On the Automated Verification of BGP Convergence

HAOYUN QIN, University of Pennsylvania, Philadelphia, PA, United States

GERALD WHITTERS, University of Pennsylvania, Philadelphia, PA, United States

BOON THAU LOO, University of Pennsylvania, Philadelphia, PA, United States

CAROLYN L TALCOTT, SRI International, Menlo Park, CA, United States

PPDP '25: Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming
September 10 - 11, 2025
Rende, Italy

On the Automated Verification of BGP Convergence

Haoyun Qin*
University of Pennsylvania
Philadelphia, PA, USA
qhy@seas.upenn.edu

Gerald Whitters*
University of Pennsylvania
Philadelphia, PA, USA
whitters@seas.upenn.edu

Boon Loo
University of Pennsylvania
Philadelphia, PA, USA
boonloo@seas.upenn.edu

Carolyn Talcott
SRI International
Menlo Park, CA, USA
carolyn.talcott@gmail.com

Abstract

The Border Gateway Protocol (BGP) is employed by autonomous systems (ASes), such as network operators or ISPs, to build routing tables. However, depending on the routing policies implemented by these ASes, BGP may fail to converge, potentially rendering the network inoperative. This paper introduces a workflow that leverages SMT solvers and rewriting tools to automate the verification of BGP convergence within a given AS network. We encode the convergence conditions defined by the Metarouting theoretical framework as an SMT problem. While SMT solvers can automatically determine whether BGP will converge, they do not generate counterexample traces in cases of divergence. To overcome this shortcoming, we propose a sound divergence criterion. We also construct an executable model for verifying BGP convergence, which can be automated using the Maude rewriting tool to produce witness traces in divergent scenarios. The effectiveness of our approach is demonstrated through a series of experiments.

CCS Concepts

• **Theory of computation → Automated reasoning**; **Logic and verification**; **Equational logic and rewriting**; **Verification by model checking**; • **Networks → Routing protocols**; **Formal specifications**; **Protocol testing and verification**; • **Mathematics of computing → Solvers**.

Keywords

BGP, Maude, SMT, Metarouting

ACM Reference Format:
Haoyun Qin, Gerald Whitters, Boon Loo, and Carolyn Talcott. 2025. On the Automated Verification of BGP Convergence. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming (PPDP '25), September 10–11, 2025, Rende, Italy.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3756907.3756920

*Both authors contributed equally to this research.

1 Introduction

Due to the scale and concurrent nature of modern networks, identifying configuration bugs is a significant challenge. When such issues go undetected, they can lead to severe and sometimes catastrophic network failures. The widely used Border Gateway Protocol (BGP) enables autonomous systems (ASes) to build routing tables according to their packet forwarding policies, also known as preference relations. Prior work has shown that poorly configured or conflicting policies can prevent BGP from converging, leading to unstable routing tables and rendering the network non-operational [5, 8, 9]. We refer to the liveness property concerning whether routing tables eventually stabilize as the *BGP convergence problem.*

Reasoning about BGP convergence has long been a difficult problem due to the protocol's asynchronous execution, path-dependent decisions, and policy-driven behavior. Existing formal verification methods for BGP convergence generally fall into two categories.

(1) *Correct-by-design approaches*, such as Metarouting [7, 12], allow network engineers to ensure BGP convergence by proving that the preference relations of ASes satisfy certain monotonicity conditions [7, 12]. However, a major limitation of these methods is their lack of automation, which prevents them from efficiently detecting policy conflicts.
(2) *Model-checking approaches* [15, 16], in contrast, are designed to identify policy errors automatically. However, either the techniques proposed so far are not sound, meaning they may produce false positives, i.e., flagging problems that do not actually exist, or they are not scalable [17], not able to determine divergence even for networks with less than 5 nodes.

This paper revisits the BGP convergence problem with the goal of developing sound, scalable, and automated verification methods. The central innovation lies in combining the strengths of the two existing approaches. First, we automate correct-by-design techniques to verify whether a given network instance is guaranteed to converge. We show how SMT solvers can be employed to perform this verification automatically.

When convergence cannot be guaranteed, we turn to model-checking methods to uncover concrete counterexamples that demonstrate divergence. These counterexamples provide valuable diagnostic information, enabling network engineers to identify and correct misconfigurations.

One of the main challenges in using model checking to find a witness for BGP divergence is the potentially large size of the network. Our key insight is to leverage the SMT solver's ability to generate unsat cores, which are minimal sets of constraints responsible for unsatisfiability in the convergence check. In the context of BGP, the unsat core highlights the specific network links and policy interactions that contribute to divergence. These critical links are then used to focus the model checker's exploration, significantly reducing the overall search space and improving efficiency.

To achieve this goal, this paper makes the following key contributions:

- **SMT Encoding of Metarouting:** We present an encoding of the Metarouting criteria for BGP convergence as an SMT problem, enabling the use of SMT solvers to perform convergence checks automatically. This approach overcomes a central limitation of Metarouting, which is its lack of automation. Our approach allows engineers to systematically verify convergence conditions.
- **Sound Criterion for Divergence:** Verifying liveness properties such as BGP convergence is inherently difficult. A common approach is to reduce liveness verification to a safety check [2]. We introduce a sound safety-based criterion for detecting BGP divergence. This criterion relates to the sequence of messages exchanged between ASes and identifies conditions under which BGP fails to stabilize. It leverages links identified in the SMT solver's unsat core, i.e., candidate links that are involved in non-terminating BGP computations, as input. To the best of our knowledge, this is the first sound divergence criterion formulated for the BGP convergence problem.
- **Executable Model for BGP Verification**: We demonstrate that our divergence criterion can be fully automated. By formalizing both the criterion and BGP's operational semantics in rewriting logic, we enable executable verification using the Maude rewriting tool [3]. Furthermore, we show how the SMT-derived unsat core can be used to guide the search process, significantly reducing the state space and enabling the generation of meaningful counterexamples.

Our methods have been validated on networks combining several *networks gadgets*, i.e., network patterns, which appear in the literature. The experiments demonstrate that the methods can scale to realistic size networks containing 500 nodes and 1200 edges.

Section 2 describes by example the BGP protocol, and the BGP convergence problem including Metarouting. Section 3 provides a general overview of the verification flow proposed in this paper. Section 4 describes how SMT solvers can be used for BGP verification, while Section 5 specifies the sound criterion used for model-checking network gadgets. The proposed criterion is encoded in Maude enabling optimization as described in Section 6. Section 7 validates the proposed verification flow with several experimental results. Finally, sections 8 and 9 conclude by discussing related and future work.

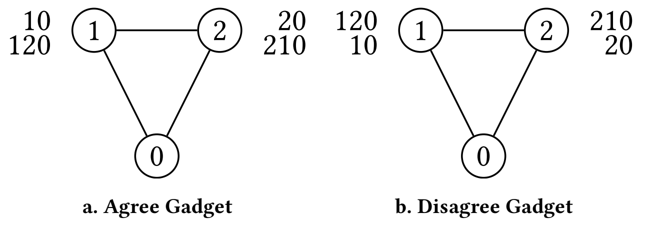


Figure 1: Example gadgets to showcase convergence and divergence in BGP

2 Border Gateway Protocol

2.1 BGP By Example

The Border Gateway Protocol (BGP) is a standardized exterior gateway protocol used for exchanging routing and reachability information between autonomous systems (ASes) on the Internet [10]. BGP is categorized as a path-vector routing protocol [6], and it determines routing decisions based on path attributes, network policies, or rule sets configured by network administrators.

We illustrate BGP using the network gadgets shown in Figure 1, while the exact algorithm is described in detail in [10]. These gadgets, called *Agree Gadget* and *Disagree Gadget*, respectively demonstrate cases where BGP converges and where it may fail to do so.

Each node of a gadget is an AS. Each of these nodes have their own policy for determining a preference on the paths for routing. This policy is represented as an ordered list of paths in the figure (written top to bottom next to the node that contains the policy). A path p that appears before a path q in such a list is preferred over q in the corresponding policy. The distinction between Agree Gadget and Disagree Gadget is that the former prefers to route paths directly to 0, while the latter prefers to route paths through a neighbor before reaching 0. A key objective of BGP is to determine the best routing paths with respect to the ASes's preference relations.

BGP is run asynchronously among all nodes in a gadget to compute a satisfying path for each node. Each node receives paths from the nodes it is directly connected to that allows it to learn valid routes to a destination. Nodes will process these paths, and for each path will determine a selected best path from all of its known valid routes. The selected path is stored in a Routing Information Base (**RIB**) for each node. When the node has no current valid path to a destination we denote this with a special path $\perp$.

For every neighbor a node has, it stores the most recent path received from that neighbor in what is known as Routing Information Base - Inbound (**RIB-IN**). Each node also keeps a FIFO queue for each of its neighbors, these queues contain the paths received from that neighbor but haven't been processed by the node yet. When a node processes a path to a destination, it checks whether its policy prefers a valid path received from a neighbor over the current path stored in its **RIB** for that destination. If so, the node updates its **RIB** with the new path and broadcasts this updated path to its neighbors.

In some cases, a node may later receive a different path from the same neighbor that had previously sent the selected path **RIB**. In this case, the path in the **RIB** is no longer a valid route. A new

Step		**Node 1**		**Node 2**	
		Node 0	**Node 2**	**Node 0**	**Node 1**
0	**RIB-IN**	$\perp$	$\perp$	$\perp$	$\perp$
	QUEUE	[(1, 0)]	[]	[(2, 0)]	[]
1	**RIB-IN**	$\boxed{(1,0)}$	$\perp$	$\perp$	$\perp$
	QUEUE	[]	[]	[(2, 0)]	[(2, 1, 0)]
2	**RIB-IN**	$\boxed{(1,0)}$	$\perp$	$\boxed{(2,0)}$	$\perp$
	QUEUE	[]	[(1, 2, 0)]	[]	[(2, 1, 0)]

Table 1: Initial BGP Steps For Agree Gadget and Disagree Gadget

one must be selected from its known paths stored in the **RIB-IN**. If there are no valid paths that can be selected from a node's **RIB-IN**, the special path $\perp$ is announced to its neighbors instead.

To showcase scenarios when BGP can find a solution and converge, as well as, when BGP might oscillate forever and diverge resulting in no solution, we explore an execution of BGP on Agree Gadget and Disagree Gadget. For both gadgets, assume that BGP is used to determine the best paths to node 0. Due to the two gadgets only differing in path preferences, the first few steps taken when executing BGP will essentially be the same. Paths will be written in the form $(n_1, n_2, ..., n_k)$ for nodes $n_1, n_2, ..., n_k$ and $k \in \mathbb{N}$. Queues will be written in the form $[p_1, p_2, \ldots, p_k]$ for paths $p_1, p_2, \ldots, p_l$ and $l \in \mathbb{N}$.

We show those steps here running it on both gadgets in parallel and will discuss the gadgets individually when their execution would begin to differ. The results of each step of BGP is shown in the tables 1, 2, and 3. We show for nodes 1 and 2 the currently stored values of the **RIB-IN** and queue for each of their neighbors. The **RIB** for each node is designated by the **RIB-IN** value that is $\boxed{\text{boxed}}$ when applicable. Initially, in step 0, node 0 will have already advertised to its neighboring nodes 1 and 2 the unit path (0). The corresponding paths, $(1,0)$ and $(2,0)$ are stored at the queues for node 1 and node 2 respectively. Next, both of these nodes can process the path. In step 1, node 1 will process path $(1,0)$ and then in step 2, node 2 will process path $(2,0)$. At both these steps, the **RIB-IN** entries at the node are $\perp$ before any processing is done. Thus, when each node processes its path in the queue, it will store this in its **RIB-IN** and then select that path as its **RIB** and announce this path to its neighbor. From step 3 and on we consider two different cases for Agree Gadget and Disagree Gadget respectively.

Agree Gadget consistently converges to a solution, regardless of the execution order in BGP. A straightforward example of its convergence can be demonstrated by continuing from the previously described steps. Let both nodes 1 and 2 process the single path in each of their non-empty queues in step 3 and in step 4 in sequence. Each queue contains a path that routes to a neighbor before finally reaching node 0. However, as discussed earlier, both nodes have a policy to prefer routes directly to 0. Since both nodes already have such a path selected as their **RIB**, when they process the path from their queues, they will store the path in the **RIB-IN**, but no new **RIB** will be selected, resulting in no new paths being announced from either node. All the queues are now exhausted, so BGP has

Step		**Node 1**		**Node 2**	
		Node 0	**Node 2**	**Node 0**	**Node 1**
3	**RIB-IN**	$\boxed{(1,0)}$	$\perp$	$\boxed{(2,0)}$	$\perp$
	QUEUE	[]	[]	[]	[(2, 1, 0)]
4	**RIB-IN**	$\boxed{(1,0)}$	$\perp$	$\boxed{(2,0)}$	$\perp$
	QUEUE	[]	[]	[]	[]

Table 2: BGP Steps for Agree Gadget

Step		**Node 1**		**Node 2**	
		Node 0	**Node 2**	**Node 0**	**Node 1**
3	**RIB-IN**	(1,0)	$\boxed{(1, 2, 0)}$	$\boxed{(2,0)}$	$\perp$
	QUEUE	[]	[]	[]	[(2, 1, 0); (2, 1, 2, 0)]
4	**RIB-IN**	(1,0)	$\boxed{(1, 2, 0)}$	(2,0)	$\boxed{(2, 1, 0)}$
	QUEUE	[]	[(1, 2, 1, 0)]	[]	[(2, 1, 2, 0)]
5	**RIB-IN**	$\boxed{(1,0)}$	(1, 2, 1, 0)	(2,0)	$\boxed{(2, 1, 0)}$
	QUEUE	[]	[]	[]	[(2, 1, 2, 0); (2, 1, 0)]
6	**RIB-IN**	$\boxed{(1,0)}$	(1, 2, 1, 0)	$\boxed{(2,0)}$	(2, 1, 2, 0)
	QUEUE	[]	[(1, 2, 0)]	[]	[(2, 1, 0)]

Table 3: BGP steps for Disagree Gadget

terminated for Agree Gadget, giving the solution $(1,0)$ as the **RIB** for node 1 and $(2,0)$ as the **RIB** for node 2.

In general, a gadget may converge for some ordering of executions for BGP and diverge for others. For example, Disagree Gadget is not guaranteed to converge and we can display an example of its divergence. We again continue from the previously discussed common steps and start at step 3. Let both nodes 1 and 2 process the single path in each of their non-empty queues in step 3 and in step 4 in sequence. This time, when each node processes the path in their queue and stores it in their **RIB-IN**, they will determine that this new path is more preferred by its policy, selecting it as its current **RIB** and sending it to its neighbor. At step 5 both nodes again only have a single path in any of its queues. Let both nodes 1 and 2 process the single path in each of their non-empty queues in step 5 and in step 6 in sequence. These paths replace the **RIB-IN** entry that is selected as the node's **RIB** but are not allowed by the policy, so each node will now need to fallback to its previous **RIB** that routed directly through node 0, and announce this path to its neighbors. The **RIB**, **RIB-IN**, and queues in step 6 are identical to those in step 2. It's clear to see that steps 3 to 6 can be repeated after step 6 indefinitely, resulting in BGP never terminating and thus diverging.

2.2 Metarouting

Metarouting [7, 12] showcases the use of Routing Algebras to design and represent routing protocols like BGP. The algebra can be written as a tuple $(\Sigma, W, \preceq, L, \phi, \oplus, f)$ [7, 12],

- Σ is a set of signatures containing the paths in the network;
- W is a set of weights used to order the elements of Σ;
- $\preceq$ is a total order on W;
- L is a set of labels for the nodes in the network;

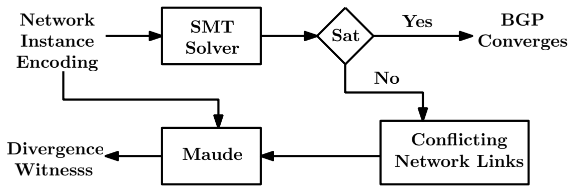


Figure 2: Verification Flow for BGP Verification

- $\phi \in \Sigma$ is the special signature representing paths not allowed by the network's policy;
- $\oplus$ is a binary operator that creates a new signature from an input of a label and a signature by prepending the node corresponding to the label to the path corresponding to the signature;
- f is a function that maps elements from Σ to elements in W.

One of the benefits of this representation is that it is sufficient to show that if a Routing Algebra is monotonic, $\forall l \in L, \forall \alpha \in \Sigma : f(\alpha) \preceq f(l \oplus \alpha)$, then the corresponding network will converge. We use $\prec$ when $a \preceq b$ and $a \neq b$.

We use Agree Gadget and Disagree Gadget from Figure 1 to provide examples for a routing algebra. As noted previously, the policy of Agree Gadget prefers to traverse directly to the destination while Disagree Gadget prefers to traverse using a neighbor before reaching the destination, otherwise the gadgets are identical. When defining a routing algebra we can use the same symbols and definitions for both gadgets except when defining the function f. We use the notation f_A for Agree Gadget and f_D for Disagree Gadget, otherwise the other symbols will be shared between the two. We have that $W = \{1, 2\}$, $L = \{0, 1, 2\}$, $\Sigma = \{(1, 0), (1, 2, 0), (2, 0), (2, 1, 0)\}$. For the functions f_A and f_D, we define them such that:

$$\begin{aligned} f_A((1,0)) &= 1 & f_D((1,0)) &= 2 \\ f_A((1,2,0)) &= 2 & f_D((1,2,0)) &= 1 \\ f_A((2,0)) &= 1 & f_D((2,0)) &= 2 \\ f_A((2,1,0)) &= 2 & f_D((2,1,0)) &= 1 \end{aligned} \tag{1}$$

Using these definitions, it is straightforward to confirm the monotonicity property for the algebra, or provide a counter example. For Agree Gadget it is clear that $1 = f_A((1, 0)) \preceq f_A((2, 1, 0)) = 2$ and $1 = f_A((2, 0)) \preceq f_A((1, 2, 0)) = 2$. Hence, the algebra is monotonic and Agree Gadget converges. On the other hand, for the algebra corresponding for Disagree Gadget to be monotonic we need the following to hold $2 = f_D((1, 0)) \preceq f_D((2, 1, 0)) = 1$, implying that $2 \preceq 1$, which is clearly false. Hence, we cannot make the same statement about convergence for Disagree Gadget.

Theorem 2.1. *[7, 12] For a network gadget $\mathcal{N}$, if a Routing Algebra corresponding to $\mathcal{N}$ is monotonic then BGP always converges for $\mathcal{N}$.*

3 Verification Workflow

Figure 2 shows the main steps and tools proposed for the BGP verification. The key insight is to combine (1) SMT solvers to automatically check for BGP convergence and (2) the Maude rewriting tool to enable the generation of a witness trace of BGP divergence. The witness trace can then be used by engineers to correct network configurations, e.g., path preference policies.

The main challenge of producing such a witness is the great size of networks which render model-checkers impractical. The solution uses SMT solver's capability of producing unsat cores to help pinpoint which part of the network might be contributing to BGP divergence, called the conflicting network links. This information is then used to enhance model-checking performance making verification feasible for larger networks.

The main steps of the verification flow are as follows and are made precise in the subsequent sections:

1. **BGP Network Instance:** The input of the verification flow is the encoding of the network instance in SMT. The encoding specifies the network topology and path preference policies. This encoding is used by both the SMT solver and Maude in subsequent steps.
2. **SMT Solver:** From the encoding of the network instance, an SMT problem is generated specifying the BGP convergence criterion specified by Metarouting [7, 12]. The satisfiability of this problem implies that BGP convergence is always guaranteed (Section 4).
3. **Conflicting Network Links Extraction:** If the SMT problem for BGP convergence is not satisfiable, then the second part of the flow starts with the objective of producing a witness for divergence. In particular, the unsat core produced by the SMT solver contains the network links that may contribute to the divergence.
4. **Maude Model-Checking:** The model-checking problem (Section 6) consists of proposed sound criterion for divergence (Section 5). It takes as inputs the conflicting network links and returns a witness of BGP divergence. Since the method is sound, but not shown to be complete, computation may not terminate.

4 Automating BGP Convergence Check with SMT

4.1 Computing a Monotonic Global Ranking

The metarouting work described previously shows that if a Routing Algebra is monotonic, then the corresponding network will converge (Theorem 2.1). Automating this check has been an open problem for which we address in this section. The key insight is instead of proving monotonicity, we check whether there is a global ranking among the paths in the network such that preserves the local preferences of paths and is monotonic.

Definition 4.1. A global ranking of paths in a network is a total ordering among the paths such that if a node prefers path p over path q then path p has a higher rank than path q. A global ranking is monotonic if $p = n \oplus q$ then path p has a higher rank than path q.

Lemma 4.2. *A routing algebra is monotonic if and only if there exists a monotonic global ranking.*

Proof. The backward direction is immediate, as if there is a global rank that is monotonic, then the algebra is also monotonic. For the forward direction, assume that a Routing Algebra is monotonic. Then we can obtain a global order by using the total order obtained by the topological sorting of the partial order of in the routing algebra that gives us the monotonic global ranking. □

We utilize an SMT solver, cvc5 [1], to find a global ranking and determine if a network will converge. cvc5 supports the defining of custom datatypes and the theory of sequences. We create a custom datatype consisting of the nodes in a given gadget. This should allow the solver to more effectively search for a solution as for any given symbol of the custom datatype, there are only finitely many choices. Paths are represented as sequences of this custom datatype. The sequence theory allows us to reason about sequences and various built in operations on sequences, e.g., concatenation, length, subsequence, etc.

First, we define two helper functions, H and T, to be used by the SMT Solver:

- H takes as input a path p and returns the first node in that path
- T takes as input a path p and returns that path without its first node $\mathrm{H}(p)$

Theories in SMT require that all functions are total, so the SMT solver is free to assign any value to H or T for inputs that these functions are not properly defined for, e.g., the empty path. Though, this shouldn't present any issues in our implementation.

We also define the function $\mathcal{S}$ that takes a path and returns an integer to represent the local policy at each node. Let p and q be paths that start from the same node n, i.e., $\mathrm{H}(p) = n = \mathrm{H}(q)$.

- If $p \neq q$ then $\mathcal{S}(p) \neq \mathcal{S}(q)$;
- If $\mathcal{S}(p) > \mathcal{S}(q)$ then node n prefers path p to path q;
- If p is the empty path then $\mathcal{S}(p) = 0$;
- If p is allowed by the policy at node n then $\mathcal{S}(p) > 0$;
- If p is not allowed by the policy at node n then $\mathcal{S}(p) < 0$.

In our examples, each policy at a node is represented as an ordered list of allowed paths. We assign the last element in the list the integer 1 and iterate backwards assigning increasingly higher integers for each subsequent member iterated through. Any non empty path not seen at one of the policies is mapped to -1.

Lastly, we define the function $\mathcal{R}$ that takes a path and returns a natural number to represent the global ranking. Let p and q be paths.

- If $p \neq q$ then $\mathcal{R}(p) \neq \mathcal{R}(q)$;
- The ranking is in ascending order of the naturals such that if $\mathcal{R}(p) < \mathcal{R}(q)$ then p is ranked higher than q;
- If p is the empty path then $\mathcal{R}(p) = 0$;
- The ranking must preserve the ordering of the local preferences for each node's policies, i.e., if $\mathrm{H}(p) = \mathrm{H}(q)$ and $\mathcal{S}(p) > \mathcal{S}(q)$ then $\mathcal{R}(p) < \mathcal{R}(q)$;
- The ranking must also be monotonic, i.e., if $\mathrm{T}(p) = q$ then $\mathcal{R}(p) > \mathcal{R}(q)$.

Let P be all the paths allowed by a node in the network. To define $\mathcal{R}$ in SMT we use the following formulas: $\forall p, q \in P$

(1) $p \neq q \implies \mathcal{R}(p) \neq \mathcal{R}(q)$
(2) $p \neq q \wedge \mathrm{H}(p) = \mathrm{H}(q) \wedge \mathcal{S}(p) < \mathcal{S}(q) \implies \mathcal{R}(p) > \mathcal{R}(q)$
(3) $\mathrm{T}(p) = q \implies \mathcal{R}(p) > \mathcal{R}(q)$

Formulas consisting of quantifiers like $\forall$ can be extremely difficult for SMT solvers to deal with. To avoid this problem, we instead iterate through all the paths in P and construct corresponding formulas for every pair $p, q \in P$ to send to the solver. (1) requires that distinct paths are given distinct global rankings, (2) requires that local preferences are preserved, and (3) requires monotonicity. If the solver returns satisfiable (SAT), then it was able to find a global ranking.

Theorem 4.3. *The SMT encoding of a gadget, described above, is satisfiable if and only if there is a monotonic routing algebra for the corresponding gadget.*

Proof. Suppose that the SMT solver returns SAT after given the formulas described above. Then a monotonic global ranking can be extracted from $\mathcal{R}$ directly as the preferences of the local policies will be preserved from the constraints correspond to (2) and the formulas corresponding to (3) force the ranking to be monotonic. It follows from Lemma 4.2, the existence of the routing algebra.

Now, suppose that such a monotonic global ranking exists. Then, there must be an ordering of the paths that preserves the preferences of the local policies and is monotonic. One can construct from this ordering a model satisfying all the formulas of the SMT encoding by using the the order for $\mathcal{R}$ and the local preferences for $\mathcal{S}$. □

The result of SAT to compute a global ranking implies that the network converges for BGP. If the solver returns unsatisfiable (UNSAT), then no such global ranking exists. The monotonicity condition is sufficient but not necessary, so it cannot be determined whether the gadget converges or diverges given an UNSAT result alone from the solver. To address this limitation we take advantage of the solver's unsat core and use this in a heuristic to attempt to find an example of divergence to both confirm the gadget does indeed diverge and to reveal the troublesome properties of the gadget that lead to the divergence so that a user may make appropriate changes to the gadget to ensure BGP converges for their network.

4.2 Extracting Links of Interest From an Unsatisfiable Core

To achieve this we use what is known as an UNSAT core. An UNSAT core is a subset of the input constraints that would lead to an UNSAT result. In particular, we query in search for a minimal UNSAT core in cvc5, a subset of the constraints UC that the solver produces UNSAT for, but any strict subset of UC produces SAT. This core may not be the minimum possible size but it can have drastically fewer elements than the original input set. Any computed UNSAT core would reveal a conflict in propositions that make it impossible to satisfy the monotonicity condition to correctly define $\mathcal{R}$. Since we form each constraint from a concrete pair of paths p, q, rather than using any quantifiers, when examining the UNSAT core the set of paths that fail to preserve monotonicity are explicitly displayed. For each path $p_i = (n_1, n_2, \ldots, n_k)$ that appears in a constraint from a minimal UNSAT core such that $k > 2$, we compute l_i as the link from node n_1 to node n_2. All such links, l_i, are passed to our implementation using Maude as described in the following section 6 to aid in locating a sequence of states from executing BGP that diverges for a given gadget.

To provide an example of what an unsat core might look like for a gadget that diverges, consider *Naughty Gadget* as shown in Figure 3. It is not difficult to show that this gadget can diverge, oscillating between processing paths from node 3 and node 4. Building the corresponding boolean constraints for this gadget as described above and sending it to the SMT solver gives an UNSAT result.

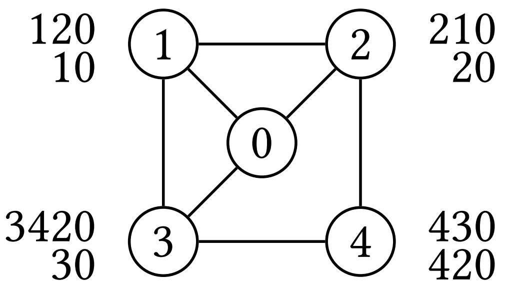


Figure 3: Naughty Gadget

Further, taking a look at the unsat core provided by the solver will look like the following:

- Score Constraints
  - $\mathcal{S}((3,0)) = 1$
  - $\mathcal{S}((3,4,2,0)) = 2$
  - $\mathcal{S}((4,2,0)) = 1$
  - $\mathcal{S}((4,3,0)) = 2$
- Local Pref Constraints
  - $\mathrm{H}((3,0)) = \mathrm{H}((3,4,2,0)) \wedge \mathcal{S}((3,0)) < \mathcal{S}((3,4,2,0)) \implies \mathcal{R}((3,0)) > \mathcal{R}((3,4,2,0))$
  - $\mathrm{H}(4,2,0)) = \mathrm{H}((4,3,0)) \wedge \mathcal{S}(4,2,0)) < \mathcal{S}((4,3,0)) \implies \mathcal{R}((4,2,0)) > \mathcal{R}((4,3,0))$
- Monotonic Constraints
  - $\mathrm{T}((3,4,2,0)) = (4,2,0) \implies \mathcal{R}((3,4,2,0)) > \mathcal{R}((4,2,0)$
  - $\mathrm{T}(4,3,0)) = (3,0) \implies \mathcal{R}((4,3,0)) > \mathcal{R}((3,0))$

It is apparent that each constraint listed containing an implication symbol has its antecedent as evaluating to true, so we focus on just the conclusion when discussing these propositions. A chain of inequalities can be created such that:

$$\begin{aligned}\mathcal{R}((3,0)) &> \mathcal{R}((3,4,2,0)) \\ &> \mathcal{R}((4,2,0)) \\ &> \mathcal{R}((4,3,0)) \\ &> \mathcal{R}((3,0))\end{aligned}$$

This would require that $\mathcal{R}(3,0) > \mathcal{R}(3,0)$, but this is a contradiction. Hence, no ranking can be found satisfying the monotonicity conditions. This is shown using only 8 of the 217 total constraints originally given to the solver. Moreover, these paths found in the unsat core only begin from nodes 3 and 4 and only directly route through nodes $2,3,4$ from its starting node, excluding the common destination 0. We compute the links: node 3 to node 4, node 4 to node 2, and node 4 to node 3 as the set of links to send to Maude as a parameter to help in the search for an oscillating sequence of states that would cause divergence, using only 3 of the 11 total links in Naughty Gadget.

5 Sound Criterion of BGP Divergence

Following the verification flow described in Section 3, if the SMT solver does not produce SAT result for convergence, the task is to provide a counterexample for engineers to use for debugging purposes if possible.

In principle, model-checkers can determine counterexamples for liveness properties such as the BGP convergence property, but these checks are notoriously difficult. We follow instead the solution of reducing the checking of a liveness property to the checking of a safety property [2]. This reduction enables the model-checker to make more effective use of invariants thus considerably improving performance.

This section introduces such a safety property. Intuitively, we observe that divergence often manifests as cycles in the flow of messages across links, where specific paths are continuously produced and consumed in a repeating pattern without ever leading to stabilization. This motivates us to study the link-level message dynamics of BGP execution and to define divergence in terms of patterns in message production and consumption.

The following subsections formalize the BGP divergence criterion. Section 5.1 reviews the trace semantics of BGP networks based on the Simple Path Vector Protocol introduced by Griffin et al. [6]. Section 5.2 introduces the divergence property based on the messages that are produced and consumed by network links. Section 5.3 illustrates the criterion with an example.

5.1 Formal Network Setup and Execution Model

We begin by formalizing the BGP network abstraction. A BGP network instance is represented by a gadget $G = (\mathcal{N}, \texttt{peer}, \mathcal{P}, S_0)$, where $\mathcal{N}$ is the finite set of nodes, `peer` maps each node to its peers, $\mathcal{P}$ is the local preference table assigning rankings to paths, and $S_0 = (\textbf{Rib}, \textbf{Rib-In}, Q)$ is the initial state of the system. Each state, including the initial state S_0, can be viewed as the gathering of the states maintained by each node $N \in \texttt{node}$, i.e.,

- **Rib**(N): the best path currently selected by N;
- **Rib-In**(N): a mapping from peers to their most recent path advertisements;
- $Q(N \Leftarrow N')$: a FIFO queue of messages pending from neighbor N' to N, per neighbor N'.

Execution Model Abstraction. The semantics of BGP route selection and message propagation follow the Simple Path Vector Protocol (SPVP) model introduced by Griffin et al. [6]. SPVP simplifies BGP to a message-passing system where each node asynchronously processes received paths, applies local preferences, and may update its selection and generate new messages.

The behavior of the BGP network is modeled via a scheduling algorithm $\mathcal{A}$, which selects, at each step, a valid transition $t = (N_r \Leftarrow N_s, P)$, representing node N_r processing path P received from neighbor N_s. A transition is said to be valid for a state if P is the head of the corresponding queue in that state. The execution of a transition results in the update of the the system state via the SPVP transition function.

Built on the above specification, an execution trace $\mathcal{T}$ is a sequence of states $S_0, S_1, \ldots, S_n$, where each transition corresponds to a valid transition applied according to an algorithm $\mathcal{A}$. Each trace implicitly induces a sequence of message productions and consumptions along each directed link.

5.2 BGP Divergence Criterion

To capture divergence in terms of observable system behavior, we introduce the notions of message production and consumption on each link.

Definition 5.1. For each directed link $N \Leftarrow N'$ and a trace $\tau = (S_A, S_1, \ldots, S_B)$, we denote,

- $P^{A \rightsquigarrow B}_{N \Leftarrow N'}$: the list of paths produced by N' and sent to N in the segment $S_A \rightsquigarrow S_B$ of the trace τ;
- $C^{A \rightsquigarrow B}_{N \Leftarrow N'}$: the list of paths consumed by N from N' in the segment $S_A \rightsquigarrow S_B$ of the trace τ.

These sequences are computed recursively over the trace, respecting the FIFO semantics of BGP message queues. The divergence of a link can further be indicated by a cyclic pattern in the production sequence that constantly replenishes the queue, thereby preventing stabilization.

We now define the central notion of a divergence point, which characterizes a trace segment that establishes recurring behavior indicative of global divergence.

Definition 5.2. A state S_B is said to be a divergence point of a prior state S_A if the following requirements are met:

(1) The control plane state, i.e., the selected paths (**Rib**) and the received advertisements (**Rib-In**), are same for both states;
(2) There exists a trace $\tau = (S_A, S_1, \ldots, S_B)$ such that for every link $N \Leftarrow N'$ that is active in the trace, the message production sequence $P^{A \rightsquigarrow B}_{N \Leftarrow N'}$, consumption sequence $C^{A \rightsquigarrow B}_{N \Leftarrow N'}$, and the S_A message queue $Q_A(N \Leftarrow N')$ exhibit the following relationship,
   - We can find three natural numbers $k_p, k_c, k_q \in \mathbb{N}$, and a basic recurrent path pattern $R = R_p + R_s$, where the prefix R_p and the suffix R_s together form the loop body R, such that,
   - the production sequence can be written as $R_s + R^{k_p} + R_p$,
   - the consumption sequence is some repetition of R, i.e., R^{k_c},
   - the original message queue is in the form of $R^{k_q} + R_p$, and
   - $k_p + 1 \geq k_c$ ensures that the queue never drains.

This condition ensures that the queue state is self-sustaining, i.e., the system can re-enter the same queue configuration at S_B after consuming and producing such configuration of paths, enabling further repetition. This is formalized by the following statements.

THEOREM 5.3. *If a trace segment $S_A \rightsquigarrow S_B$ satisfies the divergence point conditions, then there must exist a further segment $S_B \rightsquigarrow S_C$ that also ends in a divergence point S_C with respect to S_B.*

This means that any trace segment satisfying the divergence point conditions can be extended to an infinite trace.

COROLLARY 5.4. *If a BGP instance has a trace with a trace segment $S_A \rightsquigarrow S_B$ satisfying the divergence point conditions, then BGP does not always converge for the given BGP instance.*

5.3 Example

We illustrate the divergence criterion using the *Disagree Gadget* described in Section 2. The topology and local preferences are shown in Figure 1.

Starting from the initial state S_0, the following trace can be constructed,

$$S_0 \xrightarrow[(N_0)]{N_1 \Leftarrow N_0} S_1 \xrightarrow[(N_0)]{N_2 \Leftarrow N_0} S_2 \xrightarrow[(N_2,N_0)]{N_1 \Leftarrow N_2} S_3 \xrightarrow[(N_1,N_0)]{N_2 \Leftarrow N_1} \mathbf{S_4}$$
$$\xrightarrow[(N_2,N_1,N_0)]{N_1 \Leftarrow N_2} S_5 \xrightarrow[(N_1,N_2,N_0)]{N_2 \Leftarrow N_1} S_6 \xrightarrow[(N_2,N_0)]{N_1 \Leftarrow N_2} S_7 \xrightarrow[(N_1,N_0)]{N_2 \Leftarrow N_1} \mathbf{S_8},$$

where each transition $t = (N_r \Leftarrow L_s, P)$ is denoted as $S \xrightarrow[P]{N_r \Leftarrow N_s} S'$, where N_r processes a path P (the front of the queue) received from N_s. We now show that the subtrace $S_4 \rightsquigarrow S_8$ demonstrates a violation in our proposed safety property for divergence, as S_8 is a divergence point with respect to S_4.

Control Plane. First, we observe that the control plane state remains unchanged, thus satisfying condition (1) of Definition 5.2:

$$S_4.\mathbf{Rib}(N_1) = S_8.\mathbf{Rib}(N_1) = (N_1, N_2, N_0),$$

$$S_4.\mathbf{Rib}(N_2) = S_8.\mathbf{Rib}(N_2) = (N_2, N_1, N_0),$$

$$S_4.\mathbf{Rib\text{-}In}(N_1) = S_8.\mathbf{Rib\text{-}In}(N_1) = \{N_0 : (N_0), N_2 : (N_2, N_0)\},$$

$$S_4.\mathbf{Rib\text{-}In}(N_2) = S_8.\mathbf{Rib\text{-}In}(N_2) = \{N_0 : (N_0), N_1 : (N_1, N_0)\}.$$

Message Dynamics. Next, we examine the message dynamics on the two active links,

- For $N_1 \Leftarrow N_2$,

$$C^{4 \rightsquigarrow 8}_{N_1 \Leftarrow N_2} = [(N_2, N_1, N_0), (N_2, N_0)]$$

$$P^{4 \rightsquigarrow 8}_{N_1 \Leftarrow N_2} = [(N_2, N_0), (N_2, N_1, N_0)]$$

$$S_4.Q(N_1 \Leftarrow N_2) = [(N_2, N_1, N_0)]$$

- For $N_2 \Leftarrow N_1$,

$$C^{4 \rightsquigarrow 8}_{N_2 \Leftarrow N_1} = [(N_1, N_2, N_0), (N_1, N_0)]$$

$$P^{4 \rightsquigarrow 8}_{N_2 \Leftarrow N_1} = [(N_1, N_0), (N_1, N_2, N_0)]$$

$$S_4.Q(N_2 \Leftarrow N_1) = [(N_1, N_2, N_0)]$$

We now demonstrate that the divergence point condition holds for both links. Consider the following lasso recurring structure,

- for $N_1 \Leftarrow N_2$, $R_p = [(N_2, N_1, N_0)], R_s = [(N_2, N_0)], k_q = k_p = 0, k_c = 1$,
- for $N_2 \Leftarrow N_1$, $R_p = [(N_1, N_2, N_0)], R_s = [(N_1, N_0)], k_q = k_p = 0, k_c = 1$.

Thus, the divergence point condition is satisfied: the message queue maintains a self-replenishing pattern, and the system can re-enter the same configuration after each cycle, enabling indefinite repetition.

This demonstrates that BGP fails to converge in this instance, as the trace segment $S_4 \rightsquigarrow S_8$ satisfies the safety-based divergence condition.

Further Remarks. In order to automate the search for a diverging point one needs to determine the candidate trace from $S_A \rightsquigarrow S_B$. This is in principle not feasible when verifying large networks with hundreds of nodes. A key insight, as we describe in more detail in Section 6, is to provide as input the set of links that are activate in the candidate trace. This set of candidates, as illustrated in Figure 2, can be determined from the unsat core obtained from the SMT solver. Given this input of candidate links, verification is feasible as the model-checker can use this fact to reduce state-space.

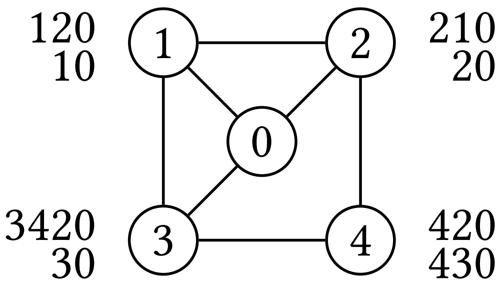


Figure 4: Bad Gadget

The proposed criterion is sound, but its completeness is still open and left to future work. Indeed the bad gadget shown in Figure 4 is an example for which the criterion does not seem to work. In particular, the model-checker fails to converge, that is, it does not terminate. It is not clear whether it is because for this gadget the problem is indeed hard or whether the criterion is incomplete. We suspect the latter.

6 Formalizing Criteria in Rewriting Logic

Building on the divergence criterion proposed in the previous section, this section describes how we formalize and implement the criterion in rewriting logic using the Maude system. Our goal is to model BGP network behavior faithfully and to enable automated search for divergence traces that satisfy our safety-based condition.

We organize the presentation into three parts: Section 6.1 describes the formal system model in Maude, Section 6.2 presents the rewrite rules for protocol execution, and Section 6.3 discusses how the search space is reduced using the SMT-generated unsat core and other rewriting heuristics.

6.1 System Modeling

We model a BGP network as a set of nodes (i.e., autonomous systems) and a booking object for our proposed divergence criterion.

Nodes: Each node is represented as an object in Maude, as illustrates the following term:

```
< N1 : NodeClass |  id : nid(1),
    rib : nid(1) nid(2) nid(0),
    rib-in : (
        (nid(1) <= nid(0)) !-> nid(0),
        (nid(1) <= nid(2)) !-> nid(2) nid(0)
    ),
    permitted : (
        (nid(2) nid(0)) :: nid(0)
    ),
    neighbours : nid(2),
    queue : (
       (nid(1) <= nid(2)) !-> nid(2) nid(1) nid(0) ) >,
```

A node term encapsulates its local state, including its identifier (`id`), selected path (`rib`), received advertisements (`rib-in`), preferences (i.e. policies) (`permitted`), and pending message queues (`queue`). For example, in the instance above for a node `nid(1)`, the path selected to node `nid(0)` passes through nodes `nid(2)`; `(nid(1) <= nid(2)) !-> (nid(2) nid(0))` denotes that node `nid(2)` advertised to node `nid(1)` its best path to node `nid(0)` as passing through node `nid(2)`; and `(nid(2) nid(0)) :: nid(0)` indicates that the path going through `nid(2)` to `nid(0)` is more preferable than directly routing to `nid(0)`, and these are the only two permitted paths; finally, the queue stores the advertisements to be processed by the node.

Bookkeeping object: In addition to the network nodes, the system also includes a bookkeeping object for our proposed divergence criterion. This object maintains trace information necessary for evaluating the proposed divergence point condition.

```
< DPC : DPClass | sz : 2,
  consume : (
       (nid(1) <= nid(2)) !-> nid(2) nid(1) nid(0)
    ++ (nid(2) <= nid(1)) !-> nid(1) nid(2) nid(0)
  ),
  produce : (
       nid(1) !-> nid(1) nid(0)
    ++ nid(2) !-> nid(2) nid(0)
  ),
  init : (
    (nid(2) <= nid(1)) !-> nid(1) nid(2) nid(0),
    (nid(1) <= nid(2)) !-> nid(2) nid(1) nid(0)
  ),
  all-rib : (
       (nid(1) nid(2) nid(0) ; nid(2) nid(1) nid(0))
    ++ (nid(1) nid(0) ; nid(2) nid(1) nid(0))
  ),
  all-rib-in : ((
      (nid(1) <= nid(0)) !-> nid(0),
      (nid(2) <= nid(0)) !-> nid(0),
      (nid(1) <= nid(2)) !-> nid(2) nid(0),
      (nid(2) <= nid(1)) !-> nid(1) nid(0)
    ) ++ (
      (nid(1) <= nid(0)) !-> nid(0),
      (nid(2) <= nid(0)) !-> nid(0),
      (nid(1) <= nid(2)) !-> nid(2) nid(1) nid(0),
      (nid(2) <= nid(1)) !-> nid(1) nid(0)
    )
  ) >
```

The above term illustrates a concrete example of a divergence point object. It maintains a history of size 2, which records the past two link consumption (`consume`), path announcements (`produce`), control plane information (`all-rib, all-rib-in`), and records the initial queue contents (`init`) two-step before the current state accordingly.

Nodes and the bookkeeping structure are wrapped in a single term to ensure full rewriting on the whole system.

```
sort BGP .
op {_} : Configuration -> BGP [ctor] .
```

To signal successful detection of divergence during a search, we introduce a special constant object:

```
op diverged : -> Configuration [ctor] .
```

This object is inserted into the configuration when the divergence point condition is met, terminating the search.

6.2 Rewrite Rules

Maude transitions are specified via rewrite rules, which model the operational semantics of BGP node behavior as defined by the SPVP model.

Rewrite Configuration. To achieve efficient bookkeeping and searching, we introduce two rewrite configurations to be specified by the user before the search starts to narrow down and divide the search space.

- *sp-links*: We categorize all message links in the system into two types: *sp-links* and *non-sp-links*, where *sp-links* are links that are involved in the trace segment between divergence points, while *non-sp-links* are the remaining links. By specifying *sp-links*, the system will search for states that oscillate only between those links. This further eliminates the need for storing history queue and control plane information of *non-sp-links*, as well as supporting us to clear the history when *non-sp-links* are touched, and can thus largely help cut the state space.
- *sp-recur-size*: The exact number of steps involved in the oscillation proof, i.e., between two divergence points. By fixing this number, the system no longer needs to save history for longer than *sp-recur-size* steps, and will populate bookkeeping state when exceeded.

Rule. The main rewrite rule takes the following form:

```
rl [process-queue] : {
  C < A : NodeClass | id : N,
    queue : ((N <= N') !-> Msg :: Tail) ;  Q, ... >
  < DPC : DPClass | ... >
}
=>
if dp-check( dpc: < DP : DPClass | ... > )
then { diverged ... }
else {
  nodes-update( ... )
  if (N <= N') in sp-links then
    dp-update( updates: ..., sys: ... )
  else
    dp-clear( sys: ... )
  fi
}
fi .
```

At each step, the rule will first check if the divergence criterion is satisfied based on the bookkeeping term. If so, a divergence flag will be inserted. Otherwise, as modeled in SPVP, a node processes the first message in one of its incoming queues through pattern matching, potentially updates its state, and produces new path announcements to neighbors if needed. The bookkeeping object is cleared whenever the rewritten link is not in the *sp-links*, and is updated otherwise.

Rules are defined to preserve asynchrony: at each rewrite step, a node processes the message from an arbitrary non-empty queue. To enable verification, the rewrite engine can be asked to search for a state where the divergence flag is inserted.

6.3 Search Space Reduction

While our rewriting logic framework enables precise modeling and analysis of BGP convergence behavior, the state space for model checking remains prohibitively large in general. This agrees with observations in the literature [17]. This section introduces two key optimizations that significantly reduce the search space and make verification feasible even for larger networks.

We identify two sources of combinatorial explosion:

- **Rewrite Configuration Combinatorics.** As introduced in Section 6.2, users must specify a set of *sp-links* – the subset of network links suspected of participating in divergence. However, for a network with *n* links, the number of such subsets is exponential in size. Exhaustively enumerating these configurations is computationally infeasible.
- **Message Interleaving Explosion.** BGP's asynchronous nature means that messages can be processed in arbitrarily different orders. For large networks, this leads to a vast number of rewrite interleavings, even when only a small portion of the network contributes to divergence.

We address these issues using two complementary techniques.

Extract sp-links from SMT unsat core. A key insight from Section 4 is that the SMT solver's unsat core pinpoints the specific constraints responsible for the failure of guaranteeing BGP convergence. These constraints correspond to path preference conflicts that arise from particular network links. Therefore, links appearing in the unsat core can be useful to be specified as the *sp-links* for the system to search for divergence. We thus use the SMT-derived unsat core to initialize the set of `sp-links` automatically. This avoids the need for manual or exhaustive enumeration and sharply reduces the search space to only those parts of the network that may be involved in the potential oscillation.

Atomic Rewrite for Non-sp-links. Our second optimization is based on the observation that links outside of *sp-links* usually do not participate in the divergence pattern. Therefore, the specific order in which messages on non-*sp-links* are processed might not affect the existence of a divergence trace, in which cases, interleavings among them are semantically irrelevant.

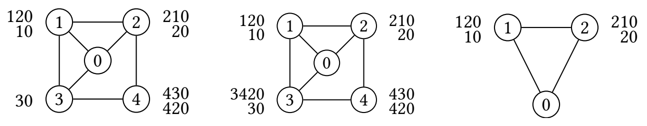
To exploit this, we introduce an atomic rewrite rule that processes the head of all non-empty non-*sp-links* in a single step. This reduces the interleaving among non-critical parts of the network and allows the rewrite engine to focus on meaningful variations in the behavior of *sp-links*.

```
crl [process-atomic] : { Conf1 } => { Conf2 }
if Nodes2 := nodes-update-all(
                          sys: Conf1,
                         links: non-empty-non-sp-links )
/\ DPC2 := dp-clear( sys: Nodes2 )
/\ Conf2 := DPC2 Nodes2 .
```

where `nodes-update-all` computes all the nodes in the configuration `Conf1` with non-*sp-links* and non-empty queues; and `dp-clear` executes the BGP algorithm, i.e., consuming the head of the nodes's queues, and adding to queues messages produced by link sources.

It must be pointed out that these heuristics are sound as no new behaviors are introduced by using atomic steps for non-*sp-links*. They, however, may lead to the tool not identifying a witnessing counterexample with the divergence. This happens, for example, with the bad gadget described at the end of Section 5.2 for which our formalization is not capable of finding a suitable counterexample.

7 Evaluation



a. Good Gadget **b. Naughty Gadget** **c. Disagree Gadget**

Figure 5: Base Gadgets

All experiments were run on a Windows 11 Home 24H2 machine, with an Intel Core i7-14700KF CPU, consisting of 64 GB of RAM, with Python 3.12.7 using Maude python bindings 1.4.0 [11] and cvc5 1.2.0. We run experiments on Good Gadget, Naughty Gadget, Disagree Gadget as seen in Figure 5. Naughty Gadget and Disagree Gadget have been discussed in previous sections, both are known to be able to diverge. Good Gadget instead is known to have a corresponding routing algebra that is monotonic and therefore always converges. We also run experiments by building larger gadgets using combinations of the three previously mentioned gadgets as base gadgets. When we create a combination in these experiments, we use gadgets where all but one are equivalent to the Good Gadget and the remaining is the gadget denoted in the first column. The total number of base gadgets used to create the gadget used for each experiment is denoted *size* and is reported in the second column. Thus, $size - 1$ gadgets are Good Gadget and the remaining gadget is the one recorded in the first column. The third column shows a tuple consisting of the number of nodes (#N), the number of links (#L), and the number of paths (#P) respectively in the corresponding gadget. The fourth column shows the amount of time in seconds it took to create the corresponding formulas for the SMT solver and then running the solver to check for convergence of the gadget, including the time it takes to compute the unsat core if necessary. The fifth and last column displays the amount of time in seconds it took to run the Maude implementation to find a trace that would oscillate resulting in the divergence. We provide experiments for when both SMT and Maude (or just SMT when the gadget converges) are able to terminate with a solution within an hour. If the gadget converges, there's no oscillating trace that can be found and we instead report **N/A** since we don't run the Maude program for these gadgets. There are three tables corresponding to the three types of gadgets we used, the base gadgets, *rooted combination* gadgets, and *nested combination* gadgets. The two combinations are constructed in a way such that the combined gadget diverges if and only if at least one of of the gadgets used to create it diverges. This is shown as we discuss how these combined gadgets are created.

Gadget	Size	(#N, #L, #P)	SMT (s)	Maude (s)
Good	1	(5, 11, 7)	0.026	**N/A**
Disagree	1	(5, 11, 8)	0.028	0.369
Naughty	1	(3, 4, 4)	0.018	0.121

Table 4: Base Gadget Results

The first combination type we denote as a *rooted combination*. We take n gadgets, $g_1, g_2, ..., g_n$ as input to construct a new gadget. First, we create a new node O as the new common destination. Then, we draw a link from each common destination O_{g_i} for g_i to O. Finally, every existing path in each gadget is appended with the new common destination O, and each O_{g_i} is given a policy for path preferences only consisting of (O_{g_i}, O).

For example, suppose we construct a rooted combination of Good Gadget as g_1 and Disagree Gadget as g_2 as seen in Figure 6. Then the path $(1, 2, 0)$ from Disagree Gadget becomes $(2 : 1, 2 : 2, 2 : 0, 0)$ by first tagging each node with the instance number 2, the index of Disagree Gadget from the list of inputs, then appending the global node 0.

It is clear that no two gadgets g_i, g_j for $i \neq j$ will have a path that traverses nodes from both gadgets. This means that we can essentially treat each gadget independently in the new combined gadget for the purposes of determining if BGP will converge or diverge. Hence, if some gadget g_i diverges then the new combined gadget will also diverge. Our machinery is able to handle gadgets constructed using the rooted combination consisting of up to 500 nodes, 1200 links, and 800 paths.

Gadget	Size	(#N, #L, #P)	SMT (s)	Maude (s)
Good	2	(11, 24, 16)	0.122	**N/A**
Naughty	2	(11, 24, 17)	0.113	1.39
Disagree	2	(9, 17, 13)	0.068	0.238
Good	5	(26, 60, 40)	0.728	**N/A**
Naughty	5	(26, 60, 41)	0.66	3.585
Disagree	5	(24, 53, 37)	0.542	0.596
Good	10	(51, 120, 80)	3.074	**N/A**
Naughty	10	(51, 120, 81)	2.636	9.601
Disagree	10	(49, 113, 77)	2.352	1.794
Good	50	(251, 600, 400)	223.523	**N/A**
Naughty	50	(251, 600, 401)	68.336	209.484
Disagree	50	(249, 593, 397)	66.953	58.908
Good	100	(501, 1200, 800)	2655.43	**N/A**
Naughty	100	(501, 1200, 801)	324.879	862.888
Disagree	100	(499, 1193, 797)	292.157	240.22

Table 5: Rooted Combination Results

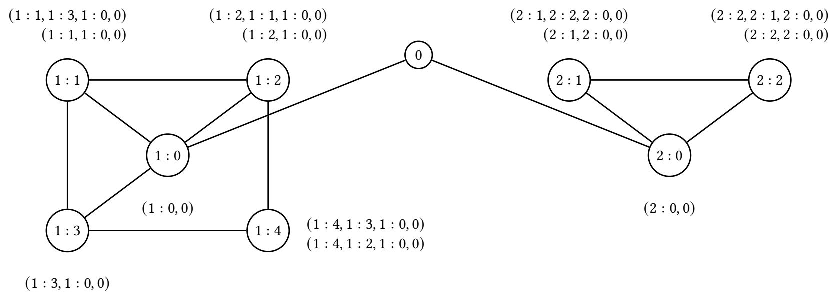


Figure 6: Rooted Combination Using Good and Disagree Gadgets

The other combination type we call a *nested combination*. This combination takes two gadgets g_a and g_b as inputs to construct a new gadget g_{ab}. The first step of this type is to replace every node in g_a that is not its common destination O_{g_a} with the nodes and links (but not paths) of gadget g_b. O_{g_a} becomes the common destination of the newly created nested combination g_{ab}. Only the nodes corresponding to the common destination in g_b, O_{g_b}, will have a link that directly connects them to O_{g_a}. We denote each newly created node as $A : B$ where A refers to a node from g_a and B refers to a node from g_b. Now, at each node $A : B$ to determine the policy for its path preference we compute the following: for each path $p_a = (u_1, u_2, \ldots, u_m, O_{g_a})$ in the policy for node A from g_a, for each path $p_b = (v_1, v_2, \ldots, v_n, O_{g_b})$ in the policy for node B from g_b, we create the path:

$$\begin{aligned} p_a \circ p_b &= (u_1, u_2, \ldots, u_m, O_{g_a}) \circ (v_1, v_2, \ldots, v_n, O_{g_b}) \\ &= (B : v_1, \ldots, B : v_n, B : O_{g_b}, u_2 : O_{g_b}, \ldots, u_m : O_{g_b}, O_{g_a}) \end{aligned}$$

An example using Disagree Gadget as g_a and Good Gadget as g_b to create a nested combination is shown in Figure 7. As an example of generating the image of the paths of a gadget, the good gadget path $(1, 3, 0)$ becomes two paths by composing the image of the disagree paths $(1, 2, 0)$ and $(1, 0)$, namely $(1 : 0, 2 : 0, 0)$ and $(1 : 0, 0)$ to the image $(1 : 1, 1 : 3, 1 : 0)$ of $(1, 3, 0)$. Thus we have:

$$(1 : 1, 1 : 3, 1 : 0) \circ (1 : 0, 2 : 0, 0) = (1 : 1, 1 : 3, 1 : 0, 2 : 0, 0)$$

and

$$(1 : 1, 1 : 3, 1 : 0) \circ (1 : 0, 0) = (1 : 1, 1 : 3, 1 : 0, 0)$$

To create larger nested combinations from more than two inputs we chain together the constructions using intermediate gadgets created as input for the next combination. For example if we want to combine gadgets g_a, g_b, g_c in a nested combination we first compute the nested combination g_{ab} using gadgets g_a and g_b, then we compute the combination using g_{ab} and g_c for the final combination.

The way these type of gadgets are constructed preserves the path preferences of the original gadgets. If path p_a was more preferred than q_a from gadget g_a, then path $p_a * p_b$ is more preferred than path $q_a * q_b$ for any paths p_b, q_b from gadget g_b. A similar argument can be made to show that path preferences are conserved for gadget g_b. Suppose that g_b diverges and we can find a trace that oscillates. For any image of g_b in the nested gadget, this oscillating trace can be translated to an oscillating trace involving only nodes of the image gadget.

Now, suppose that g_a diverges and we can find a trace that oscillates. Then, consider only the nodes that were created from O_{g_b}. This will resemble the original g_a and a similar trace can be found, thus showing divergence. Hence, if g_a or g_b diverges then g_{ab} diverges.

In our experiments we choose g_a as the gadget named in column one in the tables and all other gadgets used in the construction are chosen as the Good Gadget. We are only able to handle gadgets consisting of up to 100 nodes and 400 links, roughly a fifth of the amount of nodes for root combination before reaching the time limit of an hour. The number of paths in nested combinations grows much faster than the number of paths in rooted combinations, reaching up to 700 paths after two nested combinations on three gadgets. This seems to cause the SMT solver to become a bottleneck much faster, as the monotonicity constraints require us to iterate over every pair of paths in the gadget, making it more difficult for the solver to find a solution. The Maude implementation may find a suitable trace within the hour time if given the proper parameters as input, since the absolute number of paths does not impact the performance nearly as much as the number of nodes and links do.

Gadget	Size	(#N, #L, #P)	SMT (s)	Maude (s)
Good	2	(21, 72, 63)	4.86	**N/A**
Naughty	2	(21, 72, 71)	5.251	4.518
Disagree	2	(11, 34, 32)	0.889	0.25
Good	3	(101, 384, 623)	1744.585	**N/A**
Naughty	3	(101, 384, 687)	420.453	114.87
Disagree	3	(51, 190, 312)	96.773	2.608

Table 6: Nested Combination Results

8 Related Work

We take inspiration from related work, in particular, the Metarouting framework [7, 12] and existing Model-Checking approaches [15–17]. The main difference, however, is our goal of providing automated checks, which are sound and scalable.

Metarouting [7, 12] provides a general mathematical framework for determining whether a BGP network gadget converges. A key

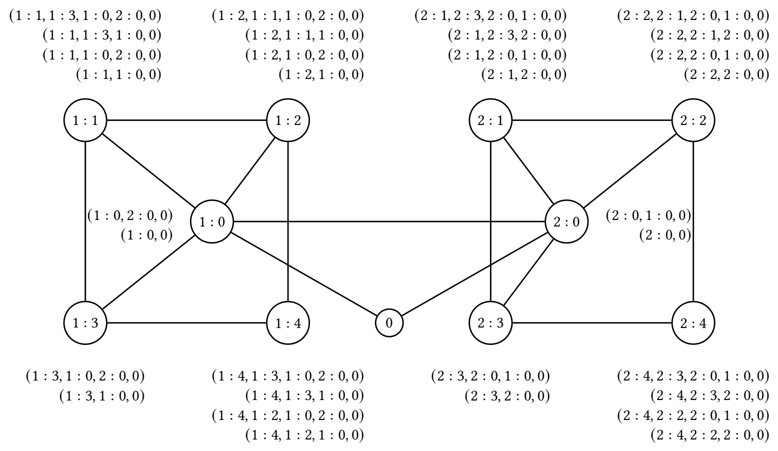


Figure 7: Nested Combination Using Disagree and Good Gadgets

limitation of the original work is the lack of fully automated methods for checking for convergence. The recent paper by Daggit and Griffin [4] has formalized the theoretical framework in Agda, which provides a means for proving convergence of network gadgets in a semi-automated fashion. The framework also enables the exploration of different convergence conditions through mechanized reasoning. We take a different approach by encoding the conditions proposed as a SMT problem. This enables the fully automated verification of network gadgets.

The second body of work uses model-checking approaches to determine BGP convergence and also to determine witnesses for divergence. Previous work [17] formalized the BGP algorithm and the BGP problem as LTL liveness formulas in Promela. However, the experiments demonstrate the complexity of the BGP convergence problem as Promela was not able to determine divergence of gadgets with less than 5 nodes. Our proposal is to reduce the liveness problem to a safety problem as suggested in the literature [2]. Model-checkers typically perform better when proving safety properties as one can exploit different types of reduction techniques. This enabled our model-checking to verify networks with hundreds of nodes.

Wang et al. [16] proposed another formalization in Maude. However, while the performance is adequate, it is not sound. In particular, it can generate false counter examples. This seems to occur in gadgets for which BGP may sometimes converge, but sometimes also diverge. This is because the definition of divergence is not precise enough. We propose, on the other hand, a definition for determining divergence that is sound and scalable.

Wang et al. [13, 14] have also proposed methods to improve the scalability of model-checking. In [14], the authors exploit the structure of network to reduce the verification problem. For example, they identify sub-nets that are duplicated which can be reduced to a single sub-net when checking for divergence. We take an alternative approach by exploiting the unsat core of the SMT solver checking the formalization of Metarouting conditions. In [13], Wang et al. encode the BGP convergence problem in Answer Set Programming. The approach scales well reaching some thousands of nodes. However, it is not capable of providing concrete examples for divergence.

This also illustrates another key difference to existing methods. We propose a verification flow that combines SMT solver and executable specification in Maude. This enables us to exploit the advantages enabled by each method, e.g., the use of SMT solvers to simplify the model-checking problem.

9 Conclusions

This paper revisits the problem of BGP convergence verification, aiming to develop a sound, scalable, and automated verification approach. The key insight is to combine different automated verification techniques. By encoding the Metarouting convergence criterion using an SMT solver, we not only automate the convergence check but also leverage the solver's ability to generate unsatisfiable cores. These cores highlight the critical links responsible for monotonicity failing to hold, which in turn helps reduce the complexity of the model-checking problem. We reduce the model-checking task to a reachability analysis of divergent points, based on a sound divergence criterion introduced in this paper.

There are several future directions. The proposed criterion is sound, but not shown to be complete. We are also considering how to express more realistic implementations of preference relations, e.g., using pseudo-code.

Acknowledgments

This research is funded in part by the Ashton Fellowship at the University of Pennsylvania, the GEM Fellowship, and NSF IIS-2436080.

References

[1] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[2] Armin Biere, Cyrille Artho, and Viktor Schuppan. 2002. Liveness Checking as Safety Checking. In *7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems, FMICS 2002, ICALP 2002 Satellite Workshop, Málaga, Spain, July 12-13, 2002 (Electronic Notes in Theoretical Computer Science, Vol. 66)*, Rance Cleaveland and Hubert Garavel (Eds.). Elsevier, 160–177. doi:10.1016/S1571-0661(04)80410-9

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All About Maude: A High-Performance Logical Framework*. LNCS, Vol. 4350. Springer.

[4] Matthew L. Daggitt and Timothy G. Griffin. 2024. Formally Verified Convergence of Policy-Rich DBF Routing Protocols. *IEEE/ACM Trans. Netw.* 32, 2 (2024), 1645–1660. doi:10.1109/TNET.2023.3326336

[5] Lixin Gao and Jennifer Rexford. 2000. Stable Internet routing without global coordination. *SIGMETRICS Perform. Eval. Rev.* 28, 1 (June 2000), 307–317. doi:10.1145/345063.339426

[6] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. 2002. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.* 10, 2 (April 2002), 232–243. doi:10.1109/90.993304

[7] Timothy G. Griffin and João Luís Sobrinho. 2005. Metarouting. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Philadelphia, Pennsylvania, USA) *(SIGCOMM '05)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/1080091.1080094

[8] Timothy G. Griffin and Gordon Wilfong. 1999. An analysis of BGP convergence properties. *SIGCOMM Comput. Commun. Rev.* 29, 4 (Aug. 1999), 277–288. doi:10.1145/316194.316231

[9] Ratul Mahajan, David Wetherall, and Tom Anderson. 2002. Understanding BGP misconfiguration. *SIGCOMM Comput. Commun. Rev.* 32, 4 (Aug. 2002), 3–16. doi:10.1145/964725.633027

[10] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. doi:10.17487/RFC4271

[11] Rubén Rubio. 2022. Maude as a Library: An Efficient All-Purpose Programming Interface. In *Rewriting Logic and Its Applications: 14th International Workshop, Revised Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 274–294. doi:10.1007/978-3-031-12441-9_14

[12] João Luis Sobrinho. 2003. Network routing with path vector protocols: theory and applications. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Karlsruhe, Germany) *(SIGCOMM '03)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/863955.863963

[13] Anduo Wang and Zhijia Chen. 2019. Internet Routing and Non-monotonic Reasoning. In *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11481)*, Marcello Balduccini, Yuliya Lierler, and Stefan Woltran (Eds.). Springer, 51–57. doi:10.1007/978-3-030-20528-7_5

[14] Anduo Wang, Alexander J. T. Gurney, Xianglong Han, Jinyan Cao, Boon Thau Loo, Carolyn L. Talcott, and Andre Scedrov. 2014. A reduction-based approach towards scaling up formal analysis of internet configurations. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*. IEEE, 637–645. doi:10.1109/INFOCOM.2014.6847989

[15] Anduo Wang, Carolyn Talcott, Alexander J. T. Gurney, Boon Thau Loo, and Andre Scedrov. 2012. Reduction-based formal analysis of BGP instances. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Tallinn, Estonia) *(TACAS'12)*. Springer-Verlag, Berlin, Heidelberg, 283–298. doi:10.1007/978-3-642-28756-5_20

[16] Anduo Wang, Carolyn Talcott, Limin Jia, Boon Thau Loo, and Andre Scedrov. 2011. Analyzing BGP instances in Maude. In *Proceedings of the Joint 13th IFIP WG 6.1 and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems* (Reykjavik, Iceland) *(FMOODS'11/FORTE'11)*. Springer-Verlag, Berlin, Heidelberg, 334–348.

[17] Ping Yin, Yinxue Ma, and Zhe Chen. 2014. Model Checking the Convergence Property of BGP Networks. *J. Softw.* 9, 6 (2014), 1619–1625. doi:10.4304/JSW.9.6.1619-1625