From Theory to Practice: Efficiently Checking BGP Configurations for Guaranteed Convergence

Luca Cittadini Massimo Rimondini Stefano Vissicchio Matteo Corea Giuseppe Di Battista
Dept. of Computer Science and Automation, Roma Tre University
{ratm,rimondin,vissicch,corea,gdb}@dia.uniroma3.it

***Abstract*—Internet Service Providers can enforce a fine-grained control of Interdomain Routing by cleverly configuring the Border Gateway Protocol. However, the price to pay for the flexibility of BGP is the lack of convergence guarantees. The literature on network protocol design introduced several sufficient conditions that routing policies should satisfy to guarantee convergence. However, a methodology to systematically check BGP policies for convergence is still missing.**

This paper presents two fundamental contributions. First, we describe a heuristic algorithm that statically checks BGP configurations for guaranteed routing convergence. Our algorithm has several highly desirable properties: *i*) it exceeds state-of-the-art algorithms by correctly reporting more configurations as stable, *ii*) it can be implemented efficiently enough to analyze Internet-scale configurations, *iii*) it is free from false positives, namely never reports a potentially oscillating configuration as stable, and *iv*) it can help spot troublesome points in a detected oscillation.

Second, we propose an architecture for a modular tool that exploits our algorithm to process native router configurations and report the presence of potential oscillations. Such a tool can effectively integrate syntactic checkers and assist operators in verifying configurations. We validate our approach using a prototype implementation and show that it scales well enough to enable Internet-scale convergence checks.

***Index Terms*—Algorithms, BGP, Network management, Routing convergence and stability.**

I. Introduction

Internet routing relies on the Border Gateway Protocol (BGP) [2]. While BGP was designed to support routing policies dictated by commercial agreements, research works (e.g., [3], [4]) have shown how this flexibility comes at the cost of convergence guarantees. Routing oscillations can and do [5], [6] happen due to BGP policy conflicts [7] or unexpected interaction with the underlying Interior Gateway Protocol [8]. Oscillations can severely impact network performance in terms of delay, jitter, and packet loss [9], [10]. Moreover, precious router resources are wasted in periodically reprocessing the same messages, and bandwidth is consumed by unnecessary routing updates travelling back and forth.

Started about one decade ago and still ongoing, lots of research efforts have addressed the routing convergence problem, that is, the problem of determining whether a given configuration is guaranteed to settle to a stable routing regardless of the protocol dynamics. However, to the best of our knowledge, a systematic methodology that exploits the results of this research to check generic routing policies for convergence is still missing, and operators are often forced to rely on “tweak and pray”.

Manuscript received April 30, 2010; revised January 22, 2011. The associate editor coordinating the review of this paper and approving it for publication was M. Brunner. This work is partially supported by MIUR PRIN project AlgoDEEP no. 2008TFBWL4, and by ESF project 10-EuroGIGA-OP-003 GraDR. A preliminary version of this paper appeared in [1].

In this paper we introduce a technique that enables operators to statically check routing configurations for guaranteed convergence. Our technique could thus be exploited to validate configurations and improve the quality of routing. A relevant use case consists in performing what-if analyses on the impact of policy changes on the global stability of the Internet (e.g., what if a large transit provider becomes a content distribution network?). Moreover, our experimental evaluation on realistic scenarios showed that in many cases our approach is efficient enough to also support online convergence checks, i.e., right after a policy change.

Our contribution is twofold. Firstly, we describe a heuristic algorithm that statically checks a BGP network for guaranteed routing stability. We prove that our algorithm has several highly desirable properties: *i*) it exceeds state of the art algorithms in that it is able to correctly report more configurations as stable, *ii*) it can be implemented efficiently enough to enable static analysis of Internet-scale BGP configurations, *iii*) it is free from false positives, meaning that configurations are only reported as stable if they do not admit potential oscillations, and *iv*) it can help in spotting the troublesome points in configurations that are not stable. Secondly, we propose an architecture for a modular tool that exploits our heuristic algorithm to process BGP configurations and return information about the potential presence of oscillations. The design of our tool is general enough to support checking the stability both of interdomain (eBGP) and of intradomain (iBGP) configurations. Observe that the lack of a central management, along with privacy concerns of the ISPs, can make it difficult to gather the routing policies required for a stability check of the Internet in its entirety. However, there exist contributions in the literature addressing the challenge of obtaining these policies (see, e.g., [11], [12]), and it is out of the scope of this paper to further investigate this problem. On the other hand, our approach can be equivalently applied to the case of iBGP configurations (see also [13]), which are usually managed in a centralized fashion and may be subject to oscillations as well [6], [8].

We validate the practical applicability of our architecture using a prototype implementation and show that both the translation of policies to an abstract model and the convergence check algorithm itself can be implemented efficiently enough to analyze Internet-scale BGP configurations.

The rest of the paper is organized as follows. The model we use to represent BGP and policy configurations is described in Section III. Our convergence check algorithm is presented in Section IV. We discuss the architecture of a convergence check tool, together with the optimization techniques that make it scalable, in Section V. A complete example of how the tool verifies a BGP configuration is analyzed in Section VI. Results of Internet-scale experiments performed using a prototype implementation are discussed in Section VII. Conclusions are drawn in Section VIII. For the sake of readability, we devote a separate Appendix to the complete proofs of the theorems.

II. Related Work

Routing convergence is renowned to be a fundamental problem in network routing [14], and still attracts significant research interest [15]. Deciding whether a BGP network is stable is known to be a computationally hard problem [4], [7], [16], and sufficient conditions [4], [8], [17] to guarantee stable routing have also been found. Based on these results, two research directions have been explored.

Several modifications to the BGP protocol have been proposed to dynamically detect and solve policy-induced oscillations (e.g., [18], [19]). Also, in [20] the authors introduce formal tools for the design of inherently stable protocols. However, there are serious difficulties in deploying substantial changes to BGP while guaranteeing service continuity. By contrast, in this paper we aim at checking whether a given BGP network is stable by simply analyzing static properties of its configuration, without considering protocol dynamics.

On the other hand, few techniques are available that address the convergence problem considering the current implementation of the protocol. An algorithm that performs convergence checks of iBGP configurations is presented in [21]. However, the algorithm assumes that the network under consideration contains only one layer of route reflectors, and cannot be easily extended to the case of eBGP configurations. Our approach is more general in both respects. Policy checkers (e.g., [22], [23]) typically execute syntactic checks and batch tests on BGP configurations. Our approach is complementary in that we explicitly focus on convergence, which also requires analyzing configuration semantics. We also overleap simulators [24], in that we are able to point out the converging portion of networks that could permanently oscillate. For these reasons, we believe our technique can effectively integrate existing checkers and visual analysis tools (e.g., [25]) to assist operators in verifying configurations.

III. A Model for Characterizing BGP Convergence

In order to study the convergence of BGP, we need a model that is able to represent generic BGP configuration policies and to analyze how their interaction influences the dynamics of the protocol. The following subsections address the matter of choosing such a model.

A. An Abstraction of BGP Policies

We choose to rely on the widely adopted formalism known as the *Stable Paths Problem* (SPP) model [4], [18]. There are several reasons for which we choose the SPP model. First of all, it is a vendor-independent, theoretically sound formalism which has been used in the literature to obtain a number of results about BGP convergence [4], [8], [17], [26]. Also, SPP can be used to model both inter-AS (eBGP) [4] and intra-AS (iBGP) [8], [26] routing dynamics, which makes it valuable to study oscillations within a single ISP. Actually, the SPP formalism is flexible enough to model virtually any policy-based path-vector routing protocol. By using an explicit ranking of the routing paths, the SPP model can represent BGP policies independently of the language constructions used to express them. This is supported by the fact that BGP is based on a fully deterministic decision process [2].

We now formally define an SPP instance.

Let $G = (V, E)$ be an undirected graph, with vertex set $V = \{0, 1, \ldots, n\}$, $n \geq 0$ and edge set E. The graph G encodes the AS-level topology of the network under consideration: vertices correspond to ASes and edges correspond to BGP peerings. Each vertex $v \in V$ attempts to establish a path to vertex 0, which represents the originator of a single network prefix.

A *path* P in G is a sequence of $k+1$ vertices $P = (v_k\ v_{k-1}\ \ldots\ v_1\ v_0)$, $v_i \in V$, $k \geq 0$, such that $(v_i, v_{i-1}) \in E$ for $0 < i \leq k$. Vertex v_{k-1} is the *next hop* of v_k in path P. We denote the empty path, representing the inability to reach 0, by ϵ. The *concatenation* of two nonempty paths $P = (v_k\ v_{k-1}\ \ldots\ v_i)$, $k \geq i$, and $Q = (v_i\ v_{i-1}\ \ldots\ v_0)$, $i \geq 0$, denoted as PQ, is the path $(v_k\ v_{k-1}\ \ldots\ v_i\ v_{i-1}\ \ldots\ v_0)$. Given that the empty path represents unreachability of 0, it makes no sense to concatenate it with any other paths, therefore we assume $P\epsilon = \epsilon P = \epsilon$.

Each vertex $v \in V$ is assigned a set of *permitted paths* $\mathcal{P}^v$. All the paths in $\mathcal{P}^v$ are simple (i.e., no repeated vertices), start from v and end in 0, and represent the paths that v can use to reach 0. The empty path is permitted at each vertex $v \neq 0$. Moreover, let $\mathcal{P}^0 = \{(0)\}$ and $\mathcal{P} = \bigcup_{v \in V} \mathcal{P}^v$.

For each vertex $v \in V$, a *ranking function* $\lambda^v : \mathcal{P}^v \to \mathbb{N}$ determines the relative level of preference $\lambda^v(P)$ assigned by v to path P. If $P_1, P_2 \in \mathcal{P}^v$ and $\lambda^v(P_2) < \lambda^v(P_1)$, then P_2 is *preferred* over P_1 Observe that we define the ranking based on a convention (i.e., the lower the better) that is opposite with respect to the one in [4]. Let $\Lambda = \{\lambda^v | v \in V\}$. Ranking functions in Λ describe the BGP routing policies.

According to the model in [4], the following conditions hold on the paths, for each vertex $v \in V - \{0\}$:

i) $\forall P \in \mathcal{P}^v, P \neq \epsilon$: $\lambda^v(P) < \lambda^v(\epsilon)$ (unreachability is the last resort);

ii) $\forall P_1, P_2 \in \mathcal{P}^v, P_1 \neq P_2 : \lambda^v(P_1) = \lambda^v(P_2) \Rightarrow P_1 = (v\ u)P_1', P_2 = (v\ u)P_2'$, (strict ranking is assumed on all the paths but those with the same next hop).

An instance of SPP is a triple $(G, \mathcal{P}, \Lambda)$.

A *path assignment* π is a function that maps each vertex $v \in V$ to a permitted path $\pi(v) \in \mathcal{P}^v$. This represents the fact that vertex v is using path $\pi(v)$ to reach 0. We have that $\pi(0) = (0)$ and, if $\pi(v) = \epsilon$, then v cannot reach vertex 0.

B. Stable Routing States

The SPP model effectively captures BGP policy definitions using a static formalism. However, in order to prove some properties that enable us to statically check for policy correctness, we also need to explicitly describe the operation of BGP. For this purpose, we rely on the widely used Simple Path Vector Protocol (SPVP) described in [18], which we now briefly recall. In SPVP, vertices asynchronously exchange messages containing paths to 0. We assume that edges introduce a finite delay on message delivery. Every time a vertex v receives a path P from a neighbor w, first of all v checks whether $(v\ w)P$ is permitted (i.e., $(v\ w)P \in \mathcal{P}^v$). If this is the case, v puts $(v\ w)P$ into a data structure $\text{rib-in}_t(v \Leftarrow w)$, which is used to store the latest path received from w. If $(v\ w)P$ is not permitted, v puts ϵ in $\text{rib-in}_t(v \Leftarrow w)$. Then, v checks whether the currently selected path, stored in $\text{rib}_{t-1}(v)$, is the currently available best path. If this is not the case, v selects the best ranked path among those in rib-in_t and stores it in $\text{rib}_t(v)$. We refer to such a path as $\text{best}_t(v) = \underset{u|(v,u)\in E}{\arg\min}\ \lambda^v((v)\text{rib-in}_t(v \Leftarrow u))$. Afterwards, v announces $\text{best}_t(v)$ to all its neighbors $u \mid (v,u) \in E$.

An *activation sequence* [18] is a (possibly infinite) sequence $\sigma = (A_0\ A_1\ \ldots A_i \ldots)$ where A_t is a set containing an ordered pair $(u,v)|(u,v) \in E$ for each vertex v that processes a message from u at time t. We say that pair (u,v) is *activated* at time t. Since the delay introduced by edges is finite, messages are eventually delivered. An activation sequence is *fair* if any ordered pair $(u,v) \mid (u,v) \in E$ is eventually activated whenever u sends a message to its neighbors. Since in real networks messages are unlikely to be indefinitely delayed, in the following we will be only considering fair activation sequences.

As SPVP operates within the network, the routing evolves through different path assignments π_t, where $\pi_t(u) = \text{rib}_t(u)$, until SPVP *converges* to a stable path assignment. In this situation every vertex has settled to its best available path and will never change its current selection. More formally, a path assignment π_t is *stable* if, for each $u \in V$, $\pi_t(u) = \text{best}_t(u)$. Once a stable path assignment is reached, no further messages are generated in the network.

Obviously, the existence of a stable path assignment is a crucial requirement for a network running BGP. For this reason, researchers have been addressing the problem of determining whether an SPP instance admits a stable path assignment [4], [7]. This problem has been proved to be NP-complete [4], and several interesting instances which do not admit any stable path assignment have been shown to exist [3], [4], [18]. Moreover, an instance may have multiple stable states [7], and, even if a stable state exists, there may still be activation sequences that give rise to oscillations [3]. From the point of view of a network operator, it is much more interesting to know whether a given BGP configuration is *guaranteed* to converge to a stable state, regardless of any possible message orderings. Several works [4], [17] have therefore considered whether SPVP is guaranteed to converge to a stable state for any fair activation sequence, also known as the *safety* problem. This problem has been proved to be CoNP-hard [16].

Given the intrinsic intractability of these crucial networking problems, a lot of research efforts have been directed to proving sufficient conditions that guarantee stability and safety [3], [4], [8], [17]. In the rest of the paper we show how to systematically exploit these sufficient conditions to perform efficient Internet-scale convergence checks.

IV. A New Greedy Algorithm

The inherent complexity of SPP pushed researchers to introduce heuristic algorithms that can efficiently verify the stability and safety of a BGP network.

In this section we first briefly recall a greedy algorithm (we call it GREEDY) that has been proposed in [4] to solve an SPP instance. Second, we propose a new greedy algorithm, called GREEDY$^+$, which we prove to exhibit highly desirable properties: *i*) it exceeds GREEDY in that it is able to correctly report more configurations as stable, *ii*) it can be implemented efficiently enough to enable static analysis of otherwise intractable Internet-scale BGP configurations, *iii*) it is free from false positives, meaning that configurations are only reported as stable if they are guaranteed to converge to a stable routing, and *iv*) it can help in spotting the troublesome points in a detected oscillation. Properties *i*) and *ii*) are unique to GREEDY$^+$. On the other hand, GREEDY satisfies properties *iii*) and *iv*), which are also inherited by GREEDY$^+$. This section also presents a comparison of GREEDY and GREEDY$^+$.

A. A Known GREEDY Algorithm

Algorithm GREEDY proceeds by iteratively growing a stable path assignment. If the algorithm terminates successfully, the path assignment defines a spanning tree that is a solution for the given SPP instance. Otherwise, GREEDY is only able to identify a stable path assignment for a subset of the vertices.

The algorithm maintains a *stable set* of vertices for which convergence is guaranteed. The stable set at iteration i of the algorithm is denoted by V_i. Vertex 0 is always in the stable set, i.e., $V_0 = \{0\}$. As the stable set grows, a path assignment π defined on the vertices in V_i is iteratively built.

We say that a path P is *compatible with a path assignment* π at iteration *i* if $P = P'(u\ v)\pi(v)$, where P' does not contain vertices in V_i, $(u,v) \in E$, and $v \in V_i$.

Algorithm GREEDY is as follows. At iteration $i > 0$, let P_v be the path with minimum $\lambda^v(P)$ among the paths at v compatible with π. If such a path does not exist, let $P_v = \epsilon$. If there exists a vertex $v \notin V_{i-1}$ such that P_v has a next hop in V_{i-1}, then construct V_i by adding v to V_{i-1} and set $\pi(v) = P_v$. If such a vertex v does not exist, then stop.

Intuitively, at each iteration vertex v is stabilized because its best compatible path directly reaches an already stabilized vertex. The algorithm terminates after at most $|V|$ iterations. A solution to the SPP instance exists if, after k iterations, GREEDY ends with $V_k = V$. The *solution* is given by the stable path assignment π.

Note that the description of GREEDY we propose here slightly differs from the one in [4], in that only a single vertex enters the stable set at each iteration. We explain in the following that this modified version is indeed equivalent to the

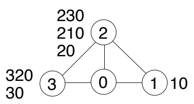


Fig. 1. DI-SAFE-GREE: An instance of SPP for which algorithm GREEDY fails to find a solution.

original algorithm. We choose to apply this slight modification in order to better introduce the improvements that allow us to overcome some shortcomings of the original GREEDY.

GREEDY can fail to find a solution even if the SPP instance under consideration is guaranteed to converge. Consider, for example, the instance DI-SAFE-GREE in Fig. 1. Permitted paths are listed next to each vertex in decreasing order of preference. It can be easily verified that any fair activation sequence of SPVP on this instance is finite. In fact, in any fair activation sequence vertices 1, 2, and 3 learn about the direct path to 0. After that, pair $(1, 2)$ is eventually activated, and 2 learns about $(2\ 1\ 0)$. Henceforth, vertex 2 will permanently be unable to select $(2\ 0)$, in turn preventing vertex 3 from choosing $(3\ 2\ 0)$. Finally, after pair $(3, 2)$ is activated, 2 switches to its best path $(2\ 3\ 0)$ and SPVP terminates, as no other message is further generated. Therefore any fair activation sequence is forcedly finite, and SPVP cannot oscillate on this instance.

We now walk through the execution of GREEDY on DI-SAFE-GREE. At the first iteration, vertex 1 enters the stable set V_1, and $\pi(1) = (1\ 0)$. At the second iteration, the algorithm forcedly stops. In fact, path $(2\ 3\ 0)$ is compatible with π because $2, 3 \notin V_1$, $0 \in V_1$, and $(3, 0) \in E$. However, even if $(2\ 3\ 0)$ is the best compatible path at vertex 2, its next hop is not in V_1. A similar argument applies to path $(3\ 2\ 0)$. Therefore, no new vertex can be added to the stable set and the algorithm stops without finding a solution, since $V_1 \neq V$.

B. *Improving* GREEDY*: the* GREEDY$^+$ *Algorithm*

We now describe the GREEDY$^+$ algorithm, which is able to successfully solve more instances than GREEDY, among which DI-SAFE-GREE, and can be implemented efficiently enough to check the convergence of Internet-scale SPP instances.

Given an SPP instance $(G = (V, E), \mathcal{P}, \Lambda)$, we say that a path P belonging to a set $\mathcal{S}$ of paths is *consistent with* $\mathcal{S}$ if either $P = \epsilon$, $P = (0)$, or $P = (v\ u)P'$ where $(v, u) \in E$, $P' \in \mathcal{S}$, and P' is consistent with $\mathcal{S}$. For example, let $V = \{0, 1, 2, 3\}$, $E = \{(1, 0), (2, 1)\}$, and $\mathcal{S} = \{(0), (1\ 0), (2\ 1\ 3\ 0)\}$: it is easy to check that (0) and $(1\ 0)$ are consistent with $\mathcal{S}$, since $(1, 0) \in E$. On the other hand, $(2\ 1\ 3\ 0)$ is not consistent with $\mathcal{S}$. In fact, even if $(2, 1) \in E$, the subpath $(1\ 3\ 0)$ is not in $\mathcal{S}$ and cannot therefore be consistent with $\mathcal{S}$. Further, for each vertex v we define a set $\bar{\mathcal{P}}^v$ of paths called *useful set*. The useful set $\bar{\mathcal{P}}^v$ is initialized with the paths in $\mathcal{P}^v$ that are consistent with $\mathcal{P}$. Let $\bar{\mathcal{P}} = \bigcup_{v \in V} \bar{\mathcal{P}}^v$.

What follows is a description of GREEDY$^+$. Let $V_0 = \{0\}$. At iteration $i > 0$, GREEDY$^+$ performs the following steps:

i) Prune all those paths that cannot be selected because a better ranked path is offered by a neighbor in the stable set. For each vertex $v \in V - V_{i-1}$ such that v has a neighbor $u \in V_{i-1}$ and there exists a path $P = (v\ u)P'$ such that $\{P'\} = \bar{\mathcal{P}}^u$, remove from $\bar{\mathcal{P}}^v$ all the paths Q such that $\lambda^v(Q) > \lambda^v(P)$. Intuitively, this step is performed because P will be always available at v.

ii) Enforce consistency. For each vertex $v \notin V_{i-1}$, remove from $\bar{\mathcal{P}}^v$ all the paths that are not consistent with $\bar{\mathcal{P}}$.

iii) Grow the stable set or stop. Let $C_i \subseteq V - V_{i-1}$ be the set of *candidate* vertices v such that the path $P \in \bar{\mathcal{P}}^v$ with minimum $\lambda^v(P)$ either has a next hop in V_{i-1}, or $P = \epsilon$. If $C_i = \oslash$, then set $V_i = V_{i-1}$ and stop. Otherwise, if $C_i \neq \oslash$, then pick an arbitrary vertex $u \in C_i$, construct V_i by adding u to V_{i-1}, and set $\bar{\mathcal{P}}^u = \{P\}$.

i	V_i	C_i	$\bar{\mathcal{P}}^1$	$\bar{\mathcal{P}}^2$	$\bar{\mathcal{P}}^3$
0	$\{0\}$	$-$	$\{(1\ 0)\}$	$\{(2\,3\,0), (2\,1\,0), (2\ 0)\}$	$\{(3\,2\,0), (3\,0)\}$
1	$\{0, 1\}$	$\{1\}$	$\{(1\ 0)\}$	$\{(2\,3\,0), (2\,1\,0), (2\ 0)\}$	$\{(3\,2\,0), (3\,0)\}$
2	$\{0, 1, 3\}$	$\{3\}$	$\{(1\ 0)\}$	$\{(2\,3\,0), (2\,1\,0)\}$	$\{(3\ 0)\}$
3	V	$\{2\}$	$\{(1\ 0)\}$	$\{(2\ 3\ 0)\}$	$\{(3\ 0)\}$
4	V	$\oslash$	$\{(1\ 0)\}$	$\{(2\ 3\ 0)\}$	$\{(3\ 0)\}$

TABLE I
A SUCCESSFUL EXECUTION OF GREEDY$^+$ ON DI-SAFE-GREE (FIG. 1). THE TABLE SHOWS SETS $V_i, C_i, \bar{\mathcal{P}}^v$ AT ITERATION i OF GREEDY$^+$. THE ROW FOR $i = 0$ CORRESPONDS TO THE INITIALIZATION STEP.

If GREEDY$^+$ stops after k iterations, its *output* consists of a stable set V_k and sets $\bar{\mathcal{P}}^v\ \forall v \in V$, with $|\bar{\mathcal{P}}^v| = 1\ \forall v \in V_k$. If $V_k = V$, GREEDY$^+$ computes a stable path assignment π for the input instance such that $\bar{\mathcal{P}}^v = \{\pi(v)\}\ \forall v \in V$. Otherwise, GREEDY$^+$ returns a *partial path assignment* on V_k, namely a path assignment π such that, for every $v \in V_k$, $\bar{\mathcal{P}}^v = \{\pi(v)\}$ and every vertex in $\pi(v)$ is in V_k. Also, $\pi(v)$ is undefined for vertices $v \notin V_k$. This partial path assignment is stable in the sense that every vertex $v \in V_k$ is guaranteed to eventually select the path in $\bar{\mathcal{P}}^v$.

GREEDY$^+$ differs from GREEDY because it exploits the useful set to prune those paths that, starting from a certain iteration, become permanently unavailable. This operation is encoded in Step *i)* of GREEDY$^+$. Indeed, if we skip this step, the set $\bar{\mathcal{P}}$ is only used to filter out inconsistent paths, and at each iteration j both the algorithms select the best path among the consistent ones having a next hop in V_j.

An example of a successful execution of GREEDY$^+$ on DI-SAFE-GREE is shown in Table I. Note that at iteration 2 path $(2\ 0)$ is evicted from $\bar{\mathcal{P}}^2$ because $(2\ 1\ 0)$ is preferred and permanently available (Step *i)*). This action puts in evidence the difference between GREEDY$^+$ and GREEDY: in fact, recall that GREEDY would have stopped at iteration 1. Step *ii)* then removes $(3\ 2\ 0)$ from $\bar{\mathcal{P}}^3$ since it is inconsistent with $\bar{\mathcal{P}}$. This allows vertex 3 to enter the stable set.

We now briefly describe some relevant properties of the GREEDY$^+$ algorithm. Formal proofs for these properties can be found in Appendix A.

Property 4.1: Let $n = |\bar{\mathcal{P}}|$ be the size of an SPP instance S. GREEDY$^+$ can be implemented to terminate on S in time that is polynomial in n.

According to this property, algorithm GREEDY$^+$ can be efficiently implemented, and is therefore suitable to be adopted in a tool that checks BGP configurations for routing stability.

Property 4.2: Consider a set C_j of vertices satisfying the criteria of Step *iii*) at an arbitrary iteration j of GREEDY$^+$. The output of GREEDY$^+$ does not change, regardless of the choice of vertex $v \in C_j$ performed at iteration j.

Property 4.2 asserts that GREEDY$^+$ is deterministic, namely at any time when multiple choices are possible, performing any of them does not alter the output. Given the similarities between GREEDY$^+$ and GREEDY, it is easy to check that the same property also holds for GREEDY. Moreover, this property confirms that the description of GREEDY given in this section and the original description given in [4] are equivalent.

Property 4.3: Consider an SPP instance S and run GREEDY$^+$ on S. Let $P \in \mathcal{P}^v$ be a path that GREEDY$^+$ deletes at iteration j. Then, for any fair activation sequence σ of SPVP on S, there exists a time t' such that $\forall t > t'$, $\pi_t(v) \neq P$.

Property 4.4: If GREEDY$^+$ terminates successfully on an instance S of SPP, then S is safe and has a unique solution.

Properties 4.3 and 4.4 actually state that GREEDY$^+$ can be used as a static, centralized, and deterministic algorithm to efficiently emulate the behavior of SPVP in the long term, thus dealing with the non-determinism that SPVP features. In our opinion, this feature of GREEDY$^+$ can be effectively exploited, e.g., by a network administrator that wants to analyze how BGP will behave in his own network. Using GREEDY$^+$ also brings another important advantage to network operators. By Property 4.3, every vertex that GREEDY$^+$ puts in the stable set always selects the same path in any fair activation sequence of SPVP. Therefore, if GREEDY$^+$ terminates with $V_k \neq V$, the source of potential oscillations must be searched in those vertices that are left out of the stable set.

Property 4.5: The set of SPP instances that GREEDY$^+$ can successfully solve is strictly larger than the set of instances that GREEDY is able to solve.

According to this property, GREEDY$^+$ exceeds existing algorithms that can be used for stability check. In fact, we have already shown that GREEDY$^+$ can solve instances for which state-of-the-art sufficient conditions for safety do not hold (instance DI-SAFE-GREE in Fig. 1 is among these). We complete the proof in the Appendix by showing that GREEDY$^+$ also solves all the instances solved by GREEDY. Note that GREEDY$^+$ is not able to solve all the instances where GREEDY fails to find a solution. Because of the computational complexity of the safety problem (see Section III), no efficient algorithm can be complete and correct at the same time. However, given the particular nature of the problem, we stress that the need to avoid false positives (i.e., networks that are mistakenly reported as safe) outweighs the risk of allowing false negatives (i.e., networks that are mistakenly reported as potentially unsafe). Characterizing the set of additional instances that GREEDY$^+$ can solve with respect to GREEDY is still an open problem. However, in Section VII we empirically assess the effectiveness of GREEDY$^+$ compared to GREEDY by using a quantitative analysis on Internet-scale BGP topologies.

V. AN AUTOMATED CONVERGENCE CHECKER

GREEDY$^+$ exhibits several desirable properties. We now show that the benefits of these properties exceed mere theoretical speculation. We propose a design of a modular tool to automatically check whether a given set of BGP configuration policies leads to guaranteed convergence, i.e., safety.

The architecture of our tool is designed to support checking arbitrary BGP configurations, regardless of whether they come from routers within a single AS (iBGP) or from different ASes (eBGP). Therefore, the description of the tool presented in this section is independent of whether we are considering eBGP or iBGP.

A. Architecture of the Checker

An architectural overview of the tool we realized is shown in Fig. 2, where sharp boxes represent data, and rounded boxes represent the main architectural components. The checker processes input data along a pipeline from BGP configurations to the final response, which tells the user whether the system is safe or not. In the latter case, the tool returns a portion of the network that can be responsible for potential oscillations. The architectural components are as follows:

- A Configuration parser, which parses input BGP configuration policies and translates them into a custom format. Depending on the origin of policy information, the input can be router configuration files, RPSL [27] policy descriptions, or topologies obtained by monitoring projects such as CAIDA [28]. This allows us to support and integrate different sources of policy information by just changing the parser component.
- An SPP generator, which takes as input the BGP policies extracted by the parser and builds an SPP instance that models the configurations.
- A Stability checker, which implements the GREEDY$^+$ algorithm and runs it on the generated SPP instance in order to check it for safety.

The SPP generator is the most challenging component to design. In fact, enumerating the permitted paths of an SPP instance may take exponential space, since the number of paths in a graph is exponential with respect to the number of vertices.

The first step performed by the checker consists in the translation of the input BGP configuration policies into a custom input-independent format. Especially when the input policies are described using vendor-specific languages, isolating this step has some important benefits: *i*) even though configuration languages continuously evolve and different vendors propose proprietary constructions, the only component of the checker that needs to be updated to accommodate these changes is the parser; and *ii*) while most configuration languages are designed with an event-condition-action approach in mind, where specific actions are undertaken whenever a particular event takes place and a set of conditions is found to hold, by using the SPP model we rely on an explicit set of ordered paths, which allows us to disclose the network-wide semantic that is hidden behind the policy configurations.

Next, an intermediate vendor-independent representation of the input BGP configurations is built. Consider that, despite the variety of routing policies contained in well-known data sets (e.g., the Internet Routing Registries) and the number of constructions supported by router vendors, it is easy to see that a full-blown configuration can be ultimately mapped to a

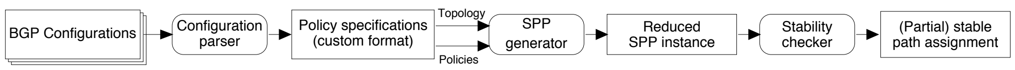


Fig. 2. Architecture of the automated convergence checker.

```
algorithm dissemination($v$)
1:  while receive $(P, \mathcal{A})$ from $w$ do
2:      $(P', \mathcal{A}') = F_{v \Leftarrow w}((P, \mathcal{A}))$
3:      if $(P', \mathcal{A}') \neq (\epsilon, \oslash)$ then
4:          $\mathcal{R}_t(v) := \mathcal{R}_{t-1}(v) \cup \{(P', \mathcal{A}')\}$
5:          if $\mathcal{R}_t(v) \neq \mathcal{R}_{t-1}(v)$ then
6:              for all $u \mid (u, v) \in E$ do
7:                  send $F_{v \Rightarrow u}(((v)P', \mathcal{A}'))$ to $u$
8:              end for
9:          end if
10:     end if
11: end while
```

Fig. 3. The dissemination algorithm used by the checker to generate the paths of the SPP instance.

set of filters. We represent a BGP *announcement* with a pair $(P, \mathcal{A})$, where P is a path and $\mathcal{A}$ is a set of BGP attributes. Before a received announcement is processed by a router u, an *import filter* $F_{u \Leftarrow v}((P, \mathcal{A}))$ is applied to the announcement; similarly, before a router u sends an announcement, an *export filter* $F_{u \Rightarrow v}((P, \mathcal{A}))$ is applied. The specification of a filter contains a predicate and a sequence of actions. The predicate is a boolean condition which can match BGP announcements based on the path and the other attributes they carry. If the predicate evaluates to true, the actions are undertaken. Possible actions include further propagating the announcement or dropping it, as well as altering, adding, or dropping the attributes carried by the announcement itself. The application of a filter returns a BGP announcement with the pertinent attribute modifications applied, or $(\epsilon, \oslash)$ if the BGP announcement is discarded.

While we extract filters from BGP configurations, we also collect information about the peerings established by each router. This allows us to build a graph $G = (V, E)$ representing the AS-level network topology.

Similarly to [22], where the processing steps of BGP are sequenced into a dissemination phase, a filtering phase, and a ranking phase, we distinguish the generation of routing paths from the actual best route selection operated by BGP.

For the dissemination, vertex 0 first starts announcing $((0), \oslash)$. We then run on the other vertices of G the distributed algorithm in Fig. 3. In this algorithm, vertices $v \in G$ exchange routing messages containing the full set of attributes (including, e.g., `as_path`, `next_hop`, `community`, etc.), apply all the configured filters, and store received announcements in sets $\mathcal{R}(v)$. Every time a new, not previously observed announcement is received by v, it is propagated over to v's neighbors. The purpose of this step is to enumerate all possible paths that comply with the import and export filters (we recall that an explicit representation of the paths is required by the SPP model). It is easy to verify that the algorithm in Fig. 3 ends in finite time, and a centralized implementation can be easily realized. A set of permitted paths $\mathcal{P}^v$ for each $v \in V$ can then be constructed starting from sets $\mathcal{R}$.

To compute path rankings, for each $v \in V$ we apply the BGP decision process to the announcements that v has collected in $\mathcal{R}(v)$ during the dissemination, and we define the ranking functions λ^v.

After executing the above steps, we define an SPP instance $S = (G, \mathcal{P}, \Lambda)$ with $\mathcal{P} = \bigcup_{v \in V} \mathcal{P}^v$ and $\Lambda = \{\lambda^v \mid v \in V\}$. We use instance S to study the convergence of the network.

B. Optimizing for Scalability

In principle, mapping vendor specific configurations to a set of explicitly permitted paths is a step that requires exponential space. On the other hand, hardcoding filter applications in the path generation process allows us to avoid generating a large number of paths. However, this is still not enough to be able to efficiently process Internet-scale configurations: we need to further reduce the paths to be generated. For this reason, during the dissemination phase and before actually generating the SPP instance, we run two pre-processing steps.

First of all, vertex 0 starts marking the path announcements it sends as *reliable*. If a vertex v receives a reliable announcement $(P, \mathcal{A})$ from a neighbor u, v applies the import filter $F_{v \Leftarrow u}((P, \mathcal{A}))$ and compares the resulting $(P', \mathcal{A}')$ with the best announcement that it could ever receive from its neighbors. If, and only if, v considers $(P', \mathcal{A}')$ as most preferred, v marks the announcement as reliable and stops considering future incoming announcements (*early stabilization*). In any case, v then applies the export filter $F_{v \Rightarrow w}((P', \mathcal{A}'))$ and further propagates the announcement to each neighbor $w \neq u$. Non-reliable paths continue to be disseminated in the standard way. This step corresponds to precomputing a subset of the stable vertices computed by GREEDY. Based on Property 4.3, a vertex v marking an announcement $(P, \mathcal{A})$ as reliable is guaranteed to select the corresponding path P. This allows us to only generate a single path for each stabilized vertex. In order to maximize the number of early stabilized vertices, we evaluate preferences based on the `local_pref`, the `as_path` length, and the `router-id` of the announcements.

Our experiments showed that early stabilization is not enough to make Internet-scale configurations tractable. Therefore, we apply an additional optimization step while generating the SPP instance: vertex v does not propagate any announcement that it considers worse than a received reliable announcement $(P, \mathcal{A})$ (*early suppression*). In fact, since paths from reliable announcements are always available, v will be unable to select an alternative path ranked worse than $(P, \mathcal{A})$. This step implements the optimizations found in GREEDY$^+$.

In order to finally generate the SPP instance, path rankings are computed by running the complete BGP decision process.

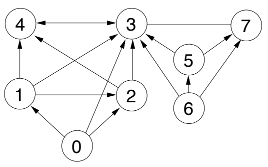


Fig. 4. The example we discuss to show the operation of the convergence checker is referred to this topology. Edges are labelled according to the commercial relationships between the Autonomous Systems (customer—▶provider, peer—peer, or sibling◀—▶sibling).

As a last step, the SPP instance is checked for guaranteed stability, i.e., safety. Observe that an efficient implementation of this step can keep track of the consistent paths in order to avoid recomputing them at each iteration.

VI. Walking through the Operation of the Convergence Checker

In this section we walk through a complete example showing how a BGP configuration is verified by our tool. Collection and parsing of the BGP configuration are omitted.

Throughout the example, we always refer to a single network prefix originated by the AS modeled as vertex 0.

A. BGP Configuration

Fig. 4 shows the BGP topology we consider in our example. The topology consists of 7 ASes, plus the prefix originator 0. To make the example more realistic, we labeled adjacencies between ASes with commercial relationships (customer—▶provider, peer—peer, or sibling◀—▶sibling) that are supposed to be honored in the configurations (see [29]–[31]). For convenience, vertices in Fig. 4 are laid out according to the customer-provider hierarchy.

We assume that all the routers in each AS implement homogeneous routing policies, so that a single BGP configuration describes the routing behavior of the whole AS. In particular, ASes in Fig. 4 implement the policies described in Fig. 5 (observe that the figure contains the policy specifications for all the 7 ASes). In order to provide a specification that is easier to read and independent of a vendor's specificities, we describe the policies using the Routing Policy Specification Language (RPSL [27]).

In Fig. 5, lines 1 through 18 define so-called `as-set` *objects*, which are groups of ASes that will be reused later on in the policy specifications. The policies adopted by each AS are described in the `aut-num` *objects* in lines 19 through 59. Each `aut-num` object states the policies applied to incoming (`import`) and outgoing (`export`) announcements separately. For example, AS0 (lines 19-22) only originates a prefix and has therefore no import policies specified. On the other hand, its export policies correctly state that it must "`announce AS0`" to each of its neighbors AS1, AS2, and AS3. Note that the exact semantic of "`announce AS0`" is "announce all the prefixes originated by AS0", which is what we would expect.

Policies may be more complex than AS0's. For example, AS1 (lines 23-26) applies different preference values to announcements that come from its providers (note the practical

```
1  as-set:  AS1:PROVIDERS
2  members: AS2, AS3, AS4
3  as-set:  AS2:CUSTOMERS
4  members: AS0, AS1
5  as-set:  AS2:PROVIDERS
6  members: AS3, AS4
7  as-set:  AS3:NEIGHBORS
8  members: AS1, AS2, AS4, AS5, AS6, AS7
9  as-set:  AS3:RESTRICTED
10 members: AS5, AS6, AS7
11 as-set:  AS4:NEIGHBORS
12 members: AS1, AS2, AS3
13 as-set:  AS5:NEIGHBORS
14 members: AS3, AS6, AS7
15 as-set:  AS6:NEIGHBORS
16 members: AS3, AS5, AS7
17 as-set:  AS7:NEIGHBORS
18 members: AS3, AS5, AS6
19 aut-num: AS0
20 export: to AS1 announce AS0
21 export: to AS2 announce AS0
22 export: to AS3 announce AS0
23 aut-num: AS1
24 import: from AS0 action pref=50; accept ANY
25 import: from AS1:PROVIDERS action pref=100; accept ANY
26 export: to AS1:PROVIDERS announce AS0
27 aut-num: AS2
28 import: from AS2:CUSTOMERS action pref=50; accept ANY
29 import: from AS2:PROVIDERS action pref=100; accept ANY
30 export: to AS2:PROVIDERS announce <^AS2:CUSTOMERS>
31 export: to AS1 announce ANY
32 aut-num: AS3
33 import: from AS3:NEIGHBORS action pref=50;
34         accept community.contains(4:50)
35 import: from AS3:NEIGHBORS action pref=100;
36         accept NOT community.contains(4:50)
37 export: to AS3:RESTRICTED announce ANY AND
38         NOT <^[AS0 AS4]>
39 export: to AS3:NEIGHBORS announce ANY
40 aut-num: AS4
41 import: from AS3 action pref=50; accept ANY
42 import: from AS2 action pref=100; accept ANY
43 import: from AS1 action pref=150; accept ANY
44 export: to AS3 action community.append(4:50);
45         announce ANY
46 export: to AS4:NEIGHBORS announce ANY
47 aut-num: AS5
48 import: from AS6 action pref=50; accept ANY
49 import: from AS5:NEIGHBORS action pref=100; accept ANY
50 export: to AS5:NEIGHBORS announce ANY
51 aut-num: AS6
52 import: from AS7 action pref=50; accept ANY
53 import: from AS6:NEIGHBORS action pref=100; accept ANY
54 export: to AS6:NEIGHBORS announce ANY
55 aut-num: AS7
56 import: from AS5 action pref=50; accept ANY
57 import: from AS7:NEIGHBORS action pref=100; accept ANY
58         AND NOT <AS6>
59 export: to AS7:NEIGHBORS announce ANY
```

Fig. 5. ASes in Fig. 4 are assumed to implement the policies described in this fragment of BGP configuration. The fragment is described using RPSL.

use of the `as-set` "`AS1:PROVIDERS`") or directly from AS0. Since in RPSL lower `pref` values correspond to higher preferences, AS1's policies in Fig. 5 implement the prefer-customer rule (see [30]) according to the topology in Fig. 4. The statement "`accept ANY`" implies that no filter is being applied on incoming announcements. Most of the policies adopted at other ASes implement the same prefer-customer rule, with a few exceptions. AS3 prefers announcements coming from its sibling AS4 (lines 33-36) because they are marked by AS4 itself with a specific community value (lines 44-45).

AS4 assigns an arbitrary ranking to its neighbors (lines 41-43). ASes 5 and 7 always prefer routes through customers (lines 48 and 56), while AS6 arbitrarily prefers one of its providers (line 52).

There are some policies that seem to have ambiguous neighbor specifications: for example, the `as-set` "`AS3:NEIGHBORS`" is used in both the policies at lines 33 and 35; "`AS3:RESTRICTED`" and "`AS3:NEIGHBORS`" have some ASes in common (lines 37 and 39); similarly for `AS6` and "`AS5:NEIGHBORS`" (lines 48 and 49), `AS7` and "`AS6:NEIGHBORS`" (lines 52 and 53), `AS5` and "`AS7:NEIGHBORS`" (lines 56 and 57). In all these cases, the specification-order rule of RPSL [27] applies to disambiguate the specification. For example, if AS3 receives from a neighboring AS an announcement containing a community `4:50`, it only applies the first matching import policy (lines 33-34), despite the existence of other policies that address the same set of neighbors.

Observe that all the policies in Fig. 5 but the one implemented by AS 7 are per-neighbor, meaning that each AS applies the same filter to all the announcements coming from a certain neighbor. Also, export policies mostly implement the selective export rules in [29]: AS1 exports to its providers only the direct route to AS0 (line 26); AS2 exports everything to its customer AS1 (line 31) and only its customer routes to its providers (line 30 – see [27] for the syntax of regular expressions on AS-paths); AS3, AS4, and AS7 may avoid applying any filters, but AS3 and AS7 still do: AS3, with respect to its neighbors AS5, AS6, and AS7, filters out all the announcements that come from AS0 or AS4 (lines 37-38); AS7, with respect to its neighbors AS3 and AS6, filters out all the announcements that have traversed AS6 (lines 57-58). Notable exceptions to the selective export rule are AS5 and AS6, since they allow routes received from a provider to be propagated to another provider.

All the violations to standard customer-provider policies discussed above have been introduced in order to build oscillatory structures in the topology of Fig. 4. In particular, ASes 3 and 4 mutually prefer each other, thus building a DISAGREE [4], while ASes 5, 6, 7 each prefer their counter-clockwise neighbor, thus forming a BAD-GADGET [4].

B. Policy Specifications

Following the architecture in Fig. 2, the next step performed by the checker is to parse the configuration in Fig. 5 and construct an internal representation that is independent of the BGP configuration given as input. Fig. 6 shows a possible representation of the policies in Fig. 5 in terms of the import and export filters described in Section V-A. In Fig. 6 we first define two handy functions for manipulating BGP attributes: $\mathbf{set}((P,\mathcal{A}),\mathtt{a},\mathtt{val})$ modifies the announcement $(P,\mathcal{A})$ by setting attribute `a` to value `val`; $\mathbf{append}((P,\mathcal{A}),\mathtt{a},\mathtt{val})$ alters the variable length attribute `a` by appending `val` at its end (for convenience, we handle `a` as a set of values).

Each filter has the following structure: upon matching a predicate on the attributes of a BGP announcement $(P,\mathcal{A})$ (left side of the $\Rightarrow$), a sequence of actions is performed and a new announcement is returned (right side of the $\Rightarrow$). In Fig. 6, all the sequences consist of a single action, and the kind of action has been annotated in italics on the right margin. Observe that `local_pref` values have been swapped because a lower `pref` in RPSL corresponds to a higher `local_pref` in BGP. If a filter returns $(\epsilon,\oslash)$, the announcement is simply dropped.

C. Generated (reduced) SPP Instance

Once the import and export filters are known, an SPP instance that models their effect can be easily built. First of all, the filters need to be analyzed in order to extract the topology of the network under consideration. Since each filter refers to a specific neighbor, we can simply enumerate the neighbors of each vertex to accomplish this task.

After that, the filters are passed to the dissemination algorithm in Fig. 3 in order to generate the paths of the SPP instance. Barely running the dissemination on the topology and filters collected from the configuration in Fig. 6 would result in the SPP instance in Fig. 7(a). Despite filter applications being hardcoded in the dissemination process, it is evident that an exponential number of paths still needs to be generated, leading to an overly large instance even in the simple example we are considering.

Fig. 7(b) shows the effectiveness of our dissemination-time optimizations. In particular, early stabilization allows to avoid generating and disseminating all the dimmed paths in Fig. 7(b). For example, at vertices 1 and 2 the direct path to 0 is the overall most preferred, therefore the announcements of $(1\ 0)$ and $(2\ 0)$ are marked as reliable and no other paths are generated at 1 and 2, which are considered stable vertices. As a consequence, none of the paths that would be obtained by the dissemination of the worse path $(2\ 1\ 0)$ are available any longer. In this example there are no other vertices that can be early stabilized, yet the gain in terms of generated paths is already noticeable.

On the other hand, early suppression prevents generating all the struck out paths in Fig. 7(b). For example, vertex 4 will be prevented from disseminating any paths that are worse than $(4\ 2\ 0)$, because this path extends the reliable path $(2\ 0)$. As a consequence, path $(3\ 4\ 1\ 0)$ will no longer be generated at 3. In a similar way, 3 will avoid disseminating paths that are worse than $(3\ 0)$, which extends the reliable path (0). As a consequence, all the paths that extend $(3\ 2\ 0)$ and $(3\ 1\ 0)$ are not generated. Note that, after applying the two optimization steps, only the empty path ϵ is left at vertices 5, 6, and 7. The final SPP instance resulting after applying the optimizations is in Fig. 7(c).

D. (Partial) Stable Path Assignment

The final step of our convergence checker consists in running on the generated SPP instance the GREEDY$^+$ algorithm described in Section IV-B. Table II shows the steps of execution of GREEDY$^+$ on the instance in Fig. 7(c). At step $i=1$ the candidate vertex 1 enters the stable set because its best path $(1\ 0)$ uses the stable vertex 0 as next hop. At the same time, paths $(3\ 2\ 0)$ and $(3\ 1\ 0)$ are removed from the useful paths at 3 because they are worse than $(3\ 0)$, which will

$\mathbf{set}((P,\mathcal{A}),\mathtt{a},\mathtt{val}) = (P,\mathcal{A}') \mid$
$\quad \mathcal{A}'[a'] = \mathcal{A}[a'], \forall a' \neq \mathtt{a} \ \wedge \ \mathcal{A}'[\mathtt{a}] = \mathtt{val}$
$\mathbf{append}((P,\mathcal{A}),\mathtt{a},\mathtt{val}) = (P,\mathcal{A}') \mid$
$\quad \mathcal{A}'[a'] = \mathcal{A}[a'], \forall a' \neq \mathtt{a} \ \wedge \ \mathcal{A}'[\mathtt{a}] = \mathcal{A}'[\mathtt{a}] \ \cup \{\mathtt{val}\}$

$F_{0\Leftarrow v, v\in\{1,2,3\}}(\cdot) : \mathit{true} \Rightarrow (\epsilon, \emptyset)$ (*drop*)
$F_{0\Rightarrow v, v\in\{1,2,3\}}(\cdot) : \mathit{true} \Rightarrow ((0), \emptyset)$ (*announce*)

$F_{1\Leftarrow 0}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 100)$ (*modify*)
$F_{1\Leftarrow v, v\in\{2,3,4\}}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 50)$ (*modify*)
$F_{1\Rightarrow v, v\in\{2,3,4\}}(((1)P,\mathcal{A})) : \mathit{true} \Rightarrow ((1)P,\mathcal{A})$ (*announce*)

$F_{2\Leftarrow v, v\in\{0,1\}}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 100)$ (*modify*)
$F_{2\Leftarrow v, v\in\{3,4\}}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 50)$ (*modify*)
$F_{2\Rightarrow v, v\in\{3,4\}}(((2)P,\mathcal{A}))$:
$\quad \mathcal{A}[\mathtt{as\_path}] = (v\ \ldots), v \in \{0,1\} \Rightarrow ((2)P,\mathcal{A})$
(*announce*)
$\quad \mathbf{else} \Rightarrow (\epsilon, \emptyset)$ (*drop*)
$F_{2\Rightarrow 1}(((2)P,\mathcal{A})) : \mathit{true} \Rightarrow ((2)P,\mathcal{A})$ (*announce*)

$F_{3\Leftarrow v, v\in\{1,2,4,5,6,7\}}((P,\mathcal{A}))$:
$\quad 4:50 \in \mathcal{A}[\mathtt{community}] \Rightarrow$
$\quad\quad \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 100)$ (*modify*)
$\quad 4:50 \notin \mathcal{A}[\mathtt{community}] \Rightarrow$
$\quad\quad \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 50)$ (*modify*)
$F_{3\Rightarrow v, v\in\{5,6,7\}}(((3)P,\mathcal{A}))$:
$\quad P = (v\ \ldots), v \notin \{0,4\} \Rightarrow ((3)P,\mathcal{A})$ (*announce*)
$\quad \mathbf{else} \Rightarrow (\epsilon, \emptyset)$ (*drop*)
$F_{3\Rightarrow v, v\in\{1,2,4\}}(((3)P,\mathcal{A})) : \mathit{true} \Rightarrow ((3)P,\mathcal{A})$ (*announce*)

$F_{4\Leftarrow 3}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 150)$ (*modify*)
$F_{4\Leftarrow 2}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 100)$ (*modify*)
$F_{4\Leftarrow 1}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 50)$ (*modify*)
$F_{4\Rightarrow 3}(((4)P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{append}(((4)P,\mathcal{A}), \mathtt{community}, 4:50)$
(*announce*)
$F_{4\Rightarrow v, v\in\{1,2\}}(((4)P,\mathcal{A})) : \mathit{true} \Rightarrow ((4)P,\mathcal{A})$ (*announce*)

$F_{5\Leftarrow 6}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 100)$ (*modify*)
$F_{5\Leftarrow v, v\in\{3,7\}}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 50)$ (*modify*)
$F_{5\Rightarrow v, v\in\{3,6,7\}}(((5)P,\mathcal{A})) : \mathit{true} \Rightarrow ((5)P,\mathcal{A})$ (*announce*)

$F_{6\Leftarrow 7}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 100)$ (*modify*)
$F_{6\Leftarrow v, v\in\{3,5\}}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 50)$ (*modify*)
$F_{6\Rightarrow v, v\in\{3,5,7\}}(((6)P,\mathcal{A})) : \mathit{true} \Rightarrow ((6)P,\mathcal{A})$ (*announce*)

$F_{7\Leftarrow 5}((P,\mathcal{A}))$:
$\quad \mathit{true} \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 100)$ (*modify*)
$F_{7\Leftarrow v, v\in\{3,6\}}((P,\mathcal{A}))$:
$\quad P \neq (Q(6)R) \Rightarrow \mathbf{set}((P,\mathcal{A}), \mathtt{local\_pref}, 50)$
(*modify*)
$\quad \mathbf{else} \Rightarrow (\epsilon, \emptyset)$ (*drop*)
$F_{7\Rightarrow v, v\in\{3,5,6\}}(((7)P,\mathcal{A})) : \mathit{true} \Rightarrow ((7)P,\mathcal{A})$ (*announce*)

Fig. 6. Custom representation of the policies in Fig. 5. Unspecified announcement attributes are assumed to have their default values.

i	V_i	C_i	$\bar{\mathcal{P}}^1$	$\bar{\mathcal{P}}^2$	$\bar{\mathcal{P}}^3$	$\bar{\mathcal{P}}^4$	$\bar{\mathcal{P}}^5,\bar{\mathcal{P}}^6,\bar{\mathcal{P}}^7$
0	$\{0\}$	—	$\{(1\ 0)\}$	$\{(2\ 0)\}$	$\{(3\,4\,2\,0),(3\,0),(3\,2\,0),(3\,1\,0)\}$	$\{(4\,3\,0),(4\,2\,0),(4\ 1\ 0)\}$	$\{\epsilon\}$
1	$\{0,1\}$	$\{1,2,5,6,7\}$	$\{(1\ 0)\}$	$\{(2\ 0)\}$	$\{(3\,4\,2\,0),(3\,0)\}$	$\{(4\,3\,0),(4\,2\,0),(4\ 1\ 0)\}$	$\{\epsilon\}$
2	$\{0,1,2\}$	$\{2,5,6,7\}$	$\{(1\ 0)\}$	$\{(2\ 0)\}$	$\{(3\,4\,2\,0),(3\,0)\}$	$\{(4\,3\,0),(4\,2\,0),(4\ 1\ 0)\}$	$\{\epsilon\}$
3	$\{0,1,2,5\}$	$\{5,6,7\}$	$\{(1\ 0)\}$	$\{(2\ 0)\}$	$\{(3\,4\,2\,0),(3\,0)\}$	$\{(4\,3\,0),(4\,2\,0)\}$	$\{\epsilon\}$
4	$\{0,1,2,5,6\}$	$\{6,7\}$	$\{(1\ 0)\}$	$\{(2\ 0)\}$	$\{(3\,4\,2\,0),(3\,0)\}$	$\{(4\,3\,0),(4\,2\,0)\}$	$\{\epsilon\}$
5	$\{0,1,2,5,6,7\}$	$\{7\}$	$\{(1\ 0)\}$	$\{(2\ 0)\}$	$\{(3\,4\,2\,0),(3\,0)\}$	$\{(4\,3\,0),(4\,2\,0)\}$	$\{\epsilon\}$
6	$\{0,1,2,5,6,7\}$	$\emptyset$	$\{(1\ 0)\}$	$\{(2\ 0)\}$	$\{(3\,4\,2\,0),(3\,0)\}$	$\{(4\,3\,0),(4\,2\,0)\}$	$\{\epsilon\}$

TABLE II
EXECUTION OF GREEDY$^+$ ON THE SPP INSTANCE IN FIG. 7(C).

always be available at 3. In a similar way, at step $i = 2$ vertex 2 enters the stable set because its best path $(2\ 0)$ has 0 as next hop. At step $i = 3$ path $(4\ 1\ 0)$ is removed from the useful paths at 4 because $(4\ 2\ 0)$ will always be available, since 2 is now stable. The algorithm then proceeds by stabilizing vertices 5, 6, and 7, since their only useful path is ϵ.

At the end we have that all the vertices but 3 and 4 have been stabilized. This means that the original BGP configuration may exhibit oscillations and the cause lies in the policies at 3 and 4. This response is correct, since we intentionally built a DISAGREE using vertices 3 and 4. Also notice that BAD-GADGET, the other oscillatory structure we put in the configuration, is reported as stable because the paths supporting its oscillation will never be available after the SPVP algorithm (i.e., BGP) has exchanged a few messages. This was, in some way, also suggested by the fact that all the paths at 5, 6, and 7 had been cleared by early stabilization and early suppression.

VII. EXPERIMENTAL RESULTS AND APPLICABILITY CONSIDERATIONS

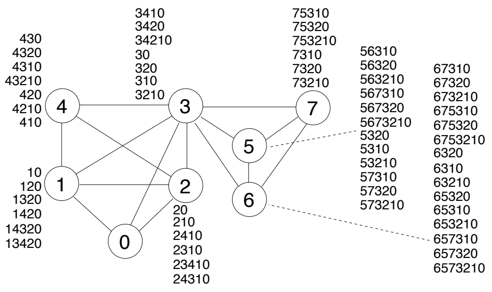
In Section V-B we described the optimization techniques we used to make our tool efficient enough to process Internet-scale configurations. We then showed the potential of these techniques in a realistic example in Section VI-C. However, it is difficult to analytically assess the effectiveness of these optimizations, since it strongly depends on the configured policies and on the network topology.

Hence, in order to validate our approach and to assess its practical applicability, we experimented with a prototype implementation of the convergence checker. Since we are interested in showing that static analysis for BGP convergence is feasible even for Internet-scale topologies, in our experiments we focused on eBGP routing policies. In principle, we could extract the policies from the Internet Routing Registries, but it is known [32] that such information is partial and in many cases inconsistent and out of date. Therefore, we chose to get the policies from the largest publicly accessible source having reasonable worldwide coverage, which is CAIDA [28].

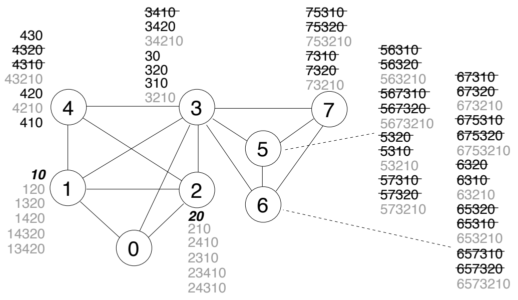
In the following, we describe the prototype and discuss the results of our experiments.

A. A Prototype Implementation

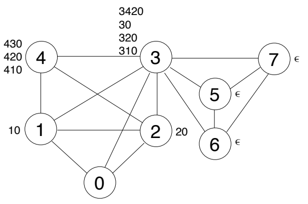
We developed a Java-based prototype implementation of the architecture in Fig. 2. This implementation is subject to some known limitations that restrict the set of BGP configuration policies it can analyze. In particular: *i*) a limited number of BGP attributes is supported, namely only the `as_path`, the `next_hop`, the `local_pref` and the `community`, and *ii*) filters can only be defined on a per-neighbor basis.



(a) SPP instance obtained by running the dissemination algorithm in Fig. 3 on the topology and policies extracted from the policy specifications in Fig. 6.



(b) Reduced SPP instance obtained by applying early stabilization (dimmed paths) and early suppression (struck out paths) during the dissemination.



(c) Final SPP instance (same as (b) with removed paths).

Fig. 7. SPP instance constructed from the policies in Fig. 6.

Limitation *i*) is not really constraining, as the combined use of `local_pref` and `community` leads to highly expressive policies, and previous studies show the widespread use of these attributes [33]. Also limitation *ii*) is not constraining because, on one hand, BGP policies are often analyzed using neighbor-specific models [12], [28], [30], and, on the other hand, using a finer granularity would add little expressiveness at the expense of manageability and performance, and we argue that these factors would impact network operation, too. Moreover, it should be considered that our architecture is designed to accommodate every kind of filter.

B. Performance and Scalability

We ran our experiments using as input the AS-level topologies from CAIDA [28]. While CAIDA datasets are unavoidably biased by the underlying inference algorithms by which they have been computed, we believe they are still a valuable data source of large-scale policy-labeled interdomain topologies, which is exactly what we need to verify the scalability of our approach. We extracted from the CAIDA dataset collected on Jan 20th, 2010 a set of smaller topologies by pruning vertices with degree lower than a threshold. We picked thresholds in the values listed in Table III, which also shows the number of vertices and edges of the obtained topologies. The last line (threshold 1) corresponds to the complete topology. All the generated graphs were connected. CAIDA datasets are annotated with information about the commercial relationships established between the ASes [29]. In order to compare with state-of-the-art tools, we implemented these relationships with BGP policies using the same approach that is hardwired in the C-BGP simulator [24]. In our experiments we assumed to originate a prefix from a given AS picked from a sample of 200 ASes having very different degree values. In particular, we picked the top 100 and the bottom 100 ASes according to CAIDA's ranking algorithm, which ranks ASes based on the size of their *customer cone* (number of direct and indirect customers). Since there is a high correlation between the customer cone and the position of an AS in the Internet hierarchy, essentially our sample is composed of ASes ranging from tier-1 providers to small stubs. The median degree of the ASes in our sample is 8, the average degree is 201, and 20% of ASes have a degree higher than 264.

Degree Threshold	Vertices	Edges	Degree Threshold	Vertices	Edges
1000	10	33	25	592	7403
500	25	209	10	1657	15672
250	48	603	5	3735	24388
100	125	1956	4	5113	28772
50	268	3953	2	21263	62756
35	397	5466	1 (complete topology)	33508	75001

TABLE III
TOPOLOGIES USED IN OUR TESTS, OBTAINED FROM CAIDA TOPOLOGIES BY PRUNING VERTICES WITH DEGREE LOWER THAN A THRESHOLD.

Our testing platform was a dual Xeon 2.66GHz with 16GB of RAM. We ran the checker once for each combination of pruned topology and originator AS. From the theoretical 2,400 runs we had to exclude the cases when the originator AS itself was removed by the pruning, dropping to 1,099 runs. For each run, we assigned 4GB of RAM to the Java VM hosting the checker. Observe that, despite the amount of memory reserved for the tool, there were cases in which the checker ran out of memory because of the excessive number of paths to be generated for the SPP instance. Every time this happened for a certain originator AS and a certain threshold, we avoided attempting the check with the same originator AS on topologies with lower threshold. Thus, we boiled down to 540 successful runs. Note that, in many cases, the SPP instances can *only* be generated using both our optimizations (GREEDY$^+$): with early stabilization alone (i.e., GREEDY) we could achieve only 105 successful runs.

The convergence check took a fraction of a second to complete in 24% of the successful runs, and 16 seconds on average. The median of running times was of 2 seconds, while the maximum was 13 minutes, recorded in 1 run only. As a term of comparison, running our implementation using early stabilization alone (i.e., GREEDY) resulted in up to 64 minutes of computing time. We could not find any correlations between the running time and other topological features such as the pruning threshold and the originator AS degree. These results already prove that our approach outperforms the state of the

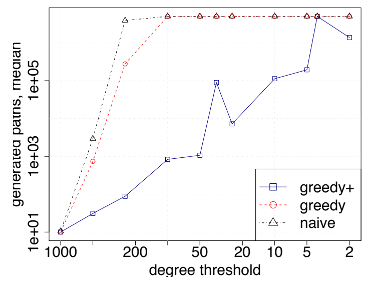


Fig. 8. Median number of generated paths, considering the top 100 ASes as originators. The plots show the values without optimizations (naive), with early stabilization (greedy) and with early stabilization and early suppression (greedy+). The X axis shows the threshold used to prune CAIDA topologies.

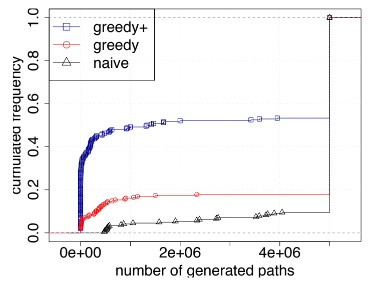


Fig. 9. Cumulative distribution function of the number of generated paths, considering the top 100 ASes and fixing the degree threshold at 2.

art: it can successfully check a larger number of Internet-scale topologies and achieves this in a very short time. These performance results show that the tool can be used for online convergence checks performed right after a policy change.

Figs. 8 and 9 show the effectiveness of the optimizations in GREEDY$^+$. Both figures refer to the experiments that considered the top 100 ASes as originators. In order to assess the feasibility of convergence checks for Internet-scale BGP configurations, we associated an arbitrary value of 5M paths to each path generation that ran out of memory. Fig. 8 plots the median number of paths in the set $\mathcal{P}$ of the generated SPP instance. Each point corresponds to a value of the degree threshold we used to prune the CAIDA topology. We used all the values in Table III down to 2, since degree-1 vertices cannot make the network unstable. It is clear that using both our optimizations (greedy+) defeats early stabilization alone (greedy) and generation of paths without optimizations (naive). Note that, starting from threshold 100, the optimizations in GREEDY$^+$ are necessary to successfully generate the SPP instance. The plot for GREEDY$^+$ exhibits some irregularities (e.g., for degree threshold 35): we ascribe this to the fact that generated paths are highly dependent on the topology, since the presence of specific vertices can cause a large number of additional paths to be generated. Fig. 9 shows the cumulative distribution function of the number of generated paths, on the topology obtained with degree threshold 2. Observe that early stabilization alone (greedy) allows us to successfully generate the paths for less than 20% of the cases. On the other hand, using both our optimizations (greedy+) allows us to successfully perform the check on more than 50% of the topology-originator pairs, and for 40% of them the check requires generating less than 200,000 paths. Moreover, in 50% of the successful runs our checker generated less than 1% of the paths generated using early stabilization alone (greedy).

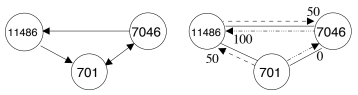


Fig. 10. This triple of vertices, on which C-BGP was unable to converge, has been pinpointed by our checker as potentially oscillating. On the left we show the commercial relationships between the ASes. On the right we show the propagation of BGP announcements and the local preference at each vertex.

The results obtained with the bottom 100 ASes in the CAIDA ranking show similar trends (plots are omitted for brevity) but different absolute values. We were able to successfully perform the check on 38% of the topology-originator pairs. This shows that the performance of our technique degrades when the originator AS is placed at the bottom levels of the Internet hierarchy, which is possibly due to the fact that stub ASes are reachable from the Internet through a very high number of possible paths. However, we stress that the optimizations of GREEDY$^+$ greatly benefit from the presence of peer-to-peer links, which are typically not captured in AS-level topologies (see, e.g., [34]).

Interestingly, compared with [1], our optimizations are still effective, despite the larger topology (26% more vertices and 39% more adjacencies) we used as input.

C. Spotting Potential Oscillations

We finally looked at the percentage of vertices of the input topology that our checker reported as safe. Interestingly, depending on the originator AS, our checker reported up to 5% of vertices as potentially unstable.

We further investigated the portions of the network which our checker reported as potentially unstable. In a separate experiment based on the data used in [1], our prototype was able to spot a triple of vertices that, if configured in accordance with the policies hardwired in C-BGP, generated a DISAGREE [4] (namely a potentially oscillating structure). The triple is depicted in Fig. 10. The left part of the figure shows the commercial relationships between the ASes, adopting the same graphical convention used in Fig. 4 (AS 11486 is a provider of AS 7046 and is a customer of AS 701, while AS 701 and AS 7046 are siblings). The right part of Fig. 10 shows the propagation of BGP announcements when AS 701 is the originator. Values next to each vertex represent the local preferences assigned to the received announcements (the higher the preferred): 50 for announcements received from a provider, 100 for announcements received from a customer, while announcements from siblings retain the preference value assigned by the neighbor AS. However, since AS 701 is the originator AS, there is no preference for its sibling AS 7046 to retain, therefore the default value 0 is applied. This creates a “policy dispute” between AS 11486 and AS 7046, which we are able to detect but causes C-BGP to loop indefinitely. In a sense, this example shows how our approach can also be useful to detect potential routing oscillations triggered by implementation or vendor-specific choices.

VIII. CONCLUSIONS

In this paper we show that an automated check for BGP convergence is feasible in practice. We describe a heuristic algorithm (GREEDY$^+$) that can be used to check the convergence of BGP in an abstract model. We prove that this algorithm has several desirable properties, among which the ability to avoid false positives (configurations mistakenly reported as safe while they are not). We use this algorithm as the base for an automated tool that is able to statically check real-world BGP configurations for guaranteed routing convergence. The tool models the configurations using the SPP formalism. Since this choice may lead to an intractably large representation to be handled, we propose optimizations that make it feasible to run the convergence checker, even online, on Internet-scale topologies. We validate our approach by performing experiments on AS-level Internet topologies from CAIDA. We are able to spot potentially oscillating portions of networks that cause state-of-the-art simulators (C-BGP) to loop indefinitely. Our results show that GREEDY$^+$ outperforms the well-known GREEDY algorithm [4]. However, characterizing the additional topologies that GREEDY$^+$ can solve is an open problem.

REFERENCES

[1] L. Cittadini, M. Rimondini, M. Corea, and G. Di Battista, "On the feasibility of static analysis for BGP convergence," in *Proc. IM*, 2009.

[2] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," IETF RFC 4271, 2006.

[3] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "Policy disputes in path-vector protocols," in *Proc. ICNP*, 1999.

[4] ——, "The stable paths problem and interdomain routing," *IEEE/ACM Trans. on Networking*, vol. 10, no. 2, pp. 232–243, 2002.

[5] M. Carlzon, "BGP oscillations when peering with loopback addresses," in *Proc. AINA*, 2006.

[6] F. Berger, "BGP oscillation," Presentation at SwiNOG 3, 2001.

[7] T. G. Griffin and G. T. Wilfong, "An analysis of BGP convergence properties," in *Proc. SIGCOMM*, 1999.

[8] ——, "On the correctness of iBGP configuration," in *Proc. SIGCOMM*, 2002.

[9] F. Wang, Z. M. Mao, J. Wang, L. Gao, and R. Bush, "A measurement study on the impact of routing events on end-to-end Internet performance," in *Proc. SIGCOMM*, 2006.

[10] N. Kushman, S. Kandula, and D. Katabi, "Can you hear me now?! It must be BGP," in *Computer Communication Review*, 2007.

[11] F. Wang and L. Gao, "On inferring and characterizing Internet routing policies," in *Proc. IMC*, 2003.

[12] G. Siganos and M. Faloutsos, "Analyzing BGP policies: Methodology and tool," in *Proc. INFOCOM*, 2004.

[13] L. Cittadini, G. Di Battista, and S. Vissicchio, "Doing don'ts: Modifying BGP attributes within an autonomous system," in *Proc. NOMS*, 2010.

[14] N. Feamster, H. Balakrishnan, and J. Rexford, "Some foundational problems in interdomain routing," in *Proc. HotNets*, 2004.

[15] M. Suchara, A. Fabrikant, and J. Rexford, "BGP safety with spurious updates," in *Proc. INFOCOM*, 2011.

[16] M. Chiesa, L. Cittadini, G. Di Battista, and S. Vissicchio, "Local transit policies and the complexity of BGP stability testing," in *Proc. INFOCOM*, 2011.

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

[18] T. G. Griffin and G. T. Wilfong, "A safe path vector protocol," in *Proc. INFOCOM*, 2000.

[19] C. T. Ee, V. Ramachandran, B.-G. Chun, K. Lakshminarayanan, and S. Shenker, "Resolving inter-domain policy disputes," in *Proc. SIGCOMM*, 2007.

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

[21] A. Flavel, M. Roughan, N. Bean, and A. Shaikh, "Where's Waldo? Practical Searches for Stability in iBGP," in *Proc. ICNP*, 2008.

[22] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *Proc. NSDI*, 2005.

[23] X. Qie and S. Narain, "Using service grammar to diagnose BGP configuration errors," *Science of Computer Programming*, vol. 53, no. 2, pp. 125–141, 2004.

[24] B. Quoitin and S. Uhlig, "Modeling the routing of an autonomous system with C-BGP," *IEEE Network*, vol. 19, no. 6, 2005.

[25] L. Colitti, G. Di Battista, F. Mariani, M. Patrignani, and M. Pizzonia, "Visualizing interdomain routing with BGPlay," *Journal of Graph Algorithms and Applications*, vol. 9, no. 1, pp. 117–148, 2005.

[26] T. G. Griffin and G. T. Wilfong, "Analysis of the MED oscillation problem in BGP," in *Proc. ICNP*, 2002.

[27] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra, "Routing Policy Specification Language (RPSL)," IETF RFC 2622, Jun 1999.

[28] CAIDA, "AS topologies annotated with AS relationships," http://www.caida.org/data/active/as-relationships/index.xml.

[29] L. Gao, "On inferring autonomous system relationships in the Internet," *IEEE/ACM Trans. on Networking*, vol. 9, no. 6, pp. 733–745, 2001.

[30] L. Gao and J. Rexford, "Stable Internet routing without global coordination," in *Proc. SIGMETRICS*, 2000.

[31] G. Di Battista, T. Erlebach, A. Hall, M. Patrignani, M. Pizzonia, and T. Schank, "Computing the types of the relationthips between autonomous systems," *IEEE/ACM Trans. on Networking*, vol. 15, no. 2, pp. 267–280, 2007.

[32] G. Di Battista, T. Refice, and M. Rimondini, "How to extract BGP peering information from the Internet routing registry," in *Proc. ACM SIGCOMM MineNet*, 2006.

[33] B. Donnet and O. Bonaventure, "On BGP communities," *ACM SIGCOMM Comp. Comm. Review*, vol. 28, no. 2, pp. 55–59, Apr 2008.

[34] X. Dimitropoulos, D. Krioukov, M. Fomenkov, B. Huffaker, Y. Hyun, kc claffy, and G. Riley, "AS relationships: Inference and validation," *ACM SIGCOMM Comp. Comm. Review*, vol. 37, no. 1, pp. 29–40, 2007.

APPENDIX A
FORMAL PROOFS

In this appendix we formally prove the properties of algorithm GREEDY$^+$ as stated in Section IV-B.

Property 4.1: Let $n = |\bar{\mathcal{P}}|$ be the size of an SPP instance S. GREEDY$^+$ can be implemented to terminate on S in time that is polynomial in n.

Proof: A trivial bound follows.

Step *i*) of GREEDY$^+$ applies to those vertices v which extend a path P offered by some neighbor u in the stable set. This step can be implemented by evaluating λ^v for all the paths in each $\bar{\mathcal{P}}^v$ and comparing its value with $\lambda^v((v\ u)P)$. This takes $O(n^3)$ time, since the length of a path is $O(n)$.

Step *ii*) enforces consistency, and can be accomplished by comparing each path in $\bar{\mathcal{P}}$ with all the others, taking $O(n^3)$.

At Step *iii*), candidate vertices can be found in $O(n^3)$ time.

Since GREEDY$^+$ executes at most $|V|$ iterations and an instance of SPP can have $O(n)$ vertices, GREEDY$^+$ can be implemented to run in $O(n^4)$ time. ■

To prove Property 4.2, we first prove the following Lemmas.

Lemma 1.1: If GREEDY$^+$ terminates after k iterations, its output is completely defined by sets V_k and $\bar{\mathcal{P}}^v\ \forall v \in V_k$.

Proof: The missing portion of the output, $\bar{\mathcal{P}}^v\ \forall v \in V - V_k$, can be uniquely constructed starting from V_k and $\bar{\mathcal{P}}^v\ \forall v \in V_k$. Consider a new instance $S' = (G', \mathcal{P}', \Lambda')$ of SPVP with $G' = G$, $\Lambda' = \Lambda$. For any $v \in V$, let $\mathcal{P}'v = \bar{\mathcal{P}}^v$ if $v \in V_k$, and let $\mathcal{P}'v = \mathcal{P}^v$ if $v \notin V_k$.

Now, initialize the stable set V_0 to V_k and execute Steps *i*) and *ii*) of GREEDY$^+$ on S'. We now show that, after doing so, $\bar{\mathcal{P}}'^v = \bar{\mathcal{P}}^v, \forall v \in V$. This is trivially true for vertices $u \in V_k$, as no path is ever removed from $\bar{\mathcal{P}}'^u$. Observe that the outcome of Step *i*) of GREEDY$^+$ only depends on the topology of the graph G', the ranking functions Λ', and the sets of useful paths $\bar{\mathcal{P}}'^v$, with $v \in V_k$. By the definition of S', at Step *i*), a path is removed from $\bar{\mathcal{P}}^v$ iff it is removed from $\bar{\mathcal{P}}'^v$. Hence, any possible difference must be due to Step *ii*).

We prove by contradiction that the output coincides also for vertices in $V - V_k$. Suppose that this is not the case, i.e., there exists some vertex $v \in V - V_k$ such that $\bar{\mathcal{P}}'^v \neq \bar{\mathcal{P}}^v$. Then, there exists a path P such that either $P \notin \bar{\mathcal{P}}'^v \wedge P \in \bar{\mathcal{P}}^v$ or $P \in \bar{\mathcal{P}}'^v \wedge P \notin \bar{\mathcal{P}}^v$. In the first case, the execution of Step *ii*) on S' has removed from $\bar{\mathcal{P}}'^v$ a path that the execution of GREEDY$^+$ on S regarded as consistent. But this is impossible, since $\forall v \in V$, $\bar{\mathcal{P}}^v \subseteq \mathcal{P}'^v$, so there can be no path that is consistent with $\bar{\mathcal{P}}$ and is not consistent with $\mathcal{P}'$. In the second case, the execution on S has removed from $\bar{\mathcal{P}}^v$ a path P that the execution on S' considered as consistent. Since it cannot be $P \notin \mathcal{P}^v$, then for P to be inconsistent with $\bar{\mathcal{P}}$, it may only be the case that $P = (v \ \ldots \ u)P_u$, where $P_u \notin \bar{\mathcal{P}}^u$ and $P_u \in \bar{\mathcal{P}}'^u$. In turn, this is only possible if there exists a path P_w such that $P_u = (u \ \ldots \ w)P_w$, with $P_w \notin \bar{\mathcal{P}}^w$ and $P_w \in \bar{\mathcal{P}}'^w$. By proceeding this way, we must eventually end up on a vertex x in V_k, possibly 0. By recalling that $\bar{\mathcal{P}}'^v = \bar{\mathcal{P}}^v$ $\forall v \in V_k$ by construction, we have a contradiction in that it should be $P_x \notin \bar{\mathcal{P}}^x$ and $P_x \in \bar{\mathcal{P}}'^x$. ■

Lemma 1.2: Consider a path P that is inconsistent with $\bar{\mathcal{P}}$ at iteration i of GREEDY$^+$. Then, P is inconsistent at any iteration $j > i$.

Proof: The property follows by observing that GREEDY$^+$ never adds new paths to $\bar{\mathcal{P}}$. ■

Lemma 1.3: At any iteration i of GREEDY$^+$, $C_i \cap V_i = V_i - V_{i-1}$.

Proof: By construction, $C_i \cap V_{i-1} = \emptyset$. Now, at iteration i a vertex is picked from C_i and added to V_{i-1} to construct V_i. Therefore, the property follows. ■

The following Lemma states the fact that, once a vertex enters the candidate set, it stays there until it is eventually moved to the stable set.

Lemma 1.4: Consider an arbitrary iteration i of GREEDY$^+$ and a vertex $v \in C_i$. Then there exists an iteration j such that $v \in C_h$ for all $i \leq h \leq j$ and $v \in V_k$ for all $k \geq j$.

Proof: Let $v \in C_i$ be a vertex such that the path $P \in \bar{\mathcal{P}}^v$ with minimum $\lambda^v(P)$ at iteration i either has a next hop in V_{i-1}, or $P = \epsilon$. Since no better path can enter $\bar{\mathcal{P}}^v$ during the execution of GREEDY$^+$ (Lemma 1.2) and P has the minimum value of λ^v among the paths in $\bar{\mathcal{P}}^v$ that are consistent with $\bar{\mathcal{P}}$, P can never be removed from $\bar{\mathcal{P}}^v$ at Step *i*) of GREEDY$^+$. Moreover, if $P = \epsilon$, by definition P is a consistent path. Otherwise, if $P = (v \ u)Q$, $u \in V_{i-1}$, $\{Q\} = \bar{\mathcal{P}}^u$, then P will remain consistent with $\bar{\mathcal{P}}$ because its next hop is $u \in V_{i-1}$, so $\bar{\mathcal{P}}^u$ will not be updated after iteration i. Thus, P cannot be removed from $\bar{\mathcal{P}}^v$ at Step *ii*). Overall, starting from iteration i, path P will always be available in $\bar{\mathcal{P}}^v$ and will always have the minimum value of λ^v. In other words, v satisfies the conditions of Step *iii*) at any iteration $h \geq i$, i.e., $v \in C_h \cup V_h$.

Since $\forall h \geq i$ we have $v \in C_h \cup V_h$, and GREEDY$^+$ only terminates when the candidate set is empty, by Lemma 1.3 there must be an iteration j at which v is picked from C_j and added to V_{j-1} to construct V_j. The statement follows by recalling that vertices never leave the stable set. ■

Property 4.2: Consider a set C_j of vertices satisfying the criteria of Step *iii*) at an arbitrary iteration j of GREEDY$^+$. The output of GREEDY$^+$ does not change, regardless of the choice of vertex $v \in C_j$ performed at iteration j.

Proof: Assume that GREEDY$^+$ terminates at iteration k and consider that, by Lemma 1.1, it is sufficient to prove the assertion for sets V_k and $\bar{\mathcal{P}}^u$ with $u \in V_k$. Consider an arbitrary vertex $v \in C_j$ at iteration $j < k$. By Lemma 1.4, we know that $v \in C_h$ for any iteration $h \geq j$, until v eventually enters the stable set. Also, as shown in the proof of Lemma 1.4, the best path $(v \ w)P$, $w \in V_h$, is always in $\bar{\mathcal{P}}^v$. Therefore, regardless of the iteration at which v is actually selected, the set $\bar{\mathcal{P}}^v$ is always updated with path $(v \ w)P$. Moreover, the set of paths that become inconsistent with $\bar{\mathcal{P}}$ after setting $\bar{\mathcal{P}}^v = \{(v \ w)P\}$ does not depend on the iteration either.

Thus, a vertex $v \in C_h$ can be picked by Step *iii*) at any iteration h of GREEDY$^+$ without affecting neither V_k nor $\bar{\mathcal{P}}^u$ $\forall u \in V_k$. Since this is true for any vertex $v \in C_h$, GREEDY$^+$ can select an arbitrary candidate vertex at each iteration h without affecting the output. ■

Property 4.3: Consider an SPP instance $S = (G = (V, E), \mathcal{P}, \Lambda)$ and run GREEDY$^+$ on S. Let $P \in \mathcal{P}^v$ be a path that GREEDY$^+$ deletes at iteration j. Then, for any fair activation sequence σ of SPVP on S, there exists a time t' such that $\forall t > t'$, $\pi_t(v) \neq P$.

Proof: The statement asserts that GREEDY$^+$ deletes only those paths that would be discarded by any fair activation sequence of SPVP. The proof is by induction on the iteration j of GREEDY$^+$. At iteration $j = 1$, since $\bar{\mathcal{P}}^u = \mathcal{P}^u$ for all $u \in V$, GREEDY$^+$ deletes a path P from $\bar{\mathcal{P}}^v$ at either Step *i*) or Step *ii*) according to the following conditions.

Deletion at Step i*)*: Since $V_0 = \{0\}$, the deletion takes place if $\lambda^v((v \ 0)) < \lambda^v(P)$. By the fairness of σ, there must exist a time t' such that $(0, v)$ is activated at t': this prevents v from selecting P after t'.

Deletion at Step ii*)*: It takes place if P is inconsistent with $\mathcal{P}$, i.e., $P = Q(w)R$ and $R \notin \mathcal{P}^w$. In this case, the statement trivially follows since $\pi_t(w) \neq R \ \forall t$.

Assume, by induction, that the assertion holds for a given iteration $j - 1$ of GREEDY$^+$. We now prove that the same property is true for the paths that are deleted during iteration j. Again, during iteration j, GREEDY$^+$ deletes a path P from $\bar{\mathcal{P}}^v$ at either Step *i*) or Step *ii*).

Deletion at Step i*)*: It takes place if there exists $u \in V_{j-1}$ such that $(v, u) \in E$ and $\lambda^v((v \ u)P') < \lambda^v(P)$, where $\{P'\} = \bar{\mathcal{P}}^u$. Observe that the induction hypothesis assures that previously deleted paths are eventually discarded after time t'. Then, by the fairness of σ, there must exist a time $t'' > t'$ such that (u, v) is activated at t'' and $(v \ u)P'$ is made available at $v \ \forall t > t''$. This prevents v from selecting path P, i.e., $\pi_t(v) \neq P \ \forall t > t''$.

Deletion at Step ii*)*: It takes place if P is inconsistent, i.e., $P = Q(w)R$ and $R \notin \bar{\mathcal{P}}^w$. By the induction hypothesis, there exists t' such that $\forall t > t' \ \pi_t(w) \neq R$. Then, by the fairness of σ, v must receive a message that withdraws the availability of R at a time $t'' > t'$. Therefore, $\pi_t(v) \neq P \ \forall t > t''$. ■

Property 4.4: If GREEDY$^+$ terminates successfully on an instance S of SPP, then S is safe and has a unique solution.

Proof: Assume that GREEDY$^+$ terminates successfully on $S = (G = (V, E), \mathcal{P}, \Lambda)$ after k iterations, and let $V_k = V$ be the stable set after the k-th iteration. By Step *iii*) of GREEDY$^+$ we know that for each vertex $v \in V_k$, $\bar{\mathcal{P}}^v$ only contains one path (possibly ϵ), because GREEDY$^+$ deleted all the permitted paths but one. By Property 4.3, for each deleted path $P = (v \ \ldots)$ there exists a time t' after which vertex v is permanently unable to select P, i.e., $\pi_t(v) \neq P$ for each $t > t'$. This implies that there exists a time $\bar{t}$ after which all the deleted paths are never selected. Therefore, for all $v \in V$ and for all $t > \bar{t}$, $\pi_t(v) = P_v$, where $\bar{\mathcal{P}}^v = \{P_v\}$. ■

We need the following lemma to prove Property 4.5.

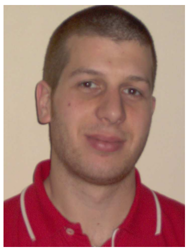
Lemma 1.5: Let S be an instance of SPP. If GREEDY terminates on S finding a path assignment π^*, then GREEDY$^+$ also terminates on S finding π^*.

Proof: By Property 4.2 we know that, when multiple vertices can enter the stable set at a given iteration, the solution computed by GREEDY$^+$ is independent on the order in which these vertices are considered. Therefore, we prove the assertion by showing that GREEDY$^+$ can find π^* by selecting vertices to put in the stable set in the very same order as GREEDY does. We show it by mapping each iteration of GREEDY to one iteration of GREEDY$^+$. In the following, we will refer to GREEDY's stable set as V_j, and to GREEDY$^+$'s stable set as V_j^+, and we will indicate with π the path assignment defined by GREEDY at a given iteration. The proof proceeds by induction on the iteration j. It is trivially true that, at $j = 0$, $V_j = V_j^+ = \{0\}$. Assume that $V_{j-1} = V_{j-1}^+$ and, without loss of generality, that the stable sets have been constructed by adding vertices in the very same order by the two algorithms. Consider vertex u that GREEDY selects at iteration j. This implies that $(u \ v)\pi(v)$ is the path with minimum λ^u among those compatible with π, for some $v \in V_{j-1}$. By the induction hypothesis, $\bar{\mathcal{P}}^v = \{\pi(v)\}$, therefore path $(u \ v)\pi(v)$ is consistent with $\bar{\mathcal{P}}$. We show that path $(u \ v)\pi(v)$ must still be in $\bar{\mathcal{P}}^u$ at iteration j. Lemma 1.2 ensures that Step *ii*) did not remove path $(u \ v)\pi(v)$ from $\bar{\mathcal{P}}^u$. That is, since path $(u \ v)\pi(v)$ is consistent with $\bar{\mathcal{P}}$ at iteration j, it was always consistent during the previous iterations. By the induction hypothesis, $\forall w \in V_{j-1}$ $\bar{\mathcal{P}}^w = \{\pi(w)\}$, therefore all the paths that are regarded as consistent by GREEDY$^+$ are necessarily compatible with π. Hence, since $(u \ v)\pi(v)$ is consistent with $\bar{\mathcal{P}}$ and has minimum λ^u among the paths compatible with π, it must also have minimum λ^u among the paths consistent with $\bar{\mathcal{P}}$. Therefore path $(u \ v)\pi(v)$ cannot be deleted at Step *i*) of GREEDY$^+$, and vertex u is a candidate to be inserted in the stable set by GREEDY$^+$.

Since, by Property 4.2, the output of GREEDY$^+$ is unaffected by the order in which vertices enter the stable set, we can assume without loss of generality that GREEDY$^+$ too selects vertex u at iteration j. This in turn implies that GREEDY$^+$ finds the same path assignment π^*. ■

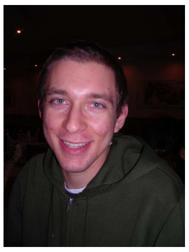
Property 4.5: The set of instances that GREEDY$^+$ can successfully solve is strictly larger than the set of instances that GREEDY is able to solve.

Proof: Lemma 1.5 proves the inclusion. The strictness is supported by DI-SAFE-GREE, which is not solved by GREEDY, as we discussed above, while it is solved by GREEDY$^+$ as shown in Table I. ■



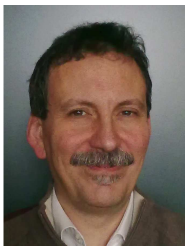
Luca Cittadini received his master degree from the Roma Tre University in 2006, and a Ph.D. degree in Computer Science and Automation from the same institution in 2010, defending the Thesis "Understanding and Detecting BGP Instabilities". During his Ph.D. he was a teaching assistant in the computer network research lab. His research activity is primarily focused on interdomain routing, including theoretical analysis of the protocol as well as active and passive measurements.

Massimo Rimondini received a Ph.D. in Computer Science and Automation at the Roma Tre University in 2007, defending the Thesis "Interdomain Routing Policies in the Internet: Inference and Analysis". He is currently a researcher at the Dept. of Computer Science and Automation of the Roma Tre University. His interests include: combinatorial problems in interdomain routing (stability, inference of commercial relationships), emulation of computer networks, network measurements. He is one of the lead developers of the network emulator Netkit.



Stefano Vissicchio received his master degree in Computer Science from the Roma Tre University in 2008. Currently, he is a Ph.D. student at the Roma Tre University. His research interests are mainly focused on network management, ranging from checking interdomain routing stability to efficient monitoring and effective visualization techniques. Some of his research results exploit router programmability.

Matteo Corea received his master degree in Computer Science from the Roma Tre University discussing the thesis "Predictive Analysis of Stability in Interdomain Routing". Since then, he worked on other project ranging from image processing and efficient computation on graphic processing units to automation in the setup of network emulators. He currently works as a software engineer, focusing on the design of large-scale web services.



Giuseppe Di Battista received a Ph.D. in Computer Science from the University of Rome "La Sapienza" and is currently Professor of Computer Science at the Dept. of Computer Science and Automation of the Roma Tre University. His interests include Computer networks, Graph Drawing, and Information Visualization. He published more than 100 papers in the above areas, co-authored a book on Graph Drawing, and has given several invited lectures worldwide. He is a founding member of the steering committee for the Graph Drawing Symposium.