Analyzing BGP Instances in Maude

Anduo Wang[1] Carolyn Talcott[2] Limin Jia[3] Boon Thau Loo[1] Andre Scedrov[1]

[1] University of Pennsylvania
[2] SRI International
[3] Carnegie-Mellon University
{anduo,boonloo}@cis.upenn.edu clt@csl.sri.com
liminjia@cmu.edu scedrov@math.upenn.edu

Abstract. Analyzing Border Gateway Protocol (BGP) instances is a crucial step in the design and implementation of safe BGP systems. Today, the analysis is a manual and tedious process. Researchers study the instances by manually constructing execution sequences, hoping to either identify an oscillation or show that the instance is safe by exhaustively examining all possible sequences. We propose to automate the analysis by using Maude, a tool based on rewriting logic. We have developed a library specifying a generalized path vector protocol, and methods to instantiate the library with customized routing policies. Protocols can be analyzed automatically by Maude, once users provide specifications of the network topology and routing policies. Using our Maude library, protocols or policies can be easily specified and checked for problems. To validate our approach, we performed safety analysis of well-known BGP instances and actual routing configurations.

Keywords: Maude, BGP gadgets, Inter-domain routing, Convergence analysis

1 Introduction

The Internet today runs on a complex routing protocol called the *Border Gateway Protocol* or *BGP* in short. BGP enables Internet-service providers (ISP) world-wide to exchange reachibility information to destinations over the Internet, and simultaneously, each ISP acts as an autonomous system that imposes its own import and export routing policies on route advertisements exchanged among neighboring ISPs.

Over the past few years, there has been a growing consensus on the complexity and fragility of BGP routing. Even when the basic routing protocol converges, conflicting policy decisions among different ISPs have led to route oscillation and slow convergence. Several empirical studies [17] have shown that there are prolonged periods in which the Internet cannot reliably route data packets to specific destinations due to routing errors induced by BGP. In response, the networking community has proposed several alternative Internet architectures [24] and policy constraints (or "safety guidelines") that guarantee protocol convergence if universally adopted [10,9,13,7,12,22].

One of the key requirements for designing new routing architectures and policy guidelines is the ability to study BGP network instances. These instances can come in the form of small topology configurations (called "gadgets"), which serve as examples of safe systems, or counterexamples showing the lack of convergence. They can also

come from actual internal router (iBGP) and border gateway (eBGP) router configurations. Today, researchers and network operators analyze these instances by manually examining execution sequences. This process is tedious and error-prone.

The main contribution of this paper is that we present an automated tool for analyzing BGP instances, and thus relieve researchers and network operators of manual analysis. Our tool uses Maude [4], a language and tool based on rewriting logic. We encode in Maude the BGP protocol as a transition system driven by rewriting rules. Consequently, we can use the high-performance rewriting engine provided by Maude to analyze BGP instances automatically. Our tool can simulate execution runs, as well as exhaustively explore all execution runs for possible divergence.

More specifically, we developed a set of Maude libraries specifying a generalized path vector protocol that is common to all BGP instances. The generalized path vector protocol utilizes a set of unspecified routing policy functions. These unspecified functions serve as the interface for specific routing policies which are formalized as *Stable Path Problems (SPP)* [14]. To use our library, users only need to input the network topology and customize routing policies functions in the form of SPP. We illustrate the use of our tool by analyzing various BGP instances.

The rest of the paper is organized as follows. Section 2 briefly reviews BGP and Maude tool. Section 3 introduces the Maude library that we have developed, as a basis for customizing SPP instances. In Section 4, we discuss how to use this library to specify a variety of BGP instances. In Section 5, we illustrate how to use the library to analyze these instances. The complete list of Maude code in our library is described in appendix A, and the details of specifying BGP instances are shown in appendix B C.

2 Background

We explain the background of our tool. First we briefly review BGP. Next we discuss the basics of Maude, the language and tool, in which our tool is built.

2.1 BGP

BGP assumes a network model in which routers are grouped into various Autonomous Systems (*AS*) administrated by Internet Server Provider (*ISP*). An individual AS exchanges route advertisements with neighboring ASes using the *path-vector* protocol. Upon receiving a route advertisement, a BGP node may choose to accept or ignore the advertisement based on its *import policy*. If the route is accepted, the node stores the route as a possible candidate. Each node selects among all candidate routes the best route to each destination based on its local route rankings. Once a best route is selected, the node advertises it to its neighbors. A BGP node may choose to export only selected routes to its neighboring ASes based on its *export policy*.

BGP systems come in two flavors: external BGP (eBGP), which establishes routes between ASes; and internal BGP (iBGP), which distributes routes within an AS. At the AS-level, a BGP system can be viewed as a network of AS nodes running eBGP. Each AS is represented by one single *router* node in the network (its internal structure ignored), and its *network state* includes its neighbors (ASes), selected best path and a

routing table. Route advertisements constitute the *routing messages* exchanged among them. Within an AS, a BGP system can be viewed as a network of two kinds of network nodes running iBGP: gateway routers and internal routers whose network states are similar to eBGP routers. iBGP allows internal routers to learn external routes (to destinations outside the AS) from gateway routers.

We model both eBGP and iBGP systems as network systems with two components: routing dynamics and routing policies. Routing dynamics specify how routers exchange routing messages, and how they update their network states accordingly. Routing policies are part of the static configuration of each router, by which the ISP operator expresses his local traffic interests and influences route decisions.

In our library, we adopt the use of Stable Paths Problems (SPP) [14] as the formal model of routing policies. An instance of the SPP S is a tuple (G, o, P, Λ), where G is a graph, o is a specific destination node [4], P is the set of permitted (usable) paths available for each node to reach o, and Λ is the ranking functions for each node. For each node v, λ^v is its ranking function, mapping its routes to natural numbers (ranks), and P^v are its permitted paths, the set of available paths to reach o. A path assignment is a function π that maps each network node $v \in V$ to a path $\pi(v) \in P^v$. A path assignment is *stable* if each node u selects a path $\pi(u)$ which is (1) the highest ranked path among its permitted paths, and (2) is consistent with the path chosen by the next-hop node. Consistency requires if $\pi(u) = (uv)P$ then for the next-hop node v, we must have $\pi(v) = P$. A solution to the SPP is a stable path assignment.

In this paper, we are interested in analyzing BGP convergence (safety) property in the SPP formalism. A BGP system converges and is said to be safe, if it produces stable routing tables, given any sequence of routing message exchanges. We can study BGP convergence by analyzing its SPP representation: SPP instance for a safe BGP system converges to a solution in all BGP executions. Note that, the existence of an SPP solution does not guarantee convergence.

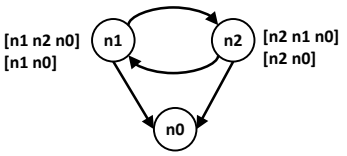

Fig. 1: Disagree Gadget.

For example, Figure 1 presents an SPP instance called the Disagree "gadget". The per-node ranking functions are $\lambda^1(\texttt{[n1 n2 n0]}) = 1$, $\lambda^1(\texttt{[n1 n0]}) = 2$, $\lambda^2(\texttt{[n2 n1 n0]}) = 1$, and $\lambda^2(\texttt{[n2 n0]}) = 2$. The permitted paths for each node are listed besides the corresponding node. The order in which the paths are listed is based on the ranking function: Nodes prefer higher ranked routes, e.g. node `n1` prefers route `[n1 n2 n0]` over `[n1 n0]`. Disagree has two stable path assignment solutions: `([n1 n2 n0], [n2 n0])` and `([n2 n1 n0], [n1 n0])`. However, Disagree is not guaranteed to converge because there exists an execution trace where route assignments keep oscillating. Consider the execution where node `n1` and `n2` update and exchange routing messages in a synchronized manner, and their network states oscillate between two unstable path assignments `([n1 n0]) ([n2 n0])` and `([n1 n2 n0] [n2 n1 n0])` forever.

[4] Assuming the Internet is symmetric, we can study its routing behavior by studying routing to a single destination.

2.2 Rewriting Logic and Maude

Rewriting logic [19] is a logical formalism that is based on two simple ideas: states of a system can be represented as elements of an algebraic data type, and the behavior of a system can be given by transitions between states described by *rewrite rules*. By algebraic data type, we mean a set whose elements are constructed from atomic elements by application of constructors. Functions on data types are defined by equations that allow one to compute the result of applying the function. A rewrite rule has the form $t \Rightarrow t'\ if\ c$ where t and t' are patterns (terms possibly containing variables) and c is a condition (a boolean term). Such a rule applies to a system state s if t can be matched to a part of s by supplying the right values for the variables, and if the condition c holds when supplied with those values. In this case the rule can be applied by replacing the part of s matching t by t' using the matching values for variables in t'.

Maude [4] is a language and tool based on rewriting logic [18]. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. Given a specification S of a concurrent system, Maude can execute this specification and allows one to observe possible behaviors of the system. One can also use the search functionality of Maude to check if a state meeting a given condition can be reached during the system's execution. Furthermore, one can model-check S to check if a temporal property is satisfied, and if not, Maude will produce a counter example. Maude also supports object-oriented specifications that enable the modeling of distributed systems as a multiset of objects that are loosely coupled by message passing. As a result, Maude is particularly amenable to the specification and analysis of network routing protocols.

3 A Maude Library for Encoding BGP Protocols

This section presents our Maude library for analyzing BGP instances. This library provides specification of the protocol dynamics that are common to BGP instances, and defines a routing policy template in terms of the Stable Path Problem (SPP) so that network designers can customize it to analyze a specific instance. Our library also provides support for detecting route oscillation.

BGP system	**Maude interpretation**
Network nodes	(Router) objects
Routing messages	Terms of type `Msg`
Global Network	Multiset of router objects and terms representing messages
Protocol dynamics	Local rewriting rules
Global network behaviors	Concurrent rewrites using local rules
Route oscillation support	A Logger object recording the histories of route assignments and rewriting rules updating the Logger object

Table 1: Overview and Interpretation of Maude Library

Our library is organized into a hierarchy of Maude modules. Table 1 presents the correspondence between concepts in BGP protocols and the Maude code. We first show how our library represents a single network state of BGP system (Section 3.1). Then we explain how to capture the dynamic behavior of a local BGP router using rewrite rules. In doing so, the global network behaviors can be viewed as concurrent applications of the local rewriting rules (Section 3.2). Finally, we discuss the component in the library that detects route oscillation (Section 3.3).

3.1 Network State

A network state is represented by a multiset of network nodes (routers) and routing messages used by routers to exchange routing information. Each network node is represented by a Maude object, whose attributes consist of its routing table, best path and neighboring table. We omit the detailed Maude sort definitions, but provide an example encoding of the network node `n1` in Disagree gadget show in Figure 1 as follows.

```
[n1 : router |
      routingTable: (source: n1,dest: n0,pv:(n1 n0),metric: 2),
      bestPath: (source: n1,dest: n0,pv:(n1 n0),metric: 2),
      nb: (mkNeigh(n0,2) mkNeigh(n2,1))]
```

The constructor for a node is `[_:_|_,_,_]`. The first two elements (`n1:router`) specify the node's id `n1`, and its object class `router`. The next three elements are the attributes. At a given state, the routing table attribute constructed from `routingTable:_` contains `n1`'s current available routes. Each routing table entry stores the routing information for one particular next-hop. Here, the routing table attribute only contains one entry `(source: n1, dest: n0, pv:(n1 n0), metric: 2)`. This route is specified by its source (`source: n1`), destination (`dest: n0`), the path vector that contains the list of nodes along the path (`pv: (n1 n0)`), and the cost of the route (`metric: 2`). This route is also used for the best path attribute, constructed from `bestPath:_`, which contains `n1`'s current best path. The last attribute is the neighbor table, constructed from `nb:_`. To extract a node's local neighbor table from the network topology, we further introduce an operator `mkNeigh`. The first argument of `mkNeigh` is the identifier of the neighboring node, and the second argument the metric associated with the link to that node. Node `n1` has two neighbors, node `n0`, the cost to which is 2 (`mkNeigh(n0,2)`); and node `n2`, the cost to which is 1 (`mkNeigh(n2,1)`).

Besides router objects, the second part of a network state is routing messages in the network. Typically, network nodes exchange routing information by sending each other routing messages carrying newly-learned routes. In our library, a routing message is constructed from `sendPacket(_,_,_,_,_)`. For example, in the Disagree gadget, the initial routing message sent by node `n1` to its neighbor `n2` is represented by message term: `sendPacket(n1,n2,n0,2,n1 n0)`. This message carries `n1`'s routes to destination `n0` with path vector `n1 n0` at cost `2`. In general, the first two arguments of `sendPacket(_,_,_,_,_)` denote the sender's identifier (node `n1`), and the receiver's identifier (node `n2`) respectively. The rest of the arguments specify the identifier of the destination (node `n0`), the metric representing the cost of the route (2), and the path vector of the routing path (`n1 n0`).

3.2 Protocol Dynamics

We now show how to specify network system dynamics in Maude. By modeling a BGP system as a concurrent system consisting of router objects (and the routing messages), to specify the global BGP evolution, we only need to specify the local rewrite rules governing the state transition of each BGP router.

A BGP node's dynamics can be captured by various equivalent state transitions. To reduce search space in analysis, we adopt a one-state transition: for each BGP node `N`, when it receives routing messages from a neighbor `S`, `N` computes the new path from the received message, updates `N`'s routing table and re-selects best path accordingly, and finally sends out routing messages carrying its new best path information if a different best path is selected. This state transition is encoded as a single rewrite rule of the following form:

```
rl [route-update] :
  sendPacket(S, N, D, C, PV)
  [ N : router | routingTable: RT, bestPath: Pb, nb: NB ]
 =>
if  (case 1) then best path re-selects (promotion)
else (if (case 2) then best path remains same
      else (if (case 3) then best path re-selection (withdraw)
            else error processing
            fi) fi) fi.
```

Here, `r1` is the identifier of this rule, and `route-update` is the name of this rule. Rule `r1` is fired when the left-hand side is matched; that is, when a node `N` consists of routingTable `RT`, bestPath `Pb`, and neighboring table `NB` receives a route advertisement message from neighbor `S`. The result of applying the rule is shown on the right-hand side: the routing message is consumed, and attributes of router `N` are updated. Based on the result of the re-selected bestPath attribute, there are three different cases for `N` to update its state as specified in the three branches. Next, we explain these three cases.

Best path promotion. In any case, node `N` needs to first compute the new path based on its neighbor `S`'s message asserting that `S` can reach `D` via a path `PV`. We define a function `newPath` that takes a routing message and the neighbor table as arguments, and returns the new path by first prepending `N` to the path announced by `S`, setting the new path attribute according to the local ranking function `lookUpRank`, and then imposing the import policy by modifying the path metric according to BGP routing policy configuration (`import` function). Here `import` and `lookUpRank` are unspecified routing policy functions. Together with `export` that we will introduce shortly, they constitute our library's specification interface for defining BGP routing policy. To specify a particular BGP instance's routing policy, the user only needs to specify `import, lookUpRank` and `export` accordingly.

The first branch (case 1) is specified below. The newly computed path is compared with the current bestPath `Pb`, if the new one is preferred over the old value `Pb`, the `bestPath` attribute will be updated to this new path. Furthermore, if the export policy allows, the new best path value will be re-advertised to all of `N`'s neighbors by sending them routing messages.

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   prefer?(newPath(sendPacket(S,N,D,C,PV),NB),Pb)==true
then
([ N : router |
   routingTable: updatedRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   bestPath: newPath(sendPacket(S,N,D,C,PV),NB),
   nb: NB ]
 multiCast(NB, export(newPath(sendPacket(S,N,D,C,PV),NB))))
```

Here the new state of `N` is obtained by updating the old `routingTable` attribute `RT` (`updateRT` function), and updating the `bestPath` attribute by setting it to the new value of `bestPath`. The `updateRT` function recursively checks the routing table, and for each next-hop entry, it either inserts the new path (`newPath(...)`) if no available route is presented; or replaces the old value with the new path. To complete the state transition, for all `N`'s neighbors, routing messages carrying the new path are generated by `multiCast` function. To impose the export routing policy, before sending the new best path, `export` is applied to the new path to filter out the routes which are intended to be hidden from neighbors. Similar to `import`, `export` is to be instantiated by the user when analyzing a particular BGP instance. If the export routing policy prohibits the new path to be announced, `export` will transform it to `emptyPath`, which `multiCast` will not generate any message.

Best path remains the same. In the second branch (`case 2`), a new path `newPath(...)` is computed from the received message as before. However, the new path is no better than the current bestPath `Pb`. But the next-hop node of the new path and `Pb` are different, implying that the new path is just an alternative path [5] for `N` to reach the destination. As a result, the current `bestPath` value `Pb` is unchanged, and only the `routingTable` will be updated with this alternative path (`newPath(...)`). No routing messages will be generated:

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   getNext(newPath(sendPacket(S,N,D,C,PV),NB))=/=getNext(Pb) and
   prefer?(Pb,newPath(sendPacket(S,N,D,C,PV),NB))==true
then
[ N : router |
   routingTable: updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   bestPath: Pb,
   nb: NB ]
```

Best path withdraw. The same as in the second branch, in case 3, the newly computed path `newPath(...)` is worse than the current bestPath `Pb`, but it is now routed through the same next-hop `S` as current bestPath `Pb`. The fact that `S` now sends a less preferred path indicates that the previous learned route `Pb` is no longer available at `S`. Therefore, we need to withdraw `Pb` by dropping `Pb` from routing table, shown as follows:

```
if getDest(newPath(sendPacket(S,N,D,C,PV),NB))==getDest(Pb) and
   getNext(newPath(sendPacket(S,N,D,C,PV),NB))==getNext(Pb) and
```

[5] Different next-hop implies the route is learned from a different neighbor.

```
   prefer?(Pb, newPath(sendPacket(S,N,D,C,PV),NB))==true
then
([ N : router |
   routingTable: updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT),
   newBest(newPath(sendPacket(S,N,D,C,PV),NB),
           updateRT(newPath(sendPacket(S,N,D,C,PV),NB),RT)),
   nb: NB ]
 multiCast(NB,export(newBest(newPath(sendPacket(S,N,D,C,PV),NB),
                     updateRT(newPath(sendPacket(S,N,D,C,PV),NB),
                              RT))))
```

Here, `updateRT` replaces (therefore removes) the outdated `Pb` with the new path (`newPath(...)`), and `newBest` function re-computes the best path from `newPath(...)` and the remaining paths in routing table. As in case 1, to complete the state transition, the newly selected best path is sent to its neighbors by `multiCast(...)`.

3.3 Route Oscillation Detection Support

Our library also provides extra definitions to help detect route oscillation. Our method is based on the observation that if route oscillation occurs during network system evolution, there is at least one *path assignment* (at a given state for a BGP system, we define the path assignment to be the collection of best paths currently selected by all network nodes) that is visited twice. Therefore, we use the following simple heuristic: we maintain a record of all path assignments for all visited states in BGP execution, and check for recurring path assignment. Note that a path assignment (best path attribute of `router` object) only constitutes a sub-set of the entire system state (the `router` objects attributes and routing messages), consequently our heuristic based on this partial system state can have false positives: our analysis may report a false route oscillation when two states are identical only in path assignments, but not the entire system states. Nevertheless, our heuristic is sound and is still helpful in detecting all potential route oscillation: when route oscillation occurs, a recurring path assignment state must occur.

More concretely, in our Maude library, we create a global *logger object* to keep track of the history of path assignments. For each snapshot of the network state, i.e. whenever a network node makes a local state transition and updates its best path attribute, the logger object is synchronized to create a new path assignment entry that corresponds to the updated best path. We then provide a function that checks for recurring entries in the list of visited path assignments, which can be used directly in Maude's exhaustive search to detect route oscillation.

Logger object. The global logger is represented by an object `pa` of `Logger` class which has one attribute `history`. At a given state, this attribute contains a list (history) of path assignments, each entry of which contains the snapshot of the network's collection of best paths in a visited state.An example logger object for the disagree gadget is the following:

```
{pa : Logger | history: ({[n1 n2 n0] [n2 n0]}
                         {[n1 n2 n0] [n2 n1 n0]}
```

```
{[n1 n2 n0] [n2 n0]}
{[n1 n0]   [n2 n0]})}
```

The above logger records four snapshots of the Disagree's best paths. For example, the first path assignment {[n1 n2 n0] [n2 n0]} denotes the network latest state where node 1's best path to 0 is [n1 n2 n0] and node 2's best path is [n2 n0]. And line 4 {[n2 n0] [n2 n0]} records Disagree's path assignment at its initial (oldest) state. Note that, this object content actually exhibits route oscillation (line 1 and line 3) described in Section 3.2.

Synchronized logging. To log all path assignment changes, we only need to slightly modify the single rewrite rule for route update, such that whenever the rule is fired to apply local state transition for some node, the global object pa is synchronized and its path assignment is updated to reflect changes in the local node's best path attribute, shown as follows:

```
rl [route-update-logging] :
  sendPacket(S, N, D, C, PV)
  [ N : router | routingTable: RT, bestPath: Pb, nb: NB ]
  { pa : Logger | history: HIS }
=>
*** first branch: bestPath re-selects (promotion)
if ... then ...
 { pa : Logger | history:
   historyAppend(updateAt(index(N),
           [getPV(newPath(sendPacket(S,N,D,C,PV),NB))],
           head(HIS)),HIS)})
else ... fi .
```

On the left-hand side, two objects: a router N and the global logger pa are matched to trigger the transition. As described in 3.2, in the first branch of route update where the node's best path attribute is set to newPath(...), the logger pa updates its path assignment attribute as follows: First, it creates a new path assignment entry to record newPath(...) by function updateAt(...). Then, the new entry updateAt(...) is inserted into the list of previous path assignments HIS by function historyAppend. Here, the new path assignment entry updateAt(...) is computed by updating the latest path assignment entry head(HIS) with newPath(...). The rest of branches 2 and 3 are modified similarly.

Route oscillation detection. A network state is now a multiset of router objects, routing messages, and one global logger object. The function detectCycle detects re-curring path assignments, as follows:

```
eq detectCycle([ N : router | routingTable: RT,
                              bestPath: Pb,nb: NB] cf)
  = detectCycle (cf) .
eq detectCycle(message cf) = detectCycle (cf) .
eq detectCycle({ pa : Logger | history: HIS } cf)
  = containCycle? (HIS) .
```

The first two equations ignore router objects and routing messages in the network state, and the last equation examines logger `pa` by function `containCycle?` to check for recurring path assignment entries in `HIS`. We will revisit the use of `detectCycle` to search for route oscillation in Section 5.

4 Specifying BGP Instance

Given a BGP instance with its *network topology* and *routing policies*, we show how to specify the instance as a SPP in our library. We discuss examples for both eBGP and iBGP.

4.1 eBGP instance

An eBGP instance can be directly modeled by an SPP instance $S = (G, o, P, \Lambda)$: G, o specifies the instance's *network topology*, and P, Λ specifies the resulting per-node route ranking function after applying the eBGP instance's *routing policies*. Our library provides Maude definitions for each SPP element.

Network topology. An eBGP instance's initial network state is generated from its network topology, which is represented by a list of network nodes and links. Our library declares two constants `top-Nodes` and `top-BGP` to represent network nodes and links. For example, to specify the topology of the Disagree gadget, the user defines `top-Nodes, top-BGP` as follows:

```
eq top-Nodes = n1 n2 .
eq top-BGP = (n1,n0 : 2) (n1,n2 : 1) (n2,n1 : 1) (n2,n0 : 2) .
```

Here, `n0` is the identifier of the destination node (o). Each link is associated with its cost. Based on the value of `top-Nodes` and `top-BGP` that are input by the user, our library automatically generates Disagree's initial state by `init-config` function:

```
eq gadget = init-config (top-Nodes, top-BGP) .
```

The resulting `gadget` is a network state which consists of the two network router objects `n1,n2`, the four initial routing messages, and the initial logger `pa`, as shown in Section 5.1. In this initial state, the three attributes of each network node – the routing table and best-path and neighbor tables are computed as follows: `init-config` parses the BGP links in network topology (`top-BGP`), for each link `(ni,nj : M)`, a new routing table entry for `nj` with cost `M` is created, and if `nj == n0`, then set `ni`'s best path to the one-hop direct path `ni n0`, and its routing tables containing this one-hop direct route; otherwise if there is no direct link from `ni` to `n0`, set `ni`'s best path and the routing table to `emptyPath`. Initial routing messages and logger `pa` are computed in a similar manner.

Routing policy. The route ranking function Λ and permitted paths P are the result of applying three BGP policies functions: `import`, `export` and `lookUpRank`. As we have discussed in Section 3, `import`, `export`, `lookUpRank` are three user-defined functions that serve as the specification interface for routing policies.

Functions `import` and `lookUpRank` are used to compute new routing paths from a neighbor's routing message: `import` filters out un-wanted paths, and `lookUpRank` assigns a rank to the remaining permitted paths. Note that the metric value `lookUpRank (N PV)` assigned by `lookUpRank` also determines the route's preference in route selection. `export` is used to filter out routes the router would like to hide.

As an example, the policy functions for Disagree are defined as follows.

```
eq export (P) = P . eq import (P) = P .
eq lookUpRank (n1 n2 n0) = 1 . eq lookUpRank (n1 n0) = 2 .
eq lookUpRank (n2 n1 n0) = 1 . eq lookUpRank (n2 n0) = 2 .
```

The first line says Disagree does not employ additional import/export policies. Whereas the second and third line asserts that Disagree's two nodes prefers routes through each other: For example the second line encodes node `n1`'s ranking policy that it prefers path `(n1 n2 n0)` (with higher rank 1) through `n2` over the direct path `(n1 n0)` (rank 2).

4.2 iBGP Instance

Our appendix C shows our SPP encoding of iBGP instances. The main differences between an iBGP and eBGP instances are: (1) iBGP network topology distinguishes between internal routers and gateway routers. Gateway routers runs eBGP to exchange routing information with (gateway routers of) other ISPs, while simultaneously running iBGP to exchange the external routing information with internal routers in the AS. (2) iBGP routing policy utilizes a separate IGP protocol to select best route. Internal to an AS, the ISP uses its own IGP protocol to compute shortest paths among all routers. The shortest path distance between internal routers and gateway routers are used in iBGP route selection: iBGP policy requires the internal routers to pick routes with shortest distance to its gateway router.

As a result, iBGP requires encoding two types of topologies: a *signaling* topology for gateway routers and internal routers to exchange routes within the AS, and a *physical* topology on which the IGP protocol is running. Further, an additional *destination router* denoting the special SPP destination o is added as an external router which is connected with all gateway routers. In our library, we implement and run separately in Maude an IGP protocol (for computing all-pairs shortest paths) and pass the resulting shortest path distances to iBGP protocol.

5 Analysis

To analyze BGP instances, our library allows us to (1) execute the Maude specification to simulate possible execution runs; and (2) exhaustively search all execution runs to detect route oscillation.

5.1 Network Simulation

Network initialization. For any analysis, we need to first generate a BGP instance's initial network state. For a given BGP instance, we have shown how to generate its initial state `gadget` from its network topology and routing policy, as described in section 4. For example, the initial state generated for Disagree is as follows:

```
{pa : Logger | history:{[n1 n0] [n2 n0]}}
[n1 : router | routingTable: (source: n1, dest: n0,
                              pv:(n1 n0), metric: 2),
               bestPath: (source: n1, dest: n0,
                          pv:(n1 n0), metric: 2),
               nb: (mkNeigh(n0,2) mkNeigh(n2,1))]
[n2 : router | ... ]
sendPacket(n1,n0,n0,n2,n1 n0) sendPacket(n1,n2,n0,n2,n1 n0)
sendPacket(n2,n0,n0,n2,n2 n0) sendPacket(n2,n1,n0,n2,n2 n0)
```

This state consists of Disagree's initial logger object `pa` that holds the initial path assignment `[n1 n0] [n2 n0]`, two router objects `n1,n2`, and four initial routing messages.

Execution. Unlike many formal specification paradigms used in static network analysis, a Maude specification is executable. To explore *one* possible execution run from a given initial state `gadget`, we can directly use Maude's `rewrite` and `frewrite` (fair rewriting) commands. For example, we could tell Maude to execute the Disagree gadget with the following command: `frew gadget .` This command terminates and returns the following final state:

```
{pa : Logger |
    history: ({[n1 n0] [n2 n1 n0]} ... {[n1 n0] [n2 n0]})}
[n1 : router |...
    bestPath: (source: n1,dest: n0,pv:(n1 n0),metric: 2), ...]
[n2 : router |...
    bestPath: (source: n2,dest: n0,pv:(n2 n1 n0),metric: 1),...]
```

Note that this final state corresponds to one of the stable path assignments of Disagree described in Section 2, where node `n1` sets its best path to `[n1 n0]`, and node `n2` sets its best path to `[n2 n1 n0]`.

On the other hand, with the `rew` command which employs a different rewriting strategy, divergence scenario is simulated and route oscillation is observed in the simulation. This is because `frewrite` employs a depth-first position-fair rewriting strategy, while `rewrite` employs a left-most, outer-most strategy that coincides with the execution trace that leads to divergence.

5.2 Route Oscillation Detection

While Maude commands `frew/rew` explore a small portion of possible runs of the instance, the `search` command allows us to exhaustively explore the entire execution space. To exhaustively search BGP execution for route oscillation, we only need to first

input the BGP instance's network topology and routing policy to generate the corresponding initial state, as described in Section 4; and then use the `search` command to automatically search for oscillation. For example, for Disagree, we run:

```
search [1] gadget =>+ X such that detectCycle(X) = true .
```

Here, `gadget` is Disagree's initial state, and `=>+ X` tells Maude to search for any reachable network state `X` such that at that state, the logger `pa` contains recurring path assignment (`detectCycle(X)=true`). `search` command exhaustively explores Disagree runs and returns with the first Disagree state that exhibits oscillation:

```
{pa : Logger | history: ({[n1 n2 n0] [n2 n0]}
                         {[n1 n2 n0] [n2 n1 n0]}
                         {[n1 n2 n0] [n2 n0]}
                         {[n1 n0] [n2 n0]})}
[n1 : router |...] [n2 : router |...] ...
```

Here, the resulting path assignment content in `pa` exhibits an oscillation (line 1, line 3).

In general, Maude allows us to exhaustively search for violation of a safety property `P` by running the following command:

```
search initialNetwork =>+ X:Configuration such that P(X) == false.
```

which tells Maude to exhaustively search for a network state `X` that violates `P` along all possible execution traces from the initial state `initialNetwork`. If Maude returns with `No solution`, we can conclude property `P` holds for all execution traces.

5.3 Case Studies

We have analyzed well-known eBGP instances, including good gadget, bad gadget, disagree [14]. In addition, we analyze two iBGP configuration instances: a 9-node iBGP gadget [8] that is known to oscillate, and a 25-node configuration randomly extracted from the Rocketfuel [23] dataset. Rocketfuel is a well-known dataset on actual iBGP configurations that are made available to the networking community. Given that an ISP has complete knowledge of its internal router configurations, the Rocketfuel experiment presents a practical use case for using our tool to check an actual BGP configuration instance for safety.

For each BGP instance, we simulate its possible executions using rewriting commands (*Simulation*), and check for route oscillation using exhaustive search (*Exhaustive*). We summarize our analysis results are as follows:

We have carried out these analysis on a laptop with 1.9 GB memory and 2.40GHz dual-cores running Debian 5.0.6. The version of Maude is Maude 2.4. While route oscillation detection explores the entire state space of the instance execution, the analysis time for rewriting based execution are measured for only one possible terminating execution (that converges to a stable path assignment).

Here we summarize findings from our case studies. Single-trace simulation is helpful in finding permanent routing oscillation. When simulating the execute trace that

	Disagree	**Bad**	**Good**	**9-node iBGP**	**25-node iBGP**
Simulation	2	NA	4	20	31
Exhaustive	2,10,No	181,641,No	10997,37692,Yes	20063,52264,No	723827,177483,Yes

Table 2: Summary of BGP analysis in Maude. In the first row, each entry shows the simulation time in milliseconds. In the second row, for each entry, the first value denotes exhaustive search time in milliseconds, the second denotes number of states explored, and the third on whether our tool determines the instances to be safe ("Yes") or unsafe ("No").

diverges, Maude does not terminate (e.g., in executing Bad gadget [6]). However, simulation can miss temporary oscillations which are only manifested on a particular executing trace. When Maude terminates, single-trace simulation time increases when network size grows. On the other hand, exhaustive search always provides a solid safety proof. For instances of similar network size, the search time for a safe instance (`good`) is considerably longer than that of an unsafe instance (`bad`). For instances of different sizes, as network size grows, exhaustive search time grows exponentially. Nevertheless, even for the 25-node scenario, exhaustive search can be completed in 12 minutes. As future work, we are going to scale our analysis technique to larger networks.

6 Related Work

Maude is a widely used tool for a variety of protocol analysis. In addition to our use of Maude for analyzing BGP instances, there is also a huge literature of using Maude for other complex systems, such as security protocols [11] , real-time systems [20], and active networking [5].

Theorem proving and model checking techniques have been applied to formal verification of network protocols. For instance, in [3], a routing protocol standard is formalized in the SPIN model checker and HOL theorem prover, where SPIN is used to check convergence of small network instances, and HOL is used to generalize the convergence proof for arbitrary network instances. Their work focuses on basic intra-AS routing protocols such as the distance-vector protocol, and does not consider policy interactions that occur in inter-AS routing protocols such as BGP. However, while our proofs are automated by Maude's built-in simulation and exhaustive search capabilities, we are restricted to analyzing specific network instances. As future work, we plan to generalize our instance-based proofs towards more general properties on BGP stability, by leveraging Maude's connection with existing theorem provers such as PVS [21].

Arye *et al.* [2] has attempted a similar formalization of eBGP gadgets in SPP using the Alloy [1] tool. Our approach differs from theirs in the overall goal of the formalization: Ayre *et al.* uses Alloy to synthesize eBGP instances that exhibit certain behaviors such as divergence, whereas our approach takes an eBGP instance as input and analyzes it via simulation runs and exhaustive search. Our overall goal is to provide an easy-to-use library in Maude that eases the entire process of specifying and analyzing a BGP

[6] Bad gadget always diverges and does not have any stable path assignment, therefore, when we simulate bad gadget with rewriting, Maude does not terminate, and we do not record the statistics.

instance. Besides, in addition to eBGP gadgets, our library also supports iBGP instances and handles iBGP route ranking generation based on a separate IGP protocol 4.2.

On the practical front, recent advances in model checking network protocol systems include *MaceMC* [16] and *CMC* [6] by imposing constraints on network implementations. While our analysis technique also uses state exploration of network instances, the major difference is that our method is aimed at analysis in the design phase instead of on actual implementation. Moreover, unlike checking a given implementation, our formal specifications of routing protocols and policies are extensible, and can be further customized to refine the analysis of various network properties.

7 Conclusion and Future Work

This paper presents our development of a Maude library for specifying and analyzing BGP instances. Our work aims to automate an important task for network designers when designing BGP protocols and safe policy guidelines. Our library uses Maude's object-based specification language and enables the user to easily generate Maude specification by only requiring them to define the network topology and routing policies. To validate the feasibility of our library, we explored a variety of well-known BGP gadgets and an actual BGP instance obtained from the Rocketfuel dataset, and demonstrated the use of Maude's analysis capabilities to detect possible divergence. All Maude code described in this paper is available at `http://netdb.cis.upenn.edu/discotec11`.

In addition to integrating our framework with the PVS theorem prover, our ongoing work includes: (1) more case studies on BGP instances and recent guidelines to explore the limits of our library, leading to possible extensions of our Maude library; and (2) releasing our tool for network designers to use.

Acknowledgment The authors would like to thank Yiqing Ren and Wenchao Zhou for their valuable help in generating the iBGP instances (from the Rocketfuel dataset) used in our analysis. This research is funded in part by NSF grants (IIS-0812270, CNS-0830949, CNS-0845552, TC-0905607, and CPS-0932397), AFOSR Grant No. FA9550-08-1-0352 and ONR Grant No. N00014-10-1-0365.

References

1. Alloy. `http://alloy.mit.edu/community/`.
2. M. Arye, R. Harrison, and R. Wang. The Next 10,000 BGP Gadgets: Lightweight Modeling for the Stable Paths Problem. Princeton COS598D course project report.
3. K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*. Springer, 2007.
5. G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The maude experience. *DARPA Information Survivability Conference and Exposition,*, 2000.

6. D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *NSDI*, 2004.
7. N. Feamster, R. Johari, and H. Balakrishnan. Implications of Autonomy for the Expressiveness of Policy Routing. In *ACM SIGCOMM*, Philadelphia, PA, August 2005.
8. A. Flavel and M. Roughan. Stable and flexible iBGP. In *ACM SIGCOMM*, 2009.
9. L. Gao, T. G. Griffin, and J. Rexford. Inherently Safe Backup Routing with BGP. In *IEEE INFOCOM*, 2001.
10. L. Gao and J. Rexford. Stable internet routing without global coordination. In *SIGMETRICS*, 2000.
11. A. Goodloe, C. A. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In *Proc. ACM WITS'05*, 2005.
12. T. G. Griffin. The stratified shortest-paths problem. In *COMSNETS*, Jan. 2010.
13. T. G. Griffin, A. Jaggard, and V. Ramacandran. Design principles of policy languages for path vector protocols. In *ACM SIGCOMM*, Aug. 2003.
14. T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE Trans. on Networking*, 10:232–243, 2002.
15. T. G. Griffin and G. Wilfong. On the correctness of IBGP configuration. SIGCOMM '02.
16. C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.
17. C. Labovitz, G. Malan, and F. Jahanian. Internet Routing Instability. *TON*, 1998.
18. Maude. `http://maude.cs.uiuc.edu/`.
19. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
20. P. C. Ölveczky and J. Meseguer. Real-time maude: A tool for simulating and analyzing real-time and hybrid systems. *Electr. Notes Theor. Comput. Sci.*, 36, 2000.
21. PVS Specification and Verification System. `http://pvs.csl.sri.com/`.
22. M. Schapira, Y. Zhu, and J. Rexford. Putting BGP on the right path: A case for next-hop routing. In *HotNets*, Oct. 2010.
23. N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *SIGCOMM'02*.
24. L. Subramanian, M. Caesar, C. T. Ee, M. Handley, M. Mao, S. Shenker, and I. Stoica. HLP: A Next-generation Interdomain Routing Protocol. In *SIGCOMM*, 2005.

A Details of Maude Code

A.1 Network State

Maude provides a built-in sort `Configuration` for the global state of a concurrent object-based system. In our library, we model BGP system state by defining two subsorts of `Configuration`: `Msg` and `Network`. `Msg` is the type for routing messages, and `Network` the type for the collection network nodes. To construct the `Network Configuration` from the constituting network nodes, we further introduce sort `Node`. `Node` is a sub-sort of `Object`, and is the type of each network node (routers) in the BGP system.

```
sorts Msg Network .
subsorts Network Msg < Configuration .
sort Node .
subsort Node < Object .
```

Given the above sorts, we can represent a BGP system state by a multiset of routing messages terms (of sort `Msg`) and network nodes terms (of sort `Node`). We will show how to construct terms representing routing messages and network nodes later in this section.

Router Object To distinguish a network node object from other Maude objects, we introduce a special Cid (class identifier) called `Router`. With `Router` Cid, we introduce the constructor of a network node (router) object. In Maude, `op` is the keyword defining constructors:

```
op Router : Oid .
op [_:_|_,_,_] : Oid Router RoutingTable BestPath Nb -> Node .
```

The constructor `[_:_|_,_,_]` takes five arguments: The first two are a unique identifier (of sort `Oid`) and a class identifier (the Cid constant `Router`). The `Oid` denotes the node's unique name in the network, the class identifier denotes the class type of the node. The last three arguments are object attributes: routing table (of sort `RoutingTable`), best path (of sort `BestPath`), and neighboring table (of sort `Nb`). Given the argument terms, a router object term (of sort `Node`) is constructed. Next we discuss a `Node` object's three attributes.

RoutingTable and BestPath attributes. At a given state, the routing table and best path attributes store node v's available paths P^v to reach destination 0, and its current best path that ranks the highest among P^v respectively. They are constructed as follows:

```
op routingTable:_ : List{Path} -> RoutingTable .
op bestPath:_ : Path -> BestPath.
```

The sort `Path` and its constructors are defined as follows:

```
sort Path .
op emptyPath : -> Path [ctor] .
op source:_,dest:_,pv:(_),metric:_ :
     Oid  Oid  ListOid  Metric -> Path .
```

There are two ways to construct a term of sort `Path`: a path is either an empty path (`emptyPath`), or a path from a source node (`Oid`) to a destination node (`Oid`) via a path vector (the list of intermediary nodes denoted by `List{Oid}`) at some cost of sort `Metric`. Sort `Metric` specify how paths are measured and compared. For example, a path metric in the SPP formalism is its rank defined in Λ. We interpret the metrics over Maude's built-in natural numbers sort `Nat`:

```
sort Metric .
subsorts Nat < Metric .
```

Neighbor attribute. A node's neighboring table holds its list of link information to reach its direct neighbors:

```
sort Neighbor .
op mkNeigh(_,_) : Oid Metric -> Neighbor.
```

Here the constructor `mkNeigh(_,_)` takes two arguments: the direct neighbor's identifier, and the cost (`Metric`) to reach this neighbor.

Example node object. The disagree gadget consists of two nodes `n1,n2` and a special destination `n0`. The two nodes are represented by two `Router` objects. We first introduce their object identifiers:

```
ops n0 n1 n2 : -> Oid .
```

At the initial state of BGP protocol execution, at node `n1` (which is connected to two neighbors `n0,n2`), the routing table contains one path `[n1 n0]` to reach `n0`, which is also node `n1`'s best path:

```
[n1 : router
   |routingTable: (source: n1, dest: n0, pv:(n1 n0), metric: 2),
    bestPath: (source: n1, dest: n0, pv:(n1 n0), metric: 2),
    nb: (mkNeigh(n0,2) mkNeigh(n2,1))]
```

Routing Messages Besides `Router` objects, the second part of a network state are routing messages in the network. Typically, network nodes exchange routing information by sending each other routing messages carrying newly-learned routing paths. In our library, it is specified as follows:

```
op sendPacket(_,_,_,_,_) : Oid Oid Oid Metric ListOid -> Msg
```

The first two `Oid` arguments denotes the sender's Oid and the receiver's Oid respectively. The rest of the arguments specify the destination Oid, metrics of the advertised path, and the path-vector (the list of network nodes along the path) of the routing path the message carries.

For example, in the disagree gadget, the initial routing message sent by node `n1` to its neighbors `n2` carrying its direct path to `n0` are is: `sendPacket(n1,n2,n0,2,n1 n0)`.

Protocol Dynamics In the following code snippet, we show the auxiliary functions used in computing new network state.

To compute a new path, `concat` is defined as follows:

```
op concat : Oid Oid Oid Metric ListOid ListNeighbor ->  Path .
eq concat(S, N, D, C, PV, NB)
     = (source: N, dest: D, pv:(N PV), metric: lookUpRank (N PV)) .
```

`eq` is a Maude keyword preceding equation definitions. Here the new path of `N`'s to reach `D` is simply by prepending `N` to `PV`: `N PV`. And the metric of (`N PV`) is determined by `N`'s routing policy, i.e., route ranking λ^N implemented by function `lookUpRank`. The definition of this function is specific to each BGP configuration instance, and is an interface between the protocol dynamics and the routing policies. We will revisit the specification of specific BGP instance's routing policies by `lookUpRank` in appendix C.

Given the newly computed route, to update a node's routing table, we use `updateRT` as follows:

```
op updateRT : Path ListPath -> ListPath .
eq updateRT (P, nil) = (P) .
eq updateRT (P, (P' RT)) =
 if ((getDest (P) == getDest (P')) and
     (getNext (P) == getNext (P')))
    then (P RT)
    else (P' updateRT (P, RT)) fi .
```

Note that, `updateRT` ensures that the routing table always keeps exactly one path from one particular next-hop neighbor.

Finally, if the best path attribute of the node changes, we use `multiCast` to generate routing messages:

```
op multiCast : ListNeighbor Path -> Configuration .
eq multiCast((NBentry NB'), (source: S, dest: D, pv:(PV), metric: C))
    = sendPacket (S, getOid(NBentry), D, C, (PV))
      multiCast (NB', (source: S, dest: D, pv:(PV), metric: C)) .
```

For each neighbor `NBentry` in `N`'s neighboring table, `multiCast` recursively generates the routing message `sendPacket(S,getOid(NBentry),D,C,(PV))`. `getOid` is an auxiliary-function that extracts the neighbor `NBentry`'s Oid.

A.2 Route Oscillation Detection Support

Logger object. The logger object consists of only one attribute: a history (list) of path assignments, each entry of which corresponds to the list of best paths for all nodes in the network at a given state. The Maude code for defining the logger object is shown below.

```
sort Logger .
op Logger : -> Cid .
op {_:_|_} : Oid Cid AttributeSet -> Logger  .

op history:_ : List{PathAssignment} -> Attribute .
op {_} : List{BP} -> PathAssignment .

sort BP .
op [_] : List{Oid} ->  BP .
```

The first three lines declares a special `Logger` object, and the constructor of the Logger, which takes three arguments, the first one is the identifier for the logger, similar to the identifiers for the router objects; the second argument is the class identifier of logger object; and the last one is the attribute of the logger. The next two lines declare the only attribute of `Logger`, which is a list (history) of `PathAssignment` elements, each of which denotes one path assignment in the network at a given state. A path assignment is a list of best path selected by each node in the network. The last two lines defines each entry in one path assignment, which is simply the best path for some network node: i.e. the list of nodes in the best path. An example logger object for Disagree is as follows:

```
{pa : Logger | history: ({[n1 n2 n0] [n2 n0]}
                         {[n1 n2 n0] [n2 n1 n0]}
                         {[n1 n2 n0] [n2 n0]}
                         {[n1 n0] [n2 n0]})}
```

Each line in the logger attribute records one snapshot of the best path assignment. And each column records the evolution of a node's best path assignment. For example, the first column shows node `n1`'s best path updates.

B SPP encoding of eBGP Instances

Network topology. The network topology G, o is represented by three constants: `n0`, `top-Nodes`, `top-BGP`:

```
op n0 : Oid .
op top-Nodes : -> ListOid .
op top-BGP : -> Topology .
```

`n0` is the specific destination o, `top-Nodes` the set of network nodes, and `top-BGP` the set of BGP links. Our library has pre-defined sort `Topology` to capture that a network topology is a set of labeled network links:

```
sorts Link Topology .
op (_,_:_) : Oid Oid Metric -> Link .

subsort Link < Topology .
op __ : Topology Topology -> Topology .
```

The first line of Maude code declares `Topology` and `Link`; The second line says a `Link` is constructed from its two end nodes, and the associated metric. The last two lines specify how `Topology` is constructed: a topology is either a single link, or recursively constructed from existing topologies.

Our Maude library automatically generates an eBGP instance's initial state based on its topology:

```
op gadget : -> Configuration .
eq gadget = init-config (top-Nodes, top-BGP) .
```

C SPP encoding of iBGP Instances

An iBGP configuration instance $C = (G_P, G_S, X)$ is defined by its physical topology G_P, signaling topology G_S, and gateway (egress) BGP nodes X. G_P represents the underlying network topology that runs a separate IGP protocol, therefore we also call G_P the *IGP topology*. G_S represents the network topology that runs iBGP. The iBGP links in G_S can be partitioned into three classes `over,down,up`: an `over` iBGP link represents a vanilla iBGP link, a `down` iBGP link represents a iBGP session from a

iBGP *reflector*[7] node to its client, and an `up` link represents that from a client to its reflector server. X represents the gateway (egress) BGP nodes from which external routes (routes to destinations outside the AS) are learned.

While the eBGP instance is usually given in the form of SPP, we need one additional translation [15] to transform iBGP instance into SPP. Because we are interested in the behavior of an iBGP instance $C = (G_P, G_S, X)$ in distributing external routing information learned from iBGP gateway routers, we define an iBGP instance's corresponding SPP representation $S = (G, o, P, \Lambda)$ as follows: o is an additional network node outside the iBGP instance, and it represents the external common destination; G is the signaling topology G_s with the additional node o and (non-BGP) links between o and egress nodes X; Λ is the function that computes IGP-distance. This is because, within an AS, for a common external destination o, all routes' AS-level metrics are same, as a result, a route can uses its IGP-distance alone to decide its rank.

To automate an iBGP instance specification in Maude, for each iBGP instance $C = (G_P, G_S, X)$, we provide additional Maude definitions to generate its SPP reorientation (G, o, P, Λ). We describe network topology G, o and routing policy Λ respectively.

Network topology. Similar to eBGP, iBGP network topology G, o is represented by constants `top-BGP, nd`. Rather than asking the user to manually input top-BGP as in eBGP instance, our library generates `top-BGP` from iBGP signaling topology as follows:

```
op top-iBGP-signal : -> Topology .
ops top-Nodes top-Xset : -> ListOid .
eq top-BGP = addExternal (top-iBGP-signal, top-Xset, nd) .
```

`top-iBGP-signal, top-Xset` stores the iBGP instance's signaling topology G_S and egress nodes X. Note that, while the metric of each link in an eBGP network topology represents the associated link cost (e.g. the IGP distance between the two nodes), the metric of link in signaling topology denotes its class: over, up, or down. So our library further includes three metric constants:

```
ops up down over : -> Metric .
```

`addExternal` is a function that takes the external destination `nd`, egress nodes `top-Xset` as input, and generates the network topology `top-BGP` by adding to the signaling topology `top-iBGP-signal` additional links between each egress node in X and external destination o:

```
op addExternal : Topology ListOid Oid -> Topology .
eq addExternal(top, (X Xset), D) =
      addExternal (top, Xset, D) (X,D : 1) .
eq addExternal (top, nil, D) = top .
```

[7] To solve scalability problem in full-mesh iBGP configuration, some iBGP nodes are elected to be route reflectors that act as focal point in iBGP sessions: the reflectors form a smaller full-mesh, and the rest of the nodes become their clients.

The resulting `top-BGP` is then used to initialize network state as in eBGP instance.

Based on the above library support, to specify the network topology of iBGP instance, the user only need to specify G_P, X by customizing constants `top-Nodes`, `top-iBGP-signal`, `top-Xset`. For example, to specify a 6-node iBGP instance [8], we write:

```
eq top-Nodes = n0 n1 n2 n3 n4 n5 .
eq top-Xset = n3 n4 n5 .
eq top-iBGP-signal =
  (n0,n1 : over) (n0,n2 : over) (n0,n3 : down)
  (n1,n0 : over) (n1,n2 : over) (n1,n4 : down)
  (n2,n0 : over) (n2,n1 : over) (n2,n5 : down)
  (n3,n0 : up) (n4,n1 : up) (n5,n2 : up) .
```

Routing policy Like eBGP instance, routing policy Λ is given by customizing `import`, `export`, and `lookUpRank`. But unlike eBGP instance where `lookUpRank` simply assigns each path its rank, iBGP routing policy is more complex.

An iBGP policy consists of two parts: First, a valid iBGP path consists of a set of (can be empty) `up` links, which is followed by zero or one `over` link followed by a set (can be empty) of `down` links. Second, for routes with same AS-level attribute, a node always prefers routes with lower IGP distance, i.e., routes with shorter distance to a egress node.

The first policy is achieved by imposing an export policy at each node, such that only routing updates from a client will be exported to all neighbors. In Maude library, this is achieved by defining `export` in the iBGP module as follows:

```
*** for routes learned from internal nodes
*** only allow routing path of the form:
***   ... up ... up (over) down down ...
eq export ((source: S, dest: D, pv:(S N N' PV), metric: C)) =
    if (getLinkMetric (S,N,top-iBGP) == over and
        getLinkMetric (N,N',top-iBGP) == over)
    then emptyPath
    else (source: S, dest: D, pv:(S N N' PV), metric: C) fi .

*** for routes learned from egress nodes, do nothing
eq export ((source: S, dest: D, pv:(S N), metric: C)) =
  (source: S, dest: D, pv:(S N), metric: C) .
```

The second iBGP policy is achieved by set a path's rank to its IGP distance:

```
eq lookUpRank (PV) =
computeIGP (head (PV), last(front(PV)), top-IGP) .
```

Here `head(PV)` is the source of the path `PV`, and `last(front(PV))` is the egress node `head(PV)` used to reach destination `last(PV)`. And `top-IGP` is the underlying IGP topology. The function `computeIGP` computes the IGP distance between the source and the egress node according to the underlying IGP topology. In our Maude library,

we implement `computeIGP` by implementing a separate IGP protocol called shortest-path protocol. In shortest-path protocol, the IGP distance between two nodes is the cost of the shortest path between them.

Based on the above library support, to specify iBGP `lookUpRank`, the user only needs to specify the underlying IGP topology G_P by providing proper definition of `top-IGP`. For example, to specify the 6-node iBGP instance, we write:

```
eq top-IGP =
  (n0,n3 : 10) (n0,n4 : 5) (n1,n4 : 10)
  (n1,n5 : 5) (n2,n5 : 10) (n2,n3 : 5) .
```

Here the link metric `5,10` specifies the IGP distance between neighboring nodes.