Toward a Lightweight Model of BGP Safety

Matvey Arye
Princeton University

Rob Harrison
Princeton University

Richard Wang
Princeton University

Pamela Zave
AT&T Labs

Jennifer Rexford
Princeton University

***Abstract*—For the past ten years, researchers have used the Stable Paths Problem (SPP) to analyze the stability properties of the Border Gateway Protocol (BGP). Analysis of SPP has revealed several combinations of topologies and routing configurations (or *gadgets*) where BGP cannot converge to a unique stable solution. Researchers typically analyze SPP by hand, using a trial-and-error process to (i) generate small SPP instances that exhibit undesirable properties and (ii) prove sufficient conditions for SPP solvability. In this paper, we present a formal, machine-readable SPP model encoded in the Alloy lightweight modeling language. The model consists of nodes with ranked sets of permitted paths, and a set of predicates for analyzing SPP instances. We also present an automated way to generate the smallest set of unique gadgets with no stable solution, or multiple stable solutions, including some previously-unknown gadgets. We use our model to verify sufficient conditions on SPP instances (e.g., "no dispute wheel implies a unique stable solution"), illustrating how our Alloy model can aid researchers in analyzing BGP.**

I. Introduction

The Border Gateway Protocol (BGP)—the Internet's interdomain routing protocol—is notoriously vulnerable to oscillation. Conflicting local policies in different Autonomous Systems (ASes) can prevent the routing system from converging to a stable selection of routes to a destination. Over the past ten years, the research community has made tremendous progress in understanding BGP *safety*, including example unsafe configurations (or *gadgets*) and sufficient conditions on local policies to ensure a safe routing system [1]–[10]. In this paper, we explore how modern model checkers can aid research on BGP safety by freeing researchers from much of the manual effort of constructing counter-examples and proofs, while providing a convenient way to explore new ideas and precisely communicate their results to others.

Research on BGP safety often uses a more abstract model of BGP-like protocols known as the Stable Paths Problem (SPP) [3]. Any BGP configuration can be translated to an SPP instance, where each node has a ranked list of permitted paths and selects the highest-ranked path consistent with its neighbors' choices. Figure 1 shows three example SPP instances. Using this static representation of the routing system, researchers can identify SPP instances that have no stable states or multiple stable states, and prove whether certain constraints are sufficient to ensure safety. As such, our first step is to construct and evaluate a model of SPP, as discussed in the rest of this paper; that is, we focus on BGP *solvability*—determining whether SPP instances are unsolvable, uniquely solvable, or multiply solvable. While unique solvability is a necessary condition for BGP safety [9], having a single stable state is *not* a sufficient condition. Ultimately, a complete treatment of BGP safety will require a dynamic model, such as the Simple Path Vector Protocol (SPVP) [2]. We plan to consider protocol dynamics as part of future work.

To analyze BGP solvability, we use Alloy [11] to create a *machine-readable* definition of SPP that is nearly as concise as the version presented in the original SPP paper [3]. Then, we use the Alloy Analyzer to automatically analyze SPP instances up to a specified size, to either verify the assertions or generate counter-examples. Sometimes we specify assertions we know to be false (e.g., "all SPP instances are uniquely solvable") to generate "counter-examples" of unsafe gadgets. Other times we specify assertions we expect to be true (e.g., "no dispute wheel implies unique solvability") and verify that it holds for all instances smaller than the given size. Developing such a model is more lightweight than developing a formal proof but the analysis is limited to a given instance size. In the process, we *automatically* generate many of the misbehaving gadgets that have dominated the BGP literature, along with some interesting new gadgets. We also automatically verify known sufficient conditions for BGP solvability, and discover some promising directions for future research.

This paper makes the following contributions. First, we provide a machine-readable SPP model immediately available for automatic, "push-button" analysis. Informally-proven assertions about SPP instances can now be quickly and accurately validated on topologies of varying sizes. Section II presents the Alloy model in a series of steps, ranging from the model of nodes and paths, to the permitted paths, and ultimately to the solvability assertions. Section III presents our iterative process for reducing counter-examples by iteratively adding constraints to the model to prevent the Alloy Analyzer from generating isomorphisms of previously-generated examples. This section also shows an example of using Alloy Analyzer to verify a sufficient condition for unique solvability. We present related work in Section IV and conclude the paper in Section V.

II. Stable Paths Problem as an Alloy Model

Before describing our Alloy model of SPP, we introduce some basic concepts of the Alloy Modeling Language and the Alloy Analyzer. The basic building blocks of an Alloy model are sets of *atoms* and relations between those atoms. A model consists of *signatures* that define these atoms and constraints, called *facts*, on these atoms. Each unique combination of atoms and relations that satisfies these constraints is called an *instance*. Finally, the programmer makes assertions about these instances using *predicates* that evaluate some property of the model. The Alloy Analyzer then reports if any instances

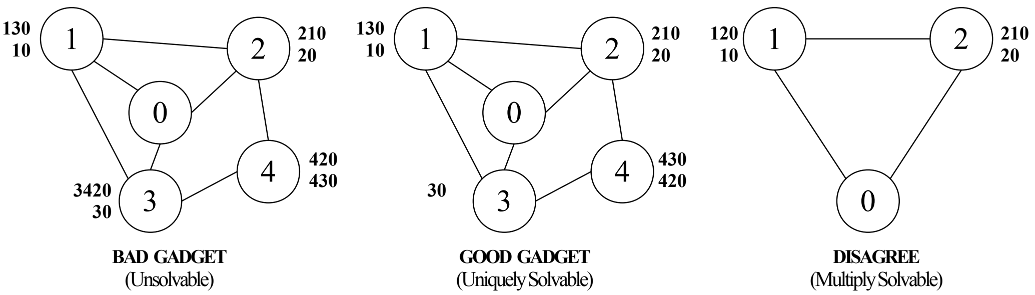


Fig. 1. We classify instances, called gadgets, of the SPP according to their solvability properties: unsolvable, solvable, and multiply solvable. The above figure shows an example of each type of instance. Numbered nodes represent abstract models of autonomous systems. The 0 node, called the destination, is special in that all other nodes share information about paths for reaching the destination. Listed next to each node is a set of paths that the node is willing to take to the destination, ordered in a highest-lowest preference. For example, Node 1 in BAD GADGET prefers the path from 1-3-0 more than its own path 1-0.

violate these assertions. Such instances are called *counter-examples*. This particular feature of Alloy makes it a good fit for studying SPP; with the appropriate constraints on paths and the (clearly erroneous) assertion that all SPP instances are solvable (or uniquely solvable), the Alloy Analyzer will generate counter-examples like the gadgets in Figure 1.

A. Describing SPP Instances in Alloy

An SPP instance consists of a set of nodes, which abstractly represent Autonomous Systems, and a set of paths between those nodes. There is one *destination node* to which all other *source nodes* select a path. Each source has a set of *permitted paths* it is willing to select; these paths are uniquely ranked and can only be the *empty path* or a sequence of source nodes ending at the destination. These core components of SPP are easily expressed in Alloy, as shown in Figure 2.[1] Nodes and paths are defined as atoms, and permitted paths are defined as a relation between a source and a sequence of paths. The position of a path in the sequence represents its ranking, where the first path is the most preferred.

```
abstract sig Node {}

one sig DstNode extends Node {}
sig SrcNodes extends Node {PermittedPaths: seq Path}

abstract sig Path {}
one sig EmptyPath extends Path {}
sig NonEmptyPaths extends Path {nodes: seq Node}
```

Fig. 2. An SPP instance consists of source nodes and a single destination node. Permitted paths are an ordered sequence of nodes or an empty path.

The SPP formalism imposes constraints on paths and the ranking of permitted paths, as captured in Figures 3 and 4, respectively. The first fact in Figure 3 requires that each non-empty path terminates at the destination node and contains no cycles. We implicitly assume a fully-connected graph, since any misbehaving gadget remains a misbehaving gadget after adding additional links. As such, we do not model the links in the topology or check that a path is feasible on the graph which simplifies the model. The second fact states that paths are unique.

```
fact validNonEmptyPaths {
  all path : NonEmptyPaths |
    let path_nodes = path.nodes |
      some path_nodes and
      !(path_nodes.hasDups) and
      path_nodes.last = DstNode }

fact noRepeatedPaths {
  all disj p1,p2 : NonEmptyPaths |
    p1.nodes != p2.nodes }
```

Fig. 3. `validNonEmptyPaths` constrains the set of non-empty paths to ensure that they consist of some nodes, have no repeated nodes, and terminate at the destination node. `noRepeatedPaths` ensures that each unique non-empty path traverses a different sequence of nodes. The | symbol is read "such that".

Finally, Figure 4 shows the facts that constrain the permitted paths at each source node. The first fact ensures that each source's set of permitted paths contains at least one non-empty path, that all non-empty paths begin with that particular source, and that the set of permitted paths contains the empty path which is always ranked last. The second fact ensures that each path included in the model is a permitted path for some source node, to avoid generating numerous equivalent SPP instances. Together, the code from Figures 2-4 completely specifies valid SPP instances. This model is nearly as concise as the original description of SPP in the literature [3], and yet has a machine-readable form amenable to automated analysis.

B. Asserting Solvability of SPP Instances

Solvability of an SPP instance is based on a set of *selected paths*. This set, also called a *selection*, contains exactly one path from the `PermittedPaths` of every `SrcNode` and thus represents a unique mapping from `SrcNodes` to a single `Path` beginning at that node. Based on the paths selected at adjacent nodes, each `SrcNode` derives a set of `choices`. This set contains all the paths the `SrcNode` could possibly

[1] While we have not provided a formal treatment of Alloy language syntax, we believe that the code is fairly intuitive on its own and should be somewhat understandable even to the unfamiliar reader.

```
fact validPermittedPaths {
 all snode: SrcNodes |
 let rankedPaths = snode.PermittedPaths |
   rankedPaths.first != EmptyPath and
   rankedPaths.last = EmptyPath and
   not (rankedPaths.hasDups) and
   all p: rankedPaths.elems |
      p.nodes.first = snode }

fact allPathsBelongToSomePermittedPath {
 all p : NonEmptyPaths |
   let first_node = p.nodes.first |
   some first_node.PermittedPaths.indsOf[p] }
```

Fig. 4. The fact validPermittedPaths constrains the set of permitted paths so that at there is at least one non-empty path per node, no path has two ranks, the lowest ranked path is the empty path, and the first node in each ranked path is the source node. allPathsBelongToSomePermittedPaths specifies that each path atom is part of a permitted path for some node.

take to get to the DstNode either directly or through one of the selected paths of its adjacent nodes. A particular selection is considered a *solution* to an instance of the SPP if, at each source node, the selected path is the highest ranked path from its set of choices. Intuitively, a solution is stable because in a dynamic setting no node would change its selected path since it is the most preferred choice given the paths available from its neighbors. An SPP instance with at least one solution is *solvable*; it is *uniquely solvable* if there is exactly one solution and *multiply solvable* if several solutions exist.

We now build a predicate that tests whether an SPP instance is *uniquely* solvable by quantifying over all possible selections and checking whether they are solutions. For simplicity, we consider a topology with three source nodes. We represent a single path by a SrcNode→Rank[2] *path tuple* where the Rank corresponds to the preference index of the path in the PermittedPaths relation of the SrcNode in the tuple. Recall that a selection contains a single path from each SrcNode's set of permitted paths. Thus a valid selection is a set of path tuples with a single tuple for every SrcNode in the instance, as seen in lines 6-8 of Figure 5. We explore all possible selections by quantifying over every possible path ranking. Each unique combination of rankings corresponds to a single unique valid selection and the one keyword on line 3 verifies that there is only a single such selection that is a solution, as required for unique solvability. By quantifying over the node rankings, rather than all sets of three paths, we substantially reduce the search space the Analyzer must explore. The predicate is specific to a topology with three source nodes because Alloy does not support second-order logic which is necessary to quantify over a variable number of items. Instead, we simply (and automatically) generate a separate predicate for each SPP instance size we analyze.

We now define what it means for a selection to be a solution.

```
1 pred OneSolvable3[] {
2   some disj node1, node2, node3: SrcNodes |
3   one rank1: (node1.PermittedPaths).inds,
4   rank2: (node2.PermittedPaths).inds,
5   rank3: (node3.PermittedPaths).inds |
6   let selection = (node1→rank1)
7                 + (node2→rank2)
8                 + (node3→rank3) |
9        SelectionIsSolution[selection] }
```

Fig. 5. oneSolvable3 checks that an SPP instance is uniquely solvable by quantifying over all possible selections and ensuring that only one is a solution. We can include multiply-solvable instances simply by changing the keyword one to some.

A selection is a solution if and only if it contains the paths that are the most highly ranked paths of the set of choices for each node. This constraint is illustrated in Figure 6.

```
pred SelectionIsSolution
  [selected: SrcNodes→seq/Int] {
    let choices = GetChoices[selected] |
    selected = GetBest[choices] }
```

Fig. 6. SelectionIsSolution takes a selection in the form of SrcNode→Rank and determines whether it is a solution. First, it generates the list of choices based on the selection. Next it tests that the selection is equivalent to the best paths for each node from of its list of choices.

Last, we define the functions GetChoices and GetBest in Figure 7. GetChoices returns all SrcNode→Rank pairs that satisfy isValidChoice—that is, all pairs where the path corresponds to the EmptyPath, goes directly to the DstNode, or is consistent with a neighbor's selected path. To find the rank of the neighbor's selected path, line 10 performs a *join*, denoted by the period, that takes the selected set of SrcNode→Rank tuples and returns the Rank in the tuple where the first element equals secondNode. The function GetBest filters the set of SrcNode→Rank tuples to return the tuple with the best Rank for each SrcNode. Line 20 performs a join to get the set of Ranks contained in the set of choices for a node.

III. Using Alloy to Analyze SPP Solvability

Our Alloy model enables us to study solvability properties of different SPP instances. We first analyze unsolvable gadgets (with no stable state) and multiply solvable gadgets (with multiple stable states), after eliminating redundant gadgets that share the same basic structure. Then, we use Alloy to verify constraints on SPP solvability. Finally, we use Alloy to explore why the dispute wheel is not a necessary condition for solvability.

A. Elimination of Redundant SPP Gadgets

Our model finds many unsolvable and multiply-solvable SPP instances. However, many of these gadgets are, in fact, very similar. Some gadgets differ only in the numbering of the nodes and associated paths, and others simply have extra paths

[2] This is Alloy notation for a two-element tuple where the first element is a SrcNode and the second is a Rank, which is not a formal type in Alloy but used here for clarity.

```
fun GetChoices [selected: SrcNodes →seq/Int]:
 SrcNodes →seq/Int {{
  node: SrcNodes, rank: seq/Int |
      isValidChoice[node, rank, selected] }}

pred isValidChoice [node: SrcNodes,
  rank: seq/Int, selected : SrcNodes →seq/Int]{
    let path = node.PermittedPaths[rank] |
    let secondNode = path.nodes[1] |
    let secondNodeSelRank = second_node.selected |
    let secondNodeSelPath =
       secondNode.PermittedPaths[secondNodeSelRank] |
     (path = EmptyPath) or
     (secondNode = DstNode) or
     (path.nodes.rest = secondNodeSelPath.nodes)

fun GetBest[choices: SrcNodes →seq/Int]:
 SrcNodes →seq/Int {{
  node: SrcNodes, bestRank: seq/Int |
  let choiceRanksForNode = node.choices |
  bestRank = min[choiceRanksForNode] }}
```

Fig. 7. Given a selection, `GetChoices` returns all choices for all nodes. `isValidChoice` tests if a choice is valid. `GetBest` filters the `SrcNode`→`Rank` tuples, returning the best tuple for each node.

```
pred Disagree {
  some node1, node2: SrcNode |
  some node1path2, node1path1
      node2path2, node2path1: Path |
  some disj i12, i11: (node1.PermittedPaths).inds |
  some disj i22, i21: (node1.PermittedPaths).inds |

  node1path2.nodes[0] = node1 &&
  node1path2.nodes[1] = node2 &&
  node1path2.nodes[2] = DstNode &&
  node1path1.nodes[0] = node1 &&
  node1path1.nodes[1] = DstNode &&
  node2path2.nodes[0] = node2 &&
  node2path2.nodes[1] = node1 &&
  node2path2.nodes[2] = DstNode &&
  node2path1.nodes[0] = node2 &&
  node2path1.nodes[1] = DstNode &&

  node1.PermittedPaths[i12] = node1path2 &&
  node1.PermittedPaths[i11] = node1path1 &&
  node2.PermittedPaths[i22] = node2path2 &&
  node2.PermittedPaths[i21] = node2path1 &&
  i12 < i11 && i22 < i21 }
```

Fig. 8. Representing a generic version of DISAGREE.

that have no influence on solvability. Manually inspecting all of these gadgets is immensely tedious and provides little insight. Instead, we need an effective way to automatically characterize and eliminate these redundant gadgets to produce a minimal set of distinct unsolvable gadgets.

The Alloy Analyzer generates counterexamples to the programmer's assertions in an arbitrary order. As a result, our model could return large gadgets with additional paths that do not impact solvability. To generate a minimal set of gadgets, we apply an iterative process where we (i) force the Analyzer to generate small gadgets (by constraining the model to a small number of source nodes and permitted paths) that violate our solvability predicate and (ii) encode each new gadget as a predicate that excludes all such gadgets on the next execution of the Analyzer. After exhausting all gadgets of a given size, we consider gadgets with one additional node or path to see if any new gadgets (besides embeddings of smaller gadgets) arise. This automated process produces a small set of generic unsovlable gadgets up to a particular target size.

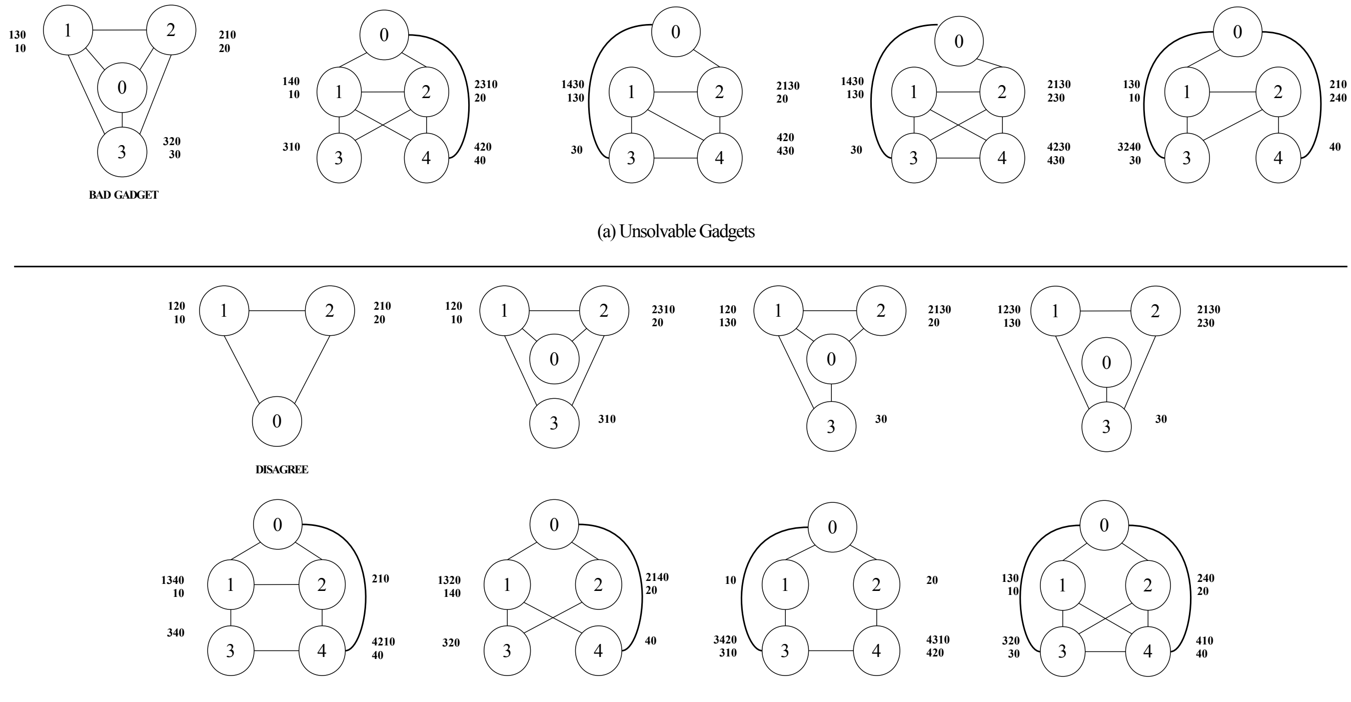
Figure 8 shows an elimination predicate for the DISAGREE gadget, where each of two source nodes prefers the path through the other node over its own direct path to the destination. To generalize the gadget, the nodes are quantified with the `some` quantifier, so any renumbering of nodes would still match the predicate. In addition the predicate specifies the *relative* order of path rankings, instead of fixed ranking numbers. As a result, any gadget that has the specified paths in the correct relative order will still match the predicate including those instances with extra paths that do not effect solvability.

Using this automated process, we fully explored gadgets with up to four source nodes, which required several days of processing. The automated analysis produced the 13 gadgets—five unsolvable instances and eight multiply-solvable instances—shown in Figure 9. The well-known DISAGREE and BAD GADGET have exactly two permitted paths per node. Many of the other gadgets are very similar to these two well-known instances, just with additional nodes. For example, the gadget to the right of DISAGREE is remarkably similar to DISAGREE except for the addition of node 3. These gadgets, which would have been difficult to generate by hand, provide useful insight when we explore new conditions for solvability.

B. Verifying a Sufficient Condition for Solvability

In addition to identifying interesting gadgets, BGP researchers also look for conditions that classify SPP instances as solvable or potentially unsolvable. The *dispute wheel* is a well-known construct for determining SPP solvability; if no set of nodes in an SPP instance form a dispute wheel, then the instance is safe—guaranteed to converge to a unique, stable solution [3]. However, some SPP instances that have a dispute wheel are still safe. Therefore, the property "no dispute wheel" is a sufficient, but not necessary, condition for safety (and unique solvability).

To verify these results, we created a predicate that checks for the existence of a dispute wheel. A dispute wheel consists of a circular chain of nodes, where each node has a *rim path* and a *pass-through path*, where the rim path of the current node always goes through the pass-through path of the next node in the chain. Any node in the chain is either a pivot node—where the rim path is more preferred than the pass-through path (which, in this case, is also called a leg path)—or a non-pivot node—where the rim path is the same as the pass-through path. A predicate that can check for the existence of a dispute wheel in systems of three source nodes is presented in Figure 10. The `DWLink` function checks that two consecutive



(a) Unsolvable Gadgets

(b) Multiply Solvable Gadgets

Fig. 9. These are the 13 simplest unique representations of all gadgets for instances of SPP under four nodes.

nodes form a valid link in the chain and the `DW3` predicate uses `DWLink` to check for chains of three nodes.

With these predicates, we can now verify well-known properties about dispute wheels. Our model verifies the assertion that no dispute wheel implies unique solvability (*i.e.*, the test for sufficiency passes). We can also verify that the presence of a dispute wheel does *not* imply that an instance is not uniquely solvable (*i.e.*, the test for necessity fails) as Alloy can provides us with many counterexamples.

C. Exploring New Conditions for Unique Solvability

We suspect that a stronger condition that is both necessary *and* sufficient could be built by starting with the dispute wheel predicate and progressively adding conditions that prevent the predicate from matching uniquely-solvable SPP instances. Using our model, we can validate (or invalidate) such new conditions for necessity and sufficiency of unique solvability in a “push-button” manner.

Many of the uniquely-solvable instances contain *latent* dispute wheels. In these instances, some nodes form a dispute wheel, but some node in the wheel also possesses a permitted path that is more preferred than the rim and pass-through paths, preventing the selection of the rim path and ultimately preventing the dispute wheel from affecting the instance’s solvability. By ignoring the rankings of paths *outside* of the wheel, the dispute wheel construction does *not* ensure that the contention actually manifests itself.

```
1  pred DWLink
2    [ currPassThruRank: seq/Int, currNode:SrcNodes,
3    nxtPassThruPath: NonEmptyPaths] {
4    some currRimRank:
5      prevs[currPassThruRank]+ currPassThruRank |
6    let currRimPath =
7      currNode.PermittedPaths[currRimRank] |
8    currRimPath in NonEmptyPath and
9    currRimPath.nodes.rest =
10     nxtPassThruPath.nodes }
11
12 pred DW3 [] {
13     some disj node1, node2, node3: SrcNode |
14     some rank1: (node1.PermittedPaths).inds |
15     some rank2: (node2.PermittedPaths).inds |
16     some rank3: (node3.PermittedPaths).inds |
17     let path1 = node1.PermittedPaths[rank1] |
18     let path2 = node2.PermittedPaths[rank2] |
19     let path3 = node2.PermittedPaths[rank3] |
20     DWLink[rank1, node1, path2] and
21     DWLink[rank2, node2, path3] and
22     DWLink[rank3, node3, path1] }
```

Fig. 10. `DWLink` checks that `currNode` and can be a link in dispute wheel given that `nxtPassThruPath` is the leg of the next node. On line 5 `prevs` is an Alloy function that returns the set of all ranks less than `currPassThruRank`. So the rim rank of the current node is either less than or equal to the pass-through rank of the node which corresponds to the current node being a pivot and non-pivot node, respectively. `DW3` checks that there is a dispute wheel of three nodes.

First, we were curious if the dispute wheel condition is necessary in cases where a latent dispute wheel could not occur. To answer this question we decided to explore the set of instances with *unembedded* dispute wheels. A dispute wheel is unembedded if all of the paths in the SPP instance are either rim paths or pass-through paths and every node is

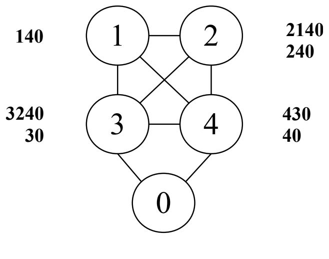


Fig. 11. A gadget with an unembedded dispute wheel that is still solvable.

either a pivot or non-pivot node. Such an instance cannot have a latent dispute wheel since all non-empty permitted paths are part of the dispute wheel. We ran a test for necessity on this set of unembedded instances. To our surprise, we found a counterexample—a solvable instance with an unembedded dispute wheel, as shown in Figure 11. This instance actually has a single stable solution: the empty path for nodes 1 and 2, and 30 and 430 for nodes 3 and 4, respectively. Yet all of the paths in the instance are part of the dispute wheel[3].

We developed a revised dispute wheel predicate to exclude instances like the one in Figure 11. The revised dispute wheel ensures that each leg path is an *independent leg*. A path is an independent leg if only the first node of the path belongs to the set of pivot and non-pivot nodes. The original dispute wheel has no such constraint—a valid leg path could be any permitted path in the SPP instance. The revised dispute wheel is more tightly constrained than the original predicate. Thus, we run a test for sufficiency to ensure that we did not over-constrain the predicate. We verified that "no revised dispute wheel" did indeed imply solvability on topologies of up to four source nodes. We verified that all SPP instances with an unembedded revised dispute wheel are not uniquely solvable. Therefore, this predicate passes the test for necessity on unembedded instances, a test which failed for the original dispute wheel condition.

We are still working on the set of constraints that is both necessary and sufficient for BGP safety. Our Alloy model has been invaluable by providing us with counterexamples to our assertions. This allows us to test our constraints much more quickly than developing proofs for each assertion, leading to a faster development cycle for our research.

IV. Related Work

Lightweight modeling of network protocols: Zave's work on verification of Chord demonstrated the utility of applying lightweight modeling to network protocols [12]. Our study analyzes *configurations* of BGP policy rather than the protocol itself. Also, Zave's work modeled protocol dynamics, whereas we consider a purely static model of BGP path selection.

BGP modeling: The static SPP model [3] and the dynamic SPVP model [2], provide rigorous ways to reason about BGP safety. Subsequent work identified conditions, some necessary and others sufficient, for ensuring BGP safety [1]–[10]. In our work, we represent SPP in the Alloy modeling language and automatically generate instances that are unsolvable (or multiply-solvable) and verify conditions for BGP solvability. Recent research shows that all multiply-solvable instances are subject to persistent routing oscillation [9]. Still, to verify the full range of known conditions for BGP safety, we need to go beyond our static model (of SPP) to a dynamic model (of SPVP).

Formal methods applied to BGP: Recent work on Formally Safe Routing (FSR) [13] supports automatic analysis of SPP instances (as well as more abstract routing policies) expressed using routing algebra. The FSR work shows how to translate SPP instances and routing policies into integer constraints, enabling the use of SMT solvers to analyze safety. Rather than analyzing existing SPP instances, our models *generate* SPP instances that (dis)obey solvability conditions.

V. Conclusions and Future Work

The end goal of our research is to use lightweight verification to improve how researchers (and perhaps ultimately practitioners) analyze BGP safety. This paper takes an important first step by modeling BGP *solvability* using the Alloy modeling language and the Alloy Analyzer. In future work, we will expand our models to capture the protocol dynamics of SPVP, so we can use Alloy to reason about *safety* and verify the relationship between the static and dynamic models of BGP. Further, accounting for SPVP dynamics permits the verification and exploration of other well-known constraints on SPP instances such as the *Gao-Rexford* conditions and *Safety Under Filtering*. We also want to use our models to explore new necessary and sufficient conditions for BGP safety.

References

[1] T. G. Griffin and G. Wilfong, "An analysis of BGP convergence properties," in *ACM SIGCOMM*, pp. 277–288, 1999.

[2] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "Policy disputes in path vector protocols," in *IEEE ICNP*, pp. 21–30, 1999.

[3] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *IEEE/ACM Trans. on Networking*, vol. 10, no. 2, pp. 232–243, 2002.

[4] L. Gao and J. Rexford, "Stable Internet routing without global coordination," *IEEE/ACM Trans. on Networking*, vol. 9, no. 6, pp. 681–692, 2001.

[5] L. Gao, T. G. Griffin, and J. Rexford, "Inherently safe backup routing with BGP," in *IEEE INFOCOM*, 2001.

[6] T. G. Griffin, A. Jaggard, and V. Ramachandran, "Design principles of policy languages for path vector protocols," in *ACM SIGCOMM*, 2003.

[7] T. G. Griffin and J. L. Sobrinho, "Metarouting," in *ACM SIGCOMM*, 2005.

[8] N. Feamster, R. Johari, and H. Balakrishnan, "Implications of autonomy for the expressiveness of policy routing," in *ACM SIGCOMM*, 2005.

[9] R. Sami, M. Schapira, and A. Zohar, "Searching for stability in interdomain routing," in *IEEE INFOCOM*, pp. 549–557, 2009.

[10] Y. Wang, M. Schapira, and J. Rexford, "Neighbor-specific BGP: More flexible routing policies while improving global stability," in *ACM SIGMETRICS*, pp. 217–228, 2009.

[11] "Alloy community." http://alloy.mit.edu/community/.

[12] P. Zave, "Why the Chord ring-maintenance protocol is not correct (Extended Abstract)," tech. rep., AT&T Research, March 2011.

[13] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott, "FSR: Formal analysis and implementation toolkit for safe interdomain routing," Tech. Rep. MS-CIS-11-10, U. Pennsylvania, May 2011.

[3] The dispute wheel includes nodes 3→2→1→4. Paths 2140, 3240, and 430 are the rim paths. Paths 240, 30, and 40 are the pass-through paths. Path 140 is both a rim and pass-through path since node 1 is a non-pivot node.