Detecting the *Unintended* in BGP Policies

Debbie Perouli*, Timothy G. Griffin†, Olaf Maennel‡, Sonia Fahmy*, Iain Phillips‡, Cristel Pelsser§
*Computer Science, Purdue University, USA; †Computer Lab, University of Cambridge, UK;
‡Computer Science, Loughborough University, UK; §IIJ, Japan
*{depe,fahmy}@purdue.edu, †tgg22@cl.cam.ac.uk, ‡{o.m.maennel,i.w.phillips}@lboro.ac.uk,§cristel@iij.ad.jp

***Abstract*—Internet Service Providers (ISPs) use routing policies to implement the requirements of business contracts, manage traffic, address security concerns and increase scalability of their network. These routing policies are often a *high-level* expression of strategies or intentions of the ISP. They have meaning when viewed from a *network-wide* perspective (*e.g.*, mark on ingress, filter on egress). However, configuring these policies for the Border Gateway Protocol (BGP) is undertaken at a *low-level*, on a *per router* basis. Unintended routing outcomes have been observed. In this work, we define a language that allows analysis of network-wide configurations at the high-level. This language aims at bridging the gap between router configurations and abstract mathematical models capable of capturing complex policies. The language can be used to verify desired properties of routing protocols and hence detect potential unintended states of BGP. The language is accompanied by a tool suite that parses router configuration languages (which by their nature are vendor-dependent) and translates them into vendor-independent representations of policies.**

I. Introduction

Configuring BGP can be extremely challenging. Each device needs its own low-level vendor-specific configuration, but BGP policy objectives are typically designed *network-wide* at a *high-level*. The research community has made significant strides in determining BGP anomalies [1], inferring policies [2], and building Internet topology models [3]. Unfortunately, most of these efforts rely on BGP data obtained from a set of monitors [4] which have limited visibility [5] and use [6]. An even greater concern is that no single ISP has sufficient information to debug certain problems [7]. The only way to detect or debug certain *unintended* policy interactions among ISPs is from a combined view of the configurations of all ISPs involved.

Available tools for validating already *existing* device configuration files [8]–[10] are limited in their scope, power, and degree of abstraction. However, recent work [11], [12] has proposed mathematically proven frameworks that enable the detection of potentially unsafe policy combinations. The stratified shortest paths problem (*SSPP*) formalization [13] allows us to push BGP policy analysis further. It enables the development of techniques to detect configurations that may potentially lead to unsafe BGP conditions.

The primary goal of this work is to bridge the gap between device configuration languages used to configure routers and mathematically sound abstractions such as the *SSPP*. This allows the correct routing decisions to be rigorously checked on real networks. It further allows validation that the deployed network matches the operators' high-level intentions. To achieve this goal, we have developed a toolset that translates router configurations into a structure usable for mathematical proof. We express this structure using a functional programming language, Haskell [14], because it provides rich support for mathematical abstractions such as semi-rings, and a convenient scheme for expressing and manipulating policies as functions.

This work has been sponsored in part by a gift from Cisco Systems. The authors would like to thank Randy Bush (IIJ) for several detailed discussions.

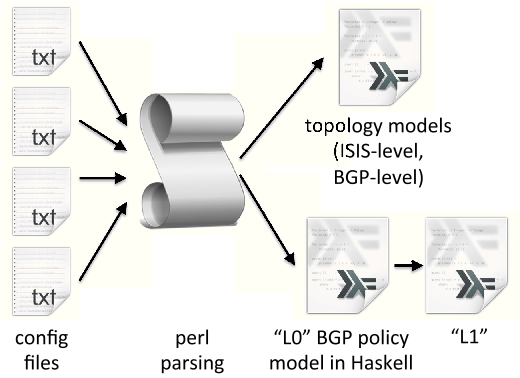

Fig. 1: Our tool suite parses raw router configuration files using Perl pattern matching, then the relevant topology and BGP policy information is translated into Haskell.

II. Policy Representation

Tools developed as part of this work convert raw device configuration files into the more abstract representations L0 and L1. Fig. 1 depicts an overview of the tools. Parsing raw configuration files mainly involves pattern matching, so it is performed via a set of Perl scripts. The output L0 representation includes policies, communities, AS paths, prefix lists, and BGP sessions. Vendor-independent keywords are used in L0, but the structure of the policies still mimics the original vendor and OS-specific configuration language. A Haskell tool converts configurations written in L0 to L1. L1 removes vendor-specific control-structures, such as `Next`, and is therefore more appropriate for further analysis.

A. Language Zero

The L0 policy representation is inspired by the Juniper syntax, where a policy is a list of *policy units* and each unit is a list of *policy terms*. A term has a list of *actions* that is executed only if the specified list of *match conditions* for

that term are satisfied. Either of the two lists can be empty. Each term also has a *control keyword* specifying whether the route matching the conditions will be accepted, rejected, or will undergo further processing by another policy unit or term (*Next*-term). The following is an example of a policy with two policy units, the first one having two policy terms.

```
l0 = [PUunit [PTterm [m1] [a1,a2] Next,
              PTterm [m2] [] Reject],
      PUunit [PTterm [m3] [b] Accept]]
```

B. Language One

The L1 canonical representation provides means to compare and compose policies. The formal definition of an L1 policy as a Haskell data structure is:

```
data BGPPolicy = BGPPolicyAtomic Action
               | BGPPolicyConditional L1Match
                   BGPPolicy BGPPolicy
               | BGPPolicySequence [BGPPolicy]
               | BGPPolicyId ID
```

BGPPolicyAtomic represents a protocol action such as set local preference, set next hop, or delete community. Typically, actions are executed only under specific conditions. For this reason, *BGPPolicyConditional* allows an L1 policy to be represented as an IF-THEN-ELSE tree. The condition statement is a predicate match on route attributes and the two policies are the THEN and the ELSE branches of the tree, respectively. *BGPPolicySequence* enables a policy to be a forest of policy trees, a list of atomic actions, or a mix of both. Instead of using the definition of a policy in a *BGPPolicySequence* or a *BGPPolicyConditional*, we can use its *QualifiedID*. *BGPPolicyId* is the constructor of the data structure which makes legal the use of a *QualifiedID* in the place of a *BGPPolicy*. Note that we keep a hash table mapping *QualifiedID*s to definitions.

C. L0 to L1 Translation

L1 makes the control flow explicit by eliminating control structures that are specific to a given configuration language, like `Next` which is similar to a `goto`. For this reason, an L1 representation is typically longer, but easier to follow. The following is the L1 equivalent of the L0 policy given in Section II-A:

```
l1 = BGPPolicyConditional (convertMatch m1) p1 p2
p1 = BGPPolicySequence [BGPPolicyAtomic a1,
                        BGPPolicyAtomic a2,
                        com]
p2 = BGPPolicyConditional (convertMatch m2)
     (BGPPolicySequence [BGPPolicyAtomic DenyAction])
     com
com = BGPPolicyConditional (convertMatch m3)
      (BGPPolicySequence [BGPPolicyAtomic b])
      (BGPPolicyAtomic NullAction)
```

Fig. 2 illustrates the control flow of policy *l1*. In this example, condition *m1* is first checked. If it is true, *p1* is executed (otherwise, *p2*). Policy *p1* is a list of three policies, two of which are atomic. *p2* is itself an if-then-else policy. Another if-then-else tree, *com*, is the last policy of the *p1* sequence and in the else branch of *p2*. Function *convertMatch* translates an L0 match statement into an L1 predicate. These predicates can be on a prefix, an AS path, or a community. *NullAction* means that the route is accepted; *DenyAction* that it is rejected.

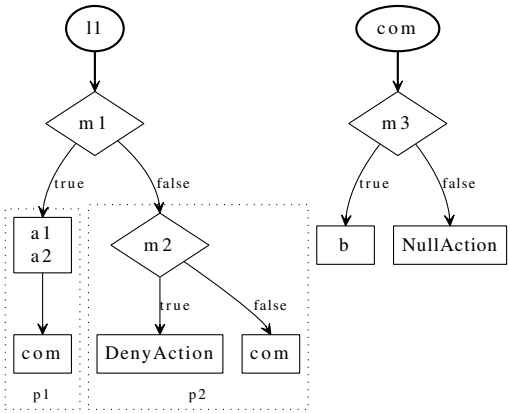

Fig. 2: Control flow of example policies *l1* and *l0*. Compared to L0, L1 makes the flow explicit by eliminating structures like *Next*.

References

[1] Y.-J. Chi, R. Oliveira, and L. Zhang, "Cyclops: the AS-level connectivity observatory," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 5–16, September 2008. [Online]. Available: http://doi.acm.org/10.1145/1452335.1452337

[2] L. Gao, "On inferring autonomous system relationships in the Internet," *IEEE/ACM Transactions on Networking*, vol. 9, no. 6, 2001.

[3] Z. Mao, L. Qiu, J. Wang, and Y. Zhang, "On AS-level path inference," in *Proc. of ACM SIGMETRICS*, June 2005.

[4] University of Oregon RouteViews project, http://www.routeviews.org/.

[5] R. Bush, O. Maennel, M. Roughan, and S. Uhlig, "Internet optometry: assessing the broken glasses in internet reachability," in *Proceedings of Internet Measurement Conference (IMC)*, 2009.

[6] M. Roughan, W. Willinger, O. Maennel, D. Perouli, and R. Bush, "10 Lessons from 10 Years of Measuring and Modeling the Internet's Autonomous Systems," *IEEE JSAC, Special Issue on "Measurement of Internet Topologies"*, vol. 29, 2011.

[7] T. G. Griffin and G. Huston, "BGP wedgies," RFC 4264, http://tools.ietf.org/html/rfc4264, 2005.

[8] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *2nd Symp. on Networked Systems Design and Implementation (NSDI)*, May 2005, http://www.gtnoise.net/projects/monitoring-diagnosis/18-rcc.

[9] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting EDGE of IP router configuration," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 21–26, 2004.

[10] A. Feldmann, "Netdb: IP network configuration debugger/database," Technical Report, AT&T Labs-Research, 1999.

[11] D. Perouli, T. G. Griffin, O. Maennel, S. Fahmy, C. Pelsser, A. Gurney, and I. Phillips, "Detecting Unsafe BGP Policies in a Flexible World," in *Proc. of IEEE ICNP, To Appear*, 2012.

[12] D. Perouli, S. Vissicchio, A. Gurney, O. Maennel, T. G. Griffin, I. Phillips, S. Fahmy, and C. Pelsser, "Reducing the Complexity of BGP Stability Analysis with Hybrid Combinatorial-Algebraic Models," in *WRiPE, To Appear*, 2012.

[13] T. Griffin, "The stratified shortest paths problem (invited paper)," in *Proc. of COMSNETS*, 2010.

[14] "Information on the haskell programming language." [Online]. Available: http://www.haskell.org/