NetPiler: Reducing Network Configuration Complexity through Policy Classification

Sihyung Lee Tina Wong Hyong S. Kim

June 29, 2007
CMU-CyLab-07-009

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

NetPiler: Reducing Network Configuration Complexity through Policy Classification

Sihyung Lee Tina Wong Hyong S. Kim
ECE Department and CyLab
Carnegie Mellon University
Pittsburgh, USA
{sihyunglee, tinawong}@cmu.edu, kim@ece.cmu.edu

***Abstract*— As a network evolves over time, multiple operators modify the network configuration, without fully considering what has been done previously. Similar policies are defined more than once, and policies that become obsolete after a transition are left in the configuration. As a result, the network configuration becomes complicated and disorganized, escalating maintenance costs and operator faults. We present a method called NetPiler that groups common policies by discovering a set of shared features and that uses the groupings instead of using each individual policy for the configuration. Such an approach removes redundancies and simplifies the configuration while preserving the intended behavior of the configuration. We apply NetPiler to the routing policy configurations from four different networks, and reduce more than 50% of BGP communities and the related commands. In addition, we show that the reduced community definitions are sufficient to satisfy the changes as the network evolves over almost two years.**

I. INTRODUCTION

Network configuration is a low-level, device-specific task. To configure a network, one needs to configure each device in the network separately. There can be hundreds of devices, thus hundreds of configuration files, each with thousands of commands. A change in one file can potentially affect other devices, or even the whole network. Often, multiple files need to be modified to make a relatively minor change in the network. In general, network configuration is complex as there are subtle dependencies in different parts of a single file. These dependencies are spread across files of multiple devices, even in a small sized network.

As a network evolves, its configurations become difficult to understand and to debug. Patches are sometimes put into configuration files to temporarily deal with a problem, and they are forgotten and left in place afterwards. Old configurations often remain to ensure the network operation works if the transition is incomplete. Configurations are edited by multiple operators with different backgrounds and working styles. Networks are also merged into a single network, complicating the combined configurations. Also, because of the low-level nature of configuration commands, the same high-level goal can be achieved in various ways in the configurations. In other words, both technical and non-technical issues can degrade the quality of a network's configuration over time.

As a result, operator errors are common and can account for more than 50% of failures in computer systems and networks [1,2,3]. Scheduled maintenance and

upgrades can account for more than 30% of network outages in Tier-1 ISPs [4]. Companies are spending more resources on the daily management and operations of their networks than on new IT services. In fact, one study has found that 80% of IT budgets in enterprise networks are used just to maintain the current operating environments [5].

There have been attempts to solve these problems. [8,9] identify potential errors in configurations, by comparing to a list of rules that are either data-mined or hard-coded. [10,11,12] propose configuration languages that are designed to be high-level, thus allowing network operators to specify their intent and translating the high-level intent into low-level implementation of configuration commands. Several researchers propose manageable network architectures [13,14,15]. Such architectures simplify management by having a central decision plane to avoid unnecessary division of configurations across a network.

In our approach, we transform the network configuration, specified either in high-level or low-level languages, into a more manageable, understandable, and evolvable configuration. In this procedure, we extract the underlying functions and dependencies from a network configuration and put it into a concise and system-independent format by reducing any redundancy. From this format, we generate a new configuration that illustrates complex inter-device and intra-device dependencies. Our approach is similar to optimizations done by a compiler of a programming language. We remove unnecessary parts that are not being used and reuse functionally equivalent parts. While doing so, we take into account the dependencies among different parts and preserve the function of the configuration. However, note that our main concern is not to optimize the length of a configuration, but to make it more understandable and manageable.

Our contributions are summarized as follows.

- We propose NetPiler, a way to find unique clusters of elements that share certain properties (or implement common functions) in a network configuration (Section II.B). Description by element groups can simplify the network configuration, independent of the description language. Even in the case where grouping is already used to configure a particular element, NetPiler can remove redundancies and simplify the grouping into unique sets. The grouping is based on our representation of elements and their properties in a graph model that captures dependencies between elements in the network configuration.
- We apply this technique on routing policy configurations to demonstrate NetPiler (Section III). We evaluate the algorithm on four national/regional providers, and we are able to reduce 90% of communities and 70% of the related commands for the best cases. Reducing the number of communities and the configuration size does not necessarily mean that the new set of communities is more meaningful and understandable. Thus, we go over a few reduction types (Section IV.C) and show that such simplification does improve the readability of the configuration. Furthermore, to show how NetPiler can be used to evolve a network's configuration in a more meaningful and maintainable way, we perform a temporal analysis on network configurations over two years to illustrate that the new set of communities can be reused in future configurations. Finally, we present a few ways to improve the algorithm (Section V).

We can apply NetPiler to any configuration language, but the configuration that results from NetPiler needs a language that allows specification of groups and

hierarchies as in object-oriented programming. This is partially supported in some configuration languages (e.g., `group` command in JUNOS [22]). Although NetPiler can be used in many different parts of a network configuration where their policies can be grouped into distinct sets, we illustrate the use of the method in restructuring BGP communities in routing policy configurations throughout the paper. This is partly because the BGP community is a widely used feature to group routes that have common properties. In addition, routing policy is one of the largest and the most complex aspects of a network configuration that has a number of dependencies within and between different components of a network. We believe the application in routing policy would better illustrate the benefits of our method. We present applications to other parts of a network configuration in Section II.C.

II. NETPILER

We first present an overview of NetPiler and show how we simplify a configuration for inter-domain routing policies and BGP communities in Section A. We describe the details of NetPiler in Section B and its applications in Section C.

A. Overview

We perform the following steps to transform a network configuration into another form. We first select the element in the configuration that is subject to the transformation. The element can be anything that can be grouped, ranging from an interface to a packet-filter. We then parse the configuration with regard to the element and construct a graph model. The model is a bipartite graph with two partite sets, the set of instances I and the set of their properties P. An instance $i \in I$ is joined by an edge with a property $p \in P$ iff i has p. We use a graph model instead of simple sets since each instance can have multiple properties such that some of the properties are properties of other instances as well. A graph is easier and more natural to represent the overlapping nature of the relationships. From the model, we identify distinct groups of instances that share common properties. Group A is comprised of a set of properties P_A that characterize the group, and a set of instances I_A, each of which has all the properties in P_A. For example, we may identify a group of interfaces i_1, i_2, i_3 that enable OSPF and have OSPF area 3, and another group i_4, i_5, i_6 that disable OSPF. If we denote the two groups A and B, $I_A=\{i_1, i_2, i_3\}$, $P_A=\{OSPF\_enabled, OSPF\_area\_3\}$, $I_B=\{i_4, i_5, i_6\}$, and $P_B=\{OSPF\_disabled\}$. The two property sets show two distinct policies associated with interfaces. Finally, we generate a new configuration that uses the groups in the specification. Note that the transformation is function-preserving in the sense that we do not alter the intended behavior of the configuration although we simplify its specification.

Before we go into more detail, we start with a fictional scenario to illustrate what the scheme can do. The scenario includes routing policy configurations using the BGP community. We first present the background of inter-domain routing policies and the BGP community, and then the scenario.

1) Inter-domain Routing and BGP communities

The BGP is a *de facto* standard inter-domain routing protocol. BGP advertises reachability to certain destinations between ASes. Such advertisement is selective in that only a subset of route advertisements received from an AS is distributed to other ASes. This is done mainly to implement a business relationship or to engineer traffic between ASes [16]. The selection of routes works by applying a route filter on the BGP

session to/from the AS. A route filter has a structure similar to the "if-then-else" chain in programming languages. It has a set of conditions followed by actions. It has a set of conditions followed by actions. Note that the exact syntax of the commands varies across different vendors, but we use the vendor independent form of if-then-else. For further details, we refer readers to [27], which gives an excellent overview of the BGP.

A BGP community refers to a group of routes that share certain properties and thus the same action is applied to the community. A community is encoded as a 32-bit field. A community influences the selection of routes by having its 32-bit string tagged to the set of route advertisements that belong to the community. If the 32-bit string matches the condition of a route filter, the required action is performed. A community implements a routing policy, which is in general described by a 3-tuple, (description of a set of routes, actions to be taken on the routes, a set of local/remote locations for the actions). For example, (prefix 1.1.0.0/16 received from neighbor 1.1.1.1, re-advertise, outbound session to neighbor 2.2.2.2) means that we want prefix 1.1.0.0/16 received from an inbound BGP session with neighbor 1.1.1.1 to be re-announced to neighbor 2.2.2.2. To implement the policy using a community, a community *C* is added to the prefix 1.1.0.0/16 by a route filter that is applied in the inbound direction of BGP sessions with neighbor 1.1.1.1. The route filter has a condition "**if** prefix equals 1.1.1.0/16", and an action "**add** *C*". Another router filter in the outbound direction with neighbor 2.2.2.2 will filter the routes by a condition "**if** there exists community *C*", and an action "**then** announce." There is a variety of community usages and more details can be found in [17,18,19].

2) Overview Example

In the scenario, we show how a network configuration becomes convoluted as communities are added and replaced ad hoc, and how we reduce the complexity. In Fig. 1(a), the oval in the center represents the network that we administer. Smaller ovals, each represents a neighboring network. Between each neighboring network and the center oval, there is a line indicating that there exists a BGP session between the two networks. A rectangle denotes a route filter applied to a BGP session and has the following format: "neighbor id/direction/" followed by an if-then-else chain. A capital letter is a community. Every if-then-else chain has an implicit deny action at the end. For example, in Fig.1(a), the left top rectangle is the route filter applied in the inbound direction of the BGP session with network *1*. Any route received from network *1* is configured to have community *A* attached. The right top rectangle is the route filter in the outbound direction toward network *4*. Routes with community *A* are to be announced to network *4*, and all other routes are not allowed by the default deny action. *P1* and *P2* each in Fig. 1(c) and (d) represents a certain collection of prefixes from network 2 and 3, respectively. The subfigures (a), (b), and (c) are in the order of evolution. We show only the first two figures and the last figure in the evolution because of space limitations.

- Initially, there are six neighboring networks and one community *A* is used to re-advertise routes from network 1, 2, and 3 to network 4, 5, and 6. For example, routes from network 6 do not have community A and thus are not advertised to network 4 and 5 (Fig.1(a)).
- The network establishes a peering relationship with three new networks, 7, 8, and 9. Community *B* is defined to re-advertise routes from 7, 8, and 9 to network 4, 5 and 6 (Fig.1(b)).

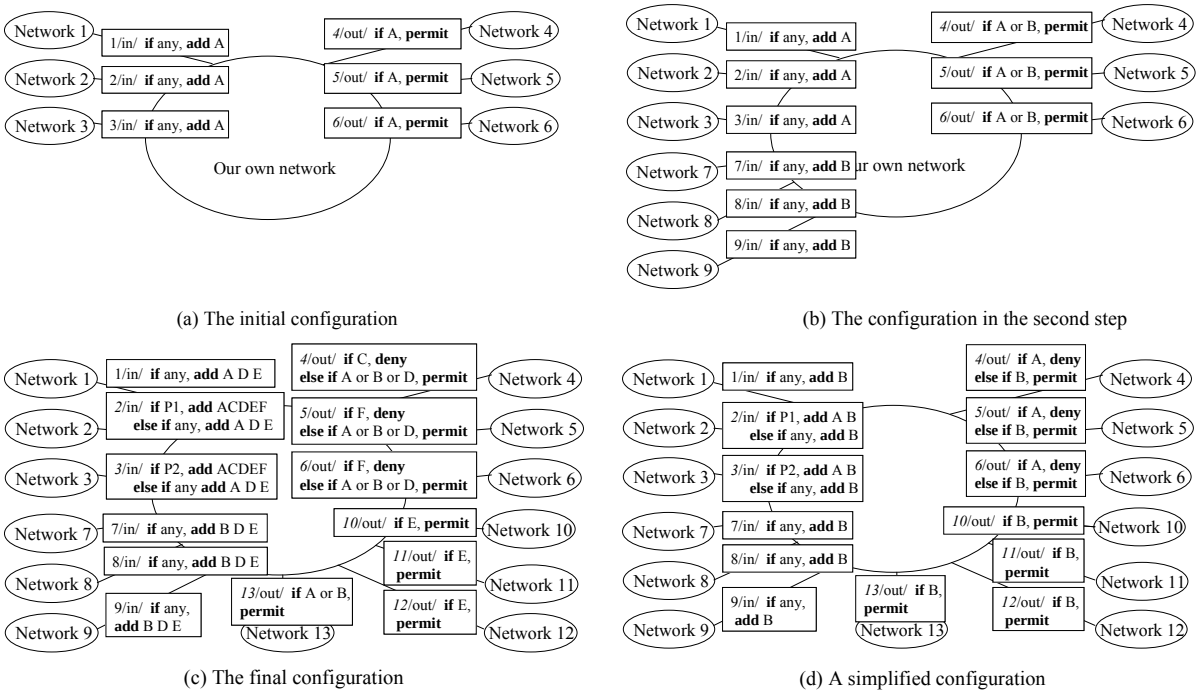


Figure 1. An example scenario on inter-domain routing and BGP community

- It is decided that IP prefix *P1* from network 2 and prefix *P2* from network 3 are not re-advertised to network 4. Community *C* is set on the IP prefixes and matched by a new outbound statement towards network 4 to deny the IP prefixes.
- There is a merger of networks and the operators decide to replace community *A* and *B* by a new community *D*. In the procedure, *A*s and *B*s remain in the configuration in order to prevent any malfunction while the migration is incomplete.
- Three new neighbors 10, 11 and 12 are added and a new community *E* is defined so that the three new neighbors receive routes advertised from network 1, 2, 3, 7, 8, and 9.
- A new neighbor session to network 13 is negotiated by a new operator. Without being aware of community *D*, the operator applies two old communities *A* and *B*.
- IP prefixes *P1* and *P2* from network 2 and 3, respectively, are no longer re-advertised to network 5 and 6 by a new community *F* (Fig.1(c)).

The resulting configuration in Fig.1(c) is much more complex than its initial form with a single community *A*. There are six communities, each of which forms a certain routing policy group (e.g., the group represented by community *F* is characterized by P_F=*{No advertise to network 5 and 6}* and has two members in I_F=*{prefix P1 from network 2, prefix P2 from network 3}*.). However, if we observe carefully, there exist two distinct policy groups, each of which can be implemented by a single community: i) *I={P1 from network 2, P2 from network 3}*, and *P={No advertise to network 4, 5, and 6}*. ii) *I={all other routes from network 1, 2, 3, 7, 8, and 9}*, and *P={Advertise to network 4, 5, 6, 10, 11, 12 and 13}*. The simplified configuration in Fig.1(d) is functionally equivalent to the intended policies. In other words, any route received from any neighbor will take the same action at any location as in Fig.1(c). Note that although

we reduce the configuration, our aim is not to minimize the length of the configuration, but to make it more meaningful and manageable.

B. Instance-Property Model and Decomposition

An element in a network configuration can be described with a set of properties associated with it. A property of the element can be either internal – properties that does not depend on other elements (e.g., interface type - Ethernet, Serial, tunnel, etc.), or external – properties that depend upon other elements (e.g., dependency upon other route filters at different locations). Our model captures such relationships between the element's instances and its properties in order to identify groups of instances sharing common properties and to simplify the configuration through grouping. We call this model *instance-property model*. In the model, a relation of an instance i having a property p is represented by two vertices i and p having an edge between them. In other words, our model is a bipartite graph with partite sets I, the set of instances, and P, the set of properties associated with the instances such that instance $i \in I$ is adjacent to a property $p \in P$ iff p characterizes i. Fig. 2(a) shows an instance-property model with five instances and seven properties. At the bottom, there is a configuration of instances i_1 through i_5. An instance is followed by ":" and its properties. Above the configuration, we show the corresponding instance-property model G. Instance i_1 has 4 properties, p_1, p_3, p_4, and p_7. In G, i_1 is incident with 4 edges that are joined with p_1, p_3, p_4, and p_7, respectively.

It is clear that an instance-property model can be described by listing each relation (i,p) represented by an edge (Fig. 2(a)). However, our goal is not to separate each single edge. We partition the edges into sets, such that each set represents a distinct group of instances that share certain properties as a unit. We call such a partition a *decomposition* of the model. Grouping similar objects and representing the objects by group improve the understandability and reusability. We define a group as follows. Group A is a nonempty set of properties P_A together with a set of instances $I_A = \{i | \forall p \in P_A, (i,p) \in E(G)\}$. G denotes the instance-property model and $E(G)$ its edge set. Since in A, every instance in one partite set I_A is adjacent to every property in the other partite set P_A, a group is equivalent to a *complete* bipartite graph. Thus, partitioning G into groups is the same as decomposing G into complete bipartite subgraphs. Fig. 2(b) presents a decomposition of G in Fig. 2(a) into 3 complete bipartite graphs (groups), A, B, and C. Below the graphs is the corresponding configuration using the 3 groups. If the instances are interfaces, then A, B, and C represent "external interface class," "interface class facing neighbor *N1*," and "interface class facing neighbor *N2*". In that case, i_2 belongs to both A and B. Such a membership is a single new group that inherits the properties from A with the addition of the properties from B. The decomposition of G is function-preserving: we do not add or delete any edges in G, and thus the intended behavior of the configuration does not change although its specification changes.

Note that there are many ways to decompose G into groups. For example, G is also decomposable into three groups A', B', and C' with their I and P sets as follows: $I_A' = \{i_1\}$, $P_A' = \{p_1, p_3, p_4, p_7\}$, $I_B' = \{i_2, i_4\}$, $P_B' = \{p_1, p_2, p_3, p_4, p_5, p_7\}$, $I_C' = \{i_3, i_5\}$, and $P_C' = \{p_1, p_3, p_4, p_6, p_7\}$. Of all possible decompositions, we look for the decomposition where each group is meaningful and thus manageable. We refer to a decomposition as being meaningful and manageable if a human operator can easily grasp the meaning of

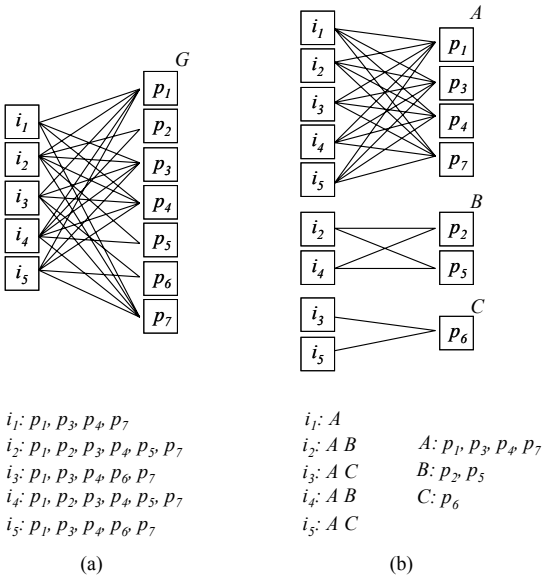


Figure 2. A decomposition of an instance-property model G (as shown in (a)) into complete bipartite subgraphs A, B, and C (as shown in (b)). The corresponding configurations appear below the graphs.

the groups – *I* and *P* sets and their relationships – and if he can reuse the groups to specify new instances or to modify existing instances with or without slight modification in the group definitions.

A meaningful decomposition for one element may not be meaningful for another element. Furthermore, one human operator may perceive a decomposition as meaningful while another may not. Thus, identification of a meaningful decomposition requires operator's input as well as domain knowledge about the instance. In Section III, we illustrate the way we find a meaningful decomposition, especially for inter-domain routing policy and the BGP community.

C. Applications of NetPiler

In this section, we investigate what aspect of a network configuration can be simplified by NetPiler. NetPiler is generally applicable to wherever there exists a notion of grouping. Even when grouping is not explicitly used, a number of elements can be grouped as in interfaces. Packet filters in a network can also be grouped into distinct sets of policies. There are cases where grouping is explicitly used with group ID. These cases include route tagging based on routing policies, packet marking/grouping based on QoS policies, and MPLS labeling based on destination prefixes/packet treatments. The instance set *I* could be a set of routes/packets, and the property set *P* could be a set of actions on the routes/packets and locations of the actions. NetPiler restructures the groups in a way that removes redundancies – equivalent groups or groups that are defined but never used – and makes the specification simpler and meaningful. We see an example of route tagging in Section III. However, identifying instances and properties is not always obvious. We need to find the minimum unit that comprises a group, and the properties that characterize the unit and a unique policy. In addition, this method might not be efficient when there is no common policy and each element is unique.

III. DEMONSTRATION WITH COMMUNITIES

A. Construction of the Instance-Property Model

At a high level, we construct the instance-property model for routing policies that are implemented by communities. We then decompose the model into groups such that each group represents a distinct routing policy as a unit and therefore is assigned a different community.

We identify an if-clause in a route filter as a vertex (both instance and property) of the instance-property model. If we think of a community in terms of a group defined in Section II.B, the members of the community (i.e. the instances of the community) are the routes tagged with the community. In a configuration, the routes are represented by sets of conditions in one or more route filters, possibly applied to different neighbors, such that each set is matched as a unit. One such set of conditions is equivalent to an if-clause. In Fig. 1(c), there are five if-clauses that represent instances of community *A*: i) all the routes from network *1*, ii) prefix *P1* form network *2*, iii) all other prefixes from network *2*, iv) prefix *P2* from network *3*, and v) all other prefixes from network *3*. The properties of the community are the dependencies on local/remote locations where the routes are matched and actions take place. Similarly in a configuration, the local/remote locations are represented by if-clauses that match the community. In Fig. 1(c), there are three if-clauses that match community *A*, and they are applied outbound to network *4*, *5*, and *6*.

The edges of the instance-property model, relationships between instances and properties, are identified as follows. There is an edge between an if-clause *i* and another if-clause *p* if the routes represented by *i* are matched by *p* via communities (i.e. if the communities attached by *i* match the condition in *p*). For example in Fig. 1(c), the routes received from network *7* have community *B*, *D*, and *E* attached by the if-clause "`7/in/ if any, add B D E`," and match the if-clause "`10/out/ if E, permit`". Therefore, the two if-clauses are joined by an edge. For an edge *(i,p)*, routes matched by *i* flow through *p* and the actions specified in *p* are taken on the routes. Note that the meaning of a relationship *(i,p)* can differ depending on the application. In the next section, we identify distinct policy groups that are represented by the dependencies among if-clauses and assign a community to each routing policy so that the community is used in its associated if-clauses.

B. Identifying Distinct and Meaningful Policies

Once an instance-property model is obtained, there are many ways to decompose the model. Naive decomposition may lead to groups whose meanings are not clear and difficult to reuse. Thus we develop a condition for each group to be meaningful and understandable.

This condition is based on the observation that a routing policy described by a community generally involves a set of routes that require an action. For example, routes from all customers might be re-advertised to all the peers and providers. A few prefixes from some customers might be AS-prepended three times when re-advertised to other peers so that those routes are not preferred. Such different sets of routes are represented by instances in our model. Thus, in order to identify distinct sets of routes that cause certain actions in concert, we identify such sets of instances.

We formalize the algorithm in Fig. 3 and present an example in Fig. 4. In a policy model *G*, we go over each property p_j and figure out the set of instances A_j that are

i_j : j-th instance, p_j : j-th property
C_{ij} : A set of communities that are added in i_j
M_{pj} : Matching condition of property p_j in Boolean logic.
$G_{j,k}$: Policy model. $G_{j,k} = 1$ if i_j is adjacent to p_k. Otherwise, $G_{j,k} = 0$.
N : Number of new communities
c_j : j-th new community
I_j : A set of instances that adds c_j
P_j : A set of properties that match c_j
$h(\ )$: Hash function associated with a hash table. If $h(A) > 0$, set A is present in the hash table and $h(A)=j$ where $I_j = A$. Otherwise, $h(A) = 0$.

Empty the hash table.
$N = 0$;
for each property p_j
    $A = \varphi$;
    for each instance i_k
        if $G_{k,j} = 1$ **then** $A = A \cup \{i_k\}$;
    if $h(A) = 0$ **then** { // create a new community
        $N = N + 1$; $h(A) = N$;
        $I_N = A$; $P_N = \{p_j\}$;
    } **else** $\{P_{h(A)} = P_{h(A)} \cup \{p_j\}$;}

Figure 3. Algorithm that identifies distinct policies based on the come-from relationship.

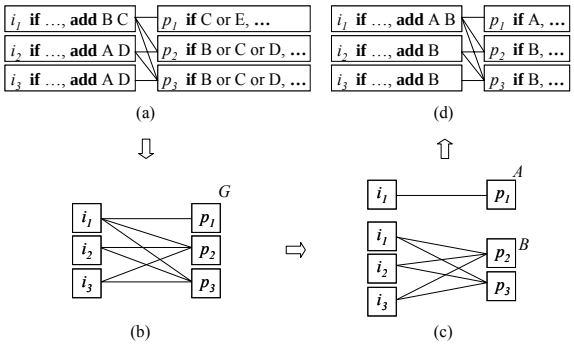


Figure 4. An example of routing policies (as shown in (a)), the corresponding instance-property model (as shown in (b)), decomposition by the come-from relationship (as shown in (c)), and the reproduced routing policies (as shown in (d)).

adjacent to p_j. A_j represents the routes that match the condition of p_j and thus are subject to the same action as described in p_j. Among all such sets, we draw distinct sets, I_1 through I_N. These sets represent distinct sets of routes that take the same action. Each I_j has its counterpart P_j, $\{p_k: A_k=I_j\}$. For each pair (I_j, P_j), all the edges between (I_j, P_j) belong to the same group and thus are assigned to the same community. In Fig. 4, there are two distinct I_j's, $I_1=\{i_1\}$ and $I_2=\{i_1, i_2, i_3\}$. The corresponding P_j's are $P_1=\{p_1\}$ and $P_2=\{p_2, p_3\}$. The two routing policy groups use community A and B, respectively. The edges in the original configuration (a) and the reproduced configuration (d) are the same and thus the transformation is function-preserving. Note that each community (group) in the reproduced configuration has a consistent meaning. In fact, a community represents a "*come-from*" relationship: routes that come-from I_j take certain actions in P_j as a unit.

Although we find that the majority of communities observe the come-from relationship, this condition might not give the most meaningful decomposition for every policy.

TABLE I. SUMMARY OF ANALYSIS

Index	Num. communities		Num. LOC	
	Before	***After***	***Before***	***After***
1	293 (113)	8	9003 (8419)	2036
2	43 (4)	4	282 (184)	194
3	45 (14)	10	2756 (1443)	1409
4	11 (4)	4	227 (126)	126

Network 1 and 2 are regional providers, and Network 3 and 4 are national providers. The number of routers are (44, 6, 13, 11) and the number of peering relationships (distinct external neighboring ASs) are (133, 39, 414, 77). The numbers in parentheses represent the numbers excluding dangling communities (as shown in Section IV.C) that do not create any edge.

IV. EVALUATION

We implement and evaluate our algorithm for communities on configurations from four different production networks. For each network, we analyze a particular snapshot between March and April 2006. Additionally for network *1* and *2*, we analyze monthly snapshots for two years to see if communities generated by our algorithm for the first snapshot could be reused over time. As shown in Table I,

- We reduce up to 90% of communities and 70% of community related commands. If we do not consider communities that do not create any edge (Section C), no reduction is possible for two networks because there is a simple set of policies, their communities are well structured, or there have not been many changes.
- More than 70% of the communities are defined by the come-from policy. There are a few exceptions and we address them in Section V.
- Most new communities are shown to be meaningful and reusable.

We describe implementation/experimental details in Section A and complexity measures to evaluate the effectiveness of our method in Section B. We then present the details of our results in Section C.

A. Experimental Setup and Implementation

First, we focus on simplification and restructuring of internal BGP communities within one administrative domain. We do not consider communities that are intended for use by external networks. However, this idea can be extended for multiple domains in the same way. In addition, predefined standard communities such as no-export and no-advertise are not subject to our simplification process.

Our implementation uses a configuration parser [6] developed for Cisco IOS and Juniper JUNOS commands. We parse routing policies related to communities and separate if-clauses into instances/properties in the format shown in Fig. 4(a). A property has a condition in Boolean logic since communities are matched based on Boolean operations (AND/OR/NOT). An instance has a list of communities attached by its corresponding if-clause. Although a community can be deleted as well, for simplicity we consider only the addition of communities. In the configurations from the four networks, we find that deletion of communities, which is rarely used, is only used to remove certain communities on routes received from/advertised to external networks. Therefore, deletion of such communities does not influence the operations of communities used within the administrative domain.

Fig. 5 shows an instance-property model representation for a configlet of Cisco IOS. In line 1, `neighbor 1.1.1.1` defines a BGP speaking neighbor with IP address 1.1.1.1. `route-map from_dora in` applies a route-filter named `from_dora` in the inbound direction of the BGP session with the neighbor. The route-filter `from_dora` is defined from line 4

Configuration in Cisco IOS syntax:

```
1   neighbor 1.1.1.1 route-map from_dora in
2   neighbor 2.2.2.2 route-map to_toto out
3
4   route-map from_dora permit 10
5    match community LIST1
6    set community 1:200 1:300
7   !
8
9   route-map to_toto permit 10
10   match community LIST2
11   set community 3:500
12  !
13
14  ip community-list standard LIST1 deny 2:444
15  ip community-list standard LIST1 2:100
16  ip community-list standard LIST2 1:200 1:300
17  ip community-list standard LIST2 1:400
```

Instance-Property Model Representation:

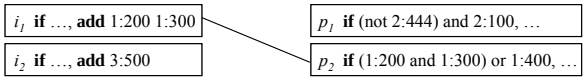


Figure 5. BGP configuration in instance-property model representation.

to line 7. The if-then clause is comprised of conditions followed by `match` and their associated actions followed by `set`. If the conditions are met on a route, the actions take place on the route and `permit` in line 4 allows the route to be redistributed. The number `10` after `permit` indicates the order of the if-then clause in a list of if-then clauses with the same route-filter `from_dora`. In line 5, the match condition is a community list named `LIST1`, defined in Line 14 and 15. The communities in LIST1 are compared with the route in the order they appear. In line 14, `deny 2:444` is a negation of community `2:444`. Thus, `LIST1` matches the route if the route does not contain community `2:444`, but contains `2:100`. Similarly, there is another `neighbor 2.2.2.2` with a route-filter `to_toto` applied in the outbound direction. The route-filter matches a community list `LIST2`. Two communities that appear on the same line means a logical AND. Thus, `LIST2` matches a route if the route contains both `1:200` and `1:300`, or `1:400`. Below the configlet, we show its instance-property model. Instance i_1 and property p_1 represent the route-filter `from_dora`, whereas i_2 and p_2 represent `to_toto`. The edge (i_1, p_2) indicates that routes redistributed through `from_dora` will match `to_toto`. Juniper JUNOS configuration has a similar condition-action structure. An if-then clause is comprised of `from` conditions and `then` actions nested in one BGP policy term.

Note that an edge can also be created by other types of filters based on prefix or AS-path attributes. In that sense, this model can be extended to a layered model with edges that are made from communities, elaborated by conditions involving prefixes, AS-path, and so forth. We do not consider such additional conditions in this paper. However, we preserve the edges made from communities while we reassign communities. Therefore, the edges will be correctly elaborated by other conditions that are not considered in this process, and the underlying routing policies are left intact.

Although the example shows communities matched by their exact values, communities can also be matched by regular expressions. We expand a regular expression by listing communities that match the regular expression in logical OR. In other words, if S is a set of communities that we consider for our clean-up process, and

Set

Network 1/ in/ 1	Network 8/ in/ 1	Network 6/ out/ 1
Network 2/ in/ 1	Network 9/ in/ 1	Network 6/ out/ 2
Network 2/ in/ 2	Network 4/ out/ 1	Network 10/out/1
Network 3/ in/ 1	Network 4/ out/ 2	Network 11/out/1
Network 3/ in/ 2	Network 5/ out/ 1	Network 12/out/1
Network 7/ in/ 1	Network 5/ out/ 2	Network 13/out/1

Match

Network 1/ in/ 1	Network 8/ in/ 1	**Network 6/ out/ 1**
Network 2/ in/ 1	Network 9/ in/ 1	Network 6/ out/ 2
Network 2/ in/ 2	**Network 4/ out/ 1**	Network 10/out/1
Network 3/ in/ 1	Network 4/ out/ 2	Network 11/out/1
Network 3/ in/ 2	**Network 5/ out/ 1**	Network 12/out/1
Network 7/ in/ 1	Network 5/ out/ 2	Network 13/out/1

(a) A diagram for the new community *A*

Set

Network 1/ in/ 1	Network 8/ in/ 1	Network 6/ out/ 1
Network 2/ in/ 1	Network 9/ in/ 1	Network 6/ out/ 2
Network 2/ in/ 2	Network 4/ out/ 1	Network 10/out/1
Network 3/ in/ 1	Network 4/ out/ 2	Network 11/out/1
Network 3/ in/ 2	Network 5/ out/ 1	Network 12/out/1
Network 7/ in/ 1	Network 5/ out/ 2	Network 13/out/1

Match

Network 1/ in/ 1	Network 8/ in/ 1	**Network 6/ out/ 1**
Network 2/ in/ 1	Network 9/ in/ 1	Network 6/ out/ 2
Network 2/ in/ 2	Network 4/ out/ 1	Network 10/out/1
Network 3/ in/ 1	Network 4/ out/ 2	Network 11/out/1
Network 3/ in/ 2	**Network 5/ out/ 1**	Network 12/out/1
Network 7/ in/ 1	Network 5/ out/ 2	Network 13/out/1

(b) A diagram for the original community *F*

Figure 6. Diagrams for new and original communities.

R is a regular expression, the matching condition is "if $\vee$(S$\cap$R)." For example, if S={1:100, 1:101, 1:200} and R=1:10(0|1), the matching condition is "if 1:100 or 1:101."

There are ways to execute multiple if-clauses in series, such as `continue` in IOS, and `next term` and `next policy` in JUNOS. However, we adhere to our representation of one if-then clause. If multiple if-clauses are executed together at all times, our algorithm will correctly identify them as belonging to the same group.

Finally, we use the following methods to see what a new community stands for. For each new and old community, we display in which if-clause the community is set and matched. In Fig. 6, a rectangle denotes an if-clause of the scenario in Fig.1. An if-clause has the following format: "neighbor id (network name or AS number) / direction/ sequence number". Below each of "Set" and "Match" are all 18 if-clauses. A bold-faced if-clause below "Set" adds the community if the condition of the if-clause is satisfied. A bold-faced if-clause below "Match" matches the community in the condition of the if-clause. The position of each if-clause is fixed, and therefore it is easy to compare two communities regarding which if-clause sets and matches the communities. For example, Fig. 6(a) represents the new community *A* in Fig. 1(d), and Fig. 6(b) the original community *F* in Fig. 1(c). The function of *A* subsumes that of *F* since all the edges created by *F* is covered by *A*. When the number of if-clauses is large, we can abstract the display. For example, we can merge if-clauses applied to the same AS into one rectangle. In addition, we give information on whether we can apply any of the methods in Section V to improve the understandability of communities based on the come-from relationship.

B. Complexity Measures

We use two measures, the number of communities and the number of LOC (Lines of Commands). The number of communities measures the total number of distinct communities that are used internally within a network. This is analogous to the vocabulary size [20], a software complexity measure that counts the number of unique operators and operands. It reflects the size of search space when writing or reading a command. As it becomes larger, the operator has to consider and compare more options to configure a community, and the configuration becomes a more complex task. The number of LOC is the sum of the number of individual communities used in conditional/action clauses. This also has its counterpart, which counts the number of

individual commands in software. The more places an operator needs to configure, the more chance to make mistakes. This is especially true when we configure communities, each of which can have dependencies and can impact tens to hundreds of BGP sessions. Furthermore, our previous work [6,7] finds a number of community-related errors, such as missing communities, using wrong communities and typos in a network where each if-clause consists of more than five communities on average. Although different communities might differ in terms of effort to configure and the number of mistakes, we use these simple definitions because multiple studies validate their correlation with the number of faults and development/maintenance time [20,21].

C. Results

Table I shows decreases in both the number of communities and the number of LOC. The decrease is noticeable in Network *1* since it has been expanding as more networks are added over the span of two years we studied and thus has gone through many changes in the past. Reducing such measures means that the resulting configuration files are shorter, but might not necessarily mean that the new set of communities is meaningful and understandable. To answer this question, we examine the configurations result of our scheme and identify how communities have changed in the following subsections. Note that some redundancies are by design, and operators can always keep certain original communities from being restructured. The operators can either exclude the original communities from the analysis, or accept only a subset of the new communities.

Each of the new communities either is equivalent to an original community, or represents policies implemented by multiple communities in the original configuration. Some of the new communities implement business relationships among transits, peers, and customers, and others implement policies intended for traffic engineering. These are common relationships found in a network, and configurations concerning communities thus naturally can be reduced according to the unique units of these relationships in a network.

Dangling Communities. The majority of communities that are removed by our algorithm (180, 39, 31, and 7 communities from network 1, 2, 3, and 4, respectively) are either added in some if-clause but never matched anywhere, or matched but never added. We call these communities dangling communities since they refer to a certain group, but do not form any edges in the instance-property model. These communities are remains of old configurations when peering relationships end. Others are defined by predicting later usage thus allowing operators to use the communities to deal with modification in peering relationships or unforeseen problems in the future. However, from our time-series analysis over a two-year period, we find that none of these communities had been modified for actual usage. These communities should be used only when they are needed. Lengthening the configuration with such communities might make the configuration harder to understand, maintain, and more prone to errors.

Subset Communities. A few communities are removed since their functions are subsumed by those of other communities. In other words, the edges created by each of the removed communities are a subset of the edges created by another community. In one network, particular routes are re-advertised to a peer based on the following matching condition.

if (*A* and *C1*) or (*A* and *C2*) or (*A* and *C3*) or …

Our algorithm detects that wherever A is attached, one of Ci's is attached as well and thus is able to simplify the condition as "**if** *A*." Furthermore, we find that some of the Ci's is always ANDed with A in conditional clauses, but is never used separately from A. Thus, not only the number of LOC is reduced, but the number of communities as well.

There are possible reasons why such communities exist: i) when communities are defined ad hoc, the dependencies created by communities and the policies implemented previously are not fully considered, and ii) communities that are replaced by others are not properly removed.

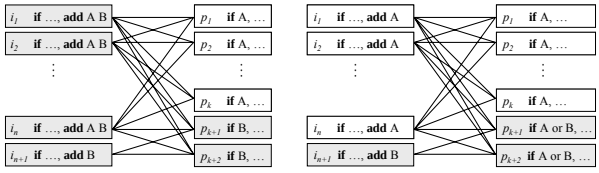
Combination of Communities. There are communities that can be combined although none of them are functionally subsumed by one another. Such communities either represent the same set of routes and match in different if-clauses, or involve different routes and match in the same if-clauses. For example, three communities are added by the same if-then clauses and thus represent the same set of routes. The communities are used so that the routes are not re-advertised to three different networks *1*, *2*, and *3*, respectively. Our algorithm combines the three communities as one by matching and adding a single community instead of the three. Such combining does not limit the flexibility of routing policies as long as we deal with the same set of routes. If we no longer need to prevent the routes from being advertised to network *2*, we can simply remove the single community from the corresponding if-clause.

Equivalent Communities. Each of the other new communities (3, 4, 7, and 4 communities from network 1, 2, 3, and 4, respectively) is equivalent to an original community. Although these communities do not contribute to the reduction, they do present an important implication as the combined communities. This implication is that the majority of routing policies comply with the come-from relationship. There are a few exceptions, which we deal with in Section V.

Time-series Analysis. Finally, even though transforming a current configuration into a concise form improves the readability, this does not guarantee that the number of communities and LOC evolves in the same way as new configurations are added in the future. Therefore, we perform an analysis on snapshots for two years (Network *1* and *2*). The result is encouraging, because it shows that configurations from a simple transformation can still be evolvable over time. During the period, the networks add and remove peering relationships periodically, and the overall number of relationships grows by roughly 25%. We find that the reduced set of communities is sufficient for this evolution. One or two communities are added and then deleted during the period to accommodate temporary peering relationships that require unique routing policies.

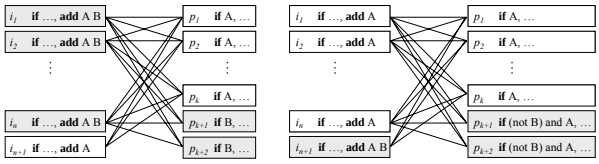
V. DISCUSSION

In this section, we first go over a few cases where the number of communities/LOC does not decrease when the new groupings reproduced by the come-from relationship do not agree with the groupings in the original configurations. Since we believe that the original groupings could be more meaningful, we present methods that restructure the new groupings into the original groupings to improve the come-from relationship. We then present other applications of our scheme.



(a) A come-from based grouping (b) A different partition with smaller sets

Figure 7. Specification with smllaer sets. The shaded instances and properties beong to community B. n, k ≫ 1.



(a) A come-from based grouping (b) A different partition with smaller sets

Figure 8. Specification with smllaer sets. The shaded instances and properties beong to community B. n, k ≫ 1.

Preference for Shorter Descriptions. The first case is related to the fact that a shorter description could be more intuitive than a longer one. For example, "All but routes from {network *1* and *2*} are to be advertised to customers." is more concise than "Routes from {network *3*, *4*, *5*, …, and *n*} are to be advertised to customers." The come-from relationship produces the latter grouping while the original configuration uses the former. Although the situation that we describe here is not common, when it happens, we observe a tendency towards using smaller *I* and *P* sets or fewer communities.

One case involves a large number of instances that are adjacent to a large number of properties and a few other instances that are adjacent to only a few of the properties. For example, most customer-originated routes are re-advertised to all the peers except a prefix that is re-advertised to only two of the peers. In Fig. 7(a), instance i_{n+1} represents the special prefix, and $\{p_{k+1}, p_{k+2}\}$ represents the two peers. The algorithm based on the come-from relationship identifies two distinct groups: i) $I_A=\{i_j|1\leq j\leq n\}$ and $P_A=\{p_j|1\leq j\leq k\}$, and ii) $I_B=\{i_j|1\leq j\leq n+1\}$ and $P_B=\{p_{k+1}, p_{k+2}\}$. However, the original configuration has two different groups: i) $I_A\text{´}=\{i_j|1\leq j\leq n\}$ and $P_A\text{´}=\{p_j|1\leq j\leq k+2\}$, and ii) $I_B\text{´}=\{i_{n+1}\}$ and $P_B\text{´}=\{p_{k+1}, p_{k+2}\}$ as shown in Fig. 7(b). Note that the new $I_B\text{´}$ set is much smaller, i.e. $|I_B\text{´}| \ll |I_B|$, but $\{p_{k+1}, p_{k+2}\}$ still receive routes from both $\{i_j|1\leq j\leq n\}$ and $\{i_{n+1}\}$ since $\{p_{k+1}, p_{k+2}\}$ now belongs to $P_A\text{´}$ as well as $P_B\text{´}$. Although the two groupings implement the same policy, the latter grouping uses a smaller set and thus has a fewer number of LOC. The shaded vertices use community *B*, and the number of LOC in Fig. 7(b) is n-2 (≫1) less than that in Fig. 7(a). This accounts for increase of LOC from 184 to 194 in network *2*. Two of the four communities represent the example policy and the latter grouping reduces the number of LOC by ten.

We also observe a similar grouping with smaller sets when the few instances have most of a large number of properties except a few properties. For example, most customer-originated routes are re-advertised to all the peers except a prefix that is *not* re-advertised to two of the peers. Fig. 8(a) illustrates the case. Note that the only difference from Fig. 8(a) is that the special prefix i_{n+1} belongs to community *A*, not *B*. Fig. 8(b) shows an alternative grouping. The shaded vertices use community *B*, and the

number of LOC in Fig. 8(b) is n-1 ($\gg$1) less than that in Fig. 8(b). $\{p_{k+1}, p_{k+2}\}$ do not receive routes from i_{n+1} through negating community *B* in their if-clauses.

The last case involves a community that can be specified by AND or OR of other communities. This idea is generally applicable to grouping any element in a network configuration where an instance/property set is an intersection/union of other instance/property sets. In one network, transits and peers receive different sets of customer routes, which produce two distinct policies and thus two communities *C1* and *C2*. There is a route collector to which both sets of customer routes are advertised. The come-from relationship identifies the latter case as the third separate policy, whereas the latter policy can actually be described as "*C1* or *C2*." Similarly, AND can be used for a policy which is an intersection of other policies.

Finer Decomposition Based on Actions. We can further partition the policies resulting from the come-from relationship in order to make their meanings clearer. Assume that a set of prefixes *P1* learned from external peers is either dropped or received a lower preference at two different remote route filters. The come-from relationship identifies the situation as one single policy, "come-from *P1*," since the prefixes always receive the same action as a unit. However, we can divide the policy into two policies: i) "come-from *P1* to be dropped," and ii) "come-from *P1* to receive a lower preference." Although operators can choose whichever way is more meaningful and convenient, if the latter is used, one can easily extend our algorithm so that come-from based policies are further partitioned according to the corresponding actions.

Misconfiguration Detection. While adding, removing, or modifying a configuration, an operator can make numerous mistakes, especially when dealing with communities that have a large number of dependencies. The operator may forget to add/delete a community or add/delete a wrong community. Such a mistake produces an additional policy distinct from the intended policies and thus stands out as a special policy by our algorithm. We can automate the detection by using data-mining techniques. For example, assume that the operator does not add a community to an if-clause in a filter outbound to a customer network, and as a results, the customer does not receive peer routes. Another community allows customer/transit routes to be announced to all the customers. *A priori* association rules mining on the set of instances A_j adjacent to each property p_j raises the A_j set for the customer as a violation of the rule "if customer/transits are in A_j, then peers are also in A_j". This method is semantic based on the instance-property model as opposed to the syntactic analysis. Thus, this method can get around false negatives/positives caused by specifications that are syntactically different but the same in their meanings.

VI. Related Work

There is a significant amount of work done to help simplify network management. We describe here the most relevant ones. [28] extracts routing policies from a network configuration and groups the policies with similar export and import characteristics. The goal is to display the routing policies to operators in a meaningful way, not to reproduce a simplified configuration. [10,11,12] propose high-level configuration languages for specific parts of a network configuration. NetPiler is language independent and thus can also simplify a high-level specification of policies if the policies can be grouped. Others have proposed new management architectures. The 4D architecture [13] has a central decision plane that pushes instructions to each router

including routing table/packet filter entries. In CONMan [14], protocol implementations expose a simple and consistent interface to the management plane. Even for these architectures, we believe that our method of transforming a configuration into a simpler form would make the configuration easier to maintain. [6,7,8,9] attempt to discover operator mistakes. A network configuration is compared with a list of predefined rules. These are useful in detecting certain types of errors, but do not remove the root causes of operator mistakes, such as the complexity or redundancies in the configuration.

[23,24] use directed graphs where an edge represents a dependency between network elements. The dependencies are used to trace the source of failures. NetPiler use dependencies as properties and group instances according to the properties to simplify the specification.

[16,17,18] study the utilization of the BGP community and categorize the usages. [19] proposes a structured way of defining communities. One of several predefined actions is encoded into a community value along with the BGP speakers associated with the action. This removes the need to manually encode the actions in route filters. NetPiler can complement this approach by reducing redundancies in the design.

Finally, our technique is similar to machine-independent optimization techniques used by compilers, such as dead-code and common-subexpression elimination [25,26]. Their transformation is also function-preserving, but is intended to speed up a program or to reduce the space taken by the compiled code, not to make the program easier to understand.

VII. CONCLUSION

We present NetPiler, a way to transform a network configuration into a simpler form, which is easier to read and update. NetPiler groups policies into a set of distinct policies, and thus removing any duplicate specifications and combining specifications that are unnecessarily decomposed. We demonstrate NetPiler for routing policies in four production networks, especially the policies implemented by the BGP community attribute, and show that up to 90% of communities and up to 70% of community-related commands are reduced. We also run NetPiler for snapshots over two years and show that the reduced set of communities can be reused and are sufficient for this evolution.

The transformation is independent of configuration languages and is applicable to any element in a network configuration if the element is configured with policies that can be grouped into finite sets. We are currently exploring the application of NetPiler to other elements of the network configuration. The first step is to include other constructs of routing policies, including prefix and AS-path based filters. We then plan to apply NetPiler to different types of elements. Packet marking/grouping is the most similar to route tagging. Packets are classified according to their QoS classes by using if-clauses (e.g., `policy-map` along with `class-map` is analogous to `route-map` in IOS) and are either marked (e.g., DSCP (Diffserv Code Point), IP precedence value, MPLS EXP field, Layer 2 CoS (Class of Service), etc.) or grouped in routing tables (e.g., QoS group in IOS). Different actions are applied to different classes of packets. Other types of elements to consider include packet filter, MPLS, and interface configurations.

REFERENCES

[1] R. Mahajan, D. wetherall, and T. Anderson, "Understanding BGP misconfigurations," in *Proc. ACM SIGCOMM*, Aug. 2002.

[2] D. Oppenheimer, A. Ganapathi and D. Patterson, "Why do Internet services fail, and what can be done about it?" in *Proc. USITS*, 2003.

[3] A. Wool, "A quantitative study of firewall configuration errors," *IEEE Computer*, June 2004.

[4] "Evaluating high availability mechanisms," Agilent Technologies White Paper, 2005.

[5] Z. Kerravala, "As the value of enterprise networks escalates, so does the need for configuraiton management," Enterprise Computing and Networking, Yankee Group, 2004.

[6] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Minerals: Using data mining to detect router misconfigurations," in *Proc. ACM SIGCOMM Workshop on Mining Network Data*, Sep. 2006.

[7] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Characterization and problem detection of routing policy configurations," CMU Technical Report, CMU-CyLab-06-010, 2006.

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

[9] A. Feldmann and J. Rexford, "IP network configuration for intradomain traffic engineering," *IEEE Network Magazine*, 2001.

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

[11] Y. Bartal, A. Mayer, K. Nissim and A. Wool, "Firmato: A novel firewall management toolkit," *ACM Transactions on Computer Systems*, 2004.

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

[13] A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM SIGCOMM Computer Communications Review*, vol. 35, no. 5, Oct. 2005

[14] Hitesh Ballani, Paul Francis, "CONMan: A step towards network manageability," in *Proc. ACM SIGCOMM*, Aug 2007

[15] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting EDGE of IP router configuration," in *Proc. HotNets-II*, 2003

[16] M. Caesar and J. Rexford, "BGP routing policies in ISP networks," *IEEE Network Magazine, special issues on inter-domain routing,* Nov/Dec. 2005.

[17] O. Bonaventure and B. Quoitin, "Common utilizations of the BGP community attribute," Internet draft, draft-bonaventure-quoitin-bgp-communities-00.txt, work in progress, June 2003.

[18] B. Quoitin and O. Bonaventure, "A survey of the utilization of the BGP community attribute," Internet draft, draft-quoitin-bgp-comm-survey-00.txt, work in progress, February 2002.

[19] B. Quoitin, S. Tandel, S. Uhlig, and O. Bonaventure, "Interdomain traffic engineering with redistribution communities," *Computer Communications Journal (Elsevier)*, vol. 27, no. 4, pp. 355-363, Mar. 2004.

[20] H. Zuse, Software Complexity: Measures and Methods, Berlin: Walter de Gruyter, 1991.

[21] S. Alexandrov, "Reliability of Compex Services," unpublished. http://www.cs.rutgers.edu/~rmartin/teaching/spring06/cs553/papers/

[22] JUNOS Configurations Guides.

http://www.juniper.net/techpubs/software/junos/junos83/index.html

[23] P. Bahl et al., "Discovering dependencies for network management," in *Proc. Hotnets*, 2006.

[24] Irene Katzela, and Mischa Schwartz, "Schemes for fault identification in communication networks," *IEEE/ACM ToN, vol. 3, no. 6*, Dec. 1995.

[25] A. V. Aho, R. Sethi and J. D. Ullman, Compilers Principles, Techniques, and Tools 2nd edition. Boston, MA: Pearson Addison-Wesley, 1986.

[26] S. S. Muchnick, Advanced Compiler Design and Implementation. San Francisco, CA: Morgan Kaufmann, 2005.

[27] J. W. Stewart, BGP4: Inter-Domain Routing in the Internet. Boston, MA: Pearson Assison-Wesley, 1998.

[28] K. Levanti, H. S. Kim, and T. Wong, "Intent-based analysis of network-wide routing policy configuration," in *Proc. ACM SIGCOMM Workshop on Internet Network Management*, Aug. 2007.