1 ROUTING PROTOCOLS

Formalizations of SRPs extend the type of the routing attributes to include a special value ∞ denoting the lack of a route. This is easily expressed in NV using option types; we define the following useful operators (functions) to deal with *optional routes*.

```
let  a `≫` f =
  match a with
  | None -> None
  | Some a -> f a

let x `⊕f` y =
  match x,y with
  | None, None -> None
  | Some _, None -> x
  | None, Some _ -> y
  | Some x, Some y -> f x y
```

Listing 1: Optional routes

1.1 Common routing protocols

```
type Arip= option ℕ

let transferrip e a =
  a ≫ (λ c. if c > 15 then None else Some (c+1))

let rankRip (c1 c2 : ℕ) =
   if c1 < c2 then Some c1 else Some c2

let merge_rip u x y = x ⊕rankRip y
```

Listing 2: Model of the RIP protocol

```
type bgp = {lp: ℕ; asLen:ℕ; communities: set ℕ }
type Aebgp = option bgp

let transferebgp e a = policy dependent

let rankBGP (a b: bgp)  =
  if a.lp > b.lp then Some a
  else if b.lp > a.lp then Some b
  else if a.asLen < b.asLen then Some a
  else Some b

let mergeebgp u a b = a ⊕rankBGP b
```

Listing 3: Model of the eBGP protocol

1.2 Building more realistic models

Building on top of the simple models developed for eBGP and RIP in the previous section, we can model a more realistic model of the control plane of modern networks.

Forwarding. While computing and reasoning about the routing solution of each node is a problem of great interest to network operators, the end goal is to reason about the forwarding behavior of the network. A node forwards traffic towards the neighbor from which it received the "best" advertisement towards the destination. We capture this, by defining a *combinator* that keeps track of this information.

```
type A_fwd = option {fwdEdge : ℕ × ℕ;  protocol : A}

let transfer_fwd e a =
  a ≫
    (λ a.
      (transfer e a.protocol) ≫
        (λ route. Some {fwdEdge = flip edge; protocol = route)

let merge_fwd u a b =
  let f = λ a b. if (merge u a.protocol b.protocol) = a.protocol then a else b in
    a ⊕_f b
```

Listing 4: SRP capturing forwarding behavior

Combining two protocols. Networks often run a combination of protocols. Each protocol runs independently and computes the best route according to the protocol's definition. Subsequently, a comparison between the protocols is performed based to select the best overall protocol. Because protocols exchange messages of different types, comparison is done based on an extra field called *administrative distance* that denotes how desirable each protocol is. Expressing these semantics in NV is straightforward:

So far, we assumed that nodes announce a single route. In reality, a node may announce multiple routes, each for a different *prefix*. In NV we can easily lift the models above to multiple prefixes using *maps*.

1.3 Data plane

As mentioned, reasoning about the behavior of data packets is the end goal of network verification; reasoning about the behavior of the control plane is a prerequisite to this. In this section, we give a semantics to the data plane in terms of another SRP, given the solution (rib) to the SRP that models the control plane. Assuming a type for the payload of packets, *the stable solution of a node denotes the set of packets that have traversed it.*

lpm is a function capturing the longest matching prefix semantics: it returns the rib entry whose prefix is the longest one that matches the destination IP of the packet. One can additionally model data plane ACLs in the transfer function, by simple adding additional predicates in this set filter operation.

1.4 Modelling iBGP

So far there is clear phase distinction between control and data plane; the control plane generates the route table and the data plane uses this to generate the forwarding table. This is no longer true when the network uses the iBGP routing protocol. iBGP is used to distribute external routes (i.e. learned through eBGP) inside an AS. To achieve this, iBGP relies on an IGP (e.g. OSPF or RIP). In particular, each node inside an AS that runs iBGP announces a (unique) prefix. When a border router learns a new external route through eBGP it sends a data packet to each iBGP router in the same AS – using the routes computed by the IGP – that encapsulates routing information for the external route. This creates a kind of "circular" dependency where the computation of the routes for a prefix depends on the forwarding behavior for another prefix. One can explicitly find the dependencies between prefixes and compute a sequence of control and data planes. Instead, we can avoid dealing with this extra complexity, by creating an SRP that represents both the control plane and the data plane. The stable state of this SRP is the state that would be reached by the semantics which make prefix dependencies explicit.

```
type A_igp = { attr_igp : A_rip; ad_igp : ℕ}
type A_egp = { attr_egp : A_ebgp; ad_egp : ℕ}
type protocolPair = { igp : A_igp; egp : A_egp; best : ℕ}
type A_pair = option protocolPair

let transfer_pair e a =
  a ≫
    (λ a.
       { igp  = transfer_igp e a.igp;
         egp = transfer_egp e a.egp;
         best = 0
       })

let rankPair u (a b : protocolPair) =
  let i = merge_igp u a.igp b.igp in
  let e = merge_egp u a.egp b.egp in
  match i, e with
  | None, None -> None
  | Some _, None ->
    Some { igp = i; egp = None; best = 0 }
  | None, Some _ ->
    Some { igp = None; egp = e; best = 1 }
  | _, _ ->
    Some { igp = i; egp = e; best = if i.ad_igp ≤ e.ad_egp then 0 else 1 }

let merge_pair u a b = a ⊕_rankPair~u b
```

Listing 5: Combining SRPs

```
type A_control = dict[(ℕ × ℕ), A_pair]

let transfer_control e a=
  map (λ x. transfer_pair e x) a

let merge_control u a b =
  combine (λ x y. merge_pair u x y) a b
```

Listing 6: Lifting SRPs to prefix-based routing

First, we use a function ibgpRel that captures the iBGP neighboring relationship between nodes:

```
(* Given a node returns the set of iBGP neighbors *)
let ibgpRel (u : node) : set node =
  match u with
  | .. -> {..}
```

Furthermore, we extend the type of BGP routes to include a boolean that indicates whether a BGP route was learned from an internal or external neighbor:

```
type bgp = {lp: ℕ; asLen:ℕ; communities: set ℕ; ibgp : bool}
type A_bgp = option bgp
```

The transfer function will ensure that routes learned via iBGP are not redistributed to other iBGP neighbors, in order to ensure that the system converges to a stable state.

```
type packet = { srcIp : ℕ; dstIp : ℕ; payload : T }
type A_data = set packet

symbolic rib : A_control

(* Note to us: It would be nice if we could write this function in a part of the language that includes more
such as finite recursion, or subtraction, but get's normalized before reaching BDDs or SMT *)

(* lpm assumes that the prefixes are installed in a canonical form (i.e. ip & length) *)
let lpm (ip : ℕ) (rib : A_control) =
  match rib[ip] with
  | Some route -> Some route
  | None -> (
    match rib[ip & 2^31 − 1] with
    | Some route -> Some route
    | None ->
      (match rib[ip & 2^30 − 1] with
       | Some route -> Some route
       | None -> ....)
    )

(* Is edge e an edge over which packet p is forwarded according to this rib *)
let isForwarding e (p : packet) (rib : A_control) =
  match lpm p.dst rib with
  | None -> false
  | Some route -> route.fwdEdge = e

let transfer_data e ps =
  { p ∈ ps | isForwarding e p rib}

let merge_data u ps_1 ps_2 =
  ps_1 ∪ ps_2
```

Listing 7: Data plane as an SRP

Operational model of iBGP. The SRP of 8 precisely models the way iBGP routes are exchanged in a network. When a device learns a route through eBGP it send one packet for each iBGP "neighbor" (an iBGP neighbor is not necessarily physically connected, function ibgp(u) returns the iBGP neighbors of node u) that encapsulates this externally learned route (see $\text{transfer}_{\text{aux}}$ iBGP case). The merge function for a node u will look for such packets and if they are destined for u it will install the routes in the RIB of the device.

An alternative model for iBGP. While the operational model faithfully captures iBGP behavior, it is not amenable to efficient simulation or automated verification for various reasons. For example, fold operations over arbitrary maps cannot be efficiently translated to BDD operations or to a FOL formula (todo: cite relevant section in NV language). To overcome this limitation, we develop an alternative (but semantically equivalent[1]) model for iBGP based on a new notion of SRP, called *Extended-SRP* (*E-SRP*).

The key idea is to avoid sending iBGP routes through data packets. Instead, we connect all iBGP neighbors with virtual links, i.e. links that may have no topological manifestation. Then, as with other routing messages, iBGP messages can be transferred over those links without needing to encapsulate them in a data packet. To account for the fact that iBGP messages should be transferred over the data plane, each device running iBGP sends a packet to the loopback address announced by its iBGP peers.

[1] define this

These packets are forwarded based on the IGP computed routes over actual links (and not the iBGP virtual links). If a packet fails to reach an iBGP neighbor then the BGP route cannot be transmitted to this neighbor.

This model of iBGP is much simpler than the operational one; it essentially only requires to change the semantics of the transfer function for BGP to account for iBGP. The rest of the operations can remain as they were before introducing iBGP, modulo the extra argument of the transfer function in an *E-SRP*, which should be ignored in all cases other than iBGP.

1.5 Equivalence between the two models

We prove that the two models compute the same control plane solutions. Further, we prove that the non-iBGP data packets traverse the same nodes.

Definition 1.1 (Attribute Equivalence)**.** *We say two attributes are equivalent, written as* $a \approx \widehat{a}$ *when:*

(1) The RIBs of the two attributes are the same:

$$a.rib = \widehat{a}.rib$$

(2) The packets of the two attributes are equal, modulo the iBGP packets:

$$\forall (p : packet).\ ((\forall r.\ p.payload \neq Route\ r) \Rightarrow (p \in a.ps \Leftrightarrow p \in \widehat{a}.ps)) \wedge$$
$$(\exists r.\ p.payload = Route\ r) \wedge p \in a.ps \Rightarrow \{pwithr.payload = Route\ \infty\} \in \widehat{a}.ps$$

The definition above is weak, because it only specifies one side of the relation between iBGP packets in $\widehat{a}$ and a. To specify the other side, we need access to the solution of the node that originated the iBGP packet.

```
type A_op = { rib : A_control; ps : A_data }

(* Encapsulates eBGP route in a packet and sends it to all iBGP neighbors *)
let transferIBGP (e : edge) (rib : A_control) (pre : ℕ × ℕ) (route : A_egp) =
  let (u,v) = e in
  let ws = ibgpRel u in (* get iBGP neighbors *))
  fold (λ acc w.
   (* encapsulate route in packet *)
    let p = { srcIp = loopback(u);  dstIp = loopback(w);  payload = Route (pre,route) } in
    if isForwarding e p rib then
      {p} ∪ acc (* send it if (u,v) is the first hop from u to w *)
    else
      acc) {} ws

let transfer_op e a =
  let ps' = transfer_data e a.ps in (* transfer data packets as always *)
  let rib' = transfer_control e a.rib in (* transfer eBGP and IGP routes as always *)
  let ps'' = foldi (λ ibgpPackets pre oroute.
                    match oroute with
                    | Some route ->
                      (match route.egp.attr_egp with
                       | None -> ibgpPackets
                       | Some r ->
                         if r.protocol.ibgp then
                           ibgpPackets
                         else
                           ibgpPackets ∪ (transferIBGP e rib pre {route with ibgp=true})
                       )
                    | None -> ibgpPackets ) ps' a.rib
  in
  { rib = rib'; ps = ps''}

let installRoute u (rib : A_control) (p : packet) =
  match p with
  | Route (pre, x) -> (
    if p.dstIp = loopback(u) then
      let routeToSrc = rib[(p.srcIp, 32)] in
      let optRib =
        rib[(p.srcIp, 32)] ≫
        (λ routeToSrc. routeToSrc.igp ≫
          (λ igpRoute. let route = Some { fwdEdge = igpRoute.fwdEdge; protocol = x  } in
                       merge_control u rib [pre |-> Some {igp = None; egp = route; best = 1}]))
      in
      match optRib with
      | None -> rib
      | Some newRib -> newRib
    )
  | _ -> rib

let merge_op u a b =
  let ps' = merge_data u a.ps b.ps in
  let rib' = merge_control u a.rib b.rib in
  let rib'' = fold (λ newRib p. installRoute u newRib p) rib' ps' in
  { rib = rib''; ps = ps'}
```

Listing 8: Operational iBGP model

```
let isIBGP u v (route : A_fwdBgp) =
  match route.attr with
  | Some route ->
    if !route.protocol.ibgp && ibgpRel(u)[v] then (* is this a route to be transmitted by iBGP? *)
      true
    else
      false
  | None -> false

let transferBGP e (route : A_bgp) curSol =
let (u,v) = e  in
if isIBGP u v then
  if curSol.ps[ibgpPacket(u,v)] then (* if the data packet from u has reached v*)
    curSol.rib[loopback(u)] ≫ (* and v has a path back to u through IGP*)
      (λ routeBack.
        if routeBack.best = 0 then
          routeBack.igp.attr.fwdEdge ≫
            (λ routeBackIGP.
              { fwdEdge = routeBackIGP.fwdEdge;
                protocol = transfer_ebgp e {route with ibgp=true}}))
        else None (* no IGP route back to u*)
  else
     None (* iBGP packet has not reached v *)
else
  transfer_ebgp e route (* eBGP route *)
```

Listing 9: ESRP based BGP model

```
  type A_alt = { rib : A_control; ps : A_data }

  let transfer_alt e a curSol =
    { rib = transfer_control e a.rib curSol;
      ps = transfer_data e a.ps
    }

  let merge_alt u a b =
    {rib = merge_control u a.rib b.rib; ps = merge_data u a.ps b.ps}
```

Listing 10: ESRP model of common routing protocols