The Next 10,000 BGP Gadgets:
Lightweight Modeling for the Stable Paths Problem

Matvey Arye, Rob Harrison, Richard Wang
Department of Computer Science
Princeton University
{arye,robh2,rwthree}@cs.princeton.edu

Abstract

For the past ten years, the de facto interdomain routing protocol, the Border Gateway Protocol (BGP), has been been studied in the framework of the Stable Paths Problem (SPP). The SPP approach revealed several combinations of node topologies and configurations in which BGP cannot converge to a stable solution. These misbehaving "gadgets" serve as counterexamples to desirable BGP behavior, which we would like to always converge on a stable solution. For just as long, researchers have been manually generating these gadgets by hand which is a labor intensive and incomplete process. We apply two different lightweight modeling techniques in order to automatically generate these misbehaving gadgets quickly and completely. We model the static Stable Paths Problem using Alloy and also the dynamic Stable Path Vector Protocol using the Promela/Spin platform. These two approaches serve as a case study both in the application of lightweight modeling to network protocols, but also as a comparison of two approaches to lightweight modeling. Using our approach we were able to generate over 10,000 instances of misbehaving gadgets.

1 Introduction

The Internet is a global network of independently administered networks. Inside these *autonomous systems*, administrators are free to employ any policies and infrastructure they please. Inside an autonomous system an interior routing protocol is executed in order to update routing information and is typically a variant of the shortest path algorithm. *Interdomain routing*, by contrast, is the sharing of routing information between ASes. Interdomain routing has an inherently different nature from intradomain routing because routing decisions are based more on local policy driven by economic factors rather than on shortest path metrics. Intradomain routing is accomplished on the Internet today exclusively by the Border Gateway Protocol (BGP).

BGP was very much born out of necessity and targeted primarily to address the practical concerns of network operators. Only after the widespread adoption of BGP did researchers realize that BGPs flexibility came at the cost of guaranteed convergence. [3] This led researchers to pose the question, "What problem is BGP solving?" Other interior routing protocols correspond to a well specified problem which they are solving: the shortest paths problem. As stated above, BGP does not solve this problem; instead, BGP solves the Stable Paths Problem (SPP). [4] In a very abbreviated form, the essence of this problem is given a set of nodes, paths among nodes, preferences among paths, and a destination, does there exist an assignment of paths to nodes such that the assignment will be stable. We will provide a formal speicification of this problem in Section 2.

Further, the Stable Path Vector Protocol (SPVP) was devised as an abstract model of BGP that solves the SPP. In creating this framework for understanding BGP, researchers were able to generate *counterexamples* to desirable behavior called *gadgets*. These misbehaving gadgets revealed a serious flaw in the operation of BGP: given certain conditions, there exist instances when BGP will never converge to a stable solution as depicted in Figure 1. Despite the fact that BGP displays the undesirable behavior, it is not actually a flaw in BGP itself, but a problem inherent to the SPP.

Currently, contriving these gadgets is a thought-intensive, inefficient, and incomplete effort. Given

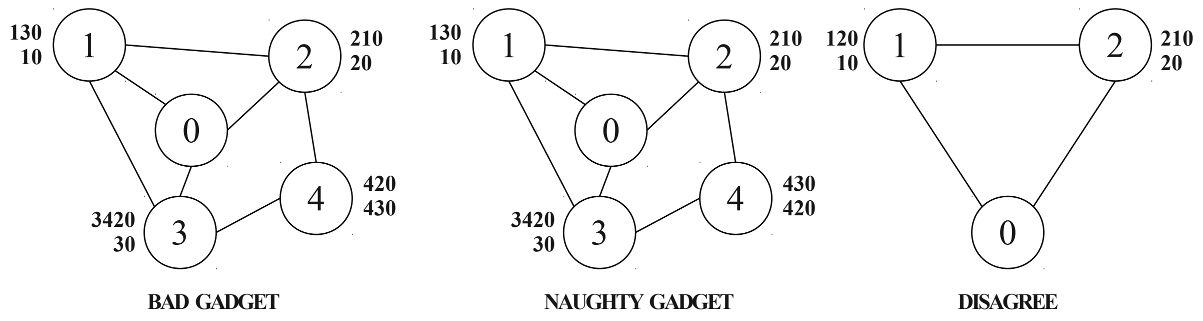


Figure 1: These seemingly simple gadgets reveal unstable instances of the Stable Paths Problem that path vector protocols are unable to solve.

the fact that the SPP and SPVP are themselves abstract models, it would seem that looking for these misbehaving gadgets is an appropriate application of model checking tools. In this work we show that by applying model enumeration tools to the SPP and model checking tools to the SPVP, we can automatically generate these gadgets far more efficiently than in current efforts. In Section 2, we cover the mathematical foundations of the SPP and relevant model checking concepts, in Section 3 we cover our implementations in Alloy and Promela/Spin, in Section 4 we present our results, and in Section 5 we describe related work and future extensions.

2 Background

2.1 The Stable Paths Problem

The SPP formalism is an abstract model that attempts to capture the essence of BGP and path vector protocols, in general, without specifically examining the implementation details of BGP itself. We begin by describing the setup of an instance of the stable paths problem. Let $G = (V, E)$ be an undirected graph of vertices[1] V and edges E where $V = \{0, 1, 2 \ldots n\}$. We define 0 as a special node called *origin*.[2] We define the set peers(u) = $\{w \mid \{u, w\} \in E\}$ which captures all the neighbors of a given node connected by a single edge.

Paths are defined as the empty path (ϵ) or a sequence of vertices. Specifically, $p = \epsilon \mid (v_k, v_{k-1}, \ldots, v_1, v_0)$ from first node v_k to last node v_0. This sequence must satisfy the following constraint: for each $i, k \geq i > 0, \{v_i, v_{i-1}\} \in E$. A path is directed from the first node (v_k) to the last nod (v_0). We define a concatenation operation on paths such that if the last node of the first path P is the same as first node in path Q then, PQ is a path. We treat concatenation of ϵ special in that $\epsilon P = P\epsilon = P$.

For each node $v \in V$, we define the set of *permitted paths* ($\mathcal{P}^v$) at node v. This set consists of a subset of all loop-free paths originating at v and terminating at the origin. For each $P \in \mathcal{P}^v, P = (vv_k \ldots v_1 0)$. The concept of a permitted path correlates to the policy aspect of BGP: there may be certain paths an AS is unwilling to take due to economic factors even though they exist. For each of these sets of permitted paths at a given node, a node-specific *ranking function* $\lambda^v : \mathcal{P}^v \rightarrow \mathbb{Z}^+$ is defined. If $P_1, P_2 \in \mathcal{P}^v$ and $\lambda^v(P_1) < \lambda^v(P_2)$ then P_2 is preffered over P_1. Let $\mathcal{P} = \bigcup_{v \in V} \mathcal{P}^v$ and $\Lambda = \{\lambda^v \mid v \in V - \{0\}\}$.

These elements form an instance S of the stable paths problem $S = (G, \mathcal{P}, \Lambda)$. We further make some assumptions to constrain our instance.

- *Origin:* $\mathcal{P}^0 = \{0\}$
- *Empty Path Permitted:* $\forall v \in V, \epsilon \in \mathcal{P}^v$
- *Empty Path is lowest:* $\forall v \in V, \lambda^v(\epsilon) = 0$ and $\forall P \neq \epsilon, \lambda^v(P) > 0$
- *Strictness:* If $P_1, P_2 \in \mathcal{P}^v, P_1 \neq P_2$ and $\lambda^v(P_1) = \lambda^v(P_2)$ then $P_1 = (v\ u)P_1', P_2 = (v\ u)P_2'$ such that u is the same next hop for both paths. Since u could only possibly have

[1]We will use the terms vertex and node interchangeably.

[2]Although it may seem counterintuitive, the terms origin and destination are both used to identify the node 0 and the terms are often used interchangeably.

one path assigned at a given time, this ensures that no two paths available to v could possibly be ranked equally. This will become more clear in a moment.

- *Simplicity:* If a given path $P \in \mathcal{P}$ then there are no repeated nodes in P. We say P is loop-free.

For a given instance S of the SPP, a function π is defined as a mapping from a node to a path in that node's set of permitted paths. This assignment is arbitrary and does not have to be the *most-preffered* path. For all $u \in V, \pi(u) : u \to \mathcal{P}^u$. Given a path assignment (π), only one path from a node's set of permitted paths is assigned to a single node.

A path assigned ($\pi(v)$) at a node $v \in$ peers(u) can be extended to a neighboring[3] node u by concatenating the edge (u, v) and the path assigned at v. We can construct a set containing all of these paths available through neighbors. The set of all possible permitted paths at u that can be formed by extending the paths assigned to the peers(u) is defined as:

$$\text{choices}(\pi, u) = \begin{cases} \{(u\ v)\ \pi(v) \mid \{u,v\} \in E\} \cap \mathcal{P}^u & u \neq 0 \\ \{(0)\} & u = 0 \end{cases}$$

Let W be some subset of permitted paths at node u. We want to identify which path is best in this subset and define the best function accordingly as:

$$\text{best}(W, u) = \begin{cases} \arg\max_{P \in W} \lambda^u(P) & W \neq \emptyset \\ \epsilon & W = \emptyset \end{cases}$$

Here $\arg\max()$ returns the the single path $P \in W$ such that the value $\lambda^u(P)$ is maximal. With these definitions we can now formulate the definitions of local and global stability. The path assignment at node u is *stable* if

$$\pi(u) = \text{best}(\text{choices}(\pi, u), u).$$

The entire path assingment is stable if this holds true at all nodes $u \in V$. Intuitively, this makes sense recalling that nodes are abstractions for autonomous systems. If we think of these nodes as ASes "choosing" a path to the origin and assume that the condition for stability is met, then no AS would want to choose a different path because each AS has the most-preferred path available given the path assignments. The stable paths problem is *solvable* if there exists a stable path assignment (π) for a given instance S.

[3]The term neighbor is used to denote a member of the set of peers(u) for a given node u.

2.2 The Stable Path Vector Protocol (SPVP)

While the SPP is a static description of the problem we are trying to solve, the SPVP is an abstraction for BGP that attempts to solve the SPP. The significance of the SPVP is that it always diverges when an instance of SPP has no solution. The SPVP is a dynamic process that runs concurrently on all nodes to solve a given instance of the SPP. To achieve this dynamism, we include in our model notions of communication between the nodes and introduce two new concepts: rib(u) and rib-in($u \Leftarrow v$). When a node u adopts a new path $P \in \mathcal{P}^u$, it simply sends that exact path in a message to its neighbors in peers(u). At a given node u, the node stores its current path to the origin in a structure called rib(u). For each neighbor $w \in$ peers(u), u stores the most currently received path from w in a stucture called rib-in($u \Leftarrow w$). We now redefine our notions of best and choices accordingly:

$$\text{choices}(u) = \{(uw)P \in \mathcal{P}^u \mid P = \text{rib-in}(u \Leftarrow w)\}$$

$$\text{best}(u) = \text{best}(\text{choices}(u), u).$$

Using these definitions we can very easily encode an algorithm that implements the SPVP as shown in Figure 2.

2.3 Lightweight Modeling

Lightweight modeling is a powerful tool that can be used to verify the logic of a piece of software or distributed system. Even though we can use these tools to identify logical inconsistencies after the fact, the best time to employ these tools is during the design phase of system development. Employing lightweight modeling early on in the development of a complicated system could reveal logical inconsistencies that lead to undersirable or unexpected system behavior. A lighweight model is an abstract model of a system's core components irrespective of implementation details combined with an automated analysis tool. This focus only on core components

```
**process** spvp($u$)
**begin**
    **receive** $P$ **from** $w \rightarrow$
    **begin**
        rib-in($u \Leftarrow w$) := $P$
        **if** rib($u$) $\neq$ best($u$) **then**
        **begin**
            rib($u$) := best($u$)
            **for each** $v \in$ peers($u$) **do**
            **begin**
                **send** rib($u$) **to** $v$
            **end**
        **end**
    **end**
**end**
```

Figure 2: The above pseudocode describes the procedure that each node runs dynamically to solve the stable paths problem

allows us to quickly construct a compact model that still captures the essential functionality we wish to verify. The use of analysis tools in lieu of more expressive theorem provers also makes the approach lightweight. Theorem provers require extensive domain specific knowledge both of the system being modeled as well as of logical formalism. Analysis tools allow us to get similar results in a more push-button manner and generate obscure test cases that humans would never have come up with. We employ two different variants of lightweight modeling: model checking with Promela/Spin and model enumeration with Alloy. Each approach seemed a more appropriate fit to the two problems, Alloy for modeling the SPP and Promela/Spin for SPVP. We will discuss these tools in more detail and their tradeoffs as applied to these problems in the following sections.

3 Implementation

We devised two different approaches of generating misbehaving gadgets. From a high-level, we chose to use Alloy to construct the definitions of stability in the SPP instances and enumerate instances that violate this definition. We chose to use Promela/Spin to model the dynamic and distributed nature of the SPVP protocol. We simulate execution of the SPVP on generated topologies to identify instances in which the SPVP diverges. These instances correspond directly to unstable SPP instances. The following subsections describe these two approaches in detail.

3.1 SPP in Alloy

3.1.1 Alloy Overview

Alloy is a declarative programming language. Declarative programming involves describing *what* the solution is as opposed to *how* to implement the solution as in imperative programming languages like C or Java. The advantage of using Alloy is that we only model key properties that are important to the problem. Alloy itself has four key properties that affect its employment.

Relation-based. In order to specify the problem space in Alloy, one defines a set of atoms and relations. Atoms are similar to objects, structs, and classes that are familiar to the imperative programmer. One can build more complex structures and specify how these atoms interact by the use of relations. Relations make Alloy easy to understand while still keeping the problem solvable. Relations also give the programmer flexibility to create a wide range of models.

No specialized logic. The lack of specialized logic for concurrency or time ensures that the models should be fairly easy to understand. It is still possible to implement these concepts through user defined atoms and relations. In Alloy, one must be very precise when specifying a model. This level of precision means that including "black-box" specialized logics may easily lead to confusion and inconsistency. Thus, it seems reasonable that Alloy does not have such specialized logics that programmers might use without properly understanding the intricate details first.

Scope and counterexamples. In Alloy, the programmer specifies a particular scope of a program and the solver will then try to find counterexamples within this scope. For instance, a hypothetical scope of an arbitrary problem might be *for exactly three nodes*. If Alloy has found no counterexamples and your encoding of the model is correct, then there are no counterexamples within that scope. Programmers do not have to spend time contriving obscure test cases because Alloy exhaustively tries all cases within a given scope.

SAT solver. Alloy uses logical reduction of these more complicated models as an input to its SAT solver backend. SAT solvers have become extremely powerful over the last decade solving millions of variables and millions of constraints in only a few seconds. As SAT solvers and computational power continue to improve substantially, Alloy will be able to solve larger scopes and more complicated models.

3.1.2 Using Alloy

In order to effectively use Alloy, one must first have a crystal clear understanding of the problem. Without this, it will be extremely difficult to setup the necessary constraints and invariants in the model that will solve the correct problem. The first step is to create a set of atoms and relations. These are the fundamental objects and relationships that will play an important role in specifying the behavior of our model.

Typically, a problem space will include uninteresting or invalid scenarios. For example, our problem space of a given graph $G = (V, E)$, we do not care about graphs that are not fully connected. In order to ignore this case, we will constrain our model to include only the exact problem space in which we are interested. Correctly specifying these constraints is an iterative process: specify too imprecisely and you end up with irrelevant counterexamples, specify too strictly and you may constrain away your entire problem yielding trivially correct results.

Once the problem space is effectively constrained to the instances in which we are interested, we create invariants that describe the desired behavior of our model. Correctly specifying these invariants, along with the constraints, is the foremost challenge in effectively wielding the power of Alloy. However, once done correctly, all that is left is to create assertions that our invariants are not violated and Alloy will enumerate any counterexamples, if they exist.

3.1.3 Solution Design

Creating an Instance of the SPP. With a clear understanding of how to reason about models in Alloy, we can begin to tackle the SPP. We first specify an instance S of the SPP, which we recall consists of $S = (G, \mathcal{P}, \Lambda)$. An instance of the graph is fairly straightforward to set up by specifying nodes as atoms and peers as relations on nodes. Notice that the current problem space that we have just created would consider every possible number of nodes in the peer relation for a given node, including itself. However, we wish to constrain our problem to proper instances of the SPP. The following Alloy code excerpt gives a good example of how we must be specific in constraining our problem.

```
fact setupGraph {
 /* Neighbor cannot be self */
 no v: Vertex | v in v.neighbors

 /* Bi-directional graph */
 all v1, v2: Vertex | v1 in v2.neighbors =>
    v2 in v1.neighbors

 /* Must be connected graph */
 all v1, v2:Vertex | v1 in v2.*neighbors
}
```

Next, we establish the set of permitted paths at each node $u(\mathcal{P}^u)$. We already have nodes so we will insert an additional relation specifying that each node has some permitted paths. Our current model is limited to two permitted paths which should suffice in generating the most well-known gadgets. With another relation, we will constrain what constitutes a permitted path: a permitted path for a given node u should begin at node u and end at the destination node 0. In addition, for each pair of nodes (u, v) in a path P, each v must be a neighbor to node u. This corresponds exactly to our definitions of paths in section 2.1.

It is very important to notice that we only needed to specify *what* a permitted path is, not an algorithm of how to create a list of all permitted paths for each node. Alloy will in fact enumerate all possible instances of two permitted paths per node when it tries to generate counterexamples, which is exactly what we want.

Next, there must be a ranking function. Since Alloy is already considering all possible instances of two permitted paths per node, we can introduce ranking implicitly based on which permitted path appears first among the two permitted paths. Now, we have successfully specified a model to generate all instances of the SPP for a given scope.

Isolating Misbehaving Gadgets. In order to proceed from creating the instance to the enumeration of misbehaving gadgets, we must revisit the definitions of the path assignment function π and the definitions

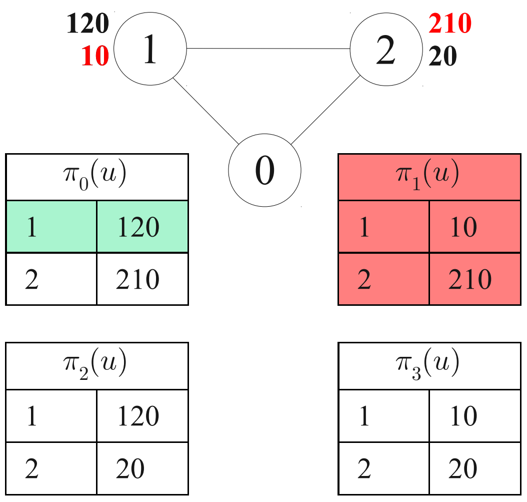


Figure 3: This graphic shows the relationship of the `piEntry` and `piTable`s of a particular instance. In this case, table $\pi_1(u)$ captures the path assignment depicted in the graph.

of stability. We recall that stability is defined as the existence of a path assignment such that each node has been assigned the best path among its choices: $\pi(u) = \text{best}(\text{choices}(\pi, u), u)$. Thus, we will generate all possible path assignments for a particular instance and then specify an invariant to check for the existence of such a path assignment.

To model the set of path assignments, we first introduce a new atom and relation that we call a `piEntry`, highlighted green in figure 3, to represent the mapping of a single node to one of its permitted paths. We also introduce a `piTable` relation, highlighted red in figure 3, to be a particular permutation of `piEntrys` for all nodes. If we specify that enough unique `piTables` are in our scope, we necessarily generate all possible path assignments for each specific instance of the SPP. In figure 3, four distinct `piTables` enumerate all possible path assignments.

The next step is to specify the appropriate invariant. Global stability is defined as the existence of a path assignment π such that for all nodes $u \in V$, the local stability property holds. Our invariant will specify exactly that. In the case of two permitted paths, if a node's path assignment is its least preferred path, then its most preferred path must not be an element of $\text{choices}(\pi, u)$. With our invariant defined, we now just specify a scope (number of nodes, path length, and number of possible path assignments) and Alloy will quickly generate counterexamples.

3.1.4 Challenges

The main challenge that we encountered was a combination of an incomplete understanding of the problem and learning to reason in a declarative manner. We reached a point in our model where we could correctly generate an instance of SPP. With this instance, we proceeded to try and specify an invariant that would identify gadgets without correctly specifying the notion of path assignments. We contiued to try and coerce the permitted path relations to generate counterexamples, but this is precisely where we went wrong. As a result of not correctly specifying the problem, we were trying to imperatively program something in a declarative model. We tried to describe *how* a node would behave, which is exactly what an imperative program would do. We were not thinking in terms of *what* the solution is. Once we realized this, we introduced the notion of `piEntrys` and `piTables` that let us identify gadgets that have no stable solution. Learning to think declaratively and understanding the degree of specificity we must employ in describing our models proved to be the most difficult challenges for us.

3.1.5 Limitations

The number of permitted paths in our model limits its operation most significantly. Currently, each node has exactly two permitted paths. This limitation originated in our early stages of development when we were having trouble correctly specifying the invariant. Reducing the complexity of the model allowed us to correctly formulate an invariant, but at the cost of generality. This limitation reduces the number of instances of the SPP that we can analyze. Generalizing the number of permitted paths is the highest priority improvement to the Alloy model.

Alloy currently generates myriad isomorphic counterexamples, especially when as we start to increase the maximum lengths of paths and number of nodes. While our initial analysis of the counterexamples suggest that they are indeed valid gadgets, reducing or eliminating these repetitive instances will significantly reduce the analytical overhead in post-processing the generated counterexamples. Although we have not implemented a solution to this problem, we are confident that additional constraints will eliminate isomorphism in future development.

3.2 SPVP in Promela/Spin

3.2.1 Promela/Spin Overview

Promela is a protocol modeling language originally developed in the early 1980's specifically for software verificaion. As it was intended to model distributed and concurrent processes, Promela includes several "baked-in" features that facilitate modeling a distributed protocol like the SPVP. Promela includes support for concurrency through atomic operations and messaging with FIFO communications channels. Promela is imperative in nature but supports only the most basic control-flow and arithmetic operations; more complicated algorithms are very difficult to construct in Promela.

Spin is a model checker that uses linear-time temporal logic to verify properties of models described in Promela. Spin can act as a simulator or an exhaustive verifier that can check both safety and liveness properties of systems. One of Spin's greatest strengths is its ability to non-deterministically explore the statespace of a given model and identify cycles in that statespace. Figure 4 shows a simple statespace diagram that illustrates this ability.

3.2.2 Design

Unlike Alloy, Promela is imperative in nature so there was no need to relearn how to think about the problem and our approach using it was straightforward. We build an instance of the SPP, run a simulation of the SPVP on the instance, and look for cycles in the statespace. As described above, Spin has cycle detection built-in so all we had to do was find the correct assertion.

Recall that the SPVP always diverges when run on an unstable instance of SPP. Revisiting the algorithm from figure 2, we see then that in an unstable instance some node u will continue to change its rib(u). If we think of the collection of rib(u) for all nodes $u \in V$ as the network state and recognize that since there are a finite number of nodes and permitted paths, this divergence necessarily implies a cycle

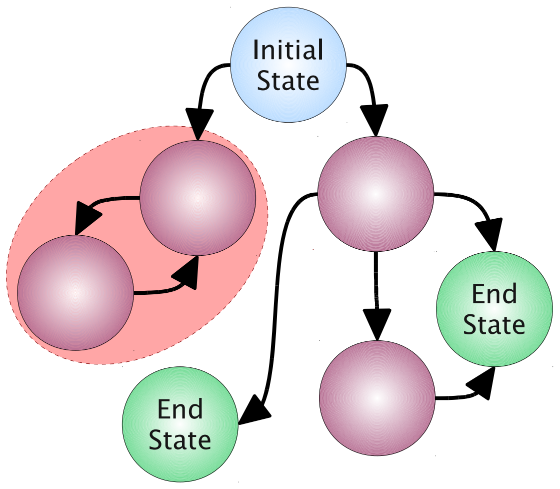


Figure 4: Spin explores statespace non-deterministically and finds inescapable cycles (circled on the left).

of state transitions. We use the Spin `accept` construct which is a formalization of the Büchi acceptance conditions [6]. Specifically, we detect the cycle where a node repeatedly changes its current rib. The following Promela code excerpt depicts the implementation of the `if-else` block in the algorithm from figure 2 and the `accept` to catch the cycle.

```
if (chosen_path != 0) &&
    (set_path != chosen_path) ->
  accept:
    set_path = chosen_path
```

This cycle detection combined with the non-deterministic exploration of state has made our Promela/Alloy approach very effective at identifying instances of the SPP on which SPVP diverges. Unfortunately, this effectiveness does come at a cost: gadget overload.

3.2.3 Challenges

Isomorphism. The single most significant challenge in modeling the SPVP in Promela/Spin is avoiding isomorphism. Isomorphism increases the time it takes the model checker to run because it must explore more instances than are actually interesting to us. Many of these instances are simple renumberings of already identified instances. Isomorphisms also increase the time it takes humans to analyze an absurd number of gadgets only to realize they are yet another isomoprhic instance (YAII). We implemented a simple heuristic isomorphic reduction that does not eliminate isomorphism altogether, but reduces the number of isomorphic instances generated by over 66%. Our heuristic constrains the nodes such that for all nodes $u, v : u > v \rightarrow \text{edges}(u) \geq \text{edges}(v)$.

Atomicity. The second most significant challenge posed in the Promela/Spin model was correctly specificying atomicity. Because Promela/Spin was intended to model concurrent and distributed systems, Spin will try to interleave operations as much as possible to explore all possible states. However, this can lead to statespace explosion for our model and consequently, slow the run-time substantially. Correctly specifying atomicity also resulted in a separate 80% reduction in runtime.

Statespace Explosion. Statespace explosion is partially the result of the preceding two challenges. Because we still need to apply more optimizations to improve our implementation, we still cannot perform an exhaustive search of the statespace without running into resource limitations on commodity hardware. Consequently, we are often forced to run only partial statespace searches.

3.2.4 Limitations

While our Promela/Spin model is a general solution to simulating a topology of n nodes, the realities of exploring this statespace on commodity hardware have confined us to very small instances of the SPP. Further, we do not have a good way to model random activation sequences. In our model we only activate nodes in strict round-robin fashion. This strategy has provided us with numerous examples, but more work is required to determine whether or not we are ignoring potential gadgets by not exploring this statespace. In additon to not exploring random activation sequences, our model cannot identify bi-stable instances of the SPP. Gadgets such as DISAGREE have two stable states between which oscillation can occur. Determining both stable states would require exploring different activation sequences. Our model explores only one (described above) activation sequence and reports stability or instability for that sequence only.

4 Results

4.1 Alloy

4.1.1 Completeness

Since Alloy exhaustively enumerates all possible combinations within a given constrained problem and scope, we believe that our model does generate all statically unsolvable instances of the SPP. We have been able to verify the existence of known gadgets such as BAD GADGET and DISAGREE as figure 5 depicts. In future work, we would also like to compare our results with those from Promela. If we can show that the results are consistent, we can be reasonably certain we have identified all gadgets in our scope.

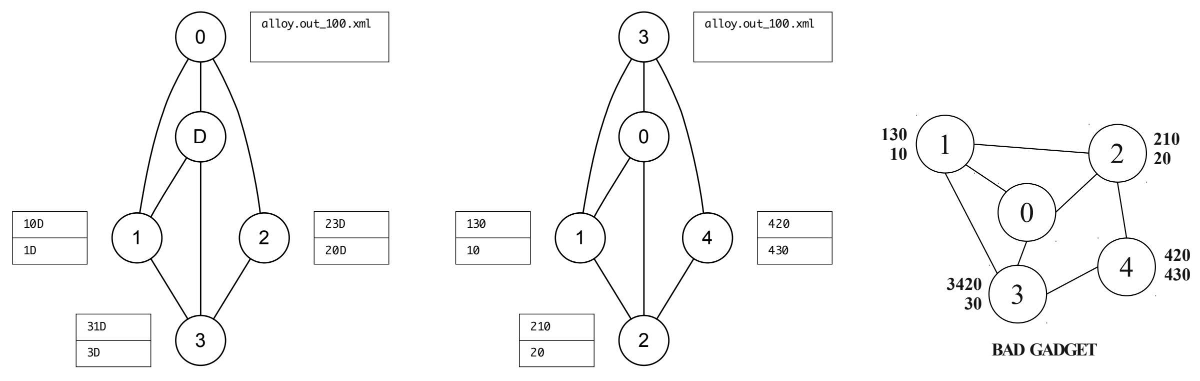
One extremely strong feature of Alloy is that the model lends itself well to modifications. Recall that DISAGREE is actally a bi-stable gadget with more than one unique globally stable solution. The fact that our model was able to identify DISAGREE was because we additionally constrained that there exists only one globally stable path assignment. This cheap extensibility should permit us to extend our model to incorporate additional constraints and models of BGP.

4.1.2 Performance

Our model performs moderately well. The following table summarizes its performance.

Problem Size	*Running Time*(sec)
$n = 4, p = 2$	4
$n = 5, p = 2$	17
$n = 6, p = 2$	515

Here, n indicates the number of nodes (including the origin) in the topology and p indicates the number of permitted paths per node. These results seem to imply that the model can definitely handle a larger search space in a reasonable amount of time. Although counterexample enumeration is not supported by Alloy, we can take advantage of Alloy's flexibility by further constraining the problem to exclude these known results. This approach introduces its own set of challenges (correctly specifying the gadgets already seen), but it should be a tractable approach.



(a) Generated (b) Isomorphic Reduction (c) Original

Figure 5: These gadgets demonstrate our model's verification of existing known gadgets. (a) shows the original generated gadget, (b) shows the isomorphic reduction of the generated gadget and is visibly identical to (c) the original BAD GADGET.

4.2 Promela/Spin

4.2.1 Completeness

Unfortunately, due to the limitations described in the section 3.2.4, we know that we are not getting complete result sets for a given problem size. We have been able to generate certain well-known gadgets. For instance, we have been able to generate DISPUTE WHEEL gadget in the $n = 4, p = 3$ case. However, we were unable to generate BAD GADGET from the $n = 5, p = 2$ case. Ostensibly, we believe this is due the inexhaustive search constrained by statespace explosion. Continued work should seek to ensure that we can get complete results in a given instance class even if those result in intractable run-times.

4.2.2 Performance

Even with the optimizations put in place thus far, our Promela/Spin model has not scaled well. The following table summarizes our results.

Problem Size	*Running Time*(sec)
$n = 4, p = 2$	31
$n = 4, p = 3$	1320
$n = 5, p = 2$	66800

Clearly, our model has reached the limit of its computational ability at this stage of development. We are hopeful that continued work in reducing isomorphism will allow us to increase the problem size and reduce runtime. On the other hand, the most interesting gadgets come in small instances so we do not need to improve the model much to reach the upper bound of our interest.

4.3 The Gadgets

In the Promela/Spin model, we were able to generate over 10,000 instances of isomorphic gadgets in the $n = 4, p = 3$ case. Of course, not all of these are interesting, but the sheer number demonstrates the power of our models, even in their nascent stages of development. The following table summarizes the purportedly unique gadgets we have generated for a given problem size using Promela/Spin.

Problem Size	*Number of Gadgets*
$n = 4, p = 2$	10
$n = 4, p = 3$	3236
$n = 5, p = 2$	70

While there may exist some isomorphism still, we have not yet had a chance to fully evaluate the results generated. Further analyis of the gadgets will be the immediate next phase of this project.

Although we have generated hundreds of Alloy gadgets as well, we have not had a concrete way of enumerating the instances thus far and we know that some counterexamples are duplicated. Recent model improvements should allow us to better constrain our counterexamples such that they are uniquely generated.

5 Related Work

5.1 Lightweight Modeling

Zave's work on lightweight verification of the Chord distributed hash table demonstrated the utility of applying lightweight modeling to network protocols. As in our work, Zave implemented a lightweight model in Alloy for an existing protocol. Zave examines both the full Chord protocol and a pure-join only model. [8] The models revealed that none of the six "provably correct" properties of Chord held. This work demonstrated the value of modeling existing network protocols for verification, but more importantly, the value in using lightweight models when designing network protocols.

5.2 SPP and Interdomain Routing

Griffin promulgated the SPP and SPVP in several independent publications beginning in 1999. [3] Griffin showed that the SPVP always diverges when run on unstable instances of the SPP. [4] Griffin also devised a construct called a dispute wheel which is another abstract representation of conflicting routing policies. Griffin shows that if no dispute wheel can be constructed, then there exists a unique stable solution to the SPP for a given instance. However, Griffin also showed that SPVP can diverge on instances where a unique stable solution exists. In these cases, if no dispute wheel exists then the SPVP will not diverge.

Gao and Rexford built upon Griffin's work by demonstrating that the imposition of certain conditions on the SPP would ensure that SPVP always converges. [2] These conditions are grounded in common business practices of internet service providers (ISP) and group the nodes into customers, providers and peers. These different groups have constrained behavior from the standard SPP model in which nodes could have arbitrary combinations of permitted paths and ranking functions.

Sami and Schapira built upon Gao and Rexford's work by showing that any topologies that obey the "Gao-Rexford Conditions" will not only be stable but will converge, worst case, polynomially in the length of the longest directed customer-provider chain. [5] Sami and Schapira also showed that bistable instances of the SPP are also subject to persistent routing oscillations.

Cittadini built on upon Griffin, Gao, and Rexford's work with the SPP and determined that the precise model of activation can lead to different outcomes. [1] Cittadini identified gadgets that have different convergence properties based on the type of activation used.

5.3 Future Work

5.3.1 Implementation Improvements

Our models certainly need additional work. The three most important implementation improvements would be to eliminate isomorphism in both models, generalize the Alloy model for $p > 2$ cases, and enable random activations for the Promela/Spin model. Of course, continued analysis of the gadgets we have generated will continue concurrently.

5.3.2 Additional Applications to BGP/SPP

Our approach should extend to other models of interdomain routing and assist in validating their claims about convergence under the new models. Imposing the Gao-Rexford conditions should be a natural extension of the constraints we are already using in the Alloy model. New models of BGP such as Neighbor Specific BGP (NS-BGP) have been proposed that impose fewer constraints than the Gao-Rexford but still guarantee safety. [7] Applying our models to this problem also seems like a natural extension of this work.

5.3.3 Existing Protocols & Protocol Design

Interdomain routing is not the only aspect of networking that would benefit from lightweight modeling. Literally hundreds of network protocols exist that have little or no formal verification of their properties. Any one of them could benefit from a thorough lightweight analysis. More importantly though, network protocol designers must learn to integrate lightweight modeling into the design process. Waiting to conduct this modeling until after protocols are "in the wild" does not make any sense at all. We fully intend to practice what we preach and are currently creating a model for Scaffold[4], a nascent data center

[4] `http://sns.cs.princeton.edu/projects/scaffold/`

networking protocol under development at Princeton.

6 Conclusions

We have demonstrated two successful lightweight modeling approaches to the stable paths problem and the stable path vector protocol. We created a static model of and conducted analysis on the SPP with Alloy. This approach was able to indentify unsolvable instances of the SPP that have no stable solution. We also created a dynamic model of the SPVP in Promela and conducted model checking with Spin. This approach was able to indentify instances of the SPP that lead to divergent behavior of the SPVP. Both approaches yielded numerous results, but the implementations of both approaches suffer from different limitations at this stage of development. We were able to generate over 10,000 non-unique misbehaving gadgets and over 3,000 unique gadgets up to a problem size of $n = 5$ nodes. We look forward to continuing to improve our current models to increase generality in the application to the SPP/SPVP, but also to extend our models to include new models of BGP and interdomain routing. This application of lightweight modeling should demonstrate that the technique is an indispensible tool in verifying network protocol logic and identifying counterexamples to desired behavior.

References

[1] L. Cittadini, G. D. Battista, and M. Rimondini. How stable is stable in interdomain routing: Efficiently detectable oscillation-free configurations. Tech. Report RT-DIA-132-2008, Dept. of Computer Science and Automation, Roma Tre University, 2008.

[2] L. Gao and J. Rexford. Stable internet routing without global coordination. *IEEE/ACM Trans. Netw.*, 9(6): 681–692, 2001. ISSN 1063-6692. doi: http://dx.doi.org/10.1109/90.974523.

[3] T. G. Griffin and G. Wilfong. An analysis of bgp convergence properties. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 277–288, New York, NY, USA, 1999. ACM. ISBN 1-58113-135-6. doi: http://doi.acm.org/10.1145/316188.316231.

[4] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, 2002. ISSN 1063-6692.

[5] R. Sami, M. Schapira, and A. Zohar. Searching for stability in interdomain routing. In *Proceedings of INFOCOM*, pages 549–557, 2009.

[6] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[7] Y. Wang, M. Schapira, and J. Rexford. Neighbor-specific bgp: More flexible routing policies while improving global stability. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 217–228, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-511-6. doi: http://doi.acm.org/10.1145/1555349.1555375.

[8] P. Zave. Lightweight verification of network protocols: The case of chord, 2009. URL `http://www2.research.att.com/~pamela/chord.pdf`.