Network Configuration Management via Model Finding

Sanjai Narain – Telcordia Technologies, Inc.

ABSTRACT

Complex, end-to-end network services are set up via the configuration method: each component has a finite number of configuration parameters each of which is set to a definite value. End-to-end network service requirements can be on connectivity, security, performance and fault-tolerance. However, there is a large conceptual gap between end-to-end requirements and detailed component configurations. To bridge this gap, a number of subsidiary requirements are created that constrain, for example, the protocols to be used, and the logical structures and associated policies to be set up at different protocol layers.

By performing different types of reasoning with these requirements, different configuration tasks are accomplished. These include configuration synthesis, configuration error diagnosis, configuration error fixing, reconfiguration as requirements or components are added and deleted, and requirement verification. However, such reasoning is currently ad hoc. Network requirements are not even precisely specified hence automation of reasoning is impossible. This is a major reason for the high cost of network management and total cost of ownership. This paper shows how to formalize and automate such reasoning using a new logical system called Alloy.

Alloy is based on the concept of model finding. Given a first-order logic formula and a domain of interpretation, Alloy tries to find whether the formula is satisfiable in that domain, i.e., whether it has a model. Alloy is used to build a Requirement Solver that takes as input a set of network components and requirements upon their configurations and determines component configurations satisfying those requirements.

This Solver is used in different ways to accomplish the above reasoning tasks. The Solver is illustrated in depth by carrying out a variety of these tasks in the context of a realistic fault-tolerant virtual private network with remote access. Alloy uses modern satisfiability solvers that solve millions of constraints in millions of variables in seconds. However, poor requirements can easily nullify such speeds. The paper outlines approaches for writing *efficient* requirements. Finally, it outlines directions for future research.

Introduction

Complex, end-to-end network services are set up via the configuration method: each component has a finite number of configuration parameters each of which is set to a definite value. End-to-end network service requirements can be on connectivity, security, performance and fault-tolerance. However, there is a large conceptual gap between end-to-end requirements and detailed component configurations. To bridge this gap, a number of subsidiary requirements are created that constrain, for example, the protocols to be used, and the logical structures and associated policies to be set up at different protocol layers.

By performing different types of reasoning with these requirements, different configuration tasks are accomplished. These include configuration synthesis, configuration error diagnosis, configuration error fixing, reconfiguration as requirements or components are added and deleted, and requirement verification. However, such reasoning is currently ad hoc. Network requirements are not even precisely specified hence automation of reasoning is impossible. This is a major reason for the high cost of network management and total cost of ownership.[1] This paper shows how to formalize and automate such reasoning using the new concept of a Requirement Solver, as shown in Figure 1.

This Solver takes as input a set of network components and requirements upon their configurations and

[1] "... operator error is the largest cause of failures ... and largest contributor to time to repair .. in two of the three (surveyed) ISPs .. configuration errors are the largest category of operator errors." [1]

"Although setup (of the trusted computing base) is much simpler than code, it is still complicated, it is usually done by less skilled people, and while code is written once, setup is different for every installation. So we should expect that it's usually wrong, and many studies confirm this expectation." [2]

"Consider this: ..the complexity (of computer systems) is growing beyond human ability to manage it..the overlapping connections, dependencies, and interacting applications call for administrative decision-making and responses faster than any human can deliver. Pinpointing root causes of failures becomes more difficult." [3]

computes as output, component configurations satisfying those requirements. The requirements can be in first-order logic (Boolean logic with quantifiers on individual variables). This logic is highly expressive and captures a very large class of practical network requirements.

The Requirement Solver is used in different ways to accomplish the above reasoning tasks. The Solver is illustrated in depth by carrying out a variety of these tasks in the context of a realistic fault-tolerant virtual private network with remote access. The reasoning tasks are accomplished as follows:

1. **Configuration Synthesis.** To determine how to configure a set of components so they satisfy a system requirement R, submit the set of components and R to the solver and take the output.
2. **Requirement Strengthening.** If a set of components satisfies a system requirement R but must now satisfy another requirement S, then to reconfigure components, submit the set of components and R∧S to the solver and take the output.
3. **Component Addition.** If a new component is to be added to a set of components already satisfying requirement R, then to configure the new component and possibly reconfigure existing components, submit the new set of components and R to the Solver and take the output.
4. **Requirement Verification.** To prove that it is impossible for an undesirable requirement U to be true when a set of components satisfies requirement R, submit the set of components and R∧U to the Solver. If the Solver cannot find a solution, the assertion is proved. Otherwise, the Solver produces a counterexample.
5. **Configuration Error Detection.** To check whether configuration of a given set of components is consistent with a requirement R, represent the configuration as a set C of constraints each of the form P=V where P is a configuration parameter and V its value. Then, submit the set of components and R∧C to the solver. If the solver cannot produce a solution, a configuration error is detected.
6. **Configuration Error Fixing.** If the configuration of a given set of components is inconsistent with a requirement R, then submit the set of components and R to the Solver and find a new solution that is as "close" as possible to the current configuration.

The Requirement Solver has been inspired by the new logical system called Alloy [4]. While Alloy is based in set theory, a subset of it also has an intuitive object-oriented interpretation: it lets one specify object types, their attributes and type of attribute values. It also lets one specify first-order logic constraints on these. Finally, it lets one specify a "scope" that defines a *finite* number of object instances of each type in a given system.

Given a specification and a scope, Alloy attempts to find values of attributes of object instances in the scope that satisfy the specification. These values together constitute a "model" of the specification in the system in the logical sense of the word "model". Alloy first compiles a specification into a propositional formula in conjunctive normal form, then uses a satisfiability solver such as Berkmin [5] or Zchaff [6] to check whether the formula is satisfiable. If so, it converts satisfying values of propositional variables back into values of attributes and displays these.

Often, more than one solution is found. Even though satisfiability is NP-complete in the worst case, in the *average* case, solutions are efficiently found. Modern satisfiability solvers can solve millions of constraints in millions of variables in seconds. However, poor requirements can easily nullify such speeds. The paper outlines approaches for writing *efficient* requirements. Finally, it outlines directions for future research.

The Solver has a direct implementation in Alloy. Network component types, attributes and values are directly modeled in Alloy. A set of network components of different types is modeled as a scope. Network system configuration is modeled as values of all component attributes in a given scope. Network requirements are modeled using first-order logic. Solutions are found by the Alloy model finder.

Subsequent sections illustrate Alloy's capabilities, describe the design of a fault-tolerant VPN with remote access and the challenges of setting it up, outline a

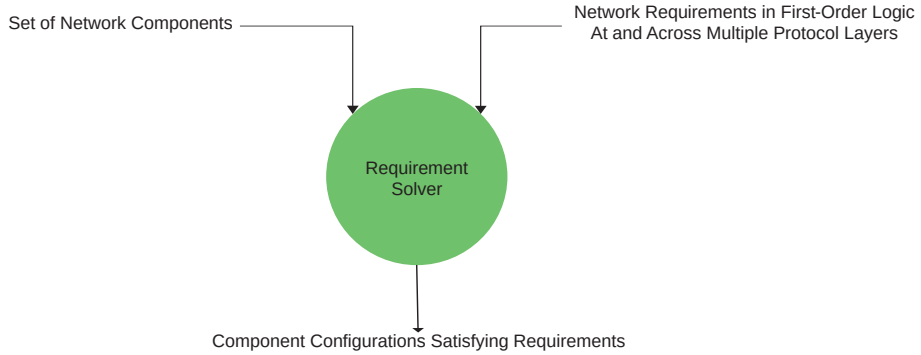

Figure 1: Requirement Solver.

formalization of the design in Alloy, describe how to accomplish, respectively, tasks 1-4 above, outline approaches for writing efficient requirements, outline relationship with previous work, summarize this work and the conclusions, and present directions for future research.

Alloy By Example

The three lines in Display 1 declare three object types: router, subnet and interface. The last type has two attributes, the chassis (of type router) to which it belongs, and network (of type subnet) on which it is placed. These attributes model configuration parameters of an interface.

The predicate spec in Display 2 defines a specification whose model we will try to find. It is a conjunction of three constraints. The first states that for every router x there is an interface y whose chassis is x, i.e., every router has at least one interface. The second states that no two non-equal interfaces on the same router are placed on the same subnet. The third states that our network contains one router, two subnets and two interfaces. These are the components we want to configure.

The Alloy command shown in Display 3 produces the model (values of configuration parameters) shown in Display 4. The first line states that the value of chassis for interface_0 is router_0 and the value of chassis for interface_1 is router_0. Similarly, for the second line. Alloy automatically creates instances of objects such as router_0, subnet_0, and subnet_1. Note that Alloy did not place interface_0 and interface_1 on the same subnet due to the second constraint. On the other hand, if we remove that constraint, Alloy produces the additional solution show in Display 5 in which both interface_0 and interface_1 are placed on the same subnet.

Fault-Tolerant Virtual Private Network With Remote Access

Our top-level goal is to synthesize a fault-tolerant network that enables hosts, including mobile hosts, at geographically distributed sites to securely collaborate. A network design for achieving this goal is now outlined. When this is implemented via configuration, one obtains a network of the type shown in Figure 2. The existence of a wide-area network, represented by the WAN router in the figure, is assumed.

Each site has a gateway router called a spoke router whose external (or public) interface is connected to the WAN and whose internal (or private) interface is connected to hosts and servers in the site. A routing protocol is run on the external interfaces of spoke and WAN routers to automatically compute routes between these interfaces. As traffic between hosts and servers on different sites is intended to be private, it cannot be allowed to flow directly over the wide area network. In order to secure it, one possibility is to set up a full-mesh of IPSec tunnels between gateway routers. However, full-mesh is not scalable since the number of tunnels increases as the square of the number of sites: for the 200 sites expected in our domain, the number of tunnels would be nearly 20,000.

A scalable alternative is a hub-and-spoke architecture as shown. A certain number of hub routers is set up. Each spoke router sets up an IPSec tunnel to each hub router. Traffic from one site to another goes via two tunnel hops, one from its spoke router to a hub router and another from the hub router to the destination site's spoke router. The number of tunnels now only increases linearly with the number of sites.

The problem, however, is that if a hub router fails, connectivity between sites is lost. This is because the

```
sig router {}
sig subnet{}
sig interface {chassis: router, network: subnet}
```

Display 1: Declaring three object types.

```
pred spec ()
      {all x:router | some y:interface | y.chassis = x}
      {no disj x1,x2:interface |
            x1.chassis=x2.chassis && x1.network = x2.network}
      {#router=1 && #subnet=2 && #interface=2}
```

Display 2: A specification with three constraints.

```
run spec for 1 router, 2 subnet, 2 interface
```

Display 3: Alloy command to create a model.

```
chassis : = {interface_0 -> router_0, interface_1 -> router_0}
network : = {interface_0 -> subnet_1, interface_1 -> subnet_0}
```

Display 4: Results of command in Display 3.

```
chassis : = {interface_0 -> router_0, interface_1 -> router_0}
network : = {interface_0 -> subnet_0, interface_1 -> subnet_0}
```

Display 5: Less constrained solution.

source spoke router will continue to send traffic through the IPSec tunnel to the failed hub router. IPSec has no notion of fault-tolerance that will enable the source spoke router to redirect traffic via another hub router.

Routing protocols such as RIP or OSPF accomplish precisely this kind of fault-tolerance, however they are incompatible with IPSec. They do not recognize an IPSec tunnel as directly connecting two routers since there can be multiple physical hops in between. The solution is to set up a new type of tunnel, called GRE, between each hub and spoke router. The purpose of GRE is to create the illusion to a routing protocol that two routers are directly connected, even when there are multiple physical hops in between.

This is done by creating new GRE interfaces at each tunnel end point and making these belong to the same point-to-point subnet. Now, if a hub router fails, a routing protocol will automatically direct traffic through another GRE tunnel to another hub router, and then to the destination. Each GRE tunnel is then secured via an IPSec tunnel. Thus, the required fault-tolerant virtual private network is set up. If two hub router failures are to be tolerated, then three hub routers are required. Then, the number of tunnels to be set up is just 600 (number of hub routers × number of spoke routers) or 3% of nearly 20,000 in the full mesh case.

This solution has a useful defense-in-depth feature: there are two separate routing domains, the external one and the overlay one. No routes are exchanged between these. Thus, even if an adversary compromises the WAN router, he cannot send a packet to a host. The WAN router does not even have a route to the host.

In order to enable remote users to securely collaborate, a remote access server is set up in "parallel" with a spoke router. A remote user connected to the WAN sets up an L2TP tunnel between his host and the server. This tunnel gives the illusion to the host of being directly connected to the internal interface of the sites' spoke router. Consequently, all traffic between the host and any other host or server on the VPN is also secured. Again, one has to ensure that two separate routing protocols run on the access server, one for the private side and one for the public. In order to realize the above design, the following types of configuration parameters need to be set:

1. Addressing: Router interface address, type and subnet mask.
2. IPSec: Tunnel end points, hash and encryption algorithms, tunnel modes, preshared keys and traffic filters.
3. OSPF: Whether it is enabled at an interface, and OSPF area and type identifiers.
4. GRE: Tunnel end points and physical end points supporting GRE tunnels.
5. Firewall: Policies at each site.
6. Remote access: Subnets to which remote access server interfaces belong and routing protocols enabled on these.

It is very hard to compute values of the above configuration parameters. The types of configuration errors that can arise are:

1. Duplicate IP addresses may be set up, or all interfaces on a subnet may not have the same type.
2. IPSec tunnels may be set up incorrectly. For example, the preshared key, hash algorithm, encryption algorithm, or authentication mode may be unequal at the two tunnel end points. Peer values may not be mirror images of each other. These errors can lead to loss of connectivity. If the wrong traffic filter is used, then sensitive data can be transmitted without being encrypted.
3. OSPF routing domain may be set up incorrectly, for example, it may not be enabled at a required interface or the area and type identifiers may be

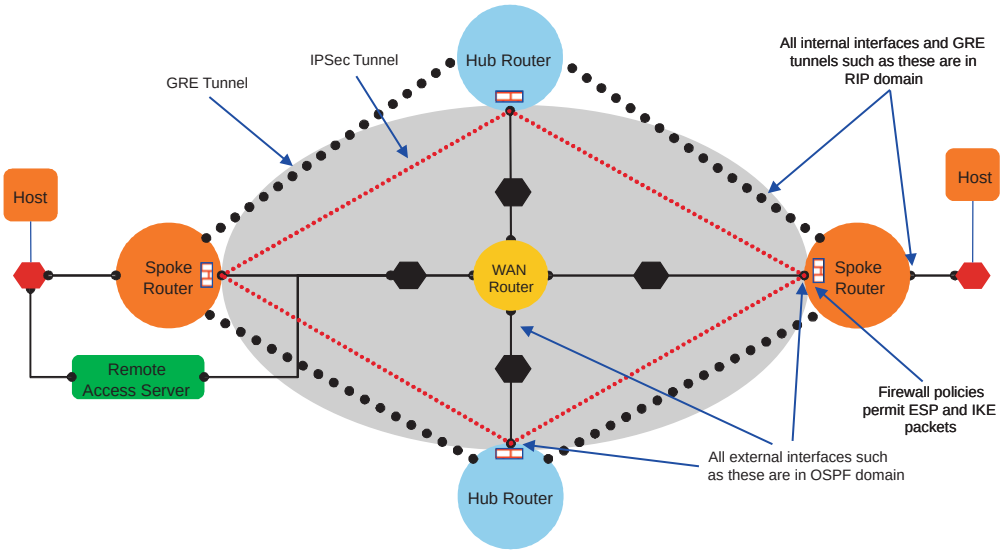

Figure 2: Fault-tolerant virtual private network with remote access.

incorrect. This can lead to incorrect routing tables and to outright isolation of subnets.
4. Routing loops may arise. If the same OSPF process is also used for routing between the gateway and WAN routers, then if it does not find a path through the physical network it will attempt to find a path through the overlay network. Since the overlay network is supported by the physical network, a routing loop will arise. This problem can be mitigated by using two distinct routing protocols, one for the overlay and another for the WAN.
5. GRE tunnels may be set up incorrectly. For example, the peer values may not be mirror images of each other, or the mapping between GRE ports and physical ports may be incorrect.
6. Firewall policies may block IPSec traffic, hence no traffic will pass through the tunnels.
7. Remote access interfaces may not belong to the correct subnets and incorrect routing protocols may be configured on these.

Before we show how to formalize the above design in Alloy, we capture its main intuitions in the following requirements:

- **RouterInterfaceRequirements**
  1. Each spoke router has internal and external interfaces
  2. Each access server has internal and external interfaces
  3. Each hub router has only external interfaces
  4. Each WAN router has only external interfaces
- **SubnettingRequirements**
  5. A router does not have more than one interface on a subnet
  6. All internal interfaces are on internal subnets
  7. All external interfaces are on external subnets
  8. Every hub and spoke router is connected to a WAN router
  9. No two non-WAN routers share a subnet
- **RoutingRequirements**
  10. RIP is enabled on all internal interfaces
  11. OSPF is enabled on all external interfaces
- **GRERequirements**
  12. There is a GRE tunnel between each hub and spoke router
  13. RIP is enabled on all GRE interfaces
- **SecureGRERequirements**
  14. For every GRE tunnel there is an IPSec tunnel between associated physical interfaces that secures all GRE traffic
- **AccessServerRequirements**
  15. There exists an access server and spoke router such that the server is attached in ‘‘parallel’’ to the router
- **AccessControlPolicyRequirements**
  16. Each hub and spoke external interface permits esp and ike packets

The interesting fact is these requirements do not specify the number of sites in the VPN. Rather, they apply to *all* sites. As new sites are added, these requirements are instantiated for the extended network to determine how to configure new components and reconfigure existing ones. This is a hard problem, in general, but that we show how to automatically solve with our approach.

Requirement Formalization In Alloy

This section presents an Alloy formalization of network component types, subtypes and their attributes. It also presents a formalization of Requirements 12 and 14. The complete formalization is available in the longer report at http://alloy.mit.edu/papers/NetConfigAlloy.pdf. Various types of routers are modeled using the following Alloy type declarations (signatures):

```
sig router {}
sig wanRouter extends router {}
sig hubRouter extends router {}
sig spokeRouter extends router {}
sig accessServer extends router {}
sig legacyRouter extends router {}
```

A generic interface just has a single attribute, the routing protocol enabled at it.

```
sig interface {routing:routingDomain}
```

A physical interface has two attributes, the router chassis on which it is mounted, and the network on which it is placed:

```
sig physicalInterface extends interface {
      chassis: router,
      network: subnet}
```

There are internal and external interfaces. External interfaces are of two types, one on hubs and one on spokes; see Display 6. There are two types of routing domains, RIP and OSPF:

```
sig routingDomain {}
sig ripDomain extends routingDomain {}
sig ospfDomain extends routingDomain {}
```

There are two types of subnets, internal and external:

```
sig subnet{}
sig internalSubnet extends subnet{}
sig externalSubnet extends subnet{}
```

There are three types of protocols, IKE, ESP and GRE:

```
sig internalInterface extends physicalInterface {}
sig externalInterface extends physicalInterface {}
sig hubExternalInterface extends externalInterface {}
sig spokeExternalInterface extends externalInterface {}
```

Display 6: Interfaces on hubs and spokes.

```
sig protocol {}
sig ike extends protocol {}
sig esp extends protocol {}
sig gre extends protocol {}
```

A firewall policy contains one of two possible permissions, permit and deny, that respectively, mean whether the firewall should allow a packet to go through or be dropped.

```
sig permission {}
sig permit extends permission {}
sig deny extends permission {}
```

A firewall policy, shown in Display 7, defines whether a packet associated with a protocol is allowed to go through or dropped, as it leaves an interface. An IPSec tunnel encrypts all packets associated with a protocol entering at either its local or its remote endpoint.

```
sig ipsecTunnel {
      local: externalInterface,
      remote: externalInterface,
      protocolToSecure: protocol}
```

A GRE tunnel encapsulates a packet into a new packet with source address that of its local endpoint and destination address that of its remote endpoint. Also, the tunnel is considered a proper link in a routing domain.

```
sig greTunnel {
      localPhysical: externalInterface,
      routing: routingDomain,
      remotePhysical: externalInterface}
```

An IP packet's attributes are its source and destination interfaces and the protocol it embodies. The precise data it carries is not modeled, since it is not relevant for our design purposes.

```
sig ipPacket {
      source:interface,
      destination:interface,
      prot:protocol}
```

Display 8 shows the Alloy version of Requirement 12. This states that between every hubExternalInterface x and spokeExternalInterface y there is a greTunnel whose local physical is x and remotePhysical is y, or vice versa.

Display 9 shows the Alloy version of Requirement 14. This states that for every greTunnel g there is an ipsecTunnel p that secures the gre protocol and whose endpoints are the same as the physical endpoints of g.

Configuration Synthesis

This section shows how to synthesize the initial network with connectivity and routing. Define:

```
PhysicalSpec =
      RouterInterfaceRequirements ∧
      SubnettingRequirements ∧
      RoutingRequirements
```

In Alloy, this would be expressed as:

```
Pred PhysicalSpec () {
      RouterInterfaceRequirements ()
      SubnettingRequirements ()
      RoutingRequirements ()}
```

Define a scope consisting of 1 hubRouter, 1 spokeRouter, 1 wanRouter, 1 internalInterface, 4 externalInterface, 1 hubExternalInterface, 1 spokeExternalInterface, 1 ripDomain, 1 ospfDomain, 3 subnet, 0 legacyRouter. These are the objects we want to configure. Now request Alloy to find a model for PhysicalSpec in the above scope. It synthesizes the network shown in Figure 3. It does so by producing the values of configuration parameters shown in Display 10. These are just the textual version of the network in Figure 3. Also note that spoke and hub routers are not directly connected, in accordance with Requirement 9.

Requirement Strengthening

In order to add an overlay network to the previous one, extend the previous scope with a GRE tunnel then request Alloy to satisfy (PhysicalSpec ∧ GRERequirements). Alloy synthesizes the network shown in Figure 4a. Alloy automatically sets up the GRE tunnel between the spoke and hub router and enables RIP routing on the GRE tunnel.

To make GRE tunnels secure, extend the previous scope with an IPSec tunnel and request Alloy to satisfy

```
sig FirewallPolicy {
       prot: protocol,
       action: permission,
       protectedInterface: physicalInterface}
```

Display 7: Firewall policy.

```
{all x:hubExternalInterface, y:spokeExternalInterface | some
g:greTunnel |
      (g.localPhysical=x && g.remotePhysical=y) or
      (g.localPhysical=y && g.remotePhysical=x)}
```

Display 8: Requirement 12 in Alloy.

```
{all g:greTunnel |
      some p:ipsecTunnel | p.protocolToSecure=gre &&
      ((p.local = g.localPhysical && p.remote = g.remotePhysical) or
       (p.local = g.localPhysical && p.remote = g.remotePhysical))}
```

Display 9: Requirement 14 in Alloy.

(PhysicalSpec ∧ GRERequirements ∧ SecureGRERequirements). Alloy synthesizes the network in Figure 4b. Alloy automatically places the IPSec tunnel between the correct physical interfaces to protect the GRE tunnel.

In order to add an access server to this network extend the previous scope with an access server, one internal interface, and one external interface and request Alloy to satisfy (PhysicalSpec ∧ GRERequirements ∧ SecureGRERequirements ∧ AccessServerRequirements). Alloy synthesizes the network in Figure 4c. Note that the access server is placed in parallel with only the spoke router, not with any other router, and has the correct routing protocols enabled on its interfaces.

Component Addition

When new components are added to an infrastructure, the logic that governs infrastructure has to be instantiated to the extended set of components. This is a nontrivial problem for humans to cope with. With Alloy, this instantiation is accomplished simply by finding a model of requirements for the existing scope extended by new components. (In order to avoid reconfiguring existing components, one can strengthen the requirement with existing configurations, each modeled as an equality P=V, P a configuration parameter and V its value). For example, in order to add a new spoke site to the previous network, extend its scope with a spoke

```
routing : =
    {externalInterface_0 -> ospfDomain_0,
     externalInterface_1 -> ospfDomain_0,
     hubExternalInterface_0 -> ospfDomain_0,
     internalInterface_0 -> ripDomain_0,
     spokeExternalInterface_0 -> ospfDomain_0}
chassis : =
    {externalInterface_0 -> wanRouter_0,
     externalInterface_1 -> wanRouter_0,
     hubExternalInterface_0 -> hubRouter_0,
     internalInterface_0 -> spokeRouter_0,
     spokeExternalInterface_0 -> spokeRouter_0}
network : =
    {externalInterface_0 -> externalSubnet_1,
     externalInterface_1 -> externalSubnet_0,
     hubExternalInterface_0 -> externalSubnet_0,
     internalInterface_0 -> internalSubnet_0,
     spokeExternalInterface_0 -> externalSubnet_1}
```

Display 10: Configuration parameters for Figure 3.

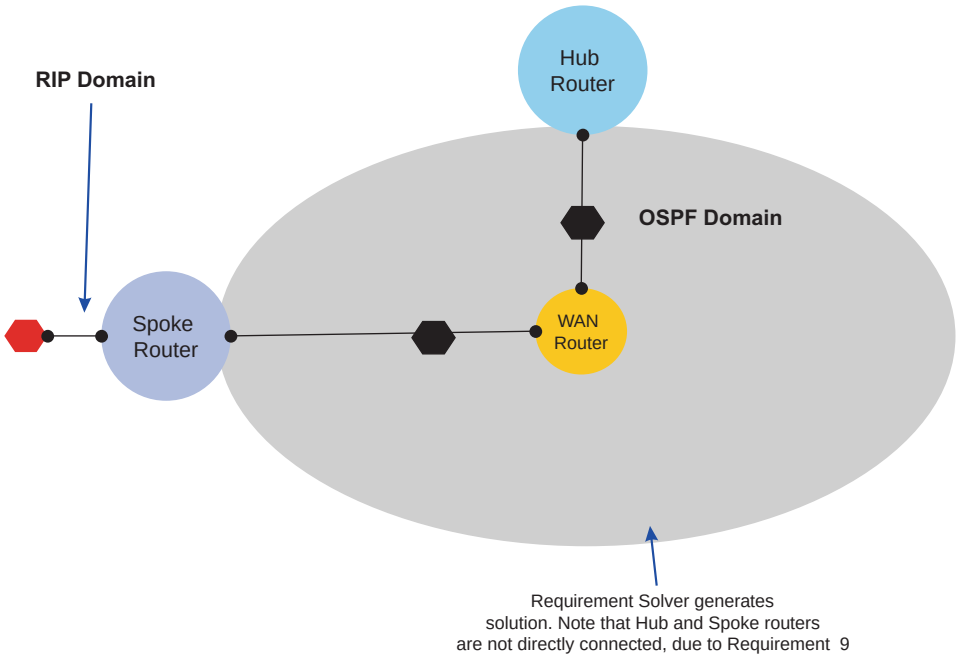

Figure 3: Configuration synthesis: Physical network.

```
(PhysicalSpec = RouterInterfaceRequirements ∧
                SubnettingRequirements ∧ RoutingRequirements)
```

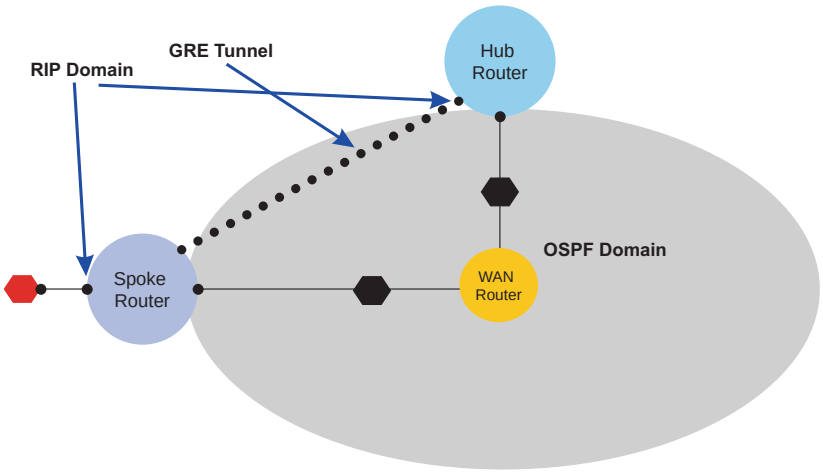

Figure 4a: Requirement strengthening: Adding overlay.
`(PhysicalSpec ∧ GRERequirements).`

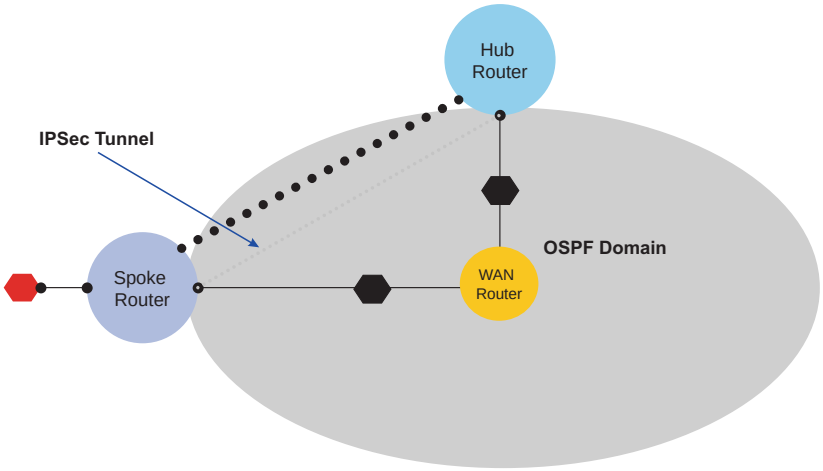

Figure 4b: Requirement strengthening: Securing overlay.
`(PhysicalSpec ∧ GRERequirements ∧ SecureGRERequirements)`

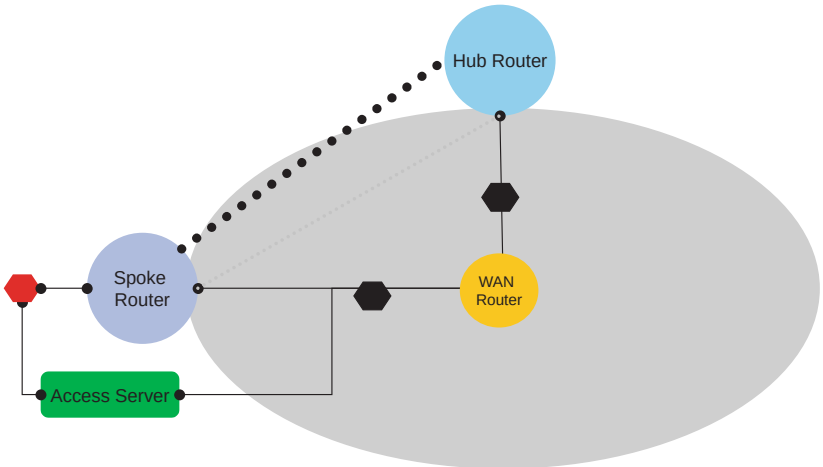

Figure 4c: Requirement strengthening: Adding remote access server.
`(PhysicalSpec ∧ GRERequirements ∧ SecureGRERequirements ∧ AccessServerRequirements)`

router, one internal subnet, one external subnet, one GRE tunnel and one IPSec tunnel. Requesting Alloy to synthesize a network satisfying (PhysicalSpec ∧ GRE-Requirements ∧ SecureGRERequirements ∧ AccessServer Requirements) in the new scope yields the network in Figure 5a.

Note that the new spoke router is physically connected just to the WAN router as required by Requirement 8. Moreover, GRE and IPSec tunnels are automatically set up between the new spoke router and hub router and physical interfaces and GRE tunnels are placed in the correct routing domains.

In order to add a new hub site to this network, extend its scope with a hub router, one external interface, one external subnet, two GRE tunnels and two IPSec tunnels. Requesting Alloy to synthesize a network satisfying (PhysicalSpec ∧ GRERequirements ∧ SecureGRE Requirements ∧ AccessServerRequirements) in the new scope yields the network in Figure 5b.

Finally, in order to permit IKE and ESP (protocols of IPSec) packets through the physical interfaces of hub and spoke routers, one can extend the above scope with eight firewall policies, then request Alloy to satisfy FullVPNSpec = (PhysicalSpec ∧ GRERequirements ∧ SecureGRERequirements ∧ AccessServerRequirements ∧ FirewallPolicyRequirements). Alloy then synthesizes the network of Figure 2 without the hosts. The reason for 8 firewall policies is that one policy is required for each IPSec tunnel endpoint.

Requirement Verification

Identifying Incorrect Firewall Policies

When we deployed the above network, we were careful to allow IKE and ESP packets to be permitted by access control lists at physical interfaces of hub and spoke routers. This was the reason for FirewallPolicyRequirements. However, we discovered that end-to-end connectivity was still not established. After considerable testing and analysis we realized that the WAN router itself was blocking IKE and ESP packets. We had not anticipated this cause. We now show how to formalize identification of this cause.

Alloy was not used for such identification, but this example illustrates how it could have been. Define a

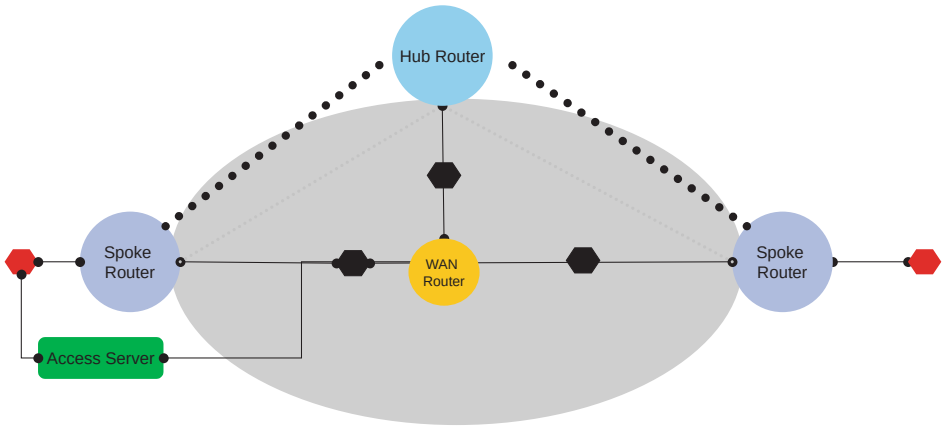

Figure 5a: Component addition: Adding new spoke router.
(PhysicalSpec ∧ GRERequirements ∧ SecureGRERequirements ∧ AccessServerRequirements)

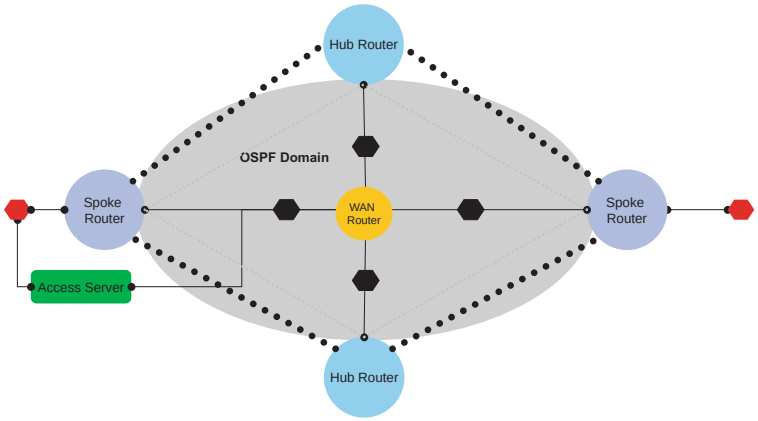

Figure 5b: Component addition: Adding new hub router.
(PhysicalSpec ∧ GRERequirements ∧ SecureGRERequirements ∧ AccessServerRequirements)

condition called BlockedIPSec capturing conditions under which an IPSec packet can be blocked, and find out how it is *possible* that (FullVPNSpec ∧ BlockedIPSec) be true. In other words, is it possible that the network be configured in a manner consistent with FullVPNSpec yet block IPSec packets? If so, we would have to modify requirements to preclude this possibility. The predicate in Display 11 states that IPSec is blocked if there is some esp or ike packet which is blocked. The predicate in Display 12 states that a packet is blocked if there is some firewall policy protecting an external interface that denies the protocol for that packet.

If we increase the scope of the last network to include 9 (more than 8) firewall policies, and request Alloy to find a model for (FullVPNSpec ∧ BlockedIPSec), Alloy produces the values in Display 13 for prot, permission and protectedInterface attributes of firewall policies. In other words, firewallPolicy_8, applied on externalInterface_0 on the WAN router, blocks ike_0.

Identifying Private Subnet Advertisement Of Private Subnets Into WAN

This section illustrates another problem that arose during deployment of our VPN solution into an existing network. Existing networks contain "legacy" routers that have no concept of internal or external subnets as do spoke routers. Thus, if our VPN is grafted into an existing network as shown in the figure above, the defense-in-depth feature mentioned in the second section is compromised. The legacy router can run the same routing protocol on both its internal and external interfaces and thereby export the internal subnet Y to the WAN. Now, if the WAN router is compromised, an adversary can send packets to the host at Y. We now show how to formalize identification of this possibility.

We define the code shown in Display 14. This predicate states that an internal subnet is advertised to the WAN if there is a legacy router with two interfaces, one attached to an internal subnet and another to an external subnet, and both have the same routing protocol enabled on them. Now, if we increase the scope of the network of Figure 5b to include one additional (legacy) router and two additional physical interfaces, and request Alloy to find a model for (FullVPNSpec ∧ internalSubnetAdvertisedToWan), Alloy produces the the code in Display 15. In other words, physicalInterface_0 and physicalInterface_1 can be placed on legacyRouter_0, one can be connected to an internal subnet, the other to an external subnet, and yet both can belong to ospfDomain_0.

Writing Efficient Requirements

Scope Splitting

One critical parameter to control in Alloy is the size of the scope. If it gets too large it should be split up and the specification changed, if necessary. Consider the following specification declaring router and interface types, and a relation chassis mapping an interface to its router. Also define EmptyCond to be an empty set of constraints to satisfy (Alloy requires *some* constraint before it can be run):

```
pred BlockedIPSec () {
some p:ipPacket, s,t:externalInterface |
            p.source = s && p.destination = t && (p.prot =
            ike or p.prot=esp) && Blocked(p)}
```

Display 11: Blocking IPSec if esp or ike packet is blocked.

```
pred Blocked(pack:ipPacket) {
            some p:firewallPolicy, x:externalInterface |
            p.protectedInterface = x &&
            p.prot=pack.prot &&
            p.action = deny
            }
```

Display 12: Block a packet if firewall policy denies its protocol.

```
prot : =                                  firewallPolicy_4 -> permit_0,
    {firewallPolicy_0 -> ike_0,           firewallPolicy_5 -> permit_0,
     firewallPolicy_1 -> ike_0,           firewallPolicy_6 -> permit_0,
     firewallPolicy_2 -> ike_0,           firewallPolicy_7 -> permit_0,
     firewallPolicy_3 -> ike_0,           firewallPolicy_8 -> deny_0}
     firewallPolicy_4 -> esp_0,     protectedInterface :  =
     firewallPolicy_5 -> esp_0,         {firewallPolicy_0 -> spokeExternalInterface_1,
     firewallPolicy_6 -> esp_0,           firewallPolicy_1 -> spokeExternalInterface_0,
     firewallPolicy_7 -> esp_0,           firewallPolicy_2 -> hubExternalInterface_1,
     firewallPolicy_8 -> ike_0}           firewallPolicy_3 -> hubExternalInterface_0,
permission: =                             firewallPolicy_4 -> spokeExternalInterface_1,
    {firewallPolicy_0 -> permit_0,        firewallPolicy_5 -> spokeExternalInterface_0,
     firewallPolicy_1 -> permit_0,        firewallPolicy_6 -> hubExternalInterface_1,
     firewallPolicy_2 -> permit_0,        firewallPolicy_7 -> hubExternalInterface_0,
     firewallPolicy_3 -> permit_0,        firewallPolicy_8 -> externalInterface_0}
```

Display 13: Model for FullVPNSpec ∧ BlockedIPSec.

```
sig router {}
sig interface {chassis: router}
pred EmptyCond () {}
```

When Alloy tries to find a model for EmptyCond in a scope consisting of 50 routers and 50 interfaces it crashes! This is because the cross product of the set of all routers and chassis' has 50×50=2500 pairs. Each subset of this product is a value of the chassis relation. Since there are 2^{2500} subsets, there are that many possible values to enumerate. We can now try splitting the scope and redefining the specification; see Diplay 16. Alloy returns a model of EmptyCond for the scope consisting of 25 hubRouters, 25 spokeRouters, 25 hubRouterInterfaces and 25 spokeRouterInterfaces in seconds! Note that the scope still contains 50 routers and 50 interfaces. But there are now "only" $2^{625} \times 2^{625} = 2^{1250}$ possible values of chassis relation, or a factor of 2^{1250} less. The scope splitting heuristic has been followed to structure the space of different routers and interfaces in the fault-tolerant VPN.

Minimizing Number Of Quantifiers In Formulas

Requirements containing quantifiers are transformed into Boolean form by instantiating quantified variables in all possible ways. The number of instantiations is the product of the number of instantiations of each quantified variable. The number of instantiations of each quantified variable is the size of the scope of that variable. In order to prevent the Boolean form from becoming excessively large, one can keep the number of quantified variables in a requirement as small as possible. For example, consider the definition of FirewallPolicyRequirements show in Display 17 in which only two explicit quantifiers appear per requirement. Display 18 shows a more compact definition in which five quantifiers appear in a single requirement.

```
pred internalSubnetAdvertisedToWan ()
      {some r:legacyRouter, x:physicalInterface, y:physicalInterface,
            s:internalSubnet, e:externalSubnet |
            x.chassis=r &&
            y.chassis=r &&
            x.network=s &&
            y.network=e &&
            x.routing=y.routing}
```

Display 14: Advertising internal subnet if several conditions met.

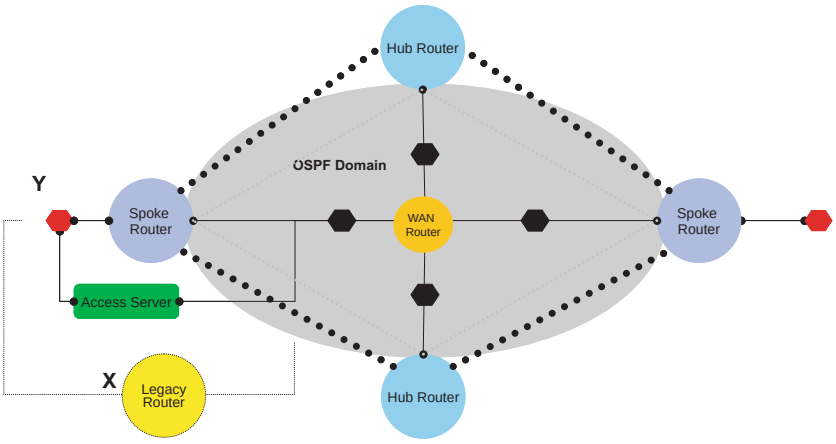

Figure 6: Advertisement of internal subnet Y into WAN by legacy router.

```
routing :  =
    {physicalInterface_0 -> ospfDomain_0,
     physicalInterface_1 -> ospfDomain_0,
     ..}
chassis :  =
    {physicalInterface_0 -> legacyRouter_0,
     physicalInterface_1 -> legacyRouter_0,
     ..}
network :  =
    {
     physicalInterface_0 -> externalSubnet_3,
     physicalInterface_1 -> internalSubnet_1,
     ..}
```

Display 15: Code for expanded specifications (FullVPNSpec ∧ internalSubnetAdvertisedToWan).

In both cases, the number of IPSec tunnels in the scope is 4 and the number of firewall policies 9. However, there is a large difference in the size of the Boolean formula produced (for the entire VPN specification). In the first case, the formula contains 216,026 clauses and 73,4262 literals, and the entire process (compilation to solution) took 2 minutes and 59 seconds. In the second case, the formula contains 601,721 clauses and 2,035,140 literals and the entire process took 8 minutes and 19 seconds.

Relationship To Previous Work

IETF's policy-based networking group [7] has similar objectives to ours. Its main contributions are vendor-neutral information models and *if-condition-then-action* rules called policies. Information models define types of objects, their attributes and possible values. These models, while important from a software development standpoint, are orthogonal to solving fundamental configuration management problems identified in this paper. These problems would remain even if we were to use all components from a single vendor.

Policy-based networking also does not enable any *declarative* representation of system logic such as Requirements 1-16. The *if-condition-then-action* rules are only procedural encodings of this logic. In effect, these rules have to do all the work of the Requirement Solver. This is a formidable undertaking. Furthermore,

```
sig hubRouter {}
sig spokeRouter {}
sig hubRouterInterface {chassis:hubRouter}
sig spokeRouterInterface {chassis:spokeRouter}
```

Display 16: Splitting the scope and redefining the specification.

```
pred FirewallPolicyRequirements ()
{(all t:ipsecTunnel | some p1:firewallPolicy |
      p1.protectedInterface = t.local &&
      p1.prot = ike &&
      p1.action = permit) &&
(all t:ipsecTunnel | some p1:firewallPolicy |
      p1.protectedInterface =t.remote &&
      p1.prot = ike &&
      p1.action = permit) &&
(all t:ipsecTunnel | some p1:firewallPolicy |
      p1.protectedInterface = t.local &&
      p1.prot = esp &&
      p1.action = permit) &&
(all t:ipsecTunnel | some p1:firewallPolicy |
      p1.protectedInterface = t.remote &&
      p1.prot = esp &&
      p1.action = permit)
(no disj p1,p2:firewallPolicy |
p1.protectedInterface=p2.protectedInterface &&
      p1.prot=p2.prot && !p1.action=p2.action)}
```

Display 17: Two explicit quantifiers per firewall requirement.

```
pred FirewallPolicyRequirements ()
{(all t:ipsecTunnel | some p1,p2,p3,p4:firewallPolicy |
      p1.protectedInterface = t.local &&
      p1.prot = ike &&
      p1.action = permit &&
      p2.protectedInterface = t.remote &&
      p2.prot = ike &&
      p2.action = permit &&
      p3.protectedInterface = t.local &&
      p3.prot = esp &&
      p3.action = permit &&
      p4.protectedInterface = t.remote &&
      p4.prot = esp &&
      p4.action = permit)&&
(no disj p1,p2:firewallPolicy |
p1.protectedInterface=p2.protectedInterface &&
      p1.prot=p2.prot && !p1.action=p2.action)}
```

Display 18: More compact version of Display 17.

verification with such procedural rules is impractical, and operations like configuration error diagnosis or fixing are not addressed. In our approach, the Requirement Solver remains unchanged. It is only the requirements or the scope that change. The Requirement Solver automatically adjusts to these changes and finds new configurations. Verification is another application of the Requirement Solver.

For the same reason, it is incorrect to assume that just the use of high-level languages like Perl and Python, often used in network management, can solve above fundamental configuration management problems. The hard part of reasoning from full first-order logic requirements still has to be programmed in these languages. It is this part that our approach automates.

Previous papers [8, 9] formalized a restricted version of the second section's requirements, in Prolog. Prolog is based in definite clauses, hence it is not possible to use it to reason with full first-order logic constraints. Examples of these are *"for every GRE tunnel there is an IPSec tunnel between associated physical interfaces that secures all GRE traffic"* and *"no two distinct interfaces on a router are on the same subnet."* The application of Prolog to system administration is thoroughly explored by Couch and Gilfix [10]. Related, widely used systems are Burgess' CFEngine [11] and Anderson's LCFG [12], but both have less expressive power than Prolog. These systems also perform robust application of configuration to components, a problem outside the scope of this paper. Recently, the need for specifying and reasoning with constraints on configurations has been amply expressed [13]. These constraints require the expressive power of full first-order logic, therefore our approach can address this need.

Summary, Conclusions & Future Directions

This paper introduces the notion of a Requirement Solver and shows how fundamental configuration management tasks can be naturally formalized using it. These tasks are configuration synthesis, requirement strengthening, component addition, configuration error diagnosis and configuration error fixing. The Solver has been inspired by, and is implemented in, the new logical system called Alloy. Alloy is based on the concept of model finding for full first-order logic in *finite* domains. Because of Alloy's use of highly efficient satisfiability solvers, there is renewed optimism for efficient reasoning in this logic, especially for configuration management. Traditional first-order logic theorem provers address the harder problem of reasoning in *infinite* domains. The Solver is illustrated in the context of a realistic fault-tolerant VPN with remote access, by working out four of above tasks. Approaches for writing efficient specifications are outlined.

Alloy's strength is efficiently sorting through complex, first-order logic constraints, provided scopes are small. On a modern PC, it requires several hours to find complete configurations for all components in an 8-site VPN: 8 spoke routers, 2 hub routers, 1 access server, 1 WAN router and associated interfaces, subnets, firewall policies, routing domains, GRE and IPSec tunnels. For context, the time taken by a human to reconcile Requirements 1-16 for all sites should be considered. Based on experiments of this paper, it is possible that Alloy be used from a traditional programming language to solve configuration management problems for networks of realistic scale and complexity. The heuristics of the ''Efficient Requirements'' section could be programmed in the programming language. Other approaches for scalability are tuning satisfiability solvers to the networking domain, improving Alloy compilers, and using divide-and-conquer approaches.

One open problem is selecting the least cost change from the current configuration as required for Configuration Error Fixing. A configuration management problem, not discussed in this paper, is migration planning: in what order to configure components so that mission-critical invariants are never violated. For example, suppose the routing protocol on all routers has to be changed from RIP to OSPF. If the only method of accessing routers to perform this change is inband, then reconfiguring the first router to which the management station is attached will effectively isolate it from all others. This is because routing protocols compute routes to routers, but since OSPF and RIP processes do not exchange information with each other, the first router will not be able to compute routes to others. The problem of the order in which to reconfigure components is fundamentally the problem of planning in artificial intelligence. The application of satisfiability solvers to this problem has been shown by Selman and Kautz [14].

Acknowledgements

I am grateful to Dr. Paul Anderson at Edinburgh, Professor Mark Burgess at University of Oslo, Professor Carla Gomes at Cornell, Professor Daniel Jackson at MIT, Dr. Gary Levin at Telcordia, Professor Sharad Malik at Princeton, and Professor Darko Marinov at University of Illinois, Urbana Champaign for very useful ideas and feedback.

Author Information

Sanjai Narain is a Senior Research Scientist in the Information Assurance and Security Department in Telcordia's Applied Research Area. His current research is on automated synthesis of secure, fault-tolerant distributed systems. This research is funded through DARPA, DISA and Department of Homeland Security. He has built security and network management systems for wireless, IP, VoIP, DSL, Dialup, ATM and SONET. The DSL loop qualification system he created was the basis for a successful Telcordia service. He won

a DARPA award for transferring technology to the Army's Future Combat Systems program. The DR. DIALUP system he created for reducing help-desk costs of Internet Service Providers was Telcordia's first product for the mass-market. Prior to joining Telcordia, he worked at RAND Corporation where he designed and implemented new discrete-event simulation techniques. His formal training is in programming languages and automated reasoning. He obtained a Ph.D. in Computer Science from University of California, Los Angeles, an M.S. in Computer Science from Syracuse University, and a B.Tech. in Electrical Engineering from Indian Institute of Technology, New Delhi. His email address is narain@research.telcordia.com .

References

[1] Oppenheimer, David, Archana Ganapathi, David A. Patterson, ''Why Internet Services Fail and What Can Be Done About These?'' *Proceedings of 4th Usenix Symposium on Internet Technologies and Systems (USITS '03)*, http://roc.cs.berkeley.edu/papers/usits03.pdf , 2003.

[2] Lampson, Butler, ''Computer Security In the Real World,'' *Proceedings of Annual Computer Security Applications Conference*, http://research.microsoft.com/lampson/64-SecurityInRealWorld/Acrobat.pdf , 2000.

[3] Horn, Paul, Senior VP, IBM Research, *Autonomic Computing IBM's Perspective on the State of Information Technology*, http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf .

[4] http://alloy.mit.edu/ .

[5] *Berkmin*, http://eigold.tripod.com/BerkMin.html .

[6] *Zchaff*, http://www.princeton.edu/~chaff/ .

[7] Moore, B., E. Ellesson, J. Strassner, A. Westerinen, ''Policy Core Information Model – Version 1 Specification,'' *IETF RFC 3060,* http://www.ietf.org/rfc/rfc3060.txt , February, 2001.

[8] Narain, S., T. Cheng, B. Coan, V. Kaul, K. Parmeswaran, W. Stephens, ''Building Autonomic Systems Via Configuration,'' *Proceedings of AMS Autonomic Computing Workshop*, Seattle, WA, http://www.argreenhouse.com/papers/narain/Autonomic.pdf , 2003.

[9] Qie, X., S. Narain, ''Using Service Grammar to Diagnose Configuration Errors in BGP-4,'' *Proceedings of Usenix Systems Administrators Conference,* San Diego, CA, 2003. Also to appear in *Science of Computer Programming Journal,* in a special issue on Network Management.

[10] Couch, A., M. Gilfix, ''It's Elementary, Dear Watson: Applying Logic Programming To Convergent System Management Processes,'' *Proceedings of Large Installations Systems Administration Conference (LISA)*, http://www.eecs.tufts.edu/~mgilfix/publications/prolog-lisa99.pdf , 1999.

[11] Burgess, M., ''A Site Configuration Engine,'' *USENIX Computing systems*, Vol. 8, Num. 3, http://www.iu.hio.no/~mark/papers/paper1.pdf , 1995.

[12] Anderson, P., A. Scobie, ''LCFG – The Next Generation,'' *Proceedings of UKUUG LISA/Winter Conference*, http://www.lcfg.org/doc/ukuug2002.pdf , 2002.

[13] Configuration Workshop, Large Installations Systems Administration Conference (LISA), http://homepages.informatics.ed.ac.uk/group/lssconf/config2004/index.html , 2004.

[14] Selman, B., H. Kautz, ''Planning As Satisfiability,'' *Proceedings of ECAI-92*, http://www.cs.cornell.edu/selman/papers/pdf/92.ecai.satplan.pdf .