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

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.

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

```
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.

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 differ-

ent 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. 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 $\Sigma^{\text{abcd}} = \{a, b, c, d\}$ where $\mathsf{ar}(a) = \mathsf{ar}(b) = 2$, $\mathsf{ar}(c) = 1$ and $\mathsf{ar}(d) = 0$. In other words, $\Sigma_2^{\text{abcd}} = \{a, b\}$, $\Sigma_1^{\text{abcd}} = \{c\}$ and $\Sigma_0^{\text{abcd}} = \{d\}$. 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 2.

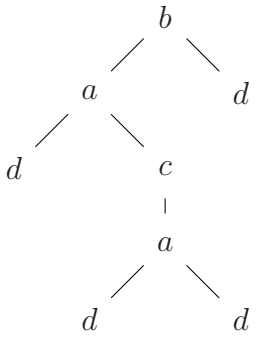


Figure 2: A tree $t \in \mathsf{T}_{\Sigma^{\text{abcd}}}$.

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 $\Sigma^{\mathrm{abc}} = \{a, b, c\}$. 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 $\mathcal{L}_{\mathrm{ac}}$ 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 $\mathcal{A}_{\mathrm{ac}} = (Q, F, \Sigma^{\mathrm{abc}}, \delta)$ 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:

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

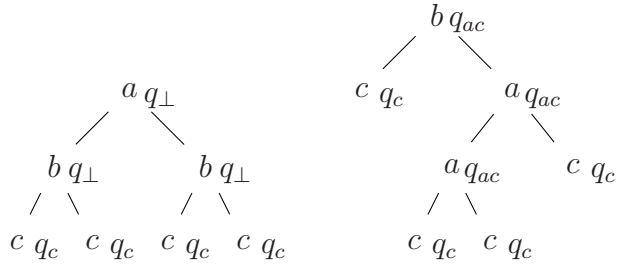
$$\begin{array}{lll} (q_c, q_c) \xrightarrow{b} q_\perp & (q_c, q_{ac}) \xrightarrow{b} q_{ac} & (q_c, q_\perp) \xrightarrow{b} q_\perp \\ (q_{ac}, q_c) \xrightarrow{b} q_{ac} & (q_{ac}, q_{ac}) \xrightarrow{b} q_{ac} & (q_{ac}, q_\perp) \xrightarrow{b} q_{ac} \\ (q_\perp, q_c) \xrightarrow{b} q_\perp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

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

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 $\mathcal{A}_{\mathrm{ac}}$ 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, $\mathcal{L}(\mathcal{A}_{\mathrm{ac}})$.



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

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

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 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.

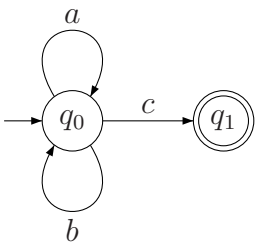


Figure 4: Word automaton recognizing $(a|b)^*c$.

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.

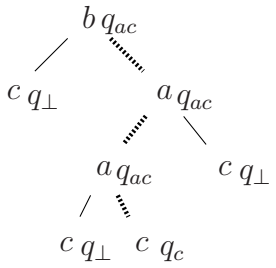


Figure 5: A run of $\mathcal{A}'_{\text{ac}}$.

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.

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.

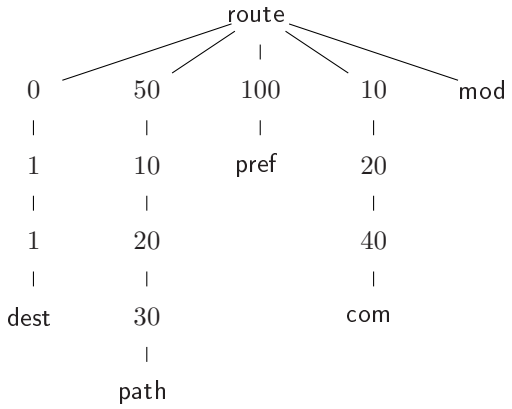


Figure 6: Tree modeling a route.

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.

- path branch: models the sequence of `ASN`s (`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 `ASN`s. 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

[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.

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: $\Sigma^{\mathcal{R}}_5 = \{\mathsf{route}\}$, $\Sigma^{\mathcal{R}}_4 = \Sigma^{\mathcal{R}}_3 = \Sigma^{\mathcal{R}}_2 = \emptyset$, $\Sigma^{\mathcal{R}}_1 = \Sigma^{\mathsf{dest}} \cup \Sigma^{\mathsf{path}} \cup \Sigma^{\mathsf{pref}} \cup \Sigma^{\mathsf{com}}$, and $\Sigma^{\mathcal{R}}_0 = \{\mathsf{dest}, \mathsf{path}, \mathsf{pref}, \mathsf{com}, \mathsf{mod}, \mathsf{acc}, \mathsf{rej}\}$.

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. $() \xrightarrow{\mathsf{dest}} q_1$, $(q_1) \xrightarrow{i} q_1$, $i \in \Sigma^{\mathsf{dest}}$,
2. $() \xrightarrow{\mathsf{path}} q_2$, $(q_2) \xrightarrow{i} q_2$, $i \in \Sigma^{\mathsf{path}}$,
3. $() \xrightarrow{\mathsf{pref}} q_3$, $(q_3) \xrightarrow{i} q'_3$, $i \in \Sigma^{\mathsf{pref}}$,
4. $() \xrightarrow{\mathsf{com}} q_4$, $(q_4) \xrightarrow{i} q_{4,i}$, $i \in \Sigma^{\mathsf{com}}$,
   $(q_{4,j}) \xrightarrow{i} q_{4,i}$, $i, j \in \Sigma^{\mathsf{com}}$ with $j > i$,
5. $() \xrightarrow{\mathsf{mod}} q_5$, $() \xrightarrow{\mathsf{acc}} q_5$, $() \xrightarrow{\mathsf{rej}} q_5$,
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 $\Sigma^{\mathcal{R}}_1 = \Sigma^{\mathsf{dest}} \cup \Sigma^{\mathsf{path}} \cup \Sigma^{\mathsf{pref}} \cup \Sigma^{\mathsf{com}}$ 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. $() \xrightarrow{\mathsf{dest}} q_0$, $() \xrightarrow{\mathsf{path}} q_0$, $() \xrightarrow{\mathsf{pref}} q_0$, $() \xrightarrow{\mathsf{com}} q_0$,
   $() \xrightarrow{\mathsf{mod}} q_0$, $() \xrightarrow{\mathsf{acc}} q_0$, $() \xrightarrow{\mathsf{rej}} q_0$,
2. $(q_0) \xrightarrow{i} q_0$, $i \in \Sigma^{\mathcal{R}}_1$,
   $(q_0, q_0, q_0, q_0, q_0) \xrightarrow{\mathsf{route}} q_f$.

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. Proposition 3.2 from Section 3.7 is an important property that we will use with $\mathcal{L} = \mathcal{R}$, when modeling predicates. We show in Section 5.5 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 restricting 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 9.

4.3. Filters seen as Tree Automata

We recall that a routing filter F is composed of a sequence of rules $(R_1, \dots R_n)$. Each rule $R = \langle P, A \rangle$ is composed of a predicate P which is a Boolean combination of atomic predicates, and of an action A which is a sequence of atomic actions. The problem studied in this article is the equivalence of two routing filters. We translate this problem to an equivalence test between two tree automata (one for each filter).

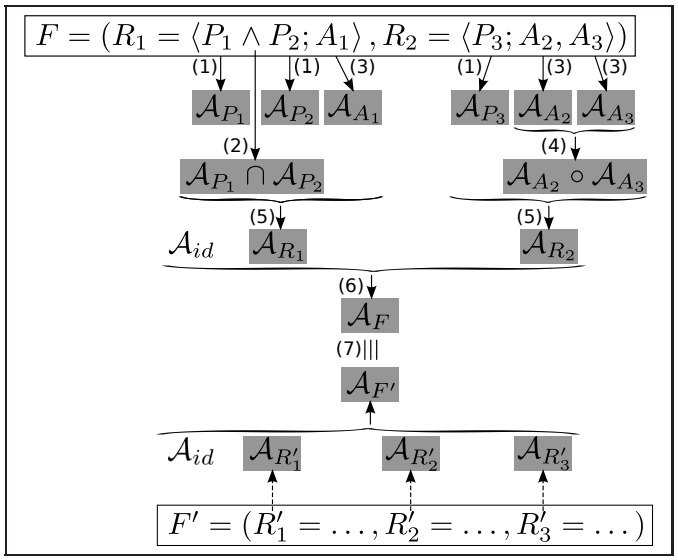


Figure 7: General approach for modeling.

The main ideas of our approach are depicted in Figure 7 and briefly described in the following paragraphs (the next sections detail the constructions). The numbers between parentheses that appear in Figure 7 refer to the list items below.

1. Each atomic predicate appearing in the predicate of a rule is modeled by a tree automaton that accepts quasi-routes (seen as trees t) satisfying the atomic predicate (see Section 4.4).
2. By Proposition 3.2, each Boolean combination of atomic predicates can be modeled by a tree automaton (see Section 4.5).
3. Each atomic action appearing in the action of a rule is modeled by a tree automaton that accepts the pairs of quasi-routes (t, t') such that t' is the image of t by the atomic action (see Section 5.2).
4. Each sequence of atomic actions, is modeled by an automaton obtained by *composition* of the automata of the atomic actions (see Section 5.3).
5. Each rule $R = \langle P, A \rangle$ is also modeled by a tree automaton that accepts pairs of quasi-routes (t, t') as follows: if t satisfies predicate P, then t' is the image of t by action A, otherwise $t' = t$. This automaton can be constructed from the automata for P and A thanks to the composition operation (see Section 5.4).
6. Finally, a routing filter $F = (R_1, \dots, R_n)$, is modeled by a tree automaton obtained by composing the automata of rules R_i. The resulting automaton is also composed with an automaton $\mathcal{A}_{id}$ that only accepts routes.
7. Two filters F, F' are equivalent if their corresponding automata are equivalent (see Theorem 3.3 and Section 5.5).

4.4. Model of an Atomic Predicate

The most important atomic predicates used in routing filters have been described in Table 2. We show in this section that each of those atomic predicates can be modeled by a tree automaton. As mentioned in Section 4.2, we make an atomic predicate automaton simpler by considering quasi-routes instead of routes. To this end, to model atomic predicate P, we build an automata $\mathcal{A}_P$ whose language is such that $\mathsf{L}(\mathcal{A}_P)_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t \text{ satisfies } P\}$. This automata can be obtained by using the same transitions as in $\mathcal{A}_{quasi\mathcal{R}}$, except for the branch concerned with predicate P. Moreover, we aim at building tree automata that are deterministic and $\mathcal{R}$-complete.

Path Membership. 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 following tree automaton $\mathcal{A}_{\mathsf{pm}}$ accepts all quasi-routes with a `path` branch that contains label x. The transitions are the following ones:

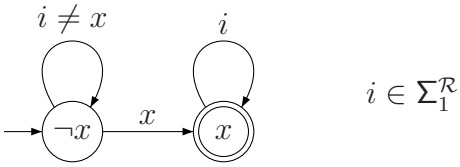


Figure 8: Word automaton recognizing words containing x.

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

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 8. 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.

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 atomic predicate. This is necessary to get a correct modeling of rules (see Section 5.5) and to optimize the modeling of predicates (see Section 9.3).

Remaining Atomic Predicates. Let us now consider the other atomic predicates P. For predicate of community membership, the treatment of

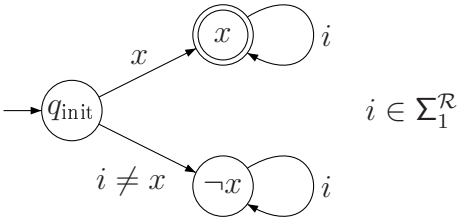


Figure 9: Word automaton recognizing words starting with x.

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 9.

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.

Regular Expressions. 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 [HU79]. 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. $() \xrightarrow{\mathsf{dest}} q_0$, $() \xrightarrow{\mathsf{pref}} q_0$, $() \xrightarrow{\mathsf{com}} q_0$, $() \xrightarrow{\mathsf{mod}} q_0$, $() \xrightarrow{\mathsf{acc}} q_0$, $() \xrightarrow{\mathsf{rej}} q_0$, $(q_0) \xrightarrow{i} q_0$, $i \in \Sigma_1^{\mathcal{R}}$,
2. $() \xrightarrow{\mathsf{path}} q_{init}$, $(p) \xrightarrow{i} q$, with transition $p \xrightarrow{i} q$ in δ,
3. $(q_0, q, q_0, q_0, q_0) \xrightarrow{\mathsf{route}} q_\top$, with $q \in F$, $(q_0, q, q_0, q_0, q_0) \xrightarrow{\mathsf{route}} q_\bot$, 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.

4.5. Model of a Predicate

A predicate P is a Boolean combination of atomic predicates P_i, $1 \leq i \leq n$. We showed in Section 4.4 how to build a deterministic and $\mathcal{R}$-complete tree automaton $\mathcal{A}_i$ for every atomic predicate P_i. In this section, we show that it is possible to build an automaton that models P, the Boolean combination of atomic predicates P_i, thanks to Proposition 3.2.

Let P_1 and P_2 be two predicates, and $\mathcal{A}_1$ and $\mathcal{A}_2$ their respective deterministic and $\mathcal{R}$-complete automata. Notice that

$$\begin{aligned}\{t \in \mathcal{R} \mid t \text{ satisfies } \neg P_1\} &= \mathcal{R} - \{t \in \mathcal{R} \mid t \text{ satisfies } P_1\} \\ &= \mathcal{R} - \mathsf{L}(\mathcal{A}_1)_{|\mathcal{R}}\end{aligned}$$

and[2]

$$\begin{aligned}\{t \in \mathcal{R} \mid t \text{ satisfies } (P_1 \wedge P_2)\} &= \{t \in \mathcal{R} \mid t \text{ satisfies } P_1\} \\ &\quad \cap \{t \in \mathcal{R} \mid t \text{ satisfies } P_2\} \\ &= (\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2))_{|\mathcal{R}}\end{aligned}$$

Therefore, by Proposition 3.2, one can build a deterministic and $\mathcal{R}$-complete automaton for $\neg P_1$, $P_1 \wedge P_2$, and $P_1 \vee P_2$.

More generally, by repeating this process, one can construct a deterministic and $\mathcal{R}$-complete automaton $\mathcal{A}_P$ modeling a predicate P that is a Boolean combination of atomic predicates $P_i, 1 \leq i \leq n$.

[2] A similar equality holds for the disjunction of the two predicates.

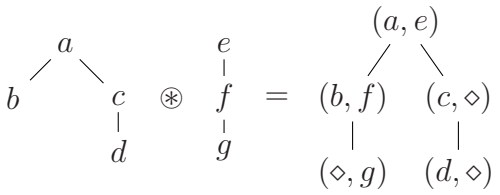


Figure 10: Example of overlay.

5. 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).

5.1. 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 10. 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 $a_1, \ldots, a_n$ the children of the a-root in t_1, and $b_1, \ldots, b_p$ 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 $(a_1, b_1), \ldots, (a_p, b_p), (a_{p+1}, \diamond), \ldots, (a_n, \diamond)$. 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 [CDG+07] 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.

5.2. 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 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, route acceptance, and route rejection.

We recall that a route t is modeled as a tree such that the last branch indicates the status of the route: mod, acc or rej. An atomic action should leave unchanged any route that has a status equal to acc or rej.

In order to have small automata for atomic actions, we are going to construct them on quasi-routes instead of routes, as we did for predicates in Sections 4.4 and 4.5. However, 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 status). 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 *pref_ add*(x) that adds 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. In the remaining of the discussion, we will consider an action as a tree relation. Let A_{rp} be a tree relation that transforms t in t' according to *pref_ add*(x). 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. $() \xrightarrow{(\mathsf{dest},\mathsf{dest})} q_0$, $() \xrightarrow{(\mathsf{path},\mathsf{path})} q_0$,
   $() \xrightarrow{(\mathsf{pref},\mathsf{pref})} q_0$, $() \xrightarrow{(\mathsf{com},\mathsf{com})} q_0$,
   $(q_0) \xrightarrow{(i,i)} q_0,\ i \in \Sigma_1^{\mathcal{R}}$,
2. $() \xrightarrow{(\mathsf{mod},\mathsf{mod})} q_{\mathrm{mod}}$,
   $() \xrightarrow{(\mathsf{acc},\mathsf{acc})} q_{\mathrm{fix}}$, $() \xrightarrow{(\mathsf{rej},\mathsf{rej})} q_{\mathrm{fix}}$,
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. $(q_0, q_0, q_{+x}, q_0, q_{\mathrm{mod}}) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f$,
   $(q_0, q_0, q_0, q_0, q_{\mathrm{fix}}) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f$.

Transitions 1 are used by the automaton $\mathcal{A}_{\mathsf{rp}}$ to check the identity relation on the dest, path and com branches. The identity is also checked for the pref branch in case the status of the route is acc or rej. Notice that a single state q_0 is enough to check identity as the automaton works on quasi-routes.

Transitions 2 memorize in state q_{mod} (resp. q_{fix}) whether the status of trees t, t' is mod (resp. acc, rej).

Transitions 3 apply the action of relative preference with states q_1 and q_{+x}. Note that non-determinism appears as there are two transitions with left-hand side $() \xrightarrow{(\mathsf{pref},\mathsf{pref})}$: either the identity relation is checked on the pref branch with transitions 1, or the action of relative preference is performed with transitions 3.

Finally, depending on the status, 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 *comm_ add*(x) that adds a community value x to the sorted sequence of communities. If x is already present, it is not added. Let A_{cm} be a tree relation that transforms t in t' according to *comm_ add*(x). 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 has one final state q_f and the following transitions:

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

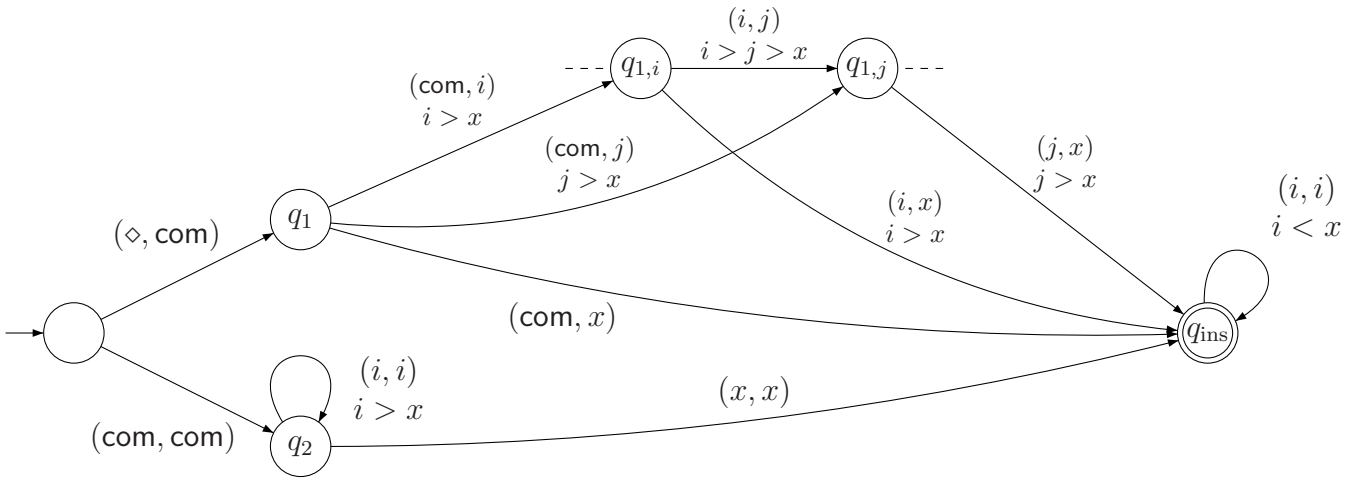


Figure 11: Transitions 3 of automaton $\mathcal{A}_{\mathsf{cm}}$ recognizing relation A_{cm}.

$() \xrightarrow{(\mathsf{pref},\mathsf{pref})} q_0, \quad () \xrightarrow{(\mathsf{com},\mathsf{com})} q_0$
$(q_0) \xrightarrow{(i,i)} q_0,\ i \in \Sigma_1^{\mathcal{R}},$

2. $() \xrightarrow{(\mathsf{mod},\mathsf{mod})} q_{\mathrm{mod}},$
$() \xrightarrow{(\mathsf{acc},\mathsf{acc})} q_{\mathrm{fix}}, \quad () \xrightarrow{(\mathsf{rej},\mathsf{rej})} q_{\mathrm{fix}},$
3. $() \xrightarrow{(\diamond,\mathsf{com})} q_1, \quad (q_1) \xrightarrow{(\mathsf{com},x)} q_{\mathrm{ins}},$
$(q_{\mathrm{ins}}) \xrightarrow{(i,i)} q_{\mathrm{ins}},\ i \in \Sigma^{\mathsf{com}}, x > i,$
$(q_1) \xrightarrow{(\mathsf{com},i)} q_{1,i},\ i \in \Sigma^{\mathsf{com}}, i > x,$
$(q_{1,j}) \xrightarrow{(j,i)} q_{1,i},\ i,j \in \Sigma^{\mathsf{com}}, j > i > x,$
$(q_{1,j}) \xrightarrow{(j,x)} q_{\mathrm{ins}},\ j \in \Sigma^{\mathsf{com}}, j > x,$
$() \xrightarrow{(\mathsf{com},\mathsf{com})} q_2,$
$(q_2) \xrightarrow{(i,i)} q_2,\ i \in \Sigma^{\mathsf{com}}, i > x,$
$(q_2) \xrightarrow{(x,x)} q_{\mathrm{ins}},$
4. $(q_0, q_0, q_0, q_{\mathrm{ins}}, q_{\mathrm{mod}}) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f,$
$(q_0, q_0, q_0, q_0, q_{\mathrm{fix}}) \xrightarrow{(\mathsf{route},\mathsf{route})} q_f.$

Transitions 1 check the identity relation on the dest, path and pref branches. Transitions 2 memorize if the status of the route is mod or acc/rej. Transitions 3 check the correct insertion of x in the com branch if the status of t, t' is mod. Non-determinism appears on the level of the com branch, depending on the current status.

Transitions 3 need some explanations. They are illustrated in Figure 11. 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. This corresponds to the middle path in Figure 11; (2) x is not the largest value and it is properly inserted (state $q_{1,j}$ remembers the last seen value j). This corresponds to the top path; (3) x is already present and is therefore not inserted. This corresponds to the bottom path.

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 Acceptance. We now consider the atomic action *accept*(). Let A_{ra} be a tree relation that transforms t in t' according to *accept*(). Except for the status, the corresponding automaton $\mathcal{A}_{\mathsf{ra}}$ checks for identity between t and t'. In case of mod status for t, it checks for acc status for t'. In case of acc or rej status for t, it checks that the status is left unchanged for t'. Automaton $\mathcal{A}_{\mathsf{ra}}$ has one final state q_f and the following transitions:

1. $() \xrightarrow{(\mathsf{dest},\mathsf{dest})} q_0, \quad () \xrightarrow{(\mathsf{path},\mathsf{path})} q_0,$
$() \xrightarrow{(\mathsf{pref},\mathsf{pref})} q_0, \quad () \xrightarrow{(\mathsf{com},\mathsf{com})} q_0,$
$(q_0) \xrightarrow{(i,i)} q_0,\ i \in \Sigma_1^{\mathcal{R}},$
2. $() \xrightarrow{(\mathsf{mod},\mathsf{acc})} q_1,$
$() \xrightarrow{(\mathsf{acc},\mathsf{acc})} q_1, \quad () \xrightarrow{(\mathsf{rej},\mathsf{rej})} q_1,$

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

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

5.3. 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 the sequel we abuse notations by writing $(t, t') \in A$ whenever action A transforms route t to the route t'.

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:

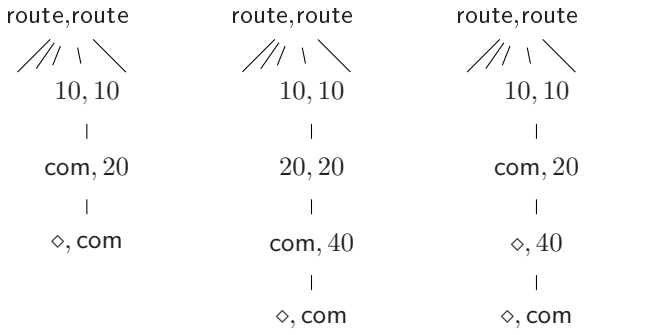
$$\mathcal{A}_A = (((\mathcal{A}_{A_1} \circ \mathcal{A}_{A_2}) \circ \mathcal{A}_{A_3}) \circ \cdots \circ \mathcal{A}_{A_n})$$

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 trees $t \circledast t'$ 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 possibly 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.



(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.

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 overlays. 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 overlay of Figure 12c. This is not the case: in Figure 12c, we can observe that a transition $() \xrightarrow{(\diamond,\mathsf{com})} (q, q')$ is needed. According to the construction process described above, this transition is part of $\mathcal{A}_A \circ \mathcal{A}_{A'}$ if there exists a label c such that transition $() \xrightarrow{(\diamond,c)} q$ exists in $\mathcal{A}_A$ and transition $() \xrightarrow{(c,\mathsf{com})} q'$ exists in $\mathcal{A}_{A'}$. A transition $() \xrightarrow{(\diamond,\mathsf{com})} q'$ exists in $\mathcal{A}_{A'}$, but the counterpart $() \xrightarrow{(\diamond,\diamond)} q$ does not exist 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[3] 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.

[3]This modification is rather simple, and detailed in the proof of Theorem 5.1.

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 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 $\{t \circledast t' \mid t, t' \in \mathcal{R}$ and $(t, t'') \in A, (t'', t') \in A'$ 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 $\mathsf{L}(\mathcal{A}_A))_{|\mathcal{R}\circledast\mathcal{R}} = \{t \circledast t' \mid t, t' \in \mathcal{R}$ and $(t, t') \in A\}$.

5.4. Modeling 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 the previous sections how to build a tree automaton $\mathcal{A}_P$ (resp. $\mathcal{A}_A$) for predicate P (resp. action A). We recall that $\mathsf{L}(\mathcal{A}_P)_{|\mathcal{R}} = \{t \in \mathcal{R} \mid t$ satisfies $P\}$, and $\mathsf{L}(\mathcal{A}_A)_{|\mathcal{R}\circledast\mathcal{R}} = \{t \circledast t' \mid t, t' \in \mathcal{R}$ 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 $\mathcal{A}_A$. This procedure is illustrated in Figure 13. When restricted to routes, $\mathcal{A}_R$ must accept exactly all $t \circledast t'$ such that t' is the image of t by A if t satisfies P, and $t' = t$ otherwise. This corresponds to the language

$$\begin{aligned}\mathsf{L}(\mathcal{A}_R)_{|\mathcal{R}\circledast\mathcal{R}} = \{\, t \circledast t' \mid\ & t, t' \in \mathcal{R} \text{ and} \\ & ((t \text{ satisfies } P \text{ and } (t, t') \in A) \text{ or} \\ & (t \text{ does not satisfy } P \text{ and } t = t'))\}\end{aligned}$$

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.

As $\mathcal{A}_P$ does not recognize a tree relation, the idea is to turn $\mathcal{A}_P$ into an automaton $\mathcal{B}_P$ recognizing a relation that "marks" a tree t when it does not satisfy P, and lets t unmarked otherwise. Marking t consists in replacing its root label route with a new label $\overline{\text{route}}$, and is denoted by $\overline{t}$. Let $\mathcal{R}'$ be the set $\mathcal{R} \cup \{\overline{t} \mid t \in \mathcal{R}\}$. The language of the needed automaton $\mathcal{B}_P$ is such that $\mathsf{L}(\mathcal{B}_P)_{|\mathcal{R}\circledast\mathcal{R}'} = \{t \circledast t \mid t$ satisfies $P\} \cup \{t \circledast \overline{t} \mid t$ does not satisfy $P\}$. This approach is illustrated in Figure 13.

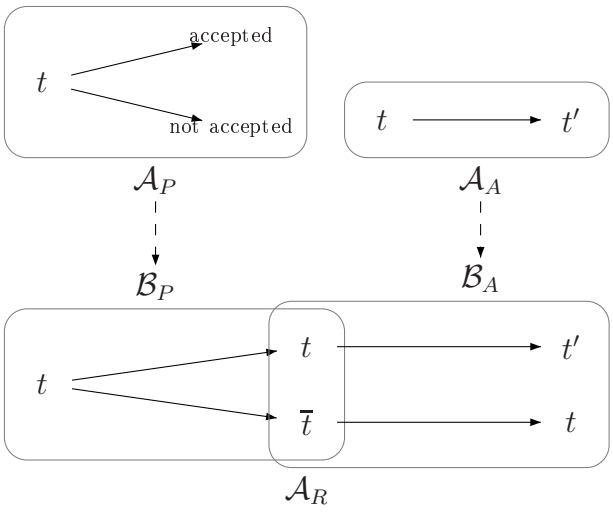


Figure 13: Composing predicate P with action A.

Similarly, we derive automaton $\mathcal{B}_A$ from automaton $\mathcal{A}_A$ such that $\mathcal{B}_A$ recognizes a relation that turns a marked tree into a unmarked tree, and turns an unmarked tree into a tree transformed by action A. Formally, $\mathsf{L}(\mathcal{B}_A)_{|\mathcal{R}'\circledast\mathcal{R}} = \{\overline{t}\circledast t \mid t \in \mathcal{R}\} \cup \{t \circledast t' \mid t, t' \in \mathcal{R}$ and $(t, t') \in A\}$.

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 Sections 4.4 and 4.5). Then we build $\mathcal{B}_P = (Q, \{q_f\}, \Sigma^{\mathcal{R}} \times \Sigma^{\mathcal{R}}, \delta')$ with

- $() \xrightarrow{(a,a)} q \in \delta'$, if $() \xrightarrow{a} q \in \delta$,
- $(p) \xrightarrow{(a,a)} q \in \delta'$, if $(p) \xrightarrow{a} q \in \delta$,

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

  $(p_1,p_2,p_3,p_4,p_5) \xrightarrow{(\mathsf{route},\overline{\mathsf{route}})} q_f \in \delta'$, 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 latter automaton two new states q_0 and q_f such that q_f is final, and the following transitions[4]:

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

5.5. Modeling 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. In the sequel we write $(t,t') \in F$ when F transforms route t into route t'. 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 (defined in Section 4.2). 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 3.3). 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 5.1. *Let F and F' be two filters. Let $\mathcal{B}_F$ and $\mathcal{B}_{F'}$ be their respective automata with $\delta_F, \delta_{F'}$ their sets of transitions. Then it can be decided in time $O(|\delta_F| \cdot |\delta_{F'}|)$ 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 9.3. Notice that this result also holds for any pair of tree relations (instead of filters) as long as they are total functions. Notice also that complexity $O(|\delta_F| \cdot |\delta_{F'}|)$ in the previous theorem can be replaced by $O(|\mathcal{B}_F|^k \cdot |\mathcal{B}_{F'}|^k \cdot |\Sigma|)$ where k is the maximal arity of Σ (see Footnote 1).

5.6. 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.
- 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

[4] These transitions are similar to the transitions of automaton $\mathcal{A}_{quasi\mathcal{R}}$ accepting quasi-routes (see Section 4.2).

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) state 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 by $\mathcal{A}_A$, 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 3.2)

- 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 such a way 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', before testing whether they are equivalent, the automata $\mathcal{A}_F$ and $\mathcal{A}_{F'}$ are modified into $\mathcal{B}_F$ and $\mathcal{B}_{F'}$ respectively in a way to treat routes, instead of quasi-routes. They are then tested for automata equivalence (see Theorem 3.3). The exponential blow-up of this test can be avoided in our context because the relations involved in filters are functional (see Theorem 5.1).

6. Prototype

We have implemented a prototype in Java, publicly available[5] under the GPLv2 licence. It implements all predicates and actions as presented in Section 2.2. Filters implementation is based on the model presented in this paper. The construction of the automata follows the inductive process described in the preceding sections, including the optimizations given in Section 9. Tree automata objects and standard operations are implemented inside a separate library, also publicly available.

We used a homemade parser to convert Cisco IOS configuration files into Java source code (see Figure 1 for an example). The parser only processes `route-maps`, `ip prefix-list`, `ip community-list` and `ip as-path access-list` clauses. Route-map `match` clauses are translated into a boolean combination of atomic predicates. Route-map `set` clauses are translated into sequences of atomic actions. Each route-map statement is converted to a single rule. Multiple route-map statements with the same identifier form a filter.

6.1. Example Run

To illustrate the operation of our prototype, this section shows how the equivalence of two Cisco IOS route-maps is tested. The two route-maps `F1` and `F2` are shown in Figure 14. Notice that even if they seem very similar, there is a slight difference.

If we provide those two route-maps to our tool, it will parse them, and produce Java code to build

[5] `https://github.com/bquoitin/eqrou`

```
ip prefix-list 1 seq 1 permit 128.0.0.0/16
ip prefix-list 1 seq 2 permit 128.1.0.0/16
ip prefix-list 1 seq 3 permit 128.2.0.0/16
ip prefix-list 1 seq 4 permit 128.3.0.0/16
!
ip community-list 1 permit 1:1
!
route-map F1 deny 10
  match community 1
!
route-map F1 permit 20
  match ip address prefix-list 1
  set local-preference 100
```

```
ip prefix-list 1 seq 1 permit 128.0.0.0/16
ip prefix-list 1 seq 2 permit 128.0.1.0/16
ip prefix-list 1 seq 3 permit 128.2.0.0/16
ip prefix-list 1 seq 4 permit 128.3.0.0/16
!
ip community-list 1 permit 1:1
!
route-map F2 deny 10
  match community 1
!
route-map F2 permit 20
  match ip address prefix-list 1
  set local-preference 100
```

Figure 14: Example IOS route-maps tested for equivalence.

the corresponding automata. Those automata are then tested for equivalence. Here, the filters are not equivalent as reported by the tool. The equivalence test took 340ms on a Intel Core 2 Duo processor running at 2.8GHz. The complete run took about 6 seconds, including parsing, generation of java code, compilation and execution.

As the filters are not equivalent, the tool produces a route that is a witness of non-equivalence. In this case, the route produced is (`DST_PREFIX` : 128.1.0.0/16, `AS_PATH` : {}, `LOCAL_PREF` : 100, `COMMUNITIES` : {}). Its image by `F1` is (`DST_PREFIX` : 128.1.0.0/16, `AS_PATH` : {}, `LOCAL_PREF` := 100, `COMMUNITIES` = {}) and the route is accepted. The image of this route by `F2` is the same but the route is rejected. This output is a good hint to track the cause of the difference between the two filters.

A route with the same attributes is accepted by `F1` and rejected by `F2`. We need to check the predicates used in the `permit` clause of our route-maps. Here the culprit is the second `ip prefix-list` statement where the permitted prefixes are different in `F1` and `F2`. Manually finding the reason why two routes are handled differently by two filters can still be difficult. In the future, the tool could be used to automatically pinpoint the rules responsible for accepting or rejecting a route.

6.2. Extensibility of the Approach

We have showed earlier how to model by automata the classical atomic predicates and actions used in routing filters. New atomic predicates can easily be incorporated, provided that they can be encoded by automata. This is true for instance for atomic predicates expressed by a regular expression as explained in Section 4.4. Similarly, new atomic actions can also be easily incorporated under the same hypothesis.

6.3. Beyond Equivalence

In the previous sections, we have explained how to test the equivalence of two filters using tools from tree automata theory. Some other related problems can also be tested with the same approach. We here list some of these problems and provide a rough idea how to solve them.

Witnesses of non-Universality. When two filters have been modeled by two automata, they are tested for equivalence by testing the equivalence of their related automata thanks to Theorem 5.1. When the filters are declared non-equivalent, it is useful to have a route that is a witness of such an non-equivalence, and more generally to have all the witnesses of non-equivalence. In the proof of Theorem 5.1 given in Section 9, we show how to construct one tree (witness) that has different images by the two filters when they are not equivalent. As a matter of fact, the proof can easily provide a tree automaton that exactly accepts all the routes that have different images by the two filters. Such an automaton modeling all the witnesses of non-universality can then be used to understand why the two filters are not equivalent. This will be explained in the next paragraph.

Behavior of a Filter under some Properties. Another interesting problem is to test whether or not a subset of routes satisfying a certain property is transformed by a filter into a subset of routes satisfying another property. For instance, we would like to test if the set of routes having community 1234 and destination included in 62.17/16 is transformed by a given filter so as to have local-preference 150 and a new community 5678. Such a problem can be solved using automata theory provided the two properties respectively imposed to the input routes and the output routes can be modeled by tree automata (this is the case of the previous example). Using standard automata properties such as in Section 3, it is possible to suitably combine the automaton modeling the filter with the two automata modeling the properties in a way to solve this problem.

Notice that such questions can also be asked (and similarly solved) about the set of witnesses of non-universality of two filters. Indeed we have explained above that this set of witnesses can be modeled by a tree automaton. We can thus have a better understanding of this set of routes by testing some properties (modeled by automata) on it. This method could help debug errors in filters.

7. Applications

In this section, we propose three applications of the equivalence test of two filters, illustrated by some examples. We start by motivating the need for an equivalence test as is, then move on to show some other, more complex, applications. We end with a longer-term application consisting of the composition and testing of distributed routing filters.

7.1. Redundant BGP Sessions in Multi-Vendor Networks

A common practice used by network operators to increase the robustness of their interdomain connectivity is to exchange routing information with neighbor AS over multiple redundant sessions. These often end on distinct physical routers. They may even terminate on routers from different vendors to decrease the risk of both routers being simultaneously affected by a common bug.

In such configurations, the routing filters deployed on both eBGP sessions are usually the same. However, checking that both routing filters are equivalent, up to now, was not a trivial task. Using equipment from different vendors also means writing routing filters using different configuration languages. Moreover, even in an environment where a single vendor is in use, configuration languages might differ among different versions of the vendor's operating system. Our approach provides a universal representation of the filters, a means to check their equivalence and to further reason about them.

In order to assess how often parallel sessions are deployed and how often routers from multiple vendors are in use, we analysed the configuration of all the routers in a large, modern, ISP network[6]. First, we determined for every neighbor AS x the number of different local routers peering with x. We observed that although 58% of AS are connected through a single session, the remaining 42% are connected using at least 2 sessions, as illustrated in Figure 15. This ISP has a neighbor AS that connects at as much as 9 different locations.

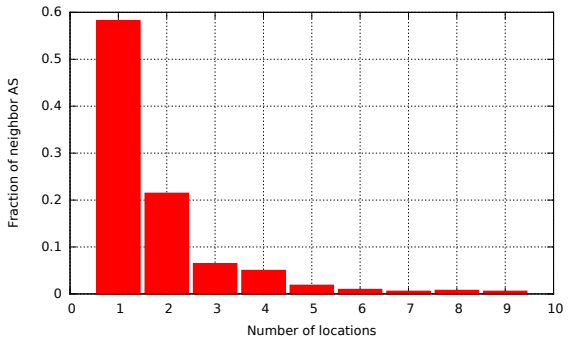


Figure 15: Number of peering locations for each neighbor AS in a large ISP.

In addition to this, 27% of the neighbor AS peer with routers that do not understand a single common configuration language. Those routers

[6] We cannot disclose the name of this ISP.

are either from different router vendors or from the same vendor but using versions of the operating system with different configuration languages (e.g. IOS versus IOS XR).

A network operator could use our tool to perform routing filters sanity check in the environment just described. For example, when new sessions are added or the business agreement and the routing policies with a neighbor AS change, the tool could be used to ensure the changes are deployed in the same way for all the sessions with that AS. Such verification could be done nightly.

7.2. Verifying Routing Filters

Today, network operators have no tool to check if a BGP routing filter works as intended. This cannot be done just by looking at the policy. Policies are often long and the semantic of the configuration languages is complex. Moreover, routers in a large network are not configured by a single person and different operators may configure a router differently to perform the same task.

Network operators will usually rely on the router operating system to check if a filter accepts, rejects or modifies a route as intended. This is typically done by injecting carefully crafted routes into a router (virtualized or in the lab) through a test session on which the filter applies and observe if the outcome of the route matches the intention. It should be noted that this approach is not practical if all routes are tested individually, as in the naive approach we described in Section 2.2. Instead of performing an exhaustive test, the network operator picks a few sample routes and limit its test to those routes, with the risk of missing corner cases of the filter that would cause an untested route to be mishandled.

As explained in Section 6.3, it is possible using our automata approach to verify if the image of one set of routes by a filter satisfies some property, provided this property can be expressed by an automaton. As an example, a network operator could want to check that all the routes with a destination prefix in some set will be rejected by the filter, or it could want to check that some communities have been added by the filter.

As this is a mid-term goal, our tool does not yet support the test of such properties. We plan to support this in the near future. It is likely to require some optimizations such as described in Section 9 to be practical. One can imagine that those properties could be expressed using languages similar to those provided by the router vendors. Moreover, the most usual properties could be pre-defined and available as libraries.

7.3. Composition of Distributed Routing Filters

Another possible application of our framework is the verification of a distributed routing filter. In a transit network, a route received through a session with a provider is typically redistributed to every customer. Most BGP sessions have inbound and outbound filters. This means a route going from one provider to a customer is processed by two different filters, defined on different routers, possibly using different languages : it is first processed by the inbound filter on the session with the provider and later by the outbound filter on the session with the customer.

The tree automata approach allows the inbound and outbound filters to be composed, resulting in a single automaton. Reasoning can then happen on this automaton: the verification of routing filters as described in Section 7.2 can be applied on it. For example, it would then be possible to check that a route transiting from a provider to a customer is marked with some community. It could also be used to verify that distributed routing filters correctly prevent some routes to be leaked from one session to another. For example, a route received from a provider is typically tagged with a special community value by the inbound filter. The outbound filter on a session with another provider should prevent a route tagged with this community value to be redistributed to another provider.

Composition of routing filters can also be used to check that the preference of a route (or a set of routes) always decreases, a property that is important for BGP to converge to a stable solution [Gri10]. This is important in the case of a confederation of ASs or when policies are applied on iBGP sessions [CBV10].

Checking the equivalence of routing filters is a key feature in being able to validate a network configuration, its changes and maintain the network in good operational shape.

8. Evaluation

8.1. Experiments

We used the prototype described in Section 6 to perform several experiments. Their results are presented in this section. Our experiments show the link between filter size, tree automaton 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 supported by a prior version of our tool (the current version supports 45 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 16 to 19. Note that the y axis is in logarithmic scale for all figures in this section. Note also that positive instances, i.e. those for which equivalence holds, are denoted by $+$, and negative instances by $\times$.

Figure 16 shows how the size of the resulting automaton varies with the filter size. We recall that the size $|\mathcal{A}|$ of an automaton $\mathcal{A}$ is the number of its states. Concerning the filters, 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 $|P|+|A|$ where $|P|$ is the number of atomic predicates of P and $|A|$ is the number of atomic actions of A. In Figure 16, 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 proved in Section 8.2.

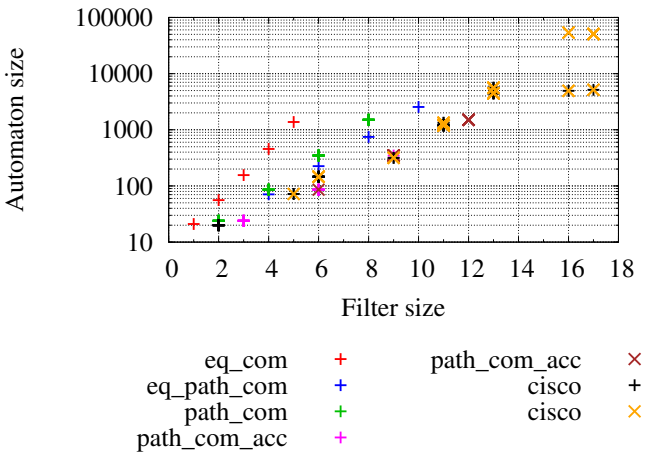


Figure 16: Size of automata.

Figure 17 shows how the time to build the automaton also follows this complexity, as described in Section 8.2. 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 16. As we will see, this is negligible compared to the time needed for testing equivalence.

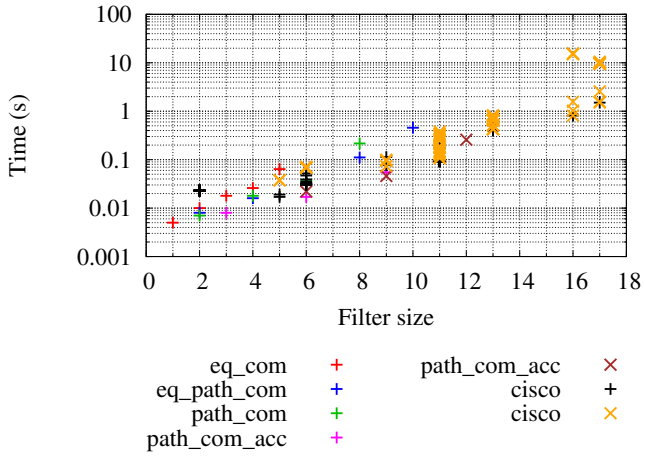


Figure 17: Time for building automata.

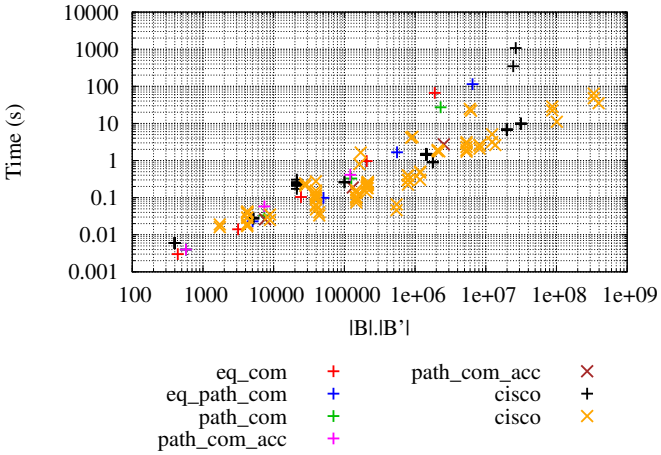


Figure 18: Testing equivalence from automata.

Figure 18 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. The figure suggests that there is a linear relationship (in logarithmic scale) between the product of the automata sizes and the running time of the algorithm, denoting a polynomial relationship between these quantities (without logarithmic scale). We state formally in Theorem 5.1 that testing the equivalence of two filters F and F' is in time $O(|\mathcal{B}_F|^k \cdot |\mathcal{B}_{F'}|^k \cdot |\Sigma|)$ when the input is the two automata $\mathcal{B}_F$ and $\mathcal{B}_{F'}$, and $k = 5$ is the maximal arity of Σ.[7]

The overall complexity of testing equivalence between two filters F and F' is thus a single exponential in the size of the filters, more precisely $O(p^{5(|F|+|F'|)} \cdot |\Sigma|)$ for a fixed p as explained in Section 8.2. Our experiments confirm this complexity, as depicted in Figure 19.

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 of an *accept*() or *reject*() action in every rule of the *cisco* filters that prevent further processing of the routes. Moreover, equivalence is generally faster on negative instances, as explained in the proof of Theorem 5.1.

[7]The alphabet Σ is either $\Sigma^{\mathcal{R}}$ or its reduction as explained in Section 9.2.

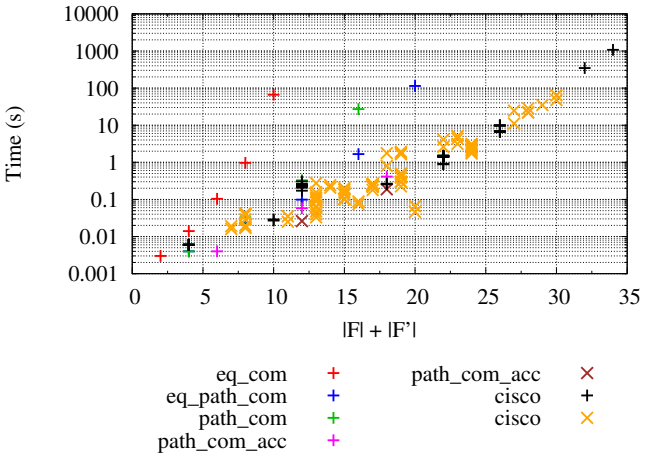


Figure 19: Testing equivalence from filters.

We also tried to compare our efficient equivalence test proposed in Theorem 5.1 with the usual equivalence test based on Boolean operations on tree automata (see Theorem 3.3). 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.

8.2. Complexity

In this section, we compare the complexity of our algorithm with the complexity of a naive algorithm.

Complexity of our algorithm. Let us evaluate the complexity of our algorithm. Consider the con-

stant p, defined as the maximal size among all atomic predicate and action automata sizes. Let us show that the automaton constructed for a filter F has size $O(p^{|F|})$ and can be constructed in time $O(p^{5|F|}\cdot|\Sigma|)$, and that testing the equivalence of two filters F, F' is done in time

$$O(p^{5(|F|+|F'|)} \cdot |\Sigma|)$$

by our algorithm.

First, when constructing the automaton $\mathcal{A}_P$ 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 3.2). Therefore the most costly operations are the union and intersection based on the synchronized product of two automata. In Proposition 3.2, it is stated that the automaton for the intersection and the union operations is built in time $O(|\delta_1| \cdot |\delta_2|)$, or equivalently in time $O(|\mathcal{A}_1|^5 \cdot |\mathcal{A}_2|^5 \cdot |\Sigma|)$ (see Footnote 1).

The worst case for automaton $\mathcal{A}_P$ is when the shape of P is $P = (((P_1\ o_1\ P_2)\ o_2\ P_3)\ o_3\ \cdots\ o_{n-1}\ P_n)$ where each P_i is an atomic predicate and each o_i is either $\vee$ or $\wedge$. In this case, the size of $\mathcal{A}_P$ is in $O(p^{|P|})$ and the time to construct it in is $O(((p^5)^2 + (p^5)^3 + \cdots + (p^5)^n) \cdot |\Sigma|)$, which is in $O(p^{5|P|} \cdot |\Sigma|)$.

Second, when constructing the automaton $\mathcal{A}_A$ 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 5.3). This results in a complexity $O(p^{|A|})$ for the size of $\mathcal{A}_A$ and $O(p^{5|A|} \cdot |\Sigma|)$ for the time to build it (with an argument similar to predicate P).

Third, the composition operation is also used for a rule and for a sequence of rules. Therefore for one rule $R = \langle P, A\rangle$, we get for automaton $\mathcal{A}_R$ a size in $O(p^{|P|+|A|}) = O(p^{|R|})$ and a time complexity to construct it in $O((p^{5|P|} + p^{5|A|} + p^{5(|P|+|A|)})\cdot|\Sigma|)$ which is in $O(p^{5|R|}\cdot|\Sigma|)$. For a filter $F = (R_1, \ldots, R_n)$, we get an automaton $\mathcal{B}_F$ of size $O(p^{|F|})$ in time $O((\Sigma_{i=1}^n p^{5|R_i|} + \Sigma_{i=2}^n p^{5(|R_1|+\cdots+|R_i|)})\cdot |\Sigma|) = O(p^{5|F|} \cdot |\Sigma|)$, yielding the announced complexities.

Finally, given two automata for two filters F, F', we can avoid a second exponential blow-up and test in time $O(p^{5(|F|+|F'|)} \cdot |\Sigma|)$ whether these filters are equivalent (see Theorems 3.3 and 5.1).

Complexity of a naive algorithm. A naive algorithm to test the equivalence of two filters F, F' consists in enumerating all the possible routes (up to a certain size) and to test if F, F' modify them into the same routes.

Let us evaluate the complexity of this algorithm. We make the hypothesis that during the application of a filter to a given route, constant time $O(1)$ is consumed by each atomic predicate (resp. action) of this filter. Therefore testing if two filters F, F' modify a given route into the same route can be performed in $O(|F| + |F'|)$. It remains to evaluate the total number of tested routes. We recall that such a route has four attributes: the `DST_PREFIX` of length bounded by l_{dest} (when written in binary), the `LOCAL_PREF` composed of one label, the `AS_PATH` of length bounded by l_{path} and the `COMMUNITIES` of length bounded by l_{com} (recall that the set `COMMUNITIES` is represented as a sorted sequence). The maximum prefix length l_{dest} equals 32 bits for IPv4. The length of the `AS_PATH` and `COMMUNITIES` is limited by the maximum size of a BGP message which is 4096 bytes. This constrains[8] l_{path} to remain below 2048 and l_{com} below 1024. We will use these bounds in our next complexity computations. We can also remember the status (accepted, rejected) of the route as given by one label. Concerning the possible values of `LOCAL_PREF`, and of the elements of `AS_PATH` and `COMMUNITIES`, we suppose that they are bounded by $c_{\mathsf{pref}} = 2^{32}$ (32-bit value), $c_{\mathsf{path}} = 2^{16}$ (16-bit `ASN`) and $c_{\mathsf{com}} = 2^{32}$ (32-bit values) respectively. Therefore, the total number of routes n_{routes} is bounded by the product $n_{\mathsf{dest}}\cdot n_{\mathsf{path}}\cdot n_{\mathsf{pref}}\cdot n_{\mathsf{com}}\cdot n_{\mathsf{stat}}$ of the numbers of attributes of each kind (including the status), such that

- $n_{\mathsf{dest}} = \sum_{i=0}^{l_{\mathsf{dest}}} 2^i = 2^{l_{\mathsf{dest}}+1} - 1$,

[8] This is a rough approximation as the message header, the destination prefix and other path attributes further limit these lengths.

- $n_{\mathsf{path}} = \sum_{i=0}^{l_{\mathsf{path}}} c_{\mathsf{path}}^i = c_{\mathsf{path}}^{l_{\mathsf{path}}+1}/(c_{\mathsf{path}} - 1)$,
- $n_{\mathsf{pref}} = c_{\mathsf{pref}}$,
- $n_{\mathsf{com}} = \sum_{i=0}^{l_{\mathsf{com}}} \binom{c_{\mathsf{com}}}{i}$ which can be bounded by $c_{\mathsf{com}}^{l_{\mathsf{com}}+1}/(c_{\mathsf{com}} - 1)$
- and $n_{\mathsf{stat}} = 2$.

It follows that the complexity of the naive algorithm is in

$$O(n_{routes}(|F| + |F'|)).$$

with n_{routes} in $O(2 \cdot 2^{l_{\mathsf{dest}}+1} \cdot c_{\mathsf{path}}^{l_{\mathsf{path}}} \cdot c_{\mathsf{pref}} \cdot c_{\mathsf{com}}^{l_{\mathsf{com}}})$. With the bounds given above, the quantity inside the O notation for n_{routes} is bounded by the huge number $2^{66} \cdot 2^{2^{16}}$.

Let us compare this complexity with our experimental results. For our largest instance with size $|F| + |F'| = 34$, our algorithm takes (in the pessimistic case) about 2^{10} seconds (see Figure 19). To be competitive, the naive algorithm has to treat $2^{66} \cdot 2^{2^{16}}$ routes in 2^{10} seconds, i.e. $2^{56} \cdot 2^{2^{16}}$ routes per second, which is far beyond what is possible. So our algorithm remains more efficient for reasonable filter sizes.

Note that our algorithm integrates the optimizations described in Section 9. The naive algorithm could also benefit from the reduction of the alphabets as described in Section 9.2. With this optimization, by inspecting the instance of size $|F| + |F'| = 34$, the previous constants c_{path}, c_{pref} and c_{com} decrease to the values $2, 1, 4$ respectively, and n_{routes} is now bounded by $2^{34} \cdot 2^{2^{12}}$. Nevertheless, the naive algorithm has to still treat the huge number of $2^{24} \cdot 2^{2^{12}}$ routes per second, to be competitive with our algorithm.

Moreover, the algorithm proposed in this paper is less dependent to changes in route models. For instance if we plan to implement IPv6 routes instead of IPv4 ones, the `DST_PREFIX` can contain 128-bit addresses instead of 32-bit ones. This will not change the way automata are built in our framework, but automata for $dst\_is(x)$ and $dst\_in(x)$ may grow by a linear factor, as the IP prefix x grows. On the opposite, the number of routes to be considered by the naive algorithm increases a lot, by a factor $2^{128-32} = 2^{96}$.

It should also be noted that adding a new attribute to the route model, that is, an additional branch to the trees, would multiply the complexity by a factor p.

9. Optimizations

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

9.1. Preprocessing Actions

When considering an action $A = (A_1, \ldots, A_n)$, 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 status. Hence A is equivalent to $(A_1, \ldots, A_i)$.

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 $pref\_sub(x)$ (see Table 3). 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.

9.2. 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 $\Sigma^{\mathcal{R}}$.

Let F and F' 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 and F' (with some refinement in the path branch).

Consider for instance the com branch. The atomic predicates possibly used by F and F' 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 Σ_F^{com} the set of labels a in Σ^{com} such that a appears in an atomic predicate/action of F or F', as $comm\_in(a)$, $comm\_add(a)$, or $comm\_remove(a)$.

If a is a community of route t such that $a \notin \Sigma_F^{com}$, 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 $a \in \Sigma_F^{com}$ in the com branch. Moreover the alphabet used by the automata on the com branch is reduced to Σ_F^{com} (instead of Σ^{com}).

The same kind of argument can be repeated for the path branch. In this case, we define by Σ_F^{path} the set of labels a in Σ^{path} such that a appears in an atomic predicate/action of F or F', 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 Σ_F^{path} a new label denoted by $\star$. This label is used at the place of each $a \in \Sigma^{path} \setminus \Sigma_F^{path}$ in the path branch. Contrarily to the com branch, we cannot forget any symbol not in Σ_F^{path}, due to predicate $path\_sub(s)$. Let us illustrate this with an example. Assume that F uses predicate $P = path\_sub(ab)$, and route t has a path branch $b(c(d(a(\mathsf{path}))))$ with $c, d \notin \Sigma_F^{path}$. 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 Σ_F^{path}.

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

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 values: 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.

9.3. Efficient Automata Operations

In this section, we come back to the automata operations used for testing equivalence of filters. In Theorems 3.1 and 3.3, the prohibitive (exponential) operation is the complementation of a non-deterministic automaton $\mathcal{A}$. In our context, instead of using these two theorems, we were able to use the more efficient counterparts given by Proposition 3.2 and Theorem 5.1.

Boolean Combination of Atomic Predicates. Proposition 3.2 shows how to avoid an exponential blow-up by working with automata that are deterministic and $\mathcal{L}$-complete. This approach has been applied to model predicates with tree automata (with $\mathcal{L} = \mathcal{R}$).

Equivalence Test of Routing Filters. As stated in Theorem 5.1, a second exponential blow-up has been avoided for the equivalence test of two tree automata, due to the functionality of filters. We here give the proof of this theorem.

Proof of Theorem 5.1. Let $\mathcal{L}$ be the set of trees $t_1 \circledast t_2$ such that

$$\exists t,\ (t, t_1) \in F, (t, t_2) \in F' \text{ and } t_1 \neq t_2.$$

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

From the definition of $\mathcal{B}_F$ and $\mathcal{B}_{F'}$, we have $\mathcal{L} = \{t_1 \circledast t_2 \mid \exists t \in \mathcal{R},\ t \circledast t_1 \in \mathsf{L}(\mathcal{B}_F) \wedge t \circledast t_2 \in \mathsf{L}(\mathcal{B}_{F'}) \wedge t_1 \neq t_2\}$. Let δ_F and $\delta_{F'}$ the respective sets of transitions of $\mathcal{B}_F$ and $\mathcal{B}_{F'}$. We now build

an automaton $\mathcal{A}$ recognizing $\mathcal{L}$ in time $O(|\delta_F| \cdot |\delta_{F'}|)$ and of size $O(|\mathcal{B}_F| \cdot |\mathcal{B}_{F'}|)$. As emptiness of tree automata is decidable in linear time, and a counterexample is constructed in case of non emptiness [CDG+07], this will prove the result.[9]

The way we construct automaton $\mathcal{A}$ is in the same vein as for the composition operation (see Section 5.3).

First we $\diamond$-fill $\mathcal{B}_F$, *i.e.* we transform $\mathcal{B}_F$ to $\mathcal{B}_F^\diamond$ such that $\mathcal{B}_F$ accepts a tree t if and only if $\mathcal{B}_F^\diamond$ accepts it with an arbitrary number of labels $(\diamond, \diamond)$ at the bottom of branches path and com. This procedure needs one more state $q_\diamond$, and the following transitions:

- $() \xrightarrow{(\diamond,\diamond)} q_\diamond$
- $(q_\diamond) \xrightarrow{(\diamond,\diamond)_1} q_\diamond$ [10]
- $(q_\diamond) \xrightarrow{(a,b)_1} q$, for each transition $() \xrightarrow{(a,b)} q$ in $\mathcal{B}_F$ with $a, b \in \{\mathsf{path}, \mathsf{com}, \diamond\}$.

We proceed similarly for $\mathcal{B}_{F'}$.

Then we build $\mathcal{A}$ from $\mathcal{B}_F^\diamond$ and $\mathcal{B}_{F'}^\diamond$. For each run of $\mathcal{B}_F^\diamond$ on $t \circledast t_1$ and each run of $\mathcal{B}_{F'}^\diamond$ 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. $(q_1, q_2)_\bot$) where q_1 (resp. q_2) is the state of the corresponding run of $\mathcal{B}_F^\diamond$ (resp. $\mathcal{B}_{F'}^\diamond$), and $\checkmark$ (resp. $\bot$) indicates whether $t_1 = t_2$ (resp. $t_1 \neq t_2$). The final states of $\mathcal{A}$ are pairs $(q_1, q_2)_\bot$ such that each q_1 (resp. q_2) is a final state in $\mathcal{B}_F^\diamond$ (resp. $\mathcal{B}_{F'}^\diamond$). 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 $\mathcal{B}_F^\diamond$ and $\mathcal{B}_{F'}^\diamond$ 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:

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

In both cases, we also propagate a previously detected difference:

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

Hence we only have to consider pairs of transitions, and the overall procedure (including the $\diamond$-filling of the automata) runs in time $O(|\delta_F| \cdot |\delta_{F'}|)$ and yields an automaton $\mathcal{A}$ of size $O(|\mathcal{B}_F| \cdot |\mathcal{B}_{F'}|)$.

Notice that $\mathcal{A}$ does not exactly recognizes $\mathcal{L}$ since the automata $\mathcal{B}_F$ and $\mathcal{B}_{F'}$ have been $\diamond$-filled. It could be $\diamond$-cleaned as explained in Section 5.3. However in this context, this is not necessary, because $\mathsf{L}(\mathcal{A}) = \emptyset$ iff $\mathcal{L} = \emptyset$. □

Remark that the worst-case complexity of Theorem 5.1 can be avoided when equivalence fails. Indeed, the rule generation can be limited to accessible states, starting from leaf-rules. Hence, once a state $(q_1, q_2)_\bot$ is generated, with q_1 (resp. q_2) a final state of $\mathcal{B}_F$ (resp. $\mathcal{B}_{F'}$), we know that filters are not equivalent.

We show in Figure 20 the code corresponding to this equivalence test. When filters are not equivalent, it generates a route (i.e. a tree) which is accepted by one filter but not by the other. It is part of the class `FilterAutomaton`, and takes a second `FilterAutomaton` as input, and a boolean indicating whether a counterexample should be built in case of non-equivalence.

We give a brief description of subroutines. Function `reachedStatesAtLeaves` returns the set of pairs $(q_1, q_2)_v$ (of Java type `EquivalenceState`, where v is a Boolean) that can be built from symbols of arity 0. These pairs initiate the saturation process. Function `equivalenceStateForRules` takes one rule of each automaton, and possibly returns a new pair $(q_1, q_2)_v$, as described in the proof. This adds new pairs to saturate. Function `filterState1` (resp. `filterState2`) returns q_1 (resp. q_2) when applied on pair $(q_1, q_2)_v$. Function `provesNonEquiv` tests whether a witness of non-equivalence has been found, i.e. whether the pair $(q_1, q_2)_v$ on which it is called is such that q_1 and q_2 are final and $v = \bot$.

[9] More precisely, the counterexample would give $t_1 \circledast t_2$ instead of t, but the algorithm can easily be adapted to identify t.

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

```
private ITerm<LabelPair> synthesizeSeparationTerm(FilterAutomaton otherFilter, boolean computeSepTerm){
    // the separation term is built by associating a term to each
    // equivalence term
    ITerm<LabelPair> sepTerm = null;
    Map<EquivalenceState,ITerm<LabelPair>> sepMap =
        new HashMap<EquivalenceState,ITerm<LabelPair>>();
    // first add diamond rules
    final FilterAutomaton automaton1 = this.addDiamondRules();
    final FilterAutomaton automaton2 = otherFilter.addDiamondRules();
    // then look for a counterexample to equivalence
    Set<EquivalenceState> agenda = new HashSet<EquivalenceState>();
    boolean equivalent = true;
    Set<EquivalenceState> reachedStates =
        reachedStatesAtLeaves(automaton1, automaton2, computeSepTerm, sepMap);
    agenda.addAll(reachedStates);
    while (equivalent && !agenda.isEmpty()) {
        final EquivalenceState state = agenda.iterator().next();
        agenda.remove(state);
        for (IRule<LabelPair,FilterState> rule1 :
            automaton1.getRulesUsingLeftState(state.filterState1())) {
            for (IRule<LabelPair,FilterState> rule2 :
                automaton2.getRulesUsingLeftState(state.filterState2())) {
                final EquivalenceState equivState =
                    equivalenceStateForRules(rule1, rule2, automaton1, automaton2, reachedStates,
                        computeSepTerm, sepMap);
                if (equivState != null) {
                    reachedStates.add(equivState);
                    agenda.add(equivState);
                    if (equivState.differs()) {
                        equivalent = !equivState.provesNonEquiv(automaton1, automaton2);
                        if (computeSepTerm && !equivalent) {
                            sepTerm = sepMap.get(equivState);
                        }
                    }
                }
            }
        }
    }
    if (!equivalent && !computeSepTerm) {
        // a dummy non-null term
        sepTerm = new Term<LabelPair>(
            this.getAlphabet(),
            new ArrayList<ITerm<LabelPair>>(),
            ActionAlphabet.REJREJ);
    }
    return sepTerm;
}
```

Figure 20: Java code for the equivalence test, returning a counterexample to equivalence when it exists, and null otherwise.

10. 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 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 performed these experiments 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.

Beyond equivalence, our modelisation allows to check properties of filters. Tree automata enjoy great expressiveness. We could check linear branches against regular expressions (for IP addresses for instance), but also express non-local properties, like: Do all accepted routes contain at least 3 communities in the com-branch, each time the pref-value is greater than 150? In this paper we only used a restricted part of tree automata theory. Many other innovative applications could arise from a deeper use of this theory.

Acknowledgments

We thank Laurent Vanbever from UCLouvain for taking part in the preliminary discussions on this work.

References

[BCC06] 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.

[BFM+05] Hagen Böhm, Anja Feldmann, Olaf Maennel, Christian Reiser, and Rüdiger Volk. Network-Wide Inter-Domain Routing Policies: Design and Realization. 34th North American Network Operator Group (NANOG) meeting, 2005.

[BRCK00] T. Bates, Y. Rekhter, R. Chandra, and D. Katz. Multiprotocol extensions for bgp-4. Internet Engineering Task Force (IETF), RFC 2858, 2000.

[CBV10] L. Cittadini, G. Di Battista, and S. Vissicchio. Doing Don'ts: Modifying BGP Attributes within an Autonomous System. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2010)*, 2010.

[CDG+07] 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.

[CGG+04] Don Caldwell, Anna Gilbert, Joel Gottlieb, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. The cutting EDGE of IP router configuration. *SIGCOMM Comput. Commun. Rev.*, 34(1):21–26, January 2004.

[CGLN09] 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.

[CR05] M. Caesar and J. Rexford. BGP routing policies in ISP networks. *IEEE Network*, 19:5–11, November 2005.

[CTL96] R. Chandra, P. Traina, and T. Li. BGP Communities Attribute. Internet Engineering Task Force (IETF), RFC 1997, 1996.

[FB05] 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.

[FGVTT04] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33(3-4):341–383, 2004.

[GJR03] Timothy G. Griffin, Aaron D. Jaggard, and Vijay Ramachandran. Design principles of policy languages for path vector protocols. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 61–72, New York, NY, USA, 2003. ACM.

[Gri10] Timothy G. Griffin. The stratified shortest-paths problem. In *Proceedings of the 2nd international conference on COMmunication systems and NETworks*, COMSNETS'10, pages 268–277, Piscataway, NJ, USA, 2010. IEEE Press.

[Hos10] Haruo Hosoya. *Foundations of XML Processing: The Tree-Automata Approach*. Cambridge Press, 2010.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Int97] Internet System Consortium (ISC). IRRToolSet. `http://irrtoolset.isc.org/`, 1997.

[KS11] W. Kumari and K. Sriram. Recommendation for Not Using AS_SET and AS_CONFED_SET in BGP. Internet Engineering Task Force (IETF), RFC 6472, 2011.

[LLW+09] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. Detecting Network-wide and Router-specific Misconfigurations through Data Mining. *IEEE/ACM Transactions on Networking (ToN)*, 17:66–79, February 2009.

[MWA02] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *In Proceedings of the ACM SIGCOMM Conference*, pages 3–16, 2002.

[PGM+12] D. Perouli, T. G. Griffin, O. Maennel, S. Fahmy, C. Pelsser, A. Gurney, and I. Phillips. Detecting Unsafe BGP Policies in a Flexible World. In *Proceedings of International Conference on Network Protocols (ICNP)*, November 2012.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). Internet Engineering Task Force (IETF), RFC 4271, 2006.

[VH09] Andreas Voellmy and Paul Hudak. Nettle: A Language for Configuring Routing Networks. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 211–235, Berlin, Heidelberg, 2009. Springer-Verlag.