route is accepted or rejected by the filter.

Equivalence. Two routing filters F_1 and F_2 are *equivalent* if and only if, for all routes, their results are equal, i.e. $F_1 \equiv F_2$ iff F_1 and F_2 define the same function. Two routes are equal if all their attributes are equal.

The above definition of the equivalence of 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 mainly depends on the size of $\mathcal{R}$. Testing the equivalence of routing filters with the above naive algorithm is clearly not practical. We provide a comparison between our approach and the naive algorithm in Section 8.2.

In this paper, we aim at providing a novel method for testing the equivalence directly *at the level of filters rather than at the level of routes.* To achieve this objective, we model

1. **routes with trees**. A tree is just a mean of encoding the values of all the attributes of a route.
2. **predicates with tree automata**. A tree automaton that models a predicate recognizes only the trees corresponding to routes satisfying the predicate.
3. **actions/filters with tree relation automata**. Actions and filters are binary relations that map a route to its image. Hence, we model actions and filters with automata that recognize a binary tree relation, that is a set of pairs of trees (a tree and its image by the relation).

The equivalence of routing filters can therefore be reduced to testing the equivalence of automata, a standard operation in Automata Theory [HU79].

It is important to note that with the proposed automata approach, a routing filter is modeled by an automaton as a relation (that maps routes to routes), and routing filters are tested to be equivalent via relations encoded as automata. The proposed test is thus not done at the level of routes but rather at the level of relations.

Other Problems. Let us mention some other related problems. When two routing filters F_1 and F_2 have been declared as not being equivalent, we could be interested to have a witness of non-equivalence, that is, a route leading to two different results by F_1 and F_2. More generally, it could be interesting to know the set of all (instead of one) witnesses of non-equivalence of two filters.

Another interesting problem is to be able to test whether or not a subset of routes satisfying a given property (for instance, routes including community 1) is transformed by a filter into a subset of routes satisfying another property (for instance, routes with local-pref value 150).

We will see in this paper that these problems can also been solved using Automata Theory, following the same approach as for the equivalence test of two filters.

3. Tree Automata

In this section, we provide the tree automata background required to fully understand the paper. We first explain what is a tree and how it can be used to encode a complex structure. Second, we recall the notion of tree automaton and illustrate it with examples. We also make a parallel between tree automata and more classical word automata. Third, we introduce two tree automata properties that are important for our model, namely *determinism* and *completion.* We illustrate these properties with examples. Finally, we explain Boolean operations on tree automata.

```
ip as-path access-list 1 permit _10_
ip as-path access-list 2 deny _10_
ip community-list 1 permit 20

route-map RM1 permit 10
  match as-path 1
  set community 20 additive
  set local-preference 200

route-map RM2 permit 20
  match as-path 2
  match community 1
  set community none
```

```
List<IFilterRule> rules = new ArrayList<FilterRule>();

final IPredicate inPath = new PathIn(10);
final List<IAction> actions1 = new ArrayList<IAction>();
actions1.add(new ComAdd(20));
actions1.add(new Accept());
rules.add(new FilterRule(inPath, actions1));

final IPredicate notInPath = new PredicateNot(inPath);
final IPredicate inComm = new CommIn(20);
final List<IAction> actions2 = new ArrayList<IAction>();
actions2.add(new ClearCommunities());
actions2.add(new Accept());
rules.add(new FilterRule(new PredicateAnd(notInPath, inComm),
                         actions2));

final List<IAction> actions3 = new ArrayList<IAction>();
actions3.add(new Reject());
rules.add(new FilterRule(null, actions3));

Filter myFilter = new Filter(rules);
```

Figure 1: Cisco IOS route-map and Java code for constructing the corresponding filter.

Those operations are required to model Boolean operations on filter predicates. These operations are also at the heart of the classical automata equivalence test.

3.1. Trees

We consider *ranked trees*, *i.e.* trees where the number of children of a node is fixed by its label. Ranked trees are useful for encoding complex, structured data such as a route composed of multiple attributes.

Let *alphabet* Σ be the finite set of *labels* that can appear in a tree. Let also ar be a function mapping each label $a \in \Sigma$ to a positive integer $\mathsf{ar}(a)$ called its *arity*. The value $\mathsf{ar}(a)$ gives the number of children of a node with label a. For convenience, we write Σ_n for the set of labels of arity n: $\Sigma_n = \{a \in \Sigma \mid \mathsf{ar}(a) = n\}$. A node labeled by $a \in \Sigma$ is called an *a-node*.

We note $a(t_1, \ldots, t_n)$ the tree rooted at a with n subtrees t_1 to t_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. A *tree language* is a subset of T_Σ.

Let us illustrate these definitions. Consider the alphabet $\Sigma^{\mathrm{abcd}} = \{a, b, c, d\}$ where $\mathsf{ar}(a) = \mathsf{ar}(b) = 2$, $\mathsf{ar}(c) = 1$ and $\mathsf{ar}(d) = 0$. In other words, $\Sigma_2^{\mathrm{abcd}} = \{a, b\}$, $\Sigma_1^{\mathrm{abcd}} = \{c\}$ and $\Sigma_0^{\mathrm{abcd}} = \{d\}$. The tree $a(d)$ does not belong to $\mathsf{T}_{\Sigma^{\mathrm{abcd}}}$, 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 $\mathsf{T}_{\Sigma^{\mathrm{abcd}}}$. It is depicted in Figure 2.

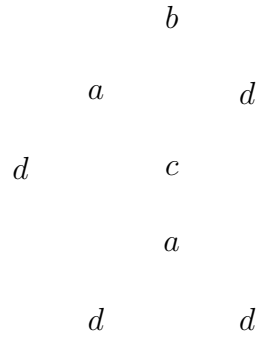


Figure 2: A tree $t \in \mathsf{T}_{\Sigma^{\mathrm{abcd}}}$.

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}. The role of a tree automaton is to recognize trees with a given structure.

Tree Automaton. 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 $|Q|$.

Run. 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 $(\rho(\pi_1), \ldots, \rho(\pi_n)) \xrightarrow{a} \rho(\pi) \in \delta$ where π_i is the i^{th} child of node π.

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 π of the tree is assigned to a final state, *i.e.* $\rho(\pi) \in F$. 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.

Recognizable Language. 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.

Equivalence. Two tree automata are *equivalent* if they recognize the same language.

3.3. Example

To illustrate the concept of a tree automaton, let us take a simple example. Consider the alphabet $\Sigma^{\text{abc}} = \{a, b, c\}$. The arity function is defined as $\mathsf{ar}(a) = \mathsf{ar}(b) = 2$, $\mathsf{ar}(c) = 0$.

Suppose we want to build an automaton that recognizes the language $\mathcal{L}_{\text{ac}}$ composed of trees over the alphabet Σ^{abc} that have at least one branch where an a-node is parent of a c-node.

We propose the tree automaton $\mathcal{A}_{\text{ac}} = (Q, F, \Sigma^{\text{abc}}, \delta)$ with $Q = \{q_c, q_{ac}, q_\perp\}$. State q_c is assigned to a c-node. State q_{ac} is assigned to a node π if and only if it belongs to a branch that contains an a-node parent of a c-node. State $q_\perp$ is assigned in every other case. There is a single final state; $F = \{q_{ac}\}$. The transitions in δ are as follows:

$$() \xrightarrow{c} q_c$$

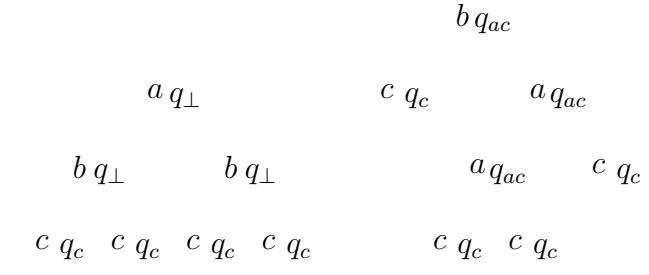
$$\begin{array}{lll} (q_c, q_c) \xrightarrow{b} q_\perp & (q_c, q_{ac}) \xrightarrow{b} q_{ac} & (q_c, q_\perp) \xrightarrow{b} q_\perp \\ (q_{ac}, q_c) \xrightarrow{b} q_{ac} & (q_{ac}, q_{ac}) \xrightarrow{b} q_{ac} & (q_{ac}, q_\perp) \xrightarrow{b} q_{ac} \\ (q_\perp, q_c) \xrightarrow{b} q_\perp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

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

A b-node is assigned state q_{ac} if and only if at least one of its child nodes was assigned q_{ac}. In every other case a b-node is assigned state $q_\perp$.

An a-node is assigned state q_{ac} if and only if at least one of its child nodes was assigned q_{ac} or q_c. If all child nodes are assigned $q_\perp$, then state $q_\perp$ is assigned to the a-node.

Figure 3 shows a run of $\mathcal{A}_{\text{ac}}$ on two different trees. The run in Figure 3a is non-accepting as the tree does not contain an a-node parent of a c-node. The state assigned to the root node, $q_\perp$ is not a final state. The run in Figure 3b is accepting. Indeed, this tree belongs to the language of the automaton, $\mathcal{L}(\mathcal{A}_{\text{ac}})$.



(a) Non-accepting run (b) Accepting run

Figure 3: Two runs of $\mathcal{A}_{\text{ac}}$.

3.4. Relation to Word Automata

Tree automata are related to more classical word automata. Tree structures subsume words, that is every word can be considered as a tree. For example, a word $a_1 a_2 \cdots a_n$ can be considered as a tree $a_n(a_{n-1}(\ldots a_1(\mathsf{nil})))$, so that a word is

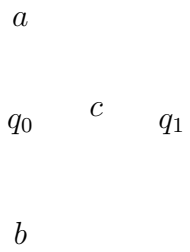


Figure 4: Word automaton recognizing $(a|b)^*c$.

mapped to a branch. We consider that each word label has arity 1 when used in the tree alphabet. A special label nil of arity 0 is also added to the tree alphabet. Note that the ordering of labels in the tree is reversed compared to that of the word. This is due to the bottom-up processing of tree automata.

Such 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 $q \xrightarrow{a} q'$. 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 [HU79] for more details.

It is also interesting to note 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 4 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 $q_0 \xrightarrow{c} q_1$.

3.5. Tree Automata Properties

Some operations on tree automata that are useful in this paper, can be realized much more efficiently when the tree automata satisfy some properties: *determinism* and *completeness*.

Determinism. 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 [CDG+07]. However, the determinization procedure is exponential in time, and yields automata of exponential size.

Completeness. 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 $t \in \mathcal{L}$. Therefore, if $\mathcal{A}$ is deterministic and $\mathcal{L}$-complete, there is exactly one run of $\mathcal{A}$ on every $t \in \mathcal{L}$.

An automaton $\mathcal{A}$ is *complete* if there is at least one transition for every left-hand side $(q_1, \ldots, q_n) \xrightarrow{a}$ where $\mathsf{ar}(a) = n$. If an automaton is complete, it is also T_Σ-complete.

Every tree automaton can easily be turned into an equivalent complete automaton by adding a (non-final) sink state q^* and transitions going to it $(q_1, \ldots, q_n) \xrightarrow{a} q^*$, for every left-hand side $(q_1, \ldots, q_n) \xrightarrow{a}$ missing in δ. We name this operation *completion*.

3.6. Example Revisited

The automaton $\mathcal{A}_{\text{ac}}$ defined in Section 3.3 is deterministic as there is a single transition for each left-hand side. The automaton is also complete as there is a transition for every possible left-hand side. In this section, we provide a non-deterministic automaton $\mathcal{A}'_{\text{ac}}$ that recognizes the same language $\mathcal{L}_{\text{ac}}$ as $\mathcal{A}_{\text{ac}}$. Recall that $\mathcal{L}_{\text{ac}}$ is the set of trees over Σ^{abc} that have at least one branch where an a-node is parent of a c-node.

To build such an automaton, let us first imagine that the automaton can guess a branch of the tree where the a-node is parent of a c-node, and then check it. Let us call this branch β. Note that the action of guessing the branch is a pure vision of the mind. The automaton is really an algebraic object and there is no reason to ask how it can guess the branch.

A run of the automaton $\mathcal{A}'_{\text{ac}}$ assigns state $q_\perp$ to every node that is not on β. On β, the automaton uses states q_c and q_{ac} to memorize that it has seen respectively a c-node or an a-node above a c-node.

State q_{ac} is the unique final state. The transitions of the automaton are as follows:

$() \xrightarrow{c} q_c$
$() \xrightarrow{c} q_\perp$

$(q_\perp, q_\perp) \xrightarrow{b} q_\perp \quad (q_\perp, q_{ac}) \xrightarrow{b} q_{ac} \quad (q_{ac}, q_\perp) \xrightarrow{b} q_{ac}$

$(q_\perp, q_\perp) \xrightarrow{a} q_\perp \quad (q_c, q_\perp) \xrightarrow{a} q_{ac} \quad (q_\perp, q_c) \xrightarrow{a} q_{ac}$
$(q_\perp, q_{ac}) \xrightarrow{a} q_{ac} \quad (q_{ac}, q_\perp) \xrightarrow{a} q_{ac}$

If the automaton guessed the wrong branch, then there is a b-node above a c-leaf which has been assigned to q_c. As no transition exists for this case, there cannot be a corresponding run for this guess.

Figure 5 shows an accepting run of $\mathcal{A}'_{\text{ac}}$ on the same tree as in Figure 3b. The branch β that has been guessed by the automaton is shown with thick lines. Every node outside the branch is mapped to state $q_\perp$. Note that there are two other accepting runs of $\mathcal{A}'_{\text{ac}}$ for this tree as there are two other branches that contain an a-node parent of a c-node.

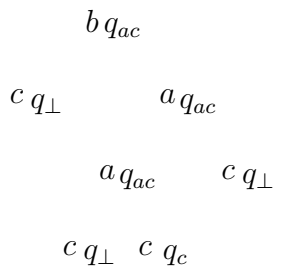


Figure 5: A run of $\mathcal{A}'_{\text{ac}}$.

The automaton $\mathcal{A}'_{\text{ac}}$ is non-deterministic. This can be observed from the transitions labeled with c. There is one for the leaf in the β branch and the other one for the leaves outside the branch. As a consequence, if a tree has more than one branch that satisfies the property checked by $\mathcal{A}'_{\text{ac}}$, then there can be multiple accepting runs for this tree. The automaton $\mathcal{A}'_{\text{ac}}$ is $\mathcal{L}_{ac}$-complete as there is at least one run for every t in $\mathcal{L}_{ac}$. However, $\mathcal{A}'_{\text{ac}}$ is not complete as there is no transition for some left-hand sides, like for the case $(q_c, q_c) \xrightarrow{a}$.

3.7. 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 $\mathsf{L}(\mathcal{A}_1) \cup \mathsf{L}(\mathcal{A}_2)$. This result is folklore [CDG+07], 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 has a size exponential in the size of the original automaton.

When the initial automata are deterministic and complete, better complexities occur for the complementation operation, as indicated in the next proposition. In this proposition, we consider the more general situation of automata that are deterministic and $\mathcal{L}$-complete. Given a tree automaton $\mathcal{A}$, we use notation $\mathsf{L}(\mathcal{A})_{|\mathcal{L}} = \mathsf{L}(\mathcal{A}) \cap \mathcal{L}$ to restrict the language of $\mathcal{A}$ to $\mathcal{L}$.

Proposition 3.2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two automata that are deterministic and $\mathcal{L}$-complete. Then one can construct automata $\mathcal{A}'_1$, $\mathcal{A}'_2$ and $\mathcal{A}'_3$ that are again deterministic and $\mathcal{L}$-complete, and such that $\mathsf{L}(\mathcal{A}'_1)_{|\mathcal{L}} = \mathcal{L} \setminus \mathsf{L}(\mathcal{A}_1)_{|\mathcal{L}}$, $\mathsf{L}(\mathcal{A}'_2)_{|\mathcal{L}} = \mathsf{L}(\mathcal{A}_1)_{|\mathcal{L}} \cap \mathsf{L}(\mathcal{A}_2)_{|\mathcal{L}}$, and $\mathsf{L}(\mathcal{A}'_3)_{|\mathcal{L}} = \mathsf{L}(\mathcal{A}_1)_{|\mathcal{L}} \cup \mathsf{L}(\mathcal{A}_2)_{|\mathcal{L}}$ respectively. Moreover $\mathcal{A}'_1$ can be constructed in time $O(|\mathcal{A}_1|)$ and with the same size as $\mathcal{A}_1$, and $\mathcal{A}'_2$, $\mathcal{A}'_3$ can be constructed in time $O(|\delta_1| \cdot |\delta_2|)$ and with size $O(|\mathcal{A}_1| \cdot |\mathcal{A}_2|)$.*

Let us give some insights about this result. As each given automaton $\mathcal{A}_i$, $i = 1, 2$, is deterministic and $\mathcal{L}$-complete, there exists a unique run for each tree $t \in \mathcal{L}$. This run is either accepting or rejecting depending on whether t belongs to $\mathsf{L}(\mathcal{A}_i)_{|\mathcal{L}}$ or not. Therefore, an automaton $\mathcal{A}'_1$ such that $\mathsf{L}(\mathcal{A}'_1)_{|\mathcal{L}} = \mathcal{L} \setminus \mathsf{L}(\mathcal{A}_1)_{|\mathcal{L}}$ is simply obtained from $\mathcal{A}_1$ by swapping its final states with its non-final states. The resulting automaton is deterministic and $\mathcal{L}$-complete. For the intersection and union operations, we use the synchronized product (as mentioned above) of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ to get automata $\mathcal{A}'_2$ and $\mathcal{A}'_3$ respectively. The announced complexities follow.

In this proposition, it is stated that the automaton for the intersection and the union operations 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 Σ.[1]

Thanks to Theorem 3.1, 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 [CDG+07]:

Theorem 3.3. *Equivalence of tree automata is decidable.*

This test is in exponential time if automata are non-deterministic [CDG+07], and in polynomial time otherwise [CGLN09].

4. Modeling Routes and Predicates

4.1. Model of a Route

We recall from Section 2 that a route is composed of the attributes `DST_PREFIX`, `AS_PATH`, `LOCAL_PREF`, `NEXT_HOP`, `MULTI_EXIT_DISC`, `COMMUNITIES` and of a status indicating if the route is still modifiable or definitely accepted or rejected by the routing filter.

A route can be modeled as a tree as shown in Figure 6. This tree has a root labeled by label route of arity 5. This node is the parent of five branches: the first four branches model some attributes and the last one models the status. It is easy to support additional attributes in the tree model of a route by adding new branches under the root node.

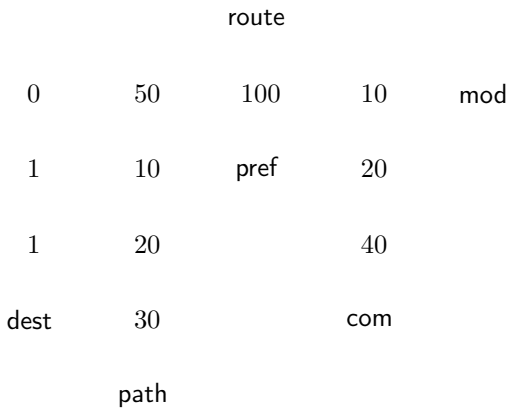


Figure 6: Tree modeling a route.

The branches shown in Figure 6 correspond to the next four attributes: a sequence of integer values (`AS_PATH`), a set of integer values (`COMMUNITIES`), a single integer (`LOCAL_PREF`) and a bitstring (`DST_PREFIX`). For clarity reasons, we choose to not present the `MULTI_EXIT_DISC` and `NEXT_HOP` in the paper as the type and the actions that can be applied to these attributes are similar to that of `LOCAL_PREF` and `DST_PREFIX` respectively.

The structure of the five branches is described in the following paragraphs along with their specific alphabet of labels of arity 1.

- dest branch: models the destination prefix (`DST_PREFIX`) written in binary, using alphabet $\Sigma^{\mathsf{dest}} = \{0, 1\}$. The most significant bit is at the bottom. For example, the route modeled on Figure 6 has the 192.0.0.0/3 destination prefix. The branch is ended by leaf dest. This leaf label is required as a tree automaton proceeds bottom-up and needs to identify on which branch it is working.

[1] We just need to store the transitions in a data structure where transitions using a given symbol of Σ are retrieved in constant time. Then we loop over all symbols of the alphabet Σ and consider pairs of transitions in $\delta_1 \times \delta_2$ using each symbol.

- path branch: models the sequence of ASNs (AS_PATH) 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 5.2). The branch uses alphabet $\Sigma^{\mathsf{path}} = [0, 2^{16}-1]$ whose labels represent 16-bit ASNs. The branch is ended with leaf path.

- pref branch: models the local preference (LOCAL_PREF). It uses a label of alphabet $\Sigma^{\mathsf{pref}} = [0, 2^{32}-1]$. The branch is ended with leaf pref.

- com branch: models the set of *community values* (COMMUNITIES) as a sorted sequence with the least number at the top of the branch. This branch uses $\Sigma^{\mathsf{com}} = [0, 2^{32}-1]$ whose labels represent communities. The branch is ended with leaf com.

- *status* branch: indicates the status of the route: either mod (modifiable), acc (accepted), or rej (rejected).

The underlying alphabet $\Sigma^{\mathcal{R}}$ used to describe routes as trees is thus decomposed as follows: $\Sigma_5^{\mathcal{R}} = \{\mathsf{route}\}$, $\Sigma_4^{\mathcal{R}} = \Sigma_3^{\mathcal{R}} = \Sigma_2^{\mathcal{R}} = \emptyset$, $\Sigma_1^{\mathcal{R}} = \Sigma^{\mathsf{dest}} \cup \Sigma^{\mathsf{path}} \cup \Sigma^{\mathsf{pref}} \cup \Sigma^{\mathsf{com}}$, and $\Sigma_0^{\mathcal{R}} = \{\mathsf{dest}, \mathsf{path}, \mathsf{pref}, \mathsf{com}, \mathsf{mod}, \mathsf{acc}, \mathsf{rej}\}$.

Although the alphabet $\Sigma^{\mathcal{R}}$ as defined at this stage is quite large, in Section 9.2, we show that only parts of the alphabets Σ^{dest}, Σ^{path}, Σ^{pref} and Σ^{com} are to be considered, depending on the routing filters submitted for equivalence. This observation will be important for performance reasons.

4.2. 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 show in Section 4.4 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 $i \in \Sigma^{\mathsf{com}}$. Its number of states can be large as the number of transitions required to check the ordering in the com branch is quadratic in the size of the com alphabet. If $|\Sigma^{\mathsf{com}}| = n$, there are $\frac{n(n-1)}{2} + n + 1$ transitions of type 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. $() \xrightarrow{\mathsf{dest}} q_0, \ () \xrightarrow{\mathsf{path}} q_0, \ () \xrightarrow{\mathsf{pref}} q_0, \ () \xrightarrow{\mathsf{com}} q_0,$ $() \xrightarrow{\mathsf{mod}} q_0, \ () \xrightarrow{\mathsf{acc}} q_0, \ () \xrightarrow{\mathsf{rej}} q_0,$