Network Configuration Validation[1]

Sanjai Narain, Rajesh Talpade, Gary Levin

Telcordia Technologies, Inc.
{narain, rrt, glevin}@research.telcordia.com

9.1 Introduction

To set up network infrastructure satisfying end-to-end requirements, it is not only necessary to run appropriate protocols on components but also to correctly configure these components. Configuration is the "glue" for logically integrating components at and across multiple protocol layers. Each component has configuration parameters each of which can be set to a definite value. However, today, the large conceptual gap between end-to-end requirements and configurations is manually bridged. This causes large numbers of configuration errors whose adverse effects on security, reliability and high cost of deployment of network infrastructure are well-documented. For example:

- "Setting it [security] up is so complicated that it's hardly ever done right. While we await a catastrophe, simpler setup is the most important step toward better security". – Turing Award winner Butler Lampson [42]
- "..human error is blamed for 50 to 80 percent of network outages". – Juniper Networks [40]
- "The biggest threat to increasingly complex systems may be systems themselves". – John Schwartz [61]
- "Things break and complex things break in complex ways". – Steve Bellovin [61]
- "We don't need hackers to break systems because they're falling apart by themselves". – Peter Neumann [61]

Thus, it is critical to develop validation tools that check whether a given configuration is consistent with the requirements it is intended to implement. Besides

[1] This material is based upon work supported by Telcordia Technologies, and Air Force Research Laboratories under contract FA8750-07-C-0030. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Telcordia Technologies or of Air Force Research Laboratories. Approved for Public Release; distribution unlimited: 88ABW-2009-3797, 27 Aug 09.

This material will appear as a chapter in the book "Guide to Reliable Internet Services and Applications" eds. Charles Kalmanek, Richard Yang and Sudip Misra, to be published by Springer in 2009.

checking consistency, configuration validation has another interesting application, namely, network testing. The usual invasive approach to testing has several limitations. It is not scalable. It consumes resources of the network and network administrators and has the potential to unleash malware into the network. Some properties such as absence of single points of failure are impractical to test as they require failing components in operational networks. A non-invasive alternative that overcomes these limitations is analyzing configurations of network components. This approach is analogous to testing software by analyzing its source code rather than by running it. This approach has been evaluated for a real enterprise.

Configuration validation is inherently hard. Requirements can be on connectivity, security, performance and reliability and span multiple components and protocols. A real infrastructure can have hundreds of components. A component's configuration file can have a couple of thousand configuration commands each setting the value of one or more configuration parameters. In general, the correctness of a component's configuration cannot be checked in isolation. One needs to evaluate global relationships into which components have been logically integrated. Configuration repair is even harder since changing configurations to make one requirement true may falsify another. The configuration change needs to be holistic in that all requirements must concurrently hold.

This chapter motivates the need for configuration validation in the context of a realistic collaboration network, proposes an abstract design of a configuration validation system, surveys current technologies for realizing this design, outlines experience with deploying such a system in a real enterprise, and outlines future research directions.

Section 9.2 discusses the challenges of configuring a realistic, decentralized collaboration network, the vulnerabilities caused by configuration errors and the benefits of using a validation system. Requirements on this network are complex to begin with. Their manual implementation can cause a large number of configuration errors. This number is compounded by the lack of a centralized configuration authority.

Section 9.3 proposes a design of a system that can not only validate the above network but also evolve to validate even more complex ones. This design consists of four subsystems. The first is a Configuration Acquisition System for extracting configuration information from components in a vendor-neutral format. The second is a Requirement Library capturing best practices and design patterns that simplify the conceptualization of end-to-end requirements. The third is a Specification Language whose syntax simplifies the specification of requirements. The fourth is an Evaluation System for efficiently evaluating requirements, for suggesting configuration repair when requirements are false, and for creating visualizations of logical relationships.

Section 9.4 discusses the Telcordia® IP Assure product [38] and the choices it has made to realize this design. It uses a parser generator for configuration acquisition. Its Requirement Library consists of requirements on integrity of logical structures, connectivity, security, performance, reliability and government policy.

Its specification language is one of visual templates. Its evaluation system uses algorithms from graph theory and constraint solving. It computes visualizations of several types of logical topologies.

Section 9.5 discusses logic-based techniques for realizing the above validation system design. Their use is particularly important for configuration repair. They simplify configuration acquisition and specification. They allow firewall subsumption, equivalence and rule redundancy analysis. These techniques are the languages Prolog, Datalog and arithmetic quantifier-free forms [53, 67, 51], the Kodkod [41] constraint solver for first-order logic of finite domains, the ZChaff [46, 27, 73] minimum-cost SAT solver for Boolean logic and Ordered Binary Decision Diagrams [12] .

Section 9.6 outlines related techniques for realizing the above validation system design. These are type-inference for configuration acquisition [47], symbolic reachability analysis [72], its implementation [3] with symbolic model checking [48], and finally, validation techniques for BGP, the Internet-wide routing protocol, and one of the most complex.

Section 9.7 contains a summary and outlines future research directions.

9.2 Configuration Validation For A Collaboration Network

This section discusses the challenges of configuring a realistic, multi enterprise *collaboration network*, the types of its vulnerabilities caused by configuration errors, the reasons these arise, and the benefits that can be derived from using a configuration validation system. Multiple communities of interest (COIs) are set up as logically partitioned virtual private networks (VPNs) overlaid on a common IP network backbone. The "nodes" of this VPN are gateway routers at each enterprise that participates in the COI. An enterprise can participate in more than one COI in which case it would have one gateway router for each COI. For each COI, agreement is reached between participating network administrators on the top-level connectivity, security, performance and reliability requirements governing the COI. Configuration of routers, firewalls and other network components to implement these requirements is up to administrators. There is no centralized configuration authority. The administrators at different enterprises in a COI negotiate with each other to ensure configuration consistency. Such decentralized networks exist in industry, academia and government and are clear candidates for the application of configuration validation tools.

Typical COI requirements are now described. The connectivity requirement is that every COI site must be reachable from every other COI site. The security requirement is twofold. First, all communication between sites must be encrypted. Second, no packets from one COI can leak into another COI. This requirement is especially important since collaborating enterprises have limited mutual trust. A site can be a part of more than one COI but the information that site is willing to

share with partners on one COI is distinct from that with partners in another COI. The performance requirement specifies the bandwidth, delay, jitter and packet loss for various types of applications. The reliability requirement specifies that connectivity be maintained in the face of link or node failure.

Since these requirements are complex, large numbers of configuration errors can be made. This number is compounded by the lack of a centralized configuration authority. The complexity has the further consequence that less experienced administrators, especially in an emergency, tend to statically route traffic directly over the IP backbone rather than correctly set up dynamic routing. But, when the emergency passes, static routes are not removed for concern of breaking the routing. Over time, this causes the COIs to become brittle in that routes cannot be automatically recomputed in the face of link or node failure.

While administrators are well aware of configuration errors and their adverse effects on the global network, they lack the tools to identify these, much less remove these. The decentralized nature of the network prevents them from obtaining a picture of the global architecture. A validation system that could identify configuration errors, make recommendations for repairing these and help understand the global relationships would be of immense value to administrators.

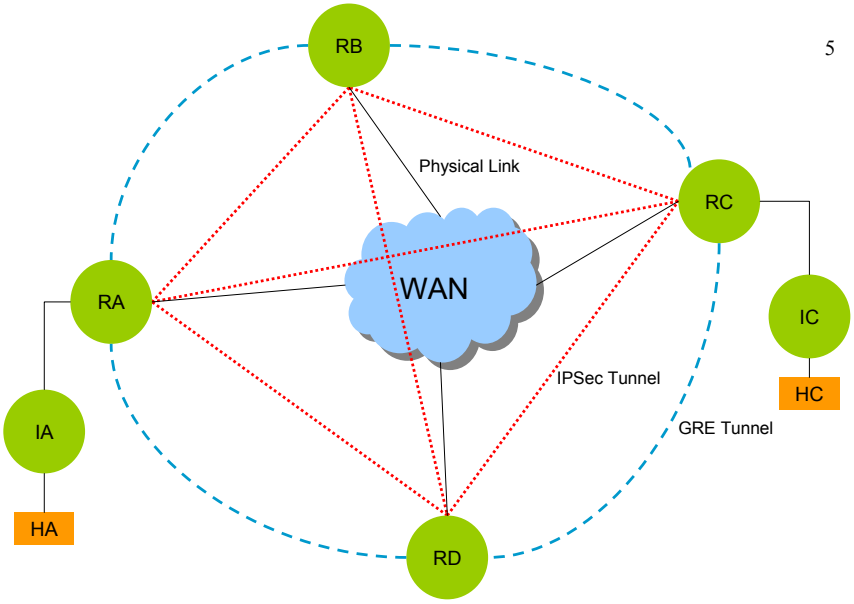

Figure 9.1 – Community of Interest Architecture

Figure 9.1 – Community of Interest Architecture shows the architecture of a typical COI with four collaborating sites A, B, C, D. Each site contains a host, an internal router and a gateway router. The first two items are shown only for sites A and C. Each gateway router is physically connected to the physical IP backbone network (WAN). Overlaid on this backbone is a network of IPSec [41] tunnels interconnecting the gateway routers. An IPSec tunnel is used to encrypt packets flowing between its endpoints. Overlaid on the IPSec network is a network of GRE [22] tunnels. A GRE tunnel provides the appearance of two routers being directly connected even though there may be many physical hops between them. The two overlay networks are "glued" together in such a way that all packets through GRE tunnels are encrypted. A routing protocol e.g., BGP [33, 36], is run over the GRE network to discover routes on this overlay. If a link or node in this network fails, BGP discovers an alternate route if possible. A packet originating at host HA destined to host HC is first directed by its internal router IA to the gateway router RA. RA encrypts the packet then finds a path to HC on the GRE network. When the packet arrives at RC it is decrypted, decapsulated and forwarded to IC. IC then forwards it to HC. All routers also run the internal routing protocol called OSPF [42]. OSPF discovers routes to destinations that are internal to a site. The OSPF process at the gateway router redistributes or injects internal routes into the BGP process. The BGP process then informs its peers at other gateway routers about these routes. Eventually, all gateway routers come to know about how to route packets to any reachable internal destination at any site.

In summary, connectivity, security and reliability requirements are satisfied by the use, respectively, of GRE, IPSec and BGP and OSPF. The security requirement that data from one COI not leak into another is satisfied implicitly. GRE reachability to a different COI is disallowed, static routes to destinations in different COIs are not set up, gateway routers at the same enterprise but belonging to different COIs are not directly connected, and BGP sessions across different COIs are not set up.

The performance requirement is satisfied by ensuring that GRE tunnels are mapped to physical links of the proper bandwidth, delay, jitter and packet loss properties, although this is not always in control of COI administrators. Avoiding one cause of packet loss, is however, in their control. This is the blocking of Maximum Transmission Unit (MTU) mismatch messages. If a router receives a packet whose size is larger than the router's configured MTU, and the packet's Do Not Fragment bit is set, the router will drop the packet. The router will also warn the sender in an ICMP protocol message that it has dropped the packet. Then the sender can reduce the size of packets its sends. However, since ICMP is the same protocol used to carry ping messages, firewalls at many sites block ICMP. The result is that the sender will continue to send packets without reducing their size and they will all be dropped by the router [68]. Packets increase in size beyond an expected MTU because GRE and IPSec encapsulations add new headers to packets. To avoid such packet loss, the MTU at all routers is set to some fixed value accounting for the encapsulation. Alternatively, ICMP packets carrying MTU mismatch messages are not blocked.

This design is captured by the following requirements:

Connectivity Requirements

1. Each site has a gateway router connected to the WAN.
2. There is a full-mesh of GRE tunnels between gateway routers.
3. Each gateway router is connected to an internal router at the same site.

Security Requirements

1. There is a full-mesh network of IPSec tunnels between all gateway routers.
2. Packets through every GRE tunnel are encrypted with an IPSec tunnel.
3. No gateway router in a COI has a static route to a destination in a different COI
4. No cross-COI physical, GRE, BGP connectivity or reachability is permitted.

Reliability Requirements

1. BGP is run on the GRE tunnel network to discover routes to destinations in different sites
2. OSPF is run within a site to discover routes to internal destinations.
3. OSPF and BGP route redistribution is set up.

Performance Requirements

1. MTU settings on all interfaces are set to be less than the expected packet size after taking into account GRE and IPSec encapsulation.
2. Alternatively, access control lists at each gateway router permit ICMP packets carrying MTU messages.

Configuration parameters that must be correctly set to implement the above requirements include:

1. IP addresses and mask of physical and GRE interfaces.
2. IP address of the local and remote BGP session end points and the autonomous system number of the remote end point.
3. Names of GRE interface and IP address of associated local and remote physical tunnel end points.
4. IP addresses of local and remote IPSec tunnel end points, encryption and hash algorithms to apply to protected packets, and the profile of packets to be protected.
5. Destination, destination mask and next hop of static routes.
6. Interfaces on which OSPF is enabled and the OSPF areas to which they belong.
7. Source and destination address ranges, protocols and port ranges of packets for access-control lists.
8. Maximum transmission units for router interfaces.

As can be imagined, a large number of errors can be made in manual computation of configuration parameter values implementing these requirements. GRE tunnels may only be configured in one direction or not at all. IPSec tunnels may only be configured in one direction or not at all. GRE and IPSec tunnels may not be “glued” together. GRE tunnels or sequences of tunnels may link routers in distinct COIs. A COI gateway router may contain static routes to a different COI, so packets could be routed to that COI via the WAN. BGP sessions may be set up between routers in different COIs so these routers may come to know about destinations behind each other. BGP sessions may only be configured in one direction or not at all. BGP sessions may not be supported by GRE tunnels so these sessions will not be established There may be single points of failure in the GRE and BGP networks. Finally, MTU settings on routers in a COI may be different leading to the possibility of packet loss. Such errors can be visualized by mapping various logical topologies. Two of these are shown below.

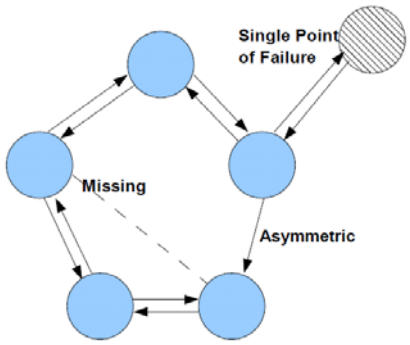

Figure 9.2 – GRE Tunnel Topology

In Figure 9.2 – GRE Tunnel Topology, nodes represent routers and edges represent a GRE edge between routers. These edges have to be set up in both directions for a GRE tunnel to be established. This graph shows two problems. First, the edge labeled "Asymmetric" has no counterpart in the reverse direction. Second, the dotted line indicates a missing tunnel. Third, the hatched router indicates a single point of GRE failure. All GRE packets to destinations to the right of this router pass through this router.

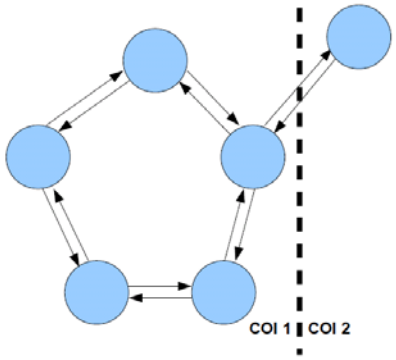

Figure 9.3 – BGP Neighbor Topology

In Figure 9.3 – BGP Neighbor Topology, nodes represent routers and links represent BGP sessions between nodes. This graph shows two problems. First, there is no full-mesh of BGP sessions within COI 1. Second, there is a BGP session between routers in two distinct COIs.

9.3 Creating a Configuration Validation System

This section outlines the design of a system that can not only validate the network of the previous section but also evolve to validate even more complex ones. As shown in **Figure 9.4 – Validation System Architecture**, this consists of a Configuration Acquisition system to acquire configuration information in a vendor-neutral format, a Requirement Library containing fundamental requirements simplifying the task of conceptualizing administrator intent, an easy to use Specification Language in which to specify requirements, and an Evaluation System to efficiently evaluate specifications in this language. These subsystems are now described.

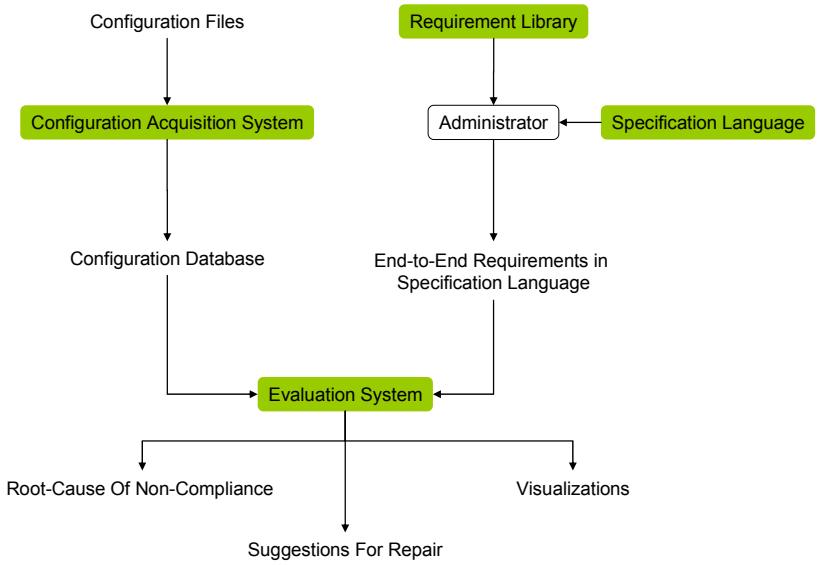

Figure 9.4 – Validation System Architecture

9.3.1 Configuration Acquisition System

Each component has associated with it a configuration file containing commands that define that component's configuration. These commands are entered by the network administrator. The most reliable method of acquiring a device's configuration information is to acquire this file, manually or automatically. Other less-reliable methods are accessing the devices' SNMP agent and querying configuration databases. SNMP agents often do not store all of the configuration informa-

tion one might be interested in. The correctness and completeness of a configuration database varies from enterprise to enterprise.

If configuration information is acquired from files then these files have to be parsed. Configuration languages have a simple syntax and semantics since they are intended to be used by network administrators who may not be expert programmers. Different vendors offer syntactically different configuration languages. However, the abstract configuration information stored in these files is the same, barring non-standard features that vendors sometimes implement. This information is associated with standardized protocols. Examples of it from the previous section are IP addresses, OSPF area identifiers, BGP neighbors, and IPSec cryptographic algorithms. This information needs to be extracted from files and stored in a vendor-neutral format database. Then, algorithms for evaluating requirements can be written just once against this database, and not once for every combination of vendor configuration language. However, configuration languages are vast, each with a very large set of features. Their syntax can change from one product release to another. Some vendors do not supply APIs to extract the abstract information. It should be possible to extract configuration information without having to understand all features of a configuration languages Extraction algorithms should be resilient to inevitable changes in configuration language syntax.

9.3.2 Requirement Library

The Requirement Library is analogous to libraries implementing fundamental algorithms in software development. The Library should capture design patterns and best practices for accomplishing fundamental goals in connectivity, security, reliability and performance. Examples of these for security can be found in [18] and for routing in [33]. These patterns can be expressed as requirements. The administrator should be easily able to conceptualize end-to-end requirements as compositions of Library requirements.

9.3.3 Specification Language

The specification language should provide an easy to use syntax for expressing end-to-end requirements. Specifications should be as close as possible in their forms to their natural language counterparts. The syntax can be text-based or visual. Since requirements are logical concepts, the syntax should allow specification of objects, attributes and constraints between these and compositions of constraints via operators such as negation, conjunction, disjunction and quantification. For example, all of these constructs appear in the Section 9.2 requirement “No gateway router in a COI has a static route to a destination in a different COI”.

9.3.4 Evaluation System

The Requirement Evaluation system should contain efficient algorithms to evaluate a requirement against configuration. These algorithms should output not just a yes/no answer but also explanations or counterexamples to guide configuration repair. Configuration repair is harder than evaluation. A set of requirements can be independently evaluated but if some are false, they cannot be independently made true. Changing the configuration to make one requirement true may falsify another. To provide further insight into reasons for truth or falsehood of requirements, this system should compute visualizations of logical relationships that are set up via configuration, analogous to visualizations of quantitative data [70].

9.4 IP Assure Validation System

This section describes the Telcordia® IP Assure product and discusses the choices made in it to implement the above abstract design of a validation system. This product aims to improve the security, availability, QoS and regulatory compliance of IP networks. It uses a parser generator for configuration acquisition. Its Requirement Library consists of well over one hundred requirements on integrity of logical structures, connectivity, security, performance, reliability and government policy. Its specification language is one of visual templates. Its evaluation system uses algorithms from graph theory and constraint solving. It also computes visualizations of several types of logical topologies. If a requirement is false, IP Assure does compute a root-cause although its computation is hand-crafted for each requirement. IP Assure does not compute a repair that concurrently satisfies all requirements.

9.4.1 Configuration Acquisition System

Section 9.3 raised three challenges in the design of a configuration acquisition system. The first was the design of a vendor-neutral database schema for storing configuration information. The second was extracting information from configuration files without having to know the entire configuration language for a given vendor. The third was making the extraction algorithms robust to inevitable changes in the configuration language. This section describes IP Assure's configuration acquisition system and sketches how well it meets these challenges.

IP Assure has defined a schema loosely modeled after DMTF [20] schemas. It uses the ANTLR [5] system to define a grammar for configuration files. The parser generated by ANTLR reads the configuration file and if successful, returns an

abstract syntax tree exposing the structure of file. This tree is then analyzed by algorithms implemented in Java to create and populate tables in its schema. Often, information in a table is assembled from information scattered in different parts of the file.

The system is illustrated in the context of a configuration file containing the following commands in Cisco's IOS configuration language:

```
hostname router1
!
interface Ethernet0
 ip address 1.1.1.1 255.255.255.0
 crypto map mapx
!
crypto map mapx 6 ipsec-isakmp
 set peer 3.3.3.3
 set transform-set transx
 match address aclx
!
crypto ipsec transform-set transx esp-3des hmac
!
ip access-list extended aclx
 permit gre host 3.3.3.3 host 4.4.4.4
```

A configuration file is a sequence of command blocks consisting of a main command followed by zero or more indented subcommands. The first command specifies the name `router1` of the router. It has no subcommands. Any line beginning with `!` is a comment line. The second command specifies an interface `Ethernet0`. It has two subcommands. The first specifies the IP address and mask of this interface. The second specifies the name `mapx` of an IPSec tunnel originating from this interface. The parameters of the IPSec tunnel are specified in the next command block. The main command specifies the name of the tunnel, `mapx`. The subcommands specify the address of the remote endpoint of the IPSec tunnel, the set `transx` of cryptographic algorithms to be used, and the profile `aclx` of the traffic that will be secured by this tunnel. The next command block defines the set `transx` as consisting of the encryption algorithm `esp-3des` and the hash algorithm `hmac`. The last command block defines the traffic profile `aclx` as any packet with protocol, source address and destination address equal to `gre`, `3.3.3.3` and `4.4.4.4` respectively.

Part of an ANTLR grammar for recognizing the above file is:

```
commands: command NL (rest=commands|EOF)
        ->^(COMMAND command $rest?);
command: ('interface') => interface_cmd
         |('crypto')    => crypto_cmd
         |('ip')        => ip_cmd
         |unparsed_cmd;
interface_cmd: 'interface' ID (LEADINGWS interface_subcmd) *
          -> ^('interface' ID interface_subcmd *)
interface_subcmd:
        'ip' 'address' a1=ADDR a2=ADDR -> ^('address' $a1 $a2)
        |'crypto' 'map' ID -> ^(CRYPTO_MAP ID)
        |unparsed_subcmd;
```

The first grammar rule states that `commands` is a sequence of 1 or more command blocks. The ^ symbol is a directive to construct the abstract syntax tree whose root is the symbol `COMMAND`, whose first child is the command block just read, and second child is the tree representing the sequence of subsequent command blocks. The next rule states that a command block begins with the keywords `interface`, `crypto` or `ip`. The symbol => means no backtracking. The last line in this rule states that if a command block does not begin with any of these identifiers, it is skipped. Skipping is done via the `unparsed_cmd` symbol. Grammar rules defining it skip all tokens till the beginning of the next command block. The last two rules define the structure of an `interface` command block. ANTLR produces a parser that processes the above file and outputs an abstract syntax tree. This tree is then analyzed to create the tables below. Note that the ipsec table assembles information from the `interface`, `crypto map`, `crypto ipsec` and `ip access-list` command blocks.

ipAddress Table

Host	Interface	Address	Mask
router1	Ethernet0	1.1.1.1	255.255.255.0

ipsec Table

Host	SrcAddr	DstAddr	EncryptAlg	HashAlg	Filter
router1	1.1.1.1	3.3.3.3	esp-3des	hmac	aclx

acl Table

Host	Filter	Protocol	SrcAddr	DstAddr	Perm
router1	aclx	gre	3.3.3.3	4.4.4.4	permit

IP Assure's vendor-neutral schema captures much of the configuration information for protocols it covers. Its skipping idea allows one to parse a file without recognizing the structure of all possible commands and command blocks. However, the idea is quite hard to get right in the ANTLR framework. One is trying to avoid writing a grammar for the skipped part of the language yet the only method one can use is to write rules defining `unparsed_cmd`.

9.4.2 Requirement Library

9.4.2.1 Requirements on integrity of logical structures

A very useful class of requirements is on the integrity of logical structures associated with different protocols. Before a group of components executing a protocol can accomplish an intended joint goal, various logical structures spanning these components must be set up. These structures are set up by making component configurations satisfy definite constraints. For example, before packets flowing between two interfaces can be secured via IPSec, the lPSec tunnel logical structure must be set up. This is done by setting IPSec configuration parameters at the two interfaces and ensuring that their values satisfy definite constraints. For example, the two interfaces must use the same hash and encryption algorithms and the remote tunnel endpoint at each interface must equal the IP address of its counterpart.

An HSRP router cluster is another example of a logical structure. HSRP stands for Hot Standby Routing Protocol [44]. It allows two or more routers to behave as a single router by offering a single virtual IP address to the outside world, on a given subnet. This address is mapped to the real address of an interface on the primary router. If this router fails, another router takes over the virtual address. Before the cluster correctly functions, however, the same virtual address and HSRP group identifier must be configured on all interfaces and the virtual and all physical addresses must belong to the same subnet.

Much more complex logical structures are set up for BGP. Different routers in an Autonomous System (AS) connect to different neighboring ASes, giving each router only a partial view of BGP routes. To allow all routers in an AS to construct a complete view of routes, routers exchange information between themselves via iBGP (internal BGP) sessions. The simplest logical structure for accomplishing this exchange is a full-mesh of iBGP sessions, one for each pair of routers. But a full-mesh is impractical for a large AS since the number of sessions grows quadratically with the number of routers. Linear growth is accomplished with a hub-and-spoke structure. All routers exchange routes with a spoke called a route-reflector. If these structures are incorrectly set up, protocol oscillations, forwarding loops, traffic blackholes and violation of business contracts can arise [31, 6, 74]. See Section 9.6.4 for more discussion of BGP validation.

IP Assure evaluates requirements on integrity of logical structures associated with all common protocols. These structures include IP subnets, GRE tunnels, IPSec tunnels, MPLS [42] tunnels, BGP full-mesh or hub-and-spoke structures, OSPF subnets and areas and HSRP router clusters.

9.4.2.2 Connectivity Requirements

Connectivity (also called reachability) is a fundamental requirement of a network. It means the existence of a path between two nodes in the network. The most obvious network is an IP network whose nodes represent subnets and routers and links represent direct connections between these. But as noted in Section 9.2, connectivity requirements are also meaningful for many other types of networks such as GRE, IPSec and BGP. IP Assure evaluates connectivity for IP, VLANs, GRE, IPSec, BGP and MPLS networks.

IP Assure also evaluates reachability in the presence of access-control policies, or lists, configured on routers or firewalls. An access-control list is a collection of rules specifying the IP packets that are permitted or denied based on their source and destination address, protocol and source and destination ports. These rules are order-dependent. Given a packet, the rules are scanned from the top-down and the permit or deny action associated with the first matching rule is taken. Even if a path exists, a given packet may fail to reach a destination because an access-control list denies that packet.

9.4.2.3 Reliability Requirements

Reliability in a network means the ability to maintain connectivity in the presence of failures of nodes or links. A single point of failure for connectivity between two nodes in a network is said to exist if a single failure causes connectivity between the two nodes to be lost. Reliability is achieved by provisioning backup resources and setting up a reliability protocol. This protocol monitors for failures and when one occurs, finds backup resources and attempts to restore connectivity using those.

Configuration errors may prevent backup resources from being provisioned. For example, in Section 9.2, some GRE tunnels were only configured in one direction, not in the other so they were unavailable for being rerouted over. Even if backup resources have been provisioned, configuration errors in the routing protocol can prevent these resources from being found. For example, in Section 9.2, BGP was simply not configured to run over some GRE tunnels so it would not find these links to reroute over.

The architecture of the fault-tolerance protocol itself can introduce a single point of failure. For example, a non-zero OSPF area may be connected to OSPF area zero by a single area-border-router. If that router fails, then OSPF will fail to discover alternate routes to another area [36] even if these exist. Similarly, unless BGP route-reflectors are replicated, they can become single points of failure [7].

Furthermore, redundant resources at one layer must be mapped to redundant resources at lower layers. For example, if all GRE tunnels originate at the same physical interface on a router then if that interface fails, all tunnels would simulta-

neously fail. Ideally, all GRE tunnels originating at a router must originate at distinct interfaces on that router.

Single points of failure can also arise out of the dependence between security and reliability.

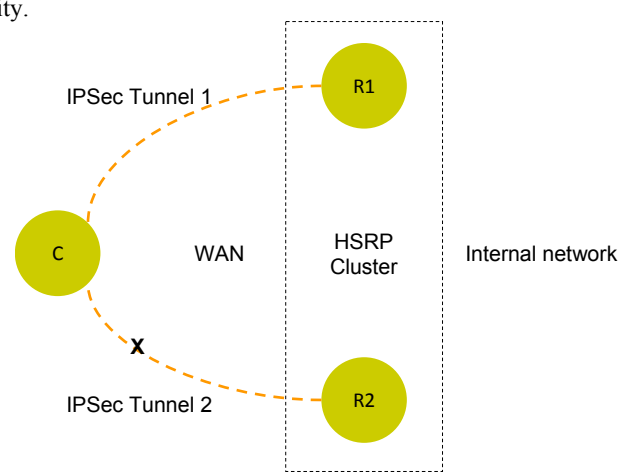

Figure 9.5 – HSRP Cluster

As shown in Figure 9.5 – HSRP Cluster, routers R1 and R2 together constitute an HSRP cluster with R1 as the primary router. This cluster forms the gateway between an enterprise's internal network on the right and the WAN on the left. For security, an IPSec tunnel is configured from R1 to the gateway router C of a collaborating site. However, this tunnel is not replicated on R2. Consequently, if R1 fails then R2 would take over the cluster's virtual address, however, IPSec connectivity to C would be lost.

Reliability requirements that IP Assure evaluates include absence of single points of failure in IP networks, with and without access-control policies; absence of single OSPF area-border-routers; and replication of IPSec tunnels in an HSRP cluster.

9.4.2.4 Security Requirements

Typical network security requirements are about data confidentiality, data integrity, authentication and access-control. IPSec is commonly used to satisfy the first three requirements and access-control lists are used to satisfy the last one. Access-control lists were discussed in Section 9.4.2.2. Components dedicated just to processing access-control lists are called firewalls. IP Assure evaluates requirements for both these technologies. For IPSec, it evaluates the tunnel integrity requirements in Section 9.4.2.1. For access-control lists, IP Assure evaluates two fundamental requirements. First, an access-control list subsumes another in that

any packet permitted by the second is also permitted by the first. A related requirement is that one list is equivalent to another in that any packet permitted by one is permitted by the other. Two lists are equivalent if each subsumes the other. An enterprise may have multiple egress firewalls. Access-control lists on these may have been set up by different administrators over different periods of time. It is useful to check that the policy governing packets that leave the enterprise are equivalent. The second requirement that IP Assure evaluates on access-control lists is that a firewall has no redundant rules. A rule is redundant if deleting it will not change the set of packets a firewall permits. Deleting redundant rules makes lists compact and easier to understand and maintain.

9.4.2.5 Performance Requirements

The DiffServ protocol allows one to specify policies for partitioning packets into different classes, and then for according them differentiated performance treatment. For example, a packet with a higher DiffServ class is given transmission priority over one with a lower. Typically, voice packets are given highest priority because of the high sensitivity of voice quality to end-to-end delays. Performance requirements that IP Assure evaluates are that all DiffServ policies on all routers are identical, and that any policy that is defined is actually used by being associated with an interface.

IP Assure also evaluates the requirement that ICMP packets are not blocked. This is a sufficient condition for avoiding packet loss due to mismatched MTU sizes and setting of Do Not Fragment bits discussed in Section 9.2.

9.4.2.6 Government Regulatory Requirements

Government regulatory requirements represent “best practices” that have evolved over a period of time. Compliance to these is deemed essential for connectivity, reliability, security and performance of an organization’s network. Compliance to certain regulations such as the Federal Information Security Management Act (FISMA) [26] is mandatory for government organizations. Two examples of a FISMA requirement are (a) alternate communications services do not share a single point of failure with primary communication services (b) all access between nodes internal to an enterprise and those external to it is mediated by a proxy server. IP Assure allows specification of a large number of FISMA requirements.

9.4.3 Specification Language

IP Assure's specification language is that of graphical templates. It offers a menu of more than 100 requirements in different categories. A user can select one or more of these to be evaluated. For each requirement, one can specify its parameters. For example, for a reachability requirement one can specify the source and destination. For an access-control list equivalence requirement one can specify the two lists. One cannot apply disjunction or quantification operators to requirements. The only way to define new requirements is to program in Java and SQL.

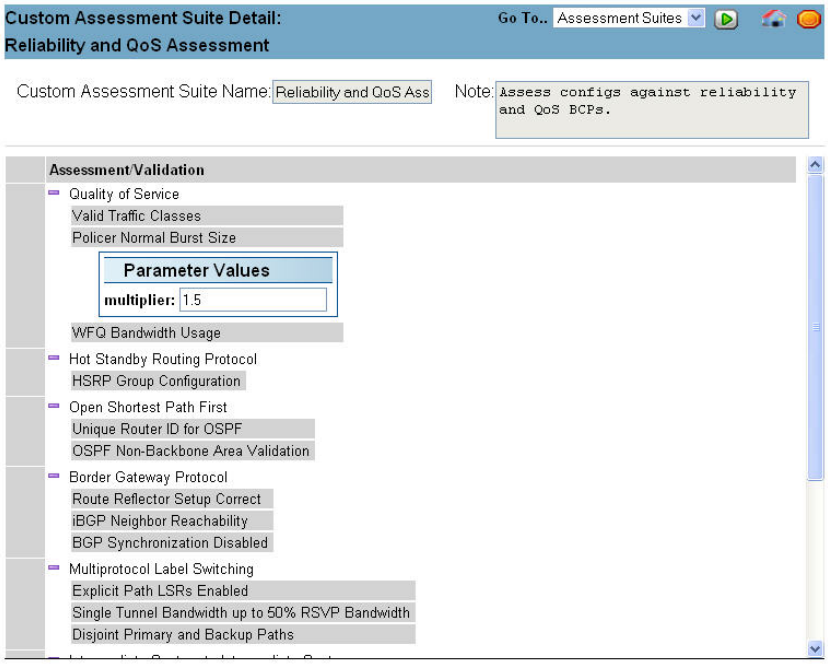

Figure 9.6 – IP Assure Requirement Specification Screen

Figure 9.6 – IP Assure Requirement Specification Screen shows a few requirement classes that can be evaluated. These are QoS (DiffServ), HSRP, OSPF, BGP and MPLS.

9.4.4 Evaluation System

Structural integrity requirements are evaluated with algorithms specialized to each requirement. In IP Assure, these algorithms are implemented with SQL and Java. The relevant tuples from the configuration database are extracted with SQL and analyzed by Java programs. For example, to evaluate whether an IPSec tunnel between two addresses `local1` and `local2` is set up, one checks that there are tuples `ipsec(h1, local1, remote1, ea1, ha1, filter1)` and `ipsec(h2, local2, remote2, ea2, ha2, filter2)` in the configuration database, and that `local1=remote2, remote1=local2, ea1=ea2, ha1=ha2` and `filter1` is a mirror image of `filter2`.

Reachability and reliability requirements for a network are evaluated by extracting the relevant graph information from the configuration database with SQL queries, then applying graph algorithms [63]. For example, given the tuple `ipAddress(host, interface, address, mask)` one creates two nodes, the router `host` and the subnet whose address is the bitwise-and of `address` and `mask`, and then creates directed edges linking these in both directions. This step is repeated for all such tuples to compute an IP network graph.

To evaluate whether a node or a link is a single point of failure, one removes it from the graph and checks whether two nodes are reachable. If not, then the deleted node or link is a single point of failure. To check reachability in the presence of access-control lists, all edges at which these lists block a given packet are deleted, and then reachability analysis is repeated for the remaining graph.

Firewall requirements cannot be evaluated by enumerating all possible packets and checking for subsumption, equivalence or redundancy. The total number of combinations of all source and destination addresses, ports, and protocols is astronomical: the total number of IPv4 source and destination address, source and destination port and protocol combinations is 2^104 (32+32+16+16+8). Instead, symbolic techniques are used. Each policy is represented as a constraint on the following fields of a packet: source and destination address, protocol and source and destination ports. The constraint is true precisely for those packets that are permitted by the firewall, taking rule ordering into account. Let `P1` and `P2` be two policies and `C1` and `C2` be, respectively, the constraints representing them. The constraint can be constructed in time linear in the number of rules. Then, `P1` is subsumed by `P2` if there is no solution to the constraint $C1 \wedge \neg C2$. To check that a rule in `P1` is redundant, delete it from `P1` and check that the resulting policy is equivalent to `P1`.

For example, let a firewall contain the following rules that, for simplicity, only check whether the source and destination addresses are in definite ranges:

```
1, 2, 3, 4, deny
5, 6, 7, 8, permit
10, 15, 15, 20, permit
```

The first rule states that any packet with source address between 1 and 2 and destination address between 3 and 4 is denied. Similarly, for the second and third rules. These are represented by the following constraint `C1` on the variables `src` and `dst`.

```
¬(1=<src ∧ src=<2 ∧ 3=<dst  ∧ dst=<4]) ∧
 (5=<src ∧ src=<6 ∧ 7=<dst  ∧ dst=<8) ∨
 (10=<src ∧ src=<15 ∧ 15=<dst∧ dst=<20)
```

This constraint states that a packet is permitted if it is not the case that its source address is in [1,2] and destination address is in [3,4] and that these fields are either in [5, 6] and [7, 8] respectively, or in [10,15] and [15, 20] respectively.

If there were another firewall with a single rule:

```
11, 12, 13, 14, permit
```

then the constraint `C2` representing it would be

```
(11=<src∧src=<12 ∧ 13=<dst∧dst=<14)
```

To check whether the first firewall subsumes the second, check that `C2∧¬C1 is` unsolvable. A constraint solver will confirm that this is so. On the other hand, the solver will compute a solution to the constraint `C1∧¬C2` as `src=5, dst=7`. Such constraints are solved by the ConfigAssure [51] system described in Section 9.5.

9.4.4.1 Proactive Evaluation

From just the configurations, IPAssure tries to guess requirements that the administrator intended and evaluates these. When its guess is correct, it saves the administrator the effort of explicitly specifying that requirement. When the guess is incorrect, the administrator can ignore the "false positive". For example, if IPSec is configured on an interface then it is a good guess that IPSec should be configured on the remote endpoint of the tunnel. Then, the IPSec structural integrity requirement is evaluated. This approach has been implemented for a number of protocols. The intent for some requirements cannot be guessed. For example, in the FISMA requirement that all communication between internal and external subnets must pass through a firewall, one cannot guess what internal and external subnets are. IP Assure allows these to be explicitly specified.

9.4.4.2 Visualization

For visualization IP Assure displays logical structures the way they are set up by configuration. Then, their integrity and defects both stand out. This approach has worked well for structures such as subnets, GRE tunnels, IPSec tunnels, OSPF areas, BGP full-meshes and hub-and-spoke structures, and HSRP router clusters. The relevant nodes and edges are extracted from the configuration database with SQL queries and the Graphviz [30] layout tool is used. For example, in Figure 9.7 – Visualization of an IP Network, nodes are routers and subnets, a link is only between a router and a subnet and represents that fact that the router has an interface on that subnet.

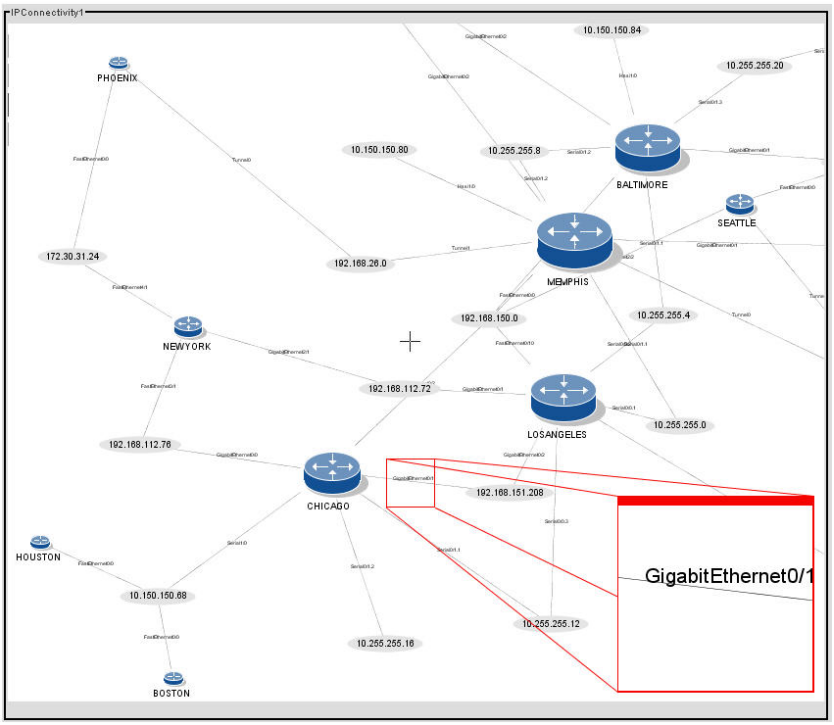

Figure 9.7 – Visualization of an IP Network

In Figure 9.8 – OSPF Topology Visualization, a link from a router to an area identifier means that router has an interface in that area. This clearly shows that Area 10 has two border routers LOSANGELES and CHICAGO linking it to Area 0. Thus, there is no single point of failure due to a single ABR discussed above. However, the figure also shows that Area 17 has only a single ABR. This is a single point of failure as outlined in Section 9.4.2.3.

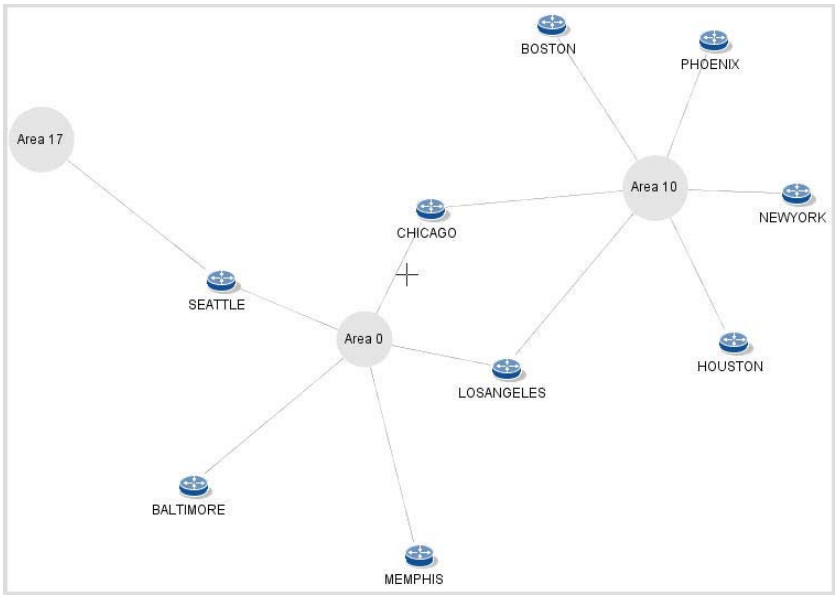

Figure 9.8 – OSPF Topology Visualization

9.5 Logic-Based Techniques For Creating A Validation System

This section describes a suite of logic-based techniques that are particularly useful for creating a validation system. They simplify configuration acquisition and requirement specification. They allow firewall subsumption, equivalence and rule redundancy analysis. Finally, they provide an efficient approach for solving the hard problem of configuration error repair. These techniques are the languages Prolog and Datalog [53, 67] and arithmetic quantifier free forms [51], the Kodkod [41] constraint solver for first-order logic, the minimum-cost ZChaff [73] SAT solver for Boolean logic and Ordered Binary Decision Diagrams [12].

The Prolog language combines a rule-based programming language and a database into a single system. Its interpreter is based on the top-down SLD-resolution [53] inference procedure. Modern Prolog implementations are highly efficient and also have tight interfaces to external languages like C, C++ and Java. Algorithms that are best encoded in these languages can be so encoded, and then called from Prolog. Prolog databases of several million tuples can be efficiently queried [66]. Datalog is a restriction of Prolog to exclude data structures.

Kodkod is a Java API for solving a first-order logic constraint. While a Boolean constraint only contains Boolean variables, a first-order logic constraint can

contain variables denoting data objects, relationships between these variables, and quantifiers upon these variables. Kodkod solves a first-order logic constraint, in finite domains, by compiling it into a Boolean logic constraint, solving it with a SAT solver and reflecting a solution back into a solution of the first-order logic constraint. If the constraint is unsolvable, Kodkod also computes a proof of unsolvability, inherited from the SAT solver. Typically, this is an unsolvable constraint that is much smaller than the original. The ZChaff SAT solver can solve millions of Boolean constraints in millions of Boolean variables in seconds. If costs are associated with the setting of a Boolean variable to true then ZChaff can also be used to compute a minimum-cost solution to a constraint.

Any Boolean constraint can be transformed into a unique, equivalent form called an Ordered Binary Decision Diagram. Thus, equivalence between two Boolean constraints can be checked by checking that their OBDDs are identical. An OBDD also has the interesting property that if it is not trivially false, then it is satisfiable. The check for satisfiability is built into the algorithm to transform a Boolean constraint into its OBDD. However, OBDDs are efficiently constructed only for constraints with a few hundred Boolean variables although this size is ample for reasoning about firewalls.

The Service Grammar system used Prolog directly for specification and validation [58, 50]. The ConfigAssure [51] system integrates Prolog with Kodkod. It allows fields in tuples in a configuration database to be variables. It computes values of these variables so that a given requirement becomes true of the database. It does so by transforming that requirement into an equivalent arithmetic quantifier-free form or QFF. A QFF is a Boolean combination of constraints formed from configuration variables, integers, the operators +, −, <, =<, =, >, >= and bitwise logic operators. This QFF is then efficiently solved by Kodkod. If ConfigAssure is unable to find a solution it outputs a proof of unsolvability, inherited from Kodkod. This proof is interpreted as a root-cause and guides configuration repair. Arithmetic quantifier-free forms constitute a good intermediate language between Boolean logic and first-order logic. Not only is it easy to express requirements in it but it can also be efficiently compiled into Boolean logic. ConfigAssure was designed to avoid, where possible, the generation of very large intermediate constraints in Kodkod's transformation of first-order logic into Boolean.

If the fields that are responsible for making a requirement false are known, then one way to repair these is as follows: replace these fields with variables and use ConfigAssure to find new values of these variables that make the requirement true. Two approaches can be used to narrow down these fields. The first exploits the proof-of-unsolvability of the falsified requirement to compute a type of root-cause. The second exploits properties of Datalog proofs and ZChaff to compute that set of fields whose cost of change is minimal. The second approach has been developed in the MulVAL [35, 55, 56] system. More generally, MulVAL is a system for enterprise security analysis using attack graphs.

Ordered Binary Decision Diagrams are an alternative to SAT solvers for evaluating firewall policy subsumption and rule redundancy with a method conceptually similar to that in Section 9.4.4.

The use of these techniques for building different parts of a validation system is now illustrated with concrete examples based on the case study in Section 9.2.

9.5.1 Configuration Acquisition By Querying

When the structure of a configuration file is simple, as it is for Cisco's IOS, then it is not necessary to write a grammar with ANTLR or PADS/ML [47]. Instead, the structure can be put into a command database and then *queried* to construct the configuration database. The query needs to refer only to that part of the command database necessary to construct a given table. All other parts are ignored. This idea provides substantial resilience to insertion of new command blocks, insertion of new subcommands in a known command block, and insertion of new keywords in subcommands.

This idea is illustrated using Prolog, although any database engine could be used. Each command block is transformed into an `ios_cmd` tuple or Prolog fact, with the structure

```
ios_cmd(FileName, MainCommand, ListOfSubCommands)
```

where `MainCommand` and each item in `ListOfSubCommands` is of the form `[NestingLevel | ListOfTokens]`. `[A|B]` means the list with head `A` and tail `B`. For example, the IOS file of Section 9.4.1, named `f` here, is transformed into the following Prolog tuples:

```
ios_cmd(f, [0, hostname, router1], []).
ios_cmd(f,
   [0, interface, 'Ethernet0'],
   [   [1, ip, address, '1.1.1.1', '255.255.255.0'],
       [1, crypto, map, mapx] ]).
ios_cmd(f,
   [0, crypto, map, mapx, 6, 'ipsec-isakmp'],
   [   [1, set, peer, '3.3.3.3'],
       [1, set, 'transform-set', transx],
       [1, match, address, aclx]]).
ios_cmd(f,
   [0,crypto,ipsec,'transform-set',
      transx,'esp-3des',hmac], []).
ios_cmd(f,
   [0, ip, 'access-list', extended, aclx],
   [   [1, permit, gre, host, '3.3.3.3',
          host, '4.4.4.4']]).
```

Note the close correspondence between the structure of command blocks in the IOS file and associated `ios_cmd` tuples. One can now write Prolog rules to con-

struct the configuration database. For instance, to construct rows for the `ipAddress` table, one can use:

```
ipAddress(H, I, A, M):-
       ios_cmd(File, [0, hostname, H|_], _),
       ios_cmd(File, [0, interface, I|_], Args),
       member(SubCmd, Args),
       subsequence([ip, address, A, M], SubCmd).
```

The syntactic convention followed in Prolog is that identifiers beginning with capital letters are variables, otherwise they are constants. The :- symbol is a shorthand for `if`. All variables are universally quantified. The rule states that `ipAddress` of an interface `I` on host `H` is `A` with mask `M` if there is a `File` containing a `hostname` command declaring host `H`, an `interface` command declaring interface `I` and a subcommand of that command declaring its address and mask to be `A` and `M` respectively.

Note that this definition is unaffected by subcommands of the `interface` command that are not of interest for computing `ipAddress`, or that are defined in a subsequent IOS release. It only tries to find a subcommand containing the sequence `[ip, address, A, M]`. It does not require that the subcommand be in a definite position in the block, or that the sequence `address A, M` appear in definite position in the `ip` subcommand. Now, where `H, I, A, M` are variables, the query `ipAddress(H, I, A, M)` will succeed with the solution `H=f`, `I='Ethernet0'`, `A='1.1.1.1'` and `M='255.255.255.0'`. Here `f` is a host, `I` is an interface on this host, and `A` and `M` its address and mask respectively.

`ipsec` is more complex but querying simplifies the assembly of information from different parts of a configuration file. For each interface, one finds the name of a crypto map `Map` applied to that interface, and then finds the corresponding crypto `map` command, from which one can extract the peer address `Peer`, the filter `Filter`, and transform-set `Transform`. These values are used to select the `crypto ipsec` command from which the `Encrypt` and `Hash` values are extracted. Thus, the `ipSecTunnel(H, Address, Peer, Encrypt, Hash, Filter)` is constructed.

```
ipsec(H, Address, Peer, Encrypt, Hash, Filter):-
    ios_cmd(File, [0, interface, I |_], Args),
    member([_, crypto, map, Map |_], Args),
    ios_cmd(File, [0, hostname, H |_], _),
    ipAddress(H, I, Address, _),
    ios_cmd(File, [0, crypto, map, Map |_], CArgs),
    member([_, set, peer, Peer |_], CArgs),
    member([_, match, address, Filter|_], CArgs),
    member([_, set, 'transform-set',
           Transform |_], CArgs),
    ios_cmd(File, [0, crypto, ipsec,
           'transform-set', Transform, Encrypt, Hash],_).
```

The `ipAddress` and `ipsec` tuples are constructed in all possible ways via Prolog backtracking. Together, these form the configuration database for these protocols.

9.5.2 Specification Language

This section shows how Prolog can be used to specify the types of requirements in the case study of Section 9.2. It has already been used to validate VPN and BGP requirements [50, 58]

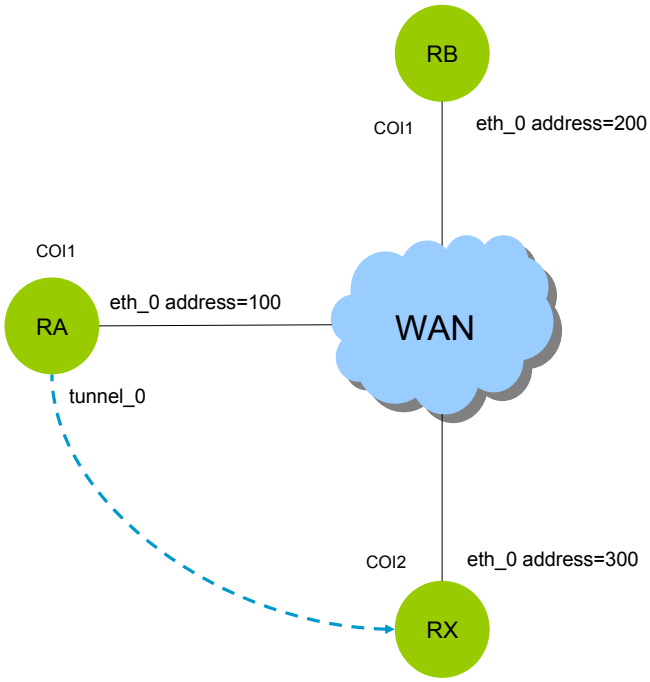

Figure 9.9 – Network Violating Security and Connectivity Requirements

As shown in Figure 9.9 – Network Violating Security and Connectivity Requirements, routers RA and RB are in the same COI but RX is in a different COI. RA's configuration violates two security requirements and one connectivity requirement. First, RA has a GRE tunnel into RX. Second, RA has a default static route using which it can forward packets destined to RX, to the WAN. Third, RA does not have a GRE tunnel into RB. All these violations need to be detected and configurations repaired.

A configuration database for the above network is represented by the following Prolog tuples:

```
static_route(ra, 0, 32, 400).
gre(ra, tunnel_0, 100, 300).
ipAddress(ra, eth_0, 100, 0).
ipAddress(rb, eth_0, 200, 0).
ipAddress(rx, eth_0, 300, 0).
coi([ra-coi1, rb-coi1, rx-coi2]).
```

The first tuple states that router `ra` has a default static route with a next hop of address `400`. Normally, a mask is a sequence of 32 bits containing a sequence of ones followed by a sequence of zeros. In the `ipAddress` tuple, a mask is represented implicitly as the number of zeros at the end of the sequence. This simplifies the computations we need. The route is called "default" because any address matches it. The second states that router `ra` has a GRE tunnel originating from GRE interface `tunnel_0` with local physical address `100` and remote physical address `300`. The third tuple states that router `ra` has a physical interface `eth_0` with address `100 and mask 0`. Similarly, for the fourth and fifth tuples. The last tuple lists the community of interest of each router. Requirements are defined with Prolog clauses, e.g.:

```
good:-gre_connectivity(ra, rb).
gre_connectivity(RX, RY):-
    gre_tunnel(RX, RY),
    route_available(RX, RY).
gre_tunnel(RX, RY):-
    gre(RX, _, _, RemoteAddr),
    ipAddress(RY, _, RemoteAddr, _).
route_available(RX, RY):-
    static_route(RX, Dest, Mask, _),
    ipAddress(RY, _, RemotePhysical, 0),
    contained(Dest, Mask, RemotePhysical, 0).
contained(Dest, Mask, Addr, M):-
    Mask>=M,
    N is ((2^32-1)<< Mask)/\Dest,
    N is ((2^32-1)<< Mask)/\Addr.
bad:-gre_tunnel(ra, rx).
bad:-route_available(ra, rx).
```

The first clause states that `good` is true provided there is GRE connectivity between routers `ra` and `rb` since they are in the same COI. The second clause states that there is GRE connectivity between any two routers `RX` and `RY` provided RX has a GRE tunnel configured to RY and a route available to RY. The third clause states that a GRE tunnel to `RY` is configured on `RX` provided there is a GRE tuple on `RX` whose remote address is that of an interface on `RY`. The fourth clause states that a route to `RY` is available on `RX` provided an address `RemotePhysical` on `RY` is contained within the address range of a static route on `RX`. The fifth clause checks this containment. `<< is` the left-shift operator and `/\` is the bitwise-and operator, not to be confused with the conjunction operator. The sixth clause

states that bad is true provided there is a gre tunnel between ra and rx since ra and rx are *not* in the same COI. The last clause states that bad is also true provided a route on ra is available for packets with a destination on rx.

We now show how to capture requirements containing quantifiers. To capture the requirement all_good that between *every* pair of routers in a COI there is GRE connectivity, we can write:

```
all_good:-not(same_coi_no_gre).
same_coi_no_gre:-same_coi(X, Y), not(gre_connectivity(X, Y)).
same_coi(X, Y):-coi(L), member(X-C, L), member(Y-C, L).
```

The first rule states all_good is true provided same_coi_no_gre is false. The second rule states that same_coi_no_gre is true provided there exist X and Y that are in the same COI but for which gre_connectivity(X, Y) is false. The last rule states that X and Y are in the same COI provided there is some COI C such that X-C and Y-C are in the COI association list L.

Similarly, we can capture the requirement no_bad that *no* router contains a route to a router in a different COI.

As previously mentioned, the MulVAL system has proposed the use of Datalog for specification and analysis of attack graphs. Datalog is a restriction of Prolog in which arguments to relations are just variables or atomic terms, i.e., no complex terms and data structures. This restriction means, in particular, that predicates such as all_good and all_pairs_gre cannot be specified and neither can subnet_id since it needs bitwise operations. However, the first five Prolog tuples above and the first three rules can be specified. This restriction, however, permits MulVAL to perform fine-grained analysis of root-causes of configuration errors and to compute strategies for their repair. This is discussed in the next subsection.

9.5.3 Evaluation For Repair

If a configuration database and requirements are expressed in Prolog then its query capability can be used to evaluate whether requirements are true. For example, the query route_available(ra, rb) is evaluated to be true by clauses for route_available, static_route and contained. The query bad succeeds for two reasons. First, the static route on ra is a default route. It forwards packets to any destination, including to destinations in a different COI. Second, a GRE tunnel to router rx is configured on ra even though rx is in a different COI. On the other hand, the query good fails. This is because the predicate gre_tunnel(ra, rb) fails. The only GRE tunnel configured on ra is to rx, not to rb.

If requirement evaluation against a configuration database is the only goal then a Prolog-based validation system is practical on a realistic scale. However, if a re-

quirement is false for a configuration database and the goal is to change some fields in some tuples so that the requirement becomes true, then Prolog is not adequate. The Prolog query (`good,not(bad)`), representing the conjunction of `good` and `not(bad)`, will simply fail. Prolog will not return new values of these fields that make the query true.

In order to efficiently compute new values of these fields, a constraint solver with the capability to compute a proof of unsolvability is needed. Such a capability is provided by the ConfigAssure system. ConfigAssure allows one to replace some fields in some tuples in a configuration database with configuration variables. These variables are *unrelated* to Prolog variables. ConfigAssure also allows one to specify a requirement R as an equivalent QFF `RC` on these configuration variables. Solving `RC` would compute new values of these fields, in effect repairing the fields.

For example, suppose we suspect that the query (`good,not(bad)`) fails because addresses and the static route mask are incorrect. We can replace all these with configuration variables to obtain the following database:

```
static_route(ra, dest(0), mask(0), 400).
gre(ra, tunnel_0, gre_a_local(0), gre_a_remote(0)).
ipAddress(ra, eth_0, ra_addr(0), 0).
ipAddress(rb, eth_0, rb_addr(0), 0).
ipAddress(rx, eth_0, rx_addr(0), 0).
coi([ra-coi1, rb-coi1, rx-coi2]).
```

Here `dest(0)`, `mask(0)`, `gre_a_local(0)`, `gre_a_remote(0)`, `ra_addr(0)`, `rb_addr(0)`, `rx_addr(0)` are all configuration variables. In order that this database satisfy `(good ∧ not(bad))`, these configuration variables must satisfy the following constraint `RC`:

```
¬gre_a_remote(0)=rx_addr(0)∧
¬contained(dest(0),mask(0),rx_addr(0),0)
   ∧ gre_a_remote(0)=rb_addr(0)
   ∧ contained(dest(0),mask(0),rb_addr(0),0)
   ∧ ¬ra_addr(0)=rb_addr(0) ∧ ¬rb_addr(0)=rx_addr(0) ∧
¬rx_addr(0)=ra_addr(0)
```

The constraint on the first two lines is equivalent to `not(bad)`. It states that `ra` should neither have a GRE tunnel nor a static route to `rx`. The constraint on the next two lines is equivalent to `good`. It states that `ra` should have both a GRE tunnel and a static route to `rb`. The constraint on the last line states that all interface addresses are unique. Solving this constraint would indeed find new values of configuration variables and hence repair the fields. However, one may change fields, such as `ra_addr(0)`, unrelated to the failure of (`good,not(bad)`). To change fields only related to failure, one can exploit the proof-of-unsolvability that ConfigAssure automatically computes when it fails to solve a requirement. This proof is a typically small and unsolvable part of the requirement, and can be taken to be a root-cause of unsolvability.

The idea is to generate a new constraint `InitVal` that is a conjunction of equations of the form `x=c` where `x` is a configuration variable that replaced a field and `c` is the initial value of that field. Now try to solve `RC∧InitVal`. Since `R` is false for the database without variables, ConfigAssure will find `RC∧InitVal` to be unsolvable and return a proof-of-unsolvability. If, in this proof, there is an equation `x=c` that is also in `InitVal`, then relax the value of `x` by deleting `x=c` from `InitVal` to create `InitVal'`. Reattempt a solution to `RC∧InitVal'` to find a new value of `x`. More than one such equation can be deleted in a single step. For example, the definition of `InitVal` for above configuration variables is:

```
dest(0)=0
∧ mask(0)=32
∧ gre_a_local(0)=100
∧ gre_a_remote(0)=300
∧ ra_addr(0)=100
∧ rb_addr(0)=200
∧ rx_addr(0)=300
```

Submitting `RC∧InitVal` to ConfigAssure generates a proof of unsolvability that `ra` should have a tunnel to `rb` but instead has one to `rx`:

```
gre_a_remote(0)=rb_addr(0) ∧ gre_a_remote(0)=300 ∧
rb_addr(0)=200
```

Deleting the second equation from `InitVal` to obtain `InitVal'` and solving `RC∧InitVal'` we obtain another proof of unsolvability that `ra` has a static route to `rx`:

```
rx_addr(0)=300 ∧ dest(0)=0 ∧ mask(0)=32 ∧
¬contained(dest(0),mask(0),rx_addr(0),0)
```

Deleting the second and third equations and solving we obtain a solution that fixes both the GRE tunnel and the static route on `ra`:

```
dest(0)=200
mask(0)=0
gre_a_remote(0)=200
gre_a_local(0)=100
ra_addr(0)=100
rb_addr(0)=200
rx_addr(0)=300
```

Values of just the first three variables needed to be recomputed. Values of others don't need to be. Note that `ra_addr(0)` never appeared in a proof of unsolvability even though it did in `RC`. Thus, its value definitely does not need to be recomputed. This is not obvious from `RC`. Note also that repair is holistic in that it satisfies both `good` and `not(bad)`.

The remaining task is generation of the constraint RC. It is accomplished by thinking about specification as a method of computing an equivalent quantifier-free formula, i.e., defining the predicate eval(Req, RC) where Req is the name of a requirement and RC is a QFF equivalent to Req. The original Prolog specification of Req in Section 9.5.2 is no longer needed. It is replaced by a *meta-level* version as follows:

```
eval(bad, or(C1, C2)):-
    eval(gre_tunnel(ra, rx), C1),
    eval(route_available(ra, rx), C2).
eval(gre_tunnel(RX, RY), RemoteAddr=Addr):-
    gre(RX, _, _, RemoteAddr),
    ipAddress(RY, _, Addr, _).
eval(route_available(RX, RY), C):-
    static_route(RX, Dest, Mask, _),
    ipAddress(RY, _, RemotePhysical, _),
    C=contained(Dest, Mask, RemotePhysical, 0).
eval(addr_unique, C):-
    andEach([not(ra_addr(0)=rb_addr(0)),
             not(rb_addr(0)=rx_addr(0)),
             not(rx_addr(0)=ra_addr(0))], C).
eval(topReq, C):-
   eval(good, G),
   eval(bad, B),
   eval(addr_unique, AU),
   andEach([G, B, AU], C).
```

These rules capture the semantics of the Prolog rules. The first states that a QFF equivalent to bad is the disjunction of C1 and C2 where C1 is the QFF equivalent to gre_tunnel(ra, rx) and C2 is the QFF equivalent to route_available(ra, rx). The second rule states that the QFF equivalent to gre_tunnel(RX, RY) is RemoteAddr=Addr where RemoteAddr is the remote physical address of a GRE tunnel on RX and Addr is the address of an interface on RY. The third rule states that the QFF equivalent to route_available(RX, RY) is C provided C is the constraint that RX contains a static route for an address on RY. The fourth rule computes the QFF for all interface addresses being unique. The last rule computes the QFF for the top-level constraint topReq.

Now, the Prolog query eval(topReq, RC) computes RC as above. As has been shown in [51] QFFs are much more expressive than Boolean logic so it is not hard to write requirements using the eval predicate.

9.5.4 Repair With MulVAL

The MulVAL system proposes an alternative, precise method of computing the fields that cause the success of an undesirable requirement provided that require-

ment is expressed in Datalog. A requirement, such as `bad`, is said to be undesirable if it enables adversary success. This method is based on the observation that any tuple in a proof of an undesirable requirement is responsible for the truth of that requirement. These tuples contain all the fields that need to be replaced by configuration variables. For example, one proof of `bad` with the original Prolog specification in Section 9.5.2 is:

```
bad
gre_tunnel(ra, rx)
gre(ra, tunnel_0, 100, 300) ∧ ipAddress(rx, eth_0, 300, 0)
```

Here each condition is implied by its successor by the use of a rule in the Prolog specification. The second proof of `bad` is:

```
bad
route_available(ra, rx)
static_route(ra,0,32,400) ∧ ipAddress(rx, eth_0,300,0) ∧
contained(0,32,300,0)
```

The tuples that contribute to the proof of `bad` are:

```
gre(ra, tunnel_0, 100, 300)   -- from the first proof
ipAddress(rx, eth_0, 300, 0)  -- from the first proof
static_route(ra, 0, 31, 400)  -- from the second proof
```

The following tuples do not contribute to the proof of `bad`:

```
ipAddress(ra, eth_0, 100, 0).
ipAddress(rb, eth_0, 200, 0).
```

The three tuples in the proof of bad contain all the fields that need to be replaced by configuration variables. Note that the address of interfaces at ra and rb do not need to be replaced.

The MulVAL system does not actually compute new values of fields. It only computes the set of tuples that should be disabled to disable all proofs of the undesirable property. A tuple can be disabled by changing its fields to different values or deleting it. But, MulVAL computes the set in an optimal way. It first derives a Boolean formula representing all of the ways in which tuples should be disabled, then solves this with a minimum-cost SAT solver. A solution represents a set of tuples to disable. For example, the Boolean formula for the above two proofs is:

```
¬ gre(ra, tunnel_0, 100, 300) ∨ ¬ipAddress(rx, eth_0, 300, 0) ∧
¬ ipAddress(rx, eth_0, 300, 0) ∨ ¬static_route(ra, 0, 32, 400)
```

The first formula states that to disable the first proof, either the `gre` tuple or the `ipAddress` tuple must be disabled. The second formula states that to disable the second proof, either the `ipAddress` or the `static_route` tuple must be disabled. Costs are associated with disabling each tuple. The minimum-cost SAT

solver computes that set of tuples whose cost of disabling is a minimum. For example, the cost of disabling the `ipAddress` tuple may be high because many requirements depend on this tuple. The cost of disabling the `static_route` and `gre tuples` may be a lot lower. It is not, in general, simple to assign cost to disabling a tuple. Furthermore, this approach only computes how to disable an undesirable requirement. It does not guarantee that disabled tuples will also not disable desirable requirements, unless these latter requirements are also expressed in Boolean logic and the combined constraint is solved.

9.5.5 Evaluating Firewall Requirements With Binary Decision Diagrams

Hamed et al. [34] evaluate firewall subsumption and rule redundancy using Ordered Binary Decision Diagrams [12]. Their algorithm is conceptually the same as in Section 9.4.4. It first transforms firewall policies into Boolean constraints upon source and destination addresses, source and destination ports and the protocol. These constraints are true only for those packets that are permitted by the firewall. These fields are represented as sequences of Boolean variables, e.g., an address field as a sequence of 32 variables and a port field as a sequence of 16 bits. The algorithm then checks whether combinations of constraints for evaluating subsumption and redundancy have a solution. Since constraints are represented as Ordered Binary Decision Diagrams, this check is straightforward. By contrast, ConfigAssure represents the above fields as integer variables and represents a policy as an arithmetic quantifier-free form constraint. It lets Kodkod transform this into a Boolean constraint and use a SAT solver to check satisfiability.

9.6 Related Work

9.6.1 Configuration Acquisition by Type Inference

Another approach to parsing configuration files is with the use of PADS/ML system [47]. Based on the functional language ML, PADS/ML describes the accepted language as if it were a type definition. PADS/ML supports the generation of parser, printer, data structure representation, and a generic interface to this representation. The generated code is in OCAML [43] language and additional tools, written in OCAML, then manipulate the internal data structure. This inter-

nal data structure is traversed to populate the relational database in the same way that the ANTLR abstract syntax tree is traversed.

Adaptive parsers are reported in [17]. These can modify the language they recognize when given examples of legal input. The inference system recognizes commands that are only handled in the abstract, much as the ANTLR grammar of IP Assure skips over some commands. Repeated instances of commands are used to generate new PADS/ML types, which are then further refined to provide access to fields in the commands. This means that as the IOS language evolves, the parser can evolve to provide an ever richer internal representation.

9.6.2 Symbolic Reachability Analysis

Instead of performing reachability analysis for each packet, a system for reachability analysis for sets of packets is described in Xie et al. [72]. This makes it is possible to evaluate a requirement such as "a change in static routes at one or more routers does not change the set of packets that can flow between two nodes". It is not feasible to evaluate such a requirement by enumerating all packets and checking reachability. In this system, the reachability upper bound is defined to be the *union* of all packets permitted by each possible forwarding path from the source to the destination. This bound models a security policy that denies some packets (i.e., those outside the upper bound) under all conceivable operational conditions. The reachability lower bound is defined to be the common set of packets allowed by *every* feasible forwarding path from the source to the destination. This bound models a resilience policy that assures the delivery of some packets despite network faults, as long as a backup forwarding path exists. Algorithms are created for estimating the reachability upper and lower bounds from a network's packet filter configurations. Moreover, the work shows it is possible to jointly reason about how packet filters, routing, and packet transformations affect reachability.

An interesting implementation of reachability analysis for sets of packets is found in the ConfigChecker [3] system. It represents the network's packet forwarding behavior as a giant state machine in which a state defines what packets are at what routers. However, the state-transition relation is not represented explicitly but rather symbolically as a constraint that must be satisfied by two states for the network to transition between these. This constraint itself is represented as an Ordered Binary Decision Diagram and input to a symbolic model checker [48]. Reachability requirements such as that above, are expressed in Computational Tree Logic [48] and the symbolic model checker used to evaluate these. The transition-relation also takes into account features such as IPSec tunnels, multicast and network address translation.

9.6.3 Alloy Specification Language

Alloy [2, 39] is a first-order relational logic system. It lets one specify object types and their attributes. It also lets one specify first-order logic constraints on these attributes. These are more expressive than Prolog constraints. Alloy solves constraints by compiling these into Kodkod and using Kodkod's constraint solver. The use of Alloy for network configuration management was explored in [49]. Alloy's specification language is very appropriate for specifying requirements. All of the requirements in Section 2 can be compactly expressed in Alloy. However, its constraint solver is inappropriate for evaluating requirements. This is because the compilation of first-order logic into Boolean logic leads to very large intermediate constraints. Kodkod addresses this problem by its partial-model optimization that exploits knowledge about parts of the solution. If the value of a variable is already known, it does not appear in the constraint that is submitted to the SAT solver. ConfigAssure follows a related approach but at a higher layer. The intuition is that given a requirement, many parts of it can be efficiently solved with non-SAT methods. Solving these parts and simplifying can yield a requirement that truly requires the power of a SAT solver. This plan is carried out by transforming a requirement into an equivalent quantifier-free form by defining the `eval` predicate for that requirement. QFFs have the property that not only is it easy to write `eval` rules, but also that QFFs are efficiently compiled and solved by Kodkod. Evaluation of parts of requirements and simplification are accomplished in the definition of `eval`.

9.6.4 BGP Validation

The Internet is, by definition, a "network of networks," and the responsibility for gluing together the tens of thousands of independently-administered networks falls to the Border Gateway Protocol (BGP) [59, 64]. A network, or *Autonomous System* (AS), uses BGP to tell neighboring networks about each block of IP addresses, it can reach; in turn, neighboring ASes propagate this information to their neighbors, allowing the entire Internet to learn how to direct packets toward their ultimate destinations. On the surface, BGP is a relatively simple path-vector routing protocol, where each router selects a single best route among those learned from its neighbors, adds its own AS number to the front of the path, and propagates the updated routing information to its neighbors for their consideration; packets flow in the reverse direction, with each router directing traffic along the chosen path in a hop-by-hop fashion.

Yet, BGP is a highly configurable protocol, giving network operators significant control over how each router selects a "best" route and whether that route is disseminated to its neighbors. The configuration of BGP across the many routers

in an AS collectively expresses a routing policy that is based on potentially complex business objectives [15]. For example, a large Internet Service Provider (ISP) uses BGP policies to direct traffic on revenue-generating paths through their own downstream customers, rather than using paths through their upstream providers. A small AS like a university campus or corporate network typically does not propagate a BGP route learned from one upstream provider to another, to avoid carrying data traffic between the two larger networks. In addition, network operators may configure BGP to filter unexpected routes that arise from configuration mistakes and malicious attacks in other ASes [52, 14]. BGP configuration also affects the scalability of the AS, where network operators choose not to propagate routes for their customers' small address blocks to reduce the size of BGP routing tables in the rest of the Internet. Finally, network operators tune their BGP configuration to direct traffic away from congested paths to balance load and improve user-perceived performance [25].

The routing policy is configured as a "route map" that consists of a sequence of clauses that match on some attributes in the BGP route and take a specific action, such as discarding the route or modifying its attributes with the goal of influencing the route-selection process. The BGP protocol defines many different attributes, and the route-selection process compares the routes one attribute at a time to ultimately identify one "best" route. This somewhat indirect mechanism for selecting and propagating routes, coupled with the large number of route attributes and route-selection steps, makes configuring BGP routing policy immensely complicated and error-prone. Network operators often use tools for automatically configuring their BGP-speaking routers [29, 11, 21]. These tools typically consist of a template that specifies the sequence of vendor-specific commands to send to the router, with parameters unique to each BGP session populated from a database; for example, these parameters might indicate a customer's name, AS number, address block(s), and the appropriate route-maps to use. When automated tools are not used, the network operators typically have configuration-checking tools to ensure that the sessions are configured correctly, and that different sessions are configured in a consistent manner [24, 16].

Configuring the BGP sessions with neighboring ASes, while important, is not the only challenge in BGP configuration. In practice, an AS consists of multiple routers in different locations; in fact, a large ISP may easily have hundreds if not thousands of routers connected by numerous links into a backbone topology. Different routers connect to different neighbor ASes, giving each router only a partial view of the candidate BGP routes. As such, large ISPs typically run BGP *inside* their networks to allow the routers to construct a more complete view of the available routes. These internal BGP (iBGP) sessions must be configured correctly to ensure that each router has all the information it needs to select routes that satisfy the AS's policy. The simplest solution is to have a "full mesh" configuration, with an iBGP session between each pair of routers. However, this approach does not scale, forcing large ISPs to introduce hierarchy by configuring *route reflectors* or *confederations* that limit the number of iBGP sessions and constrain the dissemi-

nation of routes. Each route reflector, for instance, selects a single "best route" that it disseminates to its clients; as such, the route-reflector clients do not learn all the candidate routes they would have learned in a full-mesh configuration.

When the "topology" formed by these iBGP sessions violates certain properties, routing anomalies like protocol oscillations, forwarding loops, traffic black-holes, and violations of business contracts can arise [31, 6, 74]. Fortunately, static analysis of the iBGP topology, spread over the configuration of the routers inside the AS, can detect when these problems might arise [24]. Such tools check, for instance, that the top-level route reflectors are fully connected by a "full mesh" of iBGP sessions. This prevents "signaling partitions" that could prevent some routers from learning *any* route for a destination. Static analysis can also check that route reflectors are "close" to their clients in the underlying network topology, to ensure that the route reflectors make the same routing decisions that their clients would have made with full information about the alternate routes. Finally, these tools can validate an ISP's own local rules for ensuring reliability in the face of router failures. For instance, static analysis can verify that each router is configured with at least two route-reflector parents. Collectively, these kinds of checks on the static configuration of the network can prevent a wide variety of routing anomalies.

For the most part, configuration-validation tools operate on the vendor-specific configuration commands applied to individual routers. Configuration languages vary from one vendor to another---for example, Cisco and Juniper routers have very different syntax and commands, even for relatively similar configuration tasks. Even within a single company, different router products and different generations of the router operating system have different commands and options. This makes configuration validation an immensely challenging task, where the configuration-checking tools much support a wide range of languages and commands. To address these challenges, research and standards activities have led to new BGP configuration languages that are independent of the vendor-specific command syntax [1, 71], particularly in the area of BGP routing policy. In addition to abstracting vendor-specific details, these frameworks provide some support for configuring entire networks rather than individual routers. For example, the Routing Policy Specification Language (RPSL) [1] is object-oriented, where objects contain AS-wide policy and administrative information that can be published in Internet Routing Registries [37]. Routing policy can be expressed in terms of user-friendly keywords for defining actions and groups of address blocks or AS number. Configuration-generation tools can read these specifications to generate vendor-specific commands to apply to the individual routers [37]. However, while RPSL is used for publishing information in the IRRs, many ISPs still use their own configuration tools (or manual processes) for configuring their underlying routers.

In summary, the configuration of BGP takes place at many levels---within a single router (to specify a single end-point of a BGP session with the appropriate route-maps and addresses), between pairs of routers (to ensure consistent configu-

ration of the two ends of a BGP session), across different sessions to the same neighboring AS (to ensure consistent application of the routing policy at each connection point), and across an entire AS (to ensure the iBGP topology is configured correctly). In recent years, tools have emerged for static analysis of router-configuration data to identify potential configuration mistakes, and for automated generation of the configuration commands that are sent to the routers. Still, many interesting challenges remain in raising the level of abstraction for configuring BGP, to move from the low-level focus on configuring individual routers and BGP sessions toward configuring an entire network, and from the specific details of the BGP route attributes and route-selection process to a high-level specification of an AS's routing policy. As the Internet continues to grow, and the business relationships between ASes become increasingly complex, these issues will only become more important in the years ahead.

9.6.5 Other Validation Systems

Netsys was an early software product for configuration validation. It was first acquired by Cisco Systems and then by WANDL Corporation. It contained about a hundred requirements that were evaluated against router configurations. OPNET offers validation products NetDoctor and NetMapper. These are not standalone but rather modules that need to be plugged into the base IT Sentinel system [54]. For more description of these, see [23]. None of these products offer configuration repair, reasoning about firewalls or symbolic reachability analysis. The Smart Firewalls work [13] was an early attempt at Telcordia to develop a network configuration validation system. A survey of system, not network, configuration is found in [4]. Formal methods for jointly reasoning about IPSec and firewall polices are described in [32]. A high-level configuration language is described in [45].

9.7 Summary And Directions for Future Research

To set up network infrastructure satisfying end-to-end requirements, it is not only necessary to run appropriate protocols on components but also to correctly configure these components. Configuration is the "glue" for logically integrating components at and across multiple protocol layers. Each component has a finite number of configuration parameters each of which can be set to a definite value. However, today, the large conceptual gap between end-to-end requirements and configurations is manually bridged. This causes large numbers of configuration errors whose adverse effects on security, reliability and high cost of deployment of network infrastructure are well-documented.

Thus, it is critical to develop validation tools that check whether a given configuration is consistent with the requirements it is intended to implement. Besides checking consistency, configuration validation has another interesting application, namely, network testing. The usual invasive approach to testing has several limitations. It is not scalable. It consumes resources of the network and network administrators and has the potential to unleash malware into the network. Some properties such as absence of single points of failure are impractical to test as they require failing components in operational networks. A non-invasive alternative that overcomes these limitations is analyzing configurations of network components. This approach is analogous to testing software by analyzing its source code rather than by running it. This approach has been evaluated for a real enterprise.

Configuration validation is inherently hard. Whether a component is correctly configured cannot be evaluated in isolation. Rather, the global relationships into which the component has been logically integrated with other components have to be evaluated. Configuration repair is even harder since changing configurations to make one requirement true may falsify another. The configuration change should be holistic in that it should ensure that all requirements concurrently hold.

This chapter described the challenges of configuring a typical collaboration network and the benefits of using a validation system. It then presented an abstract design of a configuration validation system. It consists of four subsystems: Configuration Acquisition System, Requirement Library, Specification Language and Evaluation System. The chapter then surveyed technologies for realizing this design. Configuration acquisition systems have been built using three approaches: parser generator, type inference and database query. Classes of requirements in their Requirements Library are logical structure integrity, connectivity, security, reliability, performance, and government regulatory. Specification languages include visual templates, Prolog, Datalog, arithmetic quantifier-free forms and Computational Tree Logic. Evaluation systems have used graph algorithms, the Kodkod constraint solver for first-order logic constraints, the ZChaff SAT solver for Boolean constraints, Binary Decision Diagrams and symbolic model checkers. Visualization of not just the IP topology but also of various other logical topologies provides useful insights into network architecture. Logic-based languages are very useful for creating a validation system, particularly for solving the hard problems of configuration repair and symbolic reasoning about requirements.

Future research needs to focus on all four components of a validation system. Robust configuration acquisition systems are critical to automated validation. The accumulated experience of building large networks is vast but largely unformalized. Formalizing these in a Requirement Library would not only raise the level of abstraction at which network requirements are written but also improve their precision. New classes of requirements, one on VLAN optimization and another on configuration complexity are reported in [65] and in [9] respectively. Specification languages that are easy to use by network administrators are also critical for broad adoption of validation systems. Logic-based languages are a good candidate despite the perception that these are too complex for administrators. These are

closest in form to the natural language requirements in network design documents. The configuration languages administrators use are already declarative in that they do not contain side effects and the ordering of commands is unimportant. Introducing logical operators, data structures and quantifiers into these is a natural step towards making these much more expressive. See [71] for a recent example of using the Haskell functional language for specifying BGP policies. High-level descriptions of component configurations could then again be composed by logical operators to describe network-wide requirements. In the nearer term, even making an implementation of the Requirement Library available as APIs in system administration languages like Perl or Python should vastly improve configuration debugging. Much greater understanding is needed of useful ways to visualize logical structures and relationships in networks. One might derive inspiration from works such as [70]. Finally, a good framework for repairing configurations was described in Section 9.5.3 but it needs to be further explored. For example, one needs to understand how the convergence of the repair procedure is affected by choice of configuration variable to relax, and how ideas of MulVAL can be generalized and combined with those of ConfigAssure. Creating the trust in network administrators before they allow automated repair of their component configurations is an open problem.

Acknowledgments We are very grateful to Jennifer Rexford, Andreas Voellmy, Richard Yang, Chuck Kalmanek, Simon Ou, Geoffrey Xie, Yitzhak Mandelbaum, Ehab Al-Shaer, Sanjay Rao, Adel El-Atawy and Paul Anderson for their contributions and comments.

9.8 References

1 Alaettinoglu C, Villamizar C, Gerich E, Kessens D, Meyer D, Bates T, Karrenberg D, Terpstra M (1999) Routing Policy Specification Language. RFC 2622
2 Alloy: http://alloy.mit.edu/
3 Al-Shaer E, Marrero W, El-Atawy A, ElBadawy K (2008) Towards Global Verification and Analysis of Network Access Control Configuration. Technical Report, TR-08-008, DePaul University http://www.mnlab.cs.depaul.edu/projects/ConfigChecker/TR-08-008/paper.pdf
4 Anderson P (2006) System Configuration. In Short Topics in System Administration ed. Rick Farrow. USENIX Association
5 ANTRL v3. http://www.antlr.org/
6 Basu A, Ong CH, Rasala A, Shepherd FB, Wilfong G (2002) Route oscillations in I-BGP with route reflection. ACM SIGCOMM
7 Bates T, Chandra R, Chen E (2000) BGP Route Reflection – An Alternative To Full Mesh IBGP. RFC 2796 http://www.faqs.org/rfcs/rfc2796
8 Bellovin R, Bush, R (2009) Configuration Management and Security. IEEE Journal on Selected Areas in Communications, Special Issue on Network Infrastructure Configuration, Vol. 27, No. 3.
9 Benson T, Akella A, Maltz, D (2009) Unraveling the Complexity of Network Management. USENIX Symposium on Network Systems Design and Implementation.

10 Berkowitz H (2000) Techniques in OSPF-Based Network. http://tools.ietf.org/html/draft-ietf-ospf-deploy-00.
11 Bohm H, Feldmann A, Maennel O, Reiser C, and Volk R (2005) Network-wide inter-domain routing policies: Design and realization. Unpublished report, http://www.net.t-labs.tu-berlin.de/papers/BFMRV-NIRP-05.pdf
12 Bryant, R. (1986) Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, C-35(8):677–691
13 Burns J, Cheng A, Gurung P, Martin D, Rajagopalan S, Rao P, Alathurai V (2001) Automatic management of network security policy. Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX II'01), volume 2, Anaheim, California
14 Butler K, Farley T, McDaniel P, Rexford J (2008) A survey of BGP security issues and solutions. In submission
15 Caesar M, Rexford J (2005) BGP routing policies in ISP networks. IEEE Network Magazine, special issue on interdomain routing
16 Caldwell D, Gilbert A, Gottlieb J, Greenberg A, Hjalmtysson G, and Rexford J (2003) The cutting EDGE of IP router configuration. ACM SIGCOMM HotNets Workshop
17 Caldwell D, Lee S, Mandelbaum Y (2008) Adaptive parsing of router configuration languages. In Proceedings of the Internet Management Workshop
18 Cheswick W, Bellovin S, Rubin A (2003) Firewalls and Internet Security: Repelling the Wily Hacker, Addison-Wesley
19 Cisco Systems (2005) DiffServ -- The Scalable End-to-End QoS Model
20 Distributed Management Task Force. http://www.dmtf.org/home
21 Enck W, Moyer T, McDaniel P, Sen S, Sebos P, Spoerel S, Greenberg A, Sung Y-W, Rao S, Aiello W, (2009) Configuration Management at Massive Scale: System Design and Experience. IEEE Journal on Selected Areas in Communications
22 Farinacci D, Li T, Hanks S, Meyer D, Traina P (2000) Generic Routing and Encapsulation. RFC 2784
23 Feamster N (2006) Proactive Techniques for Correct and Predictable Internet Routing. Ph.D. Thesis, Massachusetts Institute of Technology.
24 Feamster N, Balakrishnan H (2005) Detecting BGP configuration faults with static analysis. Symposium on Networked Systems Design and Implementation
25 Feamster N, Rexford (2007) Network-wide prediction of BGP routes. IEEE/ACM Transactions on Networking
26 Federal Information Security Management Act (2002) National Institute of Standards and Technology.
27 Fu Z, Malik S (2006) Solving the Minimum-Cost Satisfiability Problem using Branch and Bound Search. Proceedings of IEEE/ACM International Conference on Computer-Aided Design ICCAD
28 Garimella P, Sung YW, Zhang N, Rao S (2007) Characterizing VLAN usage in an Operational Network. ACM SIGCOMM Workshop on Internet Network Management
29 Gottlieb J, Greenberg A, Rexford J, Wang J (2003) Automated provisioning of BGP customers IEEE Network Magazine
30 Graphviz. http://www.graphviz.org/
31 Griffin TG and Wilfong G (2002) On the correctness of IBGP configuration. Proc. ACM SIGCOMM
32 Guttman J (1997) Filtering postures: local enforcement for global policies. Proceedings of the 1997 IEEE Symposium on Security and Privacy
33 Halabi B (1997) Internet Routing Architectures. New Riders Publishing
34 Hamed H, Al-Shaer E and Will Marrero (2005) Modeling and Verification of IPSec and VPN Security Policies, Proceedings of IEEE International Conference on Network Protocols.

35 Homer J, Ou X (2009) SAT-solving approaches to context-aware enterprise network security management. IEEE JSAC Special Issue on Network Infrastructure Configuration

36 Huitema C (1999) Routing in the Internet. Prentice Hall

37 Internet Routing Registry Toolset Project, https://www.isc.org/software/IRRtoolset

38 IP Assure. Telcordia Technologies, Inc. http://www.telcordia.com/products/ip-assure/

39 Jackson D (2006) Software Abstractions: Logic, Language, and Analysis. MIT Press

40 Juniper Networks (2008) What is Behind Network Downtime? Proactive Steps to Reduce Human Error and Improve Availability of Networks. http://www.juniper.net/solutions/literature/white_papers/200249.pdf

41 Kodkod. http://web.mit.edu/emina/www/kodkod.html

42 Lampson B (2000) Computer Security in Real World. Annual Computer Security Applications Conference. http://research.microsoft.com/Lampson/64SecurityInRealWorld/Acrobat.pdf

43 Leroy X, Doligez D, Garrigue J, Rémy D, and Vouillon J (2007) The objective caml system, release 3.10, documentation and user's manual

44 Li T, Cole B, Morton P, Li D (1998) Cisco Hot Standby Router Protocol. RFC 2281

45 Lobo J, Pappas V (2008) C2: The case for network configuration checking language. Proceedings of IEEE Workshop on Policies for Distributed Systems and Networks

46 Mahajan Y, Fu Z, Malik S (2004) Zchaff2004, An Efficient SAT Solver. Proceedings of 7th International Conference on Theory and Applications of Satisfiability Testing

47 Mandelbaum Y, Fisher K, Walker D, Fernandez M, and Gleyzer A (2007) PADS/ML: A functional data description language. ACM Symposium on Principles of Programming Language

48 McMillan K (1992) Symbolic Model Checking. Ph.D. Thesis, Computer Science Department, Carnegie Mellon University.

49 Narain S (2005) Network Configuration Management via Model-Finding. Proceedings of USENIX Large Installation System Administration (LISA) Conference

50 Narain S, Kaul V, Parmeswaran K (2003) Building Autonomic Systems via Configuration. Proceedings of AMS Autonomic Computing Workshop

51 Narain S, Levin G, Kaul V, Malik, S (2008) Declarative Infrastructure Configuration Synthesis and Debugging. Journal of Network Systems and Management, Special Issue on Security Configuration, eds. Ehab Al-Shaer, Charles Kalmanek, Felix Wu

52 Nordstrom O and Dovrolis C (2004) Beware of BGP attacks. ACM SIGCOMM Computer Communications Review, vol. 34, no. 2, pp. 1-8

53 O'Keefe R (1990) The Craft of Prolog. Addison Wesley

54 OPNET IT Sentinel. http://www.opnet.com/solutions/network_planning_operations/it_sentinel.html

55 Ou X, Boyer W, McQueen M (2006) A scalable approach to attack graph generation. In 13th ACM Conference on Computer and Communications Security (CCS)

56 Ou X, Govindavajhala S, Appel A (2005) MulVAL: A logic-based network security analyzer. 14th USENIX Security Symposium, Baltimore, MD

57 Pappas V, Wessels D, Massey D, Terzis A, Lu S, Zhang L (2009) Impact of Configuration Errors on DNS Robustness. IEEE Journal on Selected Areas in Communication

58 Qie X, Narain, S (2003) Using Service Grammar to Diagnose Configuration Errors in BGP-4. Proceedings of USENIX Systems Administrators Conference

59 Rekhter Y, Li T, Hares S (2006) A Border Gateway Protocol 4 (BGP-4), RFC 4271

60 Rosen E, Viswanathan A, Callon R (2001) Multiprotocol Label Switching Architecture. RFC 3031

61 Schwartz J (2007) Who Needs Hackers? New York Times http://www.nytimes.com/2007/09/12/technology/techspecial/12threat.html.

62 Securing Cyberspace for the 44th Presidency" (2008)CSIS Commission On Cybersecurity

63 Sedgewick R (2003) Algorithms in Java. Addison Wesley Professional
64 Stewart J (1999) BGP4: Inter-Domain Routing in the Internet. Addison-Wesley
65 Sung EY, Rao S, Xie G, and Maltz D (2008) Towards Systematic Design of Enterprise Networks. ACM CoNEXT Conference
66 SWI-Prolog Semantic Web Library. http://www.swi-prolog.org/pldoc/package/semweb.html
67 SWI-Prolog: http://www.swi-prolog.org/
68 TCP Problems with Path MTU discovery. RFC 2923
69 Torlak E, Jackson D (2007) Kodkod: A Relational Model Finder. Tools and Algorithms for Construction and Analysis of Systems (TACAS '07)
70 Tufte E (2001) The visual display of quantitative information. Graphics Press.
71 Voellmy A, Hudak P Nettle: A domain-specific language for routing configuration," http://bebop.cs.yale.edu/voellmy/nettle.html
72 Xie G, Zhan J, Maltz D, Zhang H, Greenberg A, Hjalmtysson G, and Rexford J (2005) On Static Reachability Analysis of IP Networks. IEEE INFOCOM
73 ZChaff: http://www.princeton.edu/~chaff/
74 Zhang-Shen R, Wang Y, and Rexford J (2008) Atomic routing theory: Making an AS route like a single node. Princeton University Computer Science technical report TR-827-08