

Building Autonomic Systems Via Configuration^{1 2}

Sanjai Narain, Thanh Cheng, Brian Coan, Vikram Kaul, Kirthika Parmeswaran, William Stephens
Telcordia Technologies, Inc., 445 South Street, Morristown, NJ 07960
narain@research.telcordia.com

Abstract

Large classes of autonomic (self-managing, self-healing) systems can be created by logically integrating simpler autonomic systems. The configuration method is widely used for such integration. However, there are few formalized tools in support of this method for specification, compilation, diagnosis, reasoning, and distributed provisioning. As a result, the practice of this method is very costly and can lead to security failures. This paper presents a technique called Service Grammar for building these tools based on a novel analysis of protocols and distributed algorithms in a domain of interest. The technique is illustrated in the context of a realistic adaptive virtual private network. We show how lower-layer adaptive protocols can be composed to create adaptive behavior at a higher layer, while preserving end-to-end security requirements.

1. Introduction

Commercial off the shelf components are capable of executing powerful protocols and distributed algorithms. These components are logically integrated to create systems with end-to-end functionality. A widely used technique for logical integration is configuration: components are designed to have a finite set of configuration parameters that can be set to definite values. These parameters are static in that, unlike the component's dynamic state, these do not change during the component's normal operation. By setting these to definite values, the component is customized to behave in a definite way. By appropriately configuring all system components, required end-to-end system functionality is achieved. Thus, configuration serves as a "machine language" for system synthesis. This method is used to create very large and complex systems. For example, a financial service company's wide area

network can contain about eighteen thousand routers yet end-to-end requirements on resilience, connectivity, security and performance are achieved via router configuration.

Interestingly enough, this method is also used to create adaptive systems. Components executing simple adaptive algorithms are logically integrated to create systems exhibiting complex adaptive behavior. For example, in networking there are mature adaptive protocols at the physical, link, network, middleware and application layers, and an end-to-end network derives its adaptive properties from those of all of these protocols. Since adaptation is a central to autonomic systems, the configuration method should play a central role in synthesis and analysis of autonomic systems. Many self-managing, self-healing algorithms, each for a definite purpose, will need to be integrated to create end-to-end autonomic systems.

In spite of the ubiquity of the configuration method, there are few principles to support it. It is not possible to formally specify end-to-end system functionality requirements, reason about these, refine these and translate these into component configurations. If for some reason the end-to-end functionality becomes unavailable there are no systematic tools to automatically diagnose configuration errors. System designers today manually perform these tasks. If system functionality spans multiple administrative domains, such as extranets linking multiple corporations, and there is no centralized management authority, system creation and management procedures are even more manual.

The above problems are inherently difficult. There is a large conceptual gap between global, end-to-end functionality requirements (system specification) and component configurations (machine language). Realistic systems can have a large number of components each

¹ © 2002 Telcordia Technologies, Inc. This material is based on work supported in DARPA IA&S Dynamic Coalitions program under contract number F-30602-00-C-0065. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

² Proceedings of Autonomic Computing Workshop, June 25, Seattle, WA

with many configuration parameters and possible values. Even visualizing dependencies between requirements and component configurations, at and across multiple layers of abstraction is a formidable task. Configuration error diagnosis is difficult because it requires global reasoning. In the presence of configuration errors, components continue to function correctly in isolation, but they don't work together. Diagnosing why components don't work together involves global reasoning about the logical structure of the system, which is inherently difficult. Note that these *design* problems cannot be solved just by standardizing component interfaces. Even when these are standardized, it is hardly obvious how to integrate components to create global, end-to-end functionality.

In the absence of formalized tools to support the configuration method, its practice is highly inefficient and costly³ and can create security breaches⁴. Similar conclusions are reported for the military world⁵.

This paper outlines an approach called Service Grammar for developing formalized tools to solve the above problems, and illustrates these by means of a realistic, adaptive virtual private network. These tools can be created for a particular domain by a new kind of analysis of all the protocols and distributed algorithms that arise in that domain. Thereby, synthesis and analysis of

³ Consider this: at current rates of expansion, there will not be enough skilled I/T people to keep the world's computing systems running. Some estimates for the number of I/T workers required globally to support a billion people ... put it at over 200 million, or close to the population of the entire United States.....Even if we could somehow come up with enough skilled people, the complexity 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. –Paul Horn, 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 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. – Butler Lampson, Computer Security In the Real World. *Proceedings of Annual Computer Security Applications Conference*, 2000. <http://research.microsoft.com/lampson/64-SecurityInRealWorld/Acrobat.pdf>

⁵ ...20% of a rapid insertion force is dedicated to establishing, operating, and maintaining networks. – D. Hitchcock, G. Strawn. Workshop on New Visions for Large-Scale Networks: Research and Applications. Large Scale Networking Coordinating Group Of the Interagency Working Group for Information Technology, Research and Development, March 12-14, 2001, Vienna, VA. <http://www.hpcc.gov/iwg/pca/ltn/ltn-workshop-12mar01/workshop-12mar01.pdf>

systems via the configuration method can be greatly accelerated. Service Grammar tools consist of the following:

1. A Requirements Language for *naturally* specifying end-to-end system functionality requirements. This consists of high-level constraints on component configurations that are then composed to set up end-to-end system functionality requirements. The composition is expressed by means of grammar rules. The intuition is to regard a system not as a set of components but as a set of services that, in general, span multiple components.
2. A Provisioning Engine for compiling system functionality requirements into detailed system component configurations.
3. A Diagnosis Engine for checking if system components are correctly configured to support a system functionality requirement.
4. A Distributed Provisioning Engine for enabling end-to-end functionality across multiple domains with no centralized management authority
5. An Assurance Argument Framework for arguing the correctness of system functionality requirements themselves, by examining only system configuration and other static information.

A central contribution of Service Grammar is a heuristic for identifying *natural* constraints that make up the Requirements Language. One identifies the notion of “correct configuration” associated with a distributed algorithm, i.e., the joint goal that a group of agents executing that algorithm are intended to accomplish. Whether or not they do so depends critically on how they are configured. For example, a group of agents may execute the same routing protocol such as OSPF but unless they are configured correctly, they will not achieve the joint goal of discovering network topology and computing shortest paths. Similarly, two agents may execute the same secure tunneling protocol such as IPSec but if they are not configured correctly, they will not achieve the joint goal of setting up a tunnel that achieves confidentiality, authentication and data integrity. The notion of correct configuration is already implicit in the definition of protocols and distributed algorithms in a domain since their intended use is a part of the definition. We make this “configuration logic” explicit by analyzing these definitions.

A critical observation is that correct configuration for a distributed algorithm or protocol can be defined largely independently of others. For example, it is possible to

articulate that *as far as IPSec is concerned*, what does it mean for two tunnel endpoints to be correctly configured. It is not necessary to include in this articulation, the consideration that IP packets have to be routed from one end point to another. Thus, one avoids the arbitrarily deep descent down the protocol stack in order to argue that a particular layer has been correctly configured. This observation is based upon how network administrators troubleshoot networks.

A second observation can now be made: a necessary condition for correct configuration of a system from an end-to-end standpoint is that it be correctly configured w.r.t. each of the protocols and distributed algorithms that occur in it. This is because end-to-end system functionality arises from the functionality of constituent protocols and distributed algorithms in the system, even if there are interactions between these. The grammar rules capture this observation. Thus, if the Requirements Language for all protocols and distributed algorithms in a domain can be created, then constraints from this can be composed to create end-to-end requirements for a very large class of systems. The effort to create this Language is smaller than might be imagined. For example, practically all virtual private networking requirements can be composed from those of about twenty five different protocols.

The Provisioning Engine defines how to compile a Requirements Language or composite constraint into component configurations. The Diagnosis Engine checks whether a Requirements Language or composite constraint is implied by a given system configuration. The Diagnosis Engine enables a new type of configuration error diagnosis. One can check if system components have been configured according to a specification (i.e., as intended) rather than just checking that some system invariant rules have been observed. It checks if any errors have been made due to manual compilation of end-to-end requirements.

For distributed provisioning of services across multiple domains, one sets up a controller for each domain that has the authority to configure components in that domain. A controller defines proposals for multi-domain services using the Requirements Language, and circulates it to other controllers. An agreement protocol is used by the controllers to converge on a proposal. When they do, they use the Provisioning Engine to compile proposals into component configurations, then apply the configurations that are for components in their domain. Since, each controller executes the same Provisioning Engine on the same proposal, global configuration consistency across multiple domains is enforced.

Finally, the Assurance Argument framework is used to check if specifications and grammar rules themselves are correct. The representation of a system as a collection of services simplifies the design of strategies for proving properties of the system. The novelty is that many interesting classes of system properties can be proved by examining only the system configuration and other static information about the system. It is not necessary to create models of dynamic behavior of underlying protocols and distributed algorithms. By correctly configuring a collection of components w.r.t. a protocol, we know that that collection will achieve its joint goal. Thus, proved or well-tested properties of protocols and distributed algorithms can be treated as axioms in these proofs. It is not necessary to prove these again. The individual technologies are well understood. The question is if these are put together in a definite way (as defined by the logical architecture, in turn defined by system configuration), is the intended end-to-end goal accomplished?

An interesting strategy for checking if there are security breaches is to pose the question of whether an undesirable end-to-end service is available to an adversary. One can specify this service as an end-to-end requirement then use the Diagnosis Engine to check if the requirement is true given current system configuration. If the answer is in the affirmative a security breach is detected. This idea is a part of the Smart Firewalls work⁶.

2. Application to Resilient Virtual Private Network

Service Grammar tools have been built, tested and deployed by us for the virtual private networking domain and are illustrated using a realistic example⁷. Prior illustrations also exist^{8 9 10}. We show how a protocol

⁶ Rajagopalan, S., Smart Firewalls: Automatic Management of Network Security Policy in Dynamic Networks. *Proceedings of DISCEX-II Conference, 2001.*

⁷ Cisco Systems. IPSec Virtual Private Network Resilience Solutions. http://www.cisco.com/warp/public/cc/so/neso/vpn/vpne/vpne_an.htm

⁸ Barton, M., Atkins, D., Narain, S., Ritcherson, D., Tepe, K. Integration of IP Mobility and Security For Secure Wireless Communications. *Proceedings of IEEE International Communications Conference*, New York, NY, 2002.

⁹ Narain, S., Shareef, A., Rangadurai, M. Diagnosing Configuration Errors in Virtual Private Networks. *Proceedings of IEEE International Communications Conference*, Helsinki, Finland, 2001.

¹⁰ Narain, S., Vaidyanathan, R., Moyer, S., Stephens, W., Parmeswaran, K., Shareef, A. Middleware for Building Adaptive Systems Via Configuration. *Proceedings of ACM SIGPLAN Workshop on Optimizing Middleware*, Salt Lake City, UT, June 2001.

intended for tolerating faults at a lower layer can be composed with other protocols to provide fault-tolerance at a higher layer, and furthermore, in such a way that traffic is never routed over an insecure path. Another fault-tolerance protocol provides resilience against a class of server failures. See Figure 1. The problem is to create a virtual private overlay network between four geographically distributed sites, each with Internet access through a local ISP, satisfying the following requirements:

1. It should be private in that all traffic between sites should be encrypted, authenticated and checked for integrity.
2. It should be resilient in that if an intersite link fails, traffic should be redirected along another link that is also private, never over an insecure link.
3. A web server should multicast its content to all sites.
4. The web service should also be resilient in the sense that if a server crashes, another server continues to deliver the service

CR3-CR2, CR2-CR4 or via the tunnels CR3-CR1, CR1-CR4. Suppose the first route is selected. Then, if CR2 fails, traffic will *not* be redirected via the other two tunnels. A routing protocol such as OSPF is required to perform this kind of redirection. However, running OSPF over the gateway routers is not sufficient since OSPF will not “discover” the overlay network. OSPF will not see IPsec tunnel end points as directly connected, since these end points are not, in general, on the same IP subnet (there can be many routers in between). To solve this problem, a new kind of tunneling protocol called GRE is needed. New logical (GRE) interfaces are defined on gateway routers and GRE tunnels are set up between these. GRE provides OSPF with the illusion that the tunnel end points are directly connected. Now, OSPF discovers the overlay network and provides resilience as required. Requirement 3 can be satisfied by enabling a multicast protocol such as PIM over the GRE interfaces and the local area network interfaces. Requirement 4 is satisfied by using an IP failover protocol such as Wackamole¹¹ which runs over the Spread¹² group communication system.

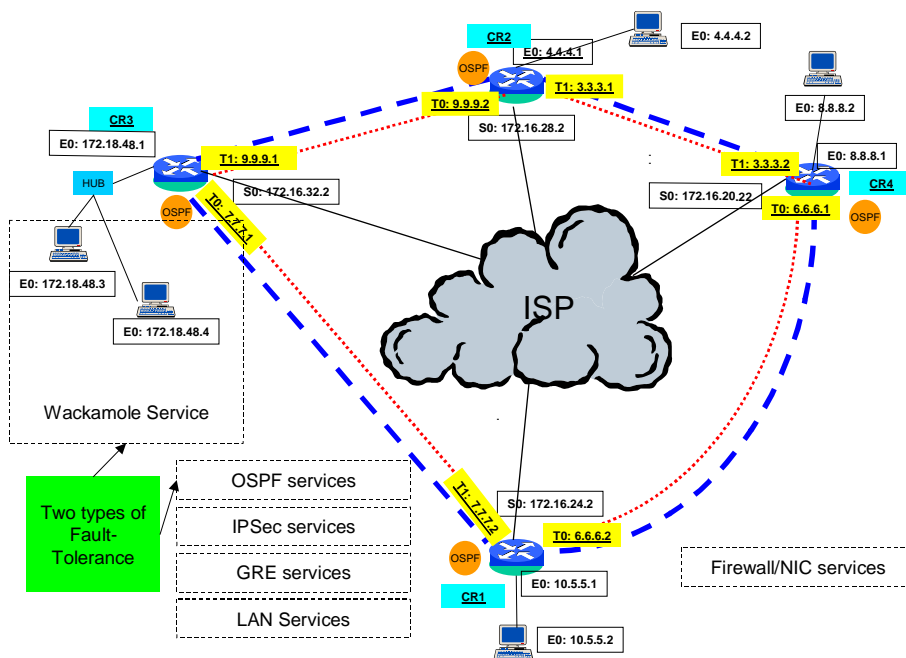


Figure 1. A Resilient VPN Overlay With Fault-Tolerant Web Server

Requirement 1 can be satisfied by setting up IPsec tunnels between gateway routers. However, satisfying Requirement 2 is harder. IPsec sets up a form of static routing. For example, when a packet arrives at CR3 destined for CR4, it is redirected either via the tunnels

¹¹ Yair Amir, Ryan Caudy, Ashima Munjal, Theo Schlossnagle, and Ciprian Tutu. [N-Way Fail-Over Infrastructure for Survivable Servers and Routers](#). Accepted to IEEE International Conference on Dependable Systems and Networks (DSN03), San Francisco, June 2003.

¹² <http://www.spread.org>

As may be imagined, configuring routers and servers in a consistent manner to satisfy the above requirements is a complex task. Commercially available tools do not even allow formal statement of the above requirements. Instead, experienced network administrators *manually* translate the above requirements into router and server configurations. Thus, many configuration errors arise. The types of configuration parameters that have to be set are:

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. PIM: Whether it is enabled on an interface.
6. Wackamole/Spread: Ports, broadcast address, group members, physical and failover IP addresses

The types of configuration errors which 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 incorrect. This can lead to incorrect routing tables and to outright isolation of subnets. Another serious problem is routing loops. 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.
4. 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.

Another problem is that GRE tunnels may not fail independently of each other. If multiple GRE tunnels are mapped on to the same physical port then all these will simultaneously fail if that port fails.

5. PIM may be set up incorrectly, e.g., over physical interfaces connecting gateway routers to the ISP, rather than over the GRE interfaces.
6. IP failover may be set up incorrectly. The two servers may not share the same virtual IP address may belong to different Spread groups, or Spread and Wackamole ports may be mismatched.

We now sketch a Service Grammar system for the above domain. The constituent protocols are IP, OSPF, GRE, IPSec, Wackamole, Spread and PIM. The configuration parameters are as outlined above. The Requirements Language contains constraints representing abstractions such as subnets, OSPF areas, GRE and IPSec tunnels, Wackamole clusters, and PIM domains. More precisely, the constraints are:

1. *subnet(Interfaces, Type, Mask, Addresses)* representing the constraint that all interfaces in *Interfaces* have type *Type*, subnet mask *Mask* and address at the corresponding position in *Addresses*.
2. *greTunnel(G1, G2, P1A, P2A)* representing the constraint that a GRE tunnel has been configured between the GRE interfaces *G1* and *G2*, supported by physical interfaces with addresses *P1A* and *P2A* respectively.
3. *ospfSubnet(Subnet, ProcessID, AreaID, AreaType)* representing the constraint that all interfaces in *Subnet* have OSPF process *ProcessID* enabled at them and belong to area *AreaID* of type *AreaType*.
4. *pim(Interfaces, Bool)* representing the constraint that all interfaces in *Interfaces* have PIM enabled or disabled at them, depending on *Bool*.
5. *ipsec(P1, P2, Policy)* representing the constraint that an IPSec tunnel has been defined between interfaces *P1* and *P2* governed by *Policy*. *Policy* specifies the name and mode of the tunnel, the preshared key, hash and encryption algorithms and the traffic filter.
6. *spreadLan(Machines, BroadcastAddress, Port, MachineAddressPairs)* representing the constraint that a Spread infrastructure has been set up on a local area network with *BroadcastAddress*.
7. *wackGroup(Machines, Port, Group, PreferredAddresses, VirtualAddresses, NotificationList)* representing the constraint that a Wackamole group has been defined over a Spread infrastructure

These are already higher-level than that of raw component configurations. These are composed into higher-level constraints using grammar rules. Such rules are implemented in Prolog by the *eval(Spec, DB)* predicate that means that constraint *Spec* is true in the configuration database *DB*. A configuration database is simply a list of equalities of the form *C:P=V* where *C:P* is configuration parameter *P* of component *C* and *V* is its value. The complete definition of *eval* is provided in the Appendix. An example is:

```
eval(secureFTMulticastOverlay, DB) :-
  eval(addressingForLansAndGRE, DB),
  eval(ospfOverLansAndGRE, DB),
  eval(protectedGREtunnels, DB),
  eval(pimOverGREtunnels, DB),
  eval(ftServer, DB).
```

which states that the constraint *secureFTMulticastOverlay* is a conjunction of five constraints: *addressingForLansAndGRE*, *ospfOverLansAndGRE*, *protectedGREtunnels*, *pimOverGREtunnels*, and *ftServer*.

Because of the relational nature of Prolog this definition serves both as the Provisioning and Diagnosis Engines. When *eval* is called with *DB* a variable then a value of *DB* is computed in which *Spec* is true, which is provisioning. If *DB* is bound then *eval* checks if *Spec* is true in *DB*, which is diagnosis.

The Provisioning Engine compiles the top-level service specification *secureFTMulticastOverlay* into detailed router and server configurations. Thereby, a large class of configuration errors is eliminated altogether.

To set up this service distributively, this specification is circulated to all domain controllers who then commit, compile the specification, and consistently configure their equipment.

To develop an assurance argument that the grammar rules themselves are correct, one defines a system property *P* and checks that for every system configuration *DB* that *secureFTMulticastOverlay* produces, *P* holds for *DB*. Formally, one shows:

$$\forall DB. eval(secureFTMulticastOverlay, DB) \supset P(DB).$$

For example, *P* can be that there are no routing loops (true in our case), or that GRE tunnels failures are independent w.r.t. port failures (false in our case). This “generate-and-test” approach works when there is a finite number of significant values of *DB* that are computed by *eval*. This is the case for our system.

When the following query is issued to Prolog:

```
?- eval(secureFTMulticastOverlay, DB)
```

it prints out the following values of configuration parameters for all components:

```
e0cr1 : addressing = [ethernet,24,10.5.5.1]
e0cr2 : addressing = [ethernet,24,4.4.4.1]
e0cr3 : addressing = [ethernet,24,172.18.48.1]
e0cr4 : addressing = [ethernet,24,8.8.8.1]
t0cr3 : addressing = [gre,24,7.7.7.1]
t1cr1 : addressing = [gre,24,7.7.7.2]
t0cr1 : addressing = [gre,24,6.6.6.2]
t0cr4 : addressing = [gre,24,6.6.6.1]
t1cr3 : addressing = [gre,24,9.9.9.1]
t0cr2 : addressing = [gre,24,9.9.9.2]
t1cr2 : addressing = [gre,24,3.3.3.1]
t1cr4 : addressing = [gre,24,3.3.3.2]
s0cr1 : addressing = [serial,24,172.16.24.2]
s0cr2 : addressing = [serial,24,172.16.28.2]
s0cr3 : addressing = [serial,24,172.16.32.2]
s0cr4 : addressing = [serial,24,172.16.20.22]
e0cr1 : ospf = [10,0,regular]
e0cr2 : ospf = [10,0,regular]
e0cr3 : ospf = [10,0,regular]
e0cr4 : ospf = [10,0,regular]
t0cr3 : ospf = [10,0,regular]
t1cr1 : ospf = [10,0,regular]
t0cr1 : ospf = [10,0,regular]
t0cr4 : ospf = [10,0,regular]
t1cr3 : ospf = [10,0,regular]
t0cr2 : ospf = [10,0,regular]
t1cr2 : ospf = [10,0,regular]
t1cr4 : ospf = [10,0,regular]
t0cr3 : gre = [t1cr1,s0cr3,s0cr1]
t1cr1 : gre = [t0cr3,s0cr1,s0cr3]
s0cr3 : ipsec : name1 =
[s0cr1,k1,des,sha,tunnel,esp,[172.16.32.2,32,172.16.24.2,32,gre]]
s0cr1 : ipsec : name1 =
[s0cr3,k1,des,sha,tunnel,esp,[172.16.24.2,32,172.16.32.2,32,gre]]
t0cr1 : gre = [t0cr4,s0cr1,s0cr4]
t0cr4 : gre = [t0cr1,s0cr4,s0cr1]
s0cr1 : ipsec : name2 =
[s0cr4,k2,des,sha,tunnel,esp,[172.16.24.2,32,172.16.20.22,32,gre]]
s0cr4 : ipsec : name2 =
[s0cr1,k2,des,sha,tunnel,esp,[172.16.20.22,32,172.16.24.2,32,gre]]
t1cr3 : gre = [t0cr2,s0cr3,s0cr2]
t0cr2 : gre = [t1cr3,s0cr2,s0cr3]
s0cr3 : ipsec : name3 =
[s0cr2,k3,des,sha,tunnel,esp,[172.16.32.2,32,172.16.28.2,32,gre]]
s0cr2 : ipsec : name3 =
[s0cr3,k3,des,sha,tunnel,esp,[172.16.28.2,32,172.16.32.2,32,gre]]
t1cr2 : gre = [t1cr4,s0cr2,s0cr4]
t1cr4 : gre = [t1cr2,s0cr4,s0cr2]
s0cr2 : ipsec : name4 =
[s0cr4,k4,des,sha,tunnel,esp,[172.16.28.2,32,172.16.20.22,32,gre]]
s0cr4 : ipsec : name4 =
[s0cr2,k4,des,sha,tunnel,esp,[172.16.20.22,32,172.16.28.2,32,gre]]
t0cr3 : pim = true
t1cr1 : pim = true
t0cr1 : pim = true
t0cr4 : pim = true
t1cr3 : pim = true
t0cr2 : pim = true
t1cr2 : pim = true
```

```

t1cr4 : pim = true
_40044 : addressing = [_40050,_40052,172.18.48.3]
m1 : spread =
[172.18.48.255,4803,[[m1,172.18.48.3],[m2,172.18.48.4]]]
_40330 : addressing = [_40336,_40338,172.18.48.4]
m2 : spread =
[172.18.48.255,4803,[[m1,172.18.48.3],[m2,172.18.48.4]]]
m2 : wackamole = [4803 :
localhost,wack1,172.18.48.100,[172.18.48.100],[cr3]]
m1 : wackamole = [4803 :
localhost,wack1,none,[172.18.48.100],[cr3]]

```

3. Relationship With Previous Work

The goal of policy-based networking is to specify end-to-end network requirements at a high-level and have detailed component configurations automatically derived. The method of achieving this goal proposed by IETF¹³ consists of two items: a vendor-neutral component information model, and policy rules of the form *if Condition then Action*. The information model defines component configuration parameters and their possible values for different protocols.

Vendor-neutral information models enable management of diverse components from a single management system. However, they do not shed light on how to solve the difficult *design* problems of Section 1. Even if a single vendor's equipment is used all problems in the above resilient VPN example would remain.

While it is tempting to design autonomic systems as collections of *if Condition then Action* rules they should be used only as a last resort. It is hard to design, reason about, implement and test collections of rules that monitor global state and take globally consistent actions. For example, it is infeasible to write *if Condition then Action* rules that monitor failures of IPSec tunnels and servers and reroute traffic over other IPSec tunnels or to other servers. The required effort would duplicate that of developing new routing and IP failover protocols. It is far more efficient, robust and *elegant* to solve the above problem by a composition of existing protocols whose properties have been established.

Service Grammar offers a framework for making the compositional approach practical. Only when the compositional approach cannot yield the required autonomic system properties, should new distributed algorithms be designed. An example of such an algorithm is that for distributed setup of Virtual Private Networks as sketched above.

¹³ Moore, B., Ellesson, E., Strassner, J., Westerinen, A. Policy Core Information Model -- Version 1 Specification, *IETF RFC 3060*, February 2001. <http://www.ietf.org/rfc/rfc3060.txt>.

An interesting question is whether the need for configuration can be eliminated via "autoconfiguration" algorithms. The answer to this question is negative because autoconfiguration algorithms themselves have to be configured to correctly operate. For example, OSPF autoconfigures routing tables yet its own configuration is non-trivial.

In the Netsys¹⁴ system for diagnosing configuration errors there is no way to describe the administrator's *intention*, i.e., *network-specific* policies and structure. It only runs a collection of network-invariant tests.

4. Summary

Large classes of autonomic systems can be created by logically integrating simpler autonomic systems. The configuration method is widely used for such integration. However, there are few formalized tools in support of this method for specification, compilation, diagnosis, reasoning, and distributed provisioning. This paper presents an approach called Service Grammar for building these tools by a novel analysis of the protocols and distributed algorithms in a domain of interest. It is illustrated in the context of a realistic adaptive virtual private network. We showed how lower-layer adaptive protocols can be composed to create adaptive behavior at a higher layer while preserving end-to-end security properties. The relationship with previous work is outlined.

Appendix

```

/*
-----
Specification of Secure Fault Tolerant Multicast Overlay
-----
*/
eval(secureFTMulticastOverlay, DB) :-
    eval(addressingForLansAndGRE, DB),
    eval(ospfOverLansAndGRE, DB),
    eval(protectedGRETunnels, DB),
    eval(pimOverGRETunnels, DB),
    eval(ftServer, DB).

lans([S1,S2,S3,S4, G1, G2, G3, G4):-
    S1=[e0cr1],S2=[e0cr2],S3=[e0cr3],
    S4=[e0cr4],G1=[s0cr1],G2=[s0cr2],
    G3=[s0cr3],G4=[s0cr4].

greTunnels([GRE1, GRE2, GRE3, GRE4):-
    GRE1=[t0cr3, t1cr1],
    GRE2=[t0cr1, t0cr4],
    GRE3=[t1cr3, t0cr2],
    GRE4=[t1cr2, t1cr4].

```

¹⁴ <http://www.cisco.com/warp/public/cc/pd/nemnsw/nesy/mn/index.shtml>

ipsecTunnels([T1, T2, T3, T4]):-

T1 = [s0cr3, s0cr1],
T2 = [s0cr1, s0cr4],
T3 = [s0cr3, s0cr2],
T4 = [s0cr2, s0cr4].

eval(addressingForLansAndGRE,DB):-

lans([S1,S2,S3,S4, G1, G2, G3, G4]),
greTunnels([GRE1,GRE2,GRE3,GRE4]),
eval(subnet(S1,ethernet,24,['10.5.5.1']),DB),
eval(subnet(S2,ethernet,24,['4.4.4.1']),DB),
eval(subnet(S3,ethernet,24,['172.18.48.1']),DB),
eval(subnet(S4,ethernet,24,['8.8.8.1']),DB),
eval(subnet(GRE1,gre,24,['7.7.7.1','7.7.7.2']),DB),
eval(subnet(GRE2,gre,24,['6.6.6.2','6.6.6.1']),DB),
eval(subnet(GRE3,gre,24,['9.9.9.1','9.9.9.2']),DB),
eval(subnet(GRE4,gre,24,['3.3.3.1','3.3.3.2']),DB),
eval(subnet(G1,serial,24,['172.16.24.2']),DB),
eval(subnet(G2,serial,24,['172.16.28.2']),DB),
eval(subnet(G3,serial,24,['172.16.32.2']),DB),
eval(subnet(G4,serial,24,['172.16.20.22']),DB).

eval(ospfOverLansAndGRE, DB):-

lans([S1, S2, S3, S4 | _]),
greTunnels([GRE1, GRE2, GRE3, GRE4]),
eval(ospfArea([S1, S2, S3, S4, GRE1,
GRE2, GRE3, GRE4], 10, 0, regular), DB).

eval(protectedGRETunnels, DB):-

greTunnels([GRE1, GRE2, GRE3, GRE4]),
eval(protectedGRETunnel(GRE1, [s0cr3, s0cr1], k1, name1), DB),
eval(protectedGRETunnel(GRE2, [s0cr1, s0cr4], k2, name2), DB),
eval(protectedGRETunnel(GRE3, [s0cr3, s0cr2], k3, name3), DB),
eval(protectedGRETunnel(GRE4, [s0cr2, s0cr4], k4, name4), DB).

eval(pimOverGRETunnels, DB):-

greTunnels(Tunnels),
eval(pimSubnets(Tunnels), DB).

eval(ftServer, DB):-

eval(wackOverSpread([m1,m2], '172.18.48.255', 4803,
[[m1, '172.18.48.3'], [m2, '172.18.48.4']], wack1,
[none, '172.18.48.100'], ['172.18.48.100'],
[cr3]), DB).

eval(P=V, DB):-member(P=V, DB).

eval(subnet([], Type, Mask, Addresses),DB).

eval(subnet([I | Interfaces], Type, Mask, [A | Addresses]), DB):-

eval(I:addressing = [Type, Mask, A], DB),
eval(subnet(Interfaces, Type, Mask, Addresses), DB).

eval(greTunnel(G1, G2, A1, A2), DB):-

eval(G1:gre=[G2, A1, A2], DB),
eval(G2:gre=[G1, A2, A1], DB).

eval(ospfSubnet([], PID, AreaID, StubID), DB).

eval(ospfSubnet([I | Rest], PID, AreaID, StubID), DB):-

eval(I:ospf=[PID, AreaID, StubID], DB),
eval(ospfSubnet(Rest, PID, AreaID, StubID), DB).

eval(ospfArea([],P, A, ST), DB).

eval(ospfArea([S/Subnets], P, A, ST), DB):-

eval(ospfSubnet(S, P, A, ST), DB),
eval(ospfArea(Subnets, P, A, ST), DB).

eval(ipsecTunnel(I1, I2, Name, Policy), DB):-

Policy = [Key, EA, HA, Mode, Protocol, Filter],
X = [I2, Key, EA, HA, Mode, Protocol, Filter],
Y = [I1, Key, EA, HA, Mode, Protocol, RevFilter],
reverseFilter(Filter, RevFilter),
eval(I1:(ipsec:Name)=X, DB),
eval(I2:(ipsec:Name)=Y, DB).

reverseFilter([], []).

reverseFilter([[Source, SMask, Dest, DMask, Protocol] | Rest], [P | RestP]):-

P = [Dest, DMask, Source, SMask, Protocol],
reverseFilter(Rest, RestP).

eval(protectedGRETunnel([G1, G2], [P1, P2], Key, Name), DB):-

eval(greTunnel(G1, G2, P1, P2), DB),
eval(P1:addressing=[_,_,P1A], DB),
eval(P2:addressing=[_,_,P2A], DB),
Policy = [Key, des, sha, tunnel, esp, [[P1A, 32, P2A, 32, gre]]],
eval(ipsecTunnel(P1, P2, Name, Policy), DB).

eval(spreadLan([], BAdd, Port, M_A_Pairs), DB).

eval(spreadLan([M|RestM], BAdd, Port, M_A_Pairs), DB):-

member([M,A], M_A_Pairs),
broadcast(A, BAdd),
eval(I:addressing=[_,_,A], DB),!,
eval(M:spread=[BAdd, Port, M_A_Pairs], DB),
eval(spreadLan(RestM, BAdd, Port, M_A_Pairs), DB).

eval(wackGroup([], Port, Group, [], VAdds, Notify), DB).

eval(wackGroup([M | RestM], Port, Group, [PrefAddr | RestPref],
VAdds, Notify), DB):-

member(PrefAddr, [none | VAdds]),
checkUniquePref(PrefAddr, RestPref),
eval(wackGroup(RestM, Port, Group, RestPref, VAdds, Notify),
DB),
eval(M:wackamole = [Port:localhost, Group, PrefAddr, VAdds,
Notify], DB).

eval(pimSubnets([], DB).

eval(pimSubnets([S | Rest]), DB):-

eval(pim(S, true), DB),
eval(pimSubnets(Rest), DB).

eval(pim([],Bool), DB).

eval(pim([Interface|Rest],Bool), DB):-

eval(Interface:pim=Bool,DB),
eval(pim(Rest,Bool), DB).

eval(wackOverSpread(Machines, BAdd, Port, M_A_Pairs, Group,
Preferences, VAdds, Notify), DB):-

eval(spreadLan(Machines, BAdd, Port, M_A_Pairs), DB),
eval(wackGroup(Machines, Port, Group, Preferences, VAdds,
Notify), DB).

checkUniquePref(none, Prefs).

checkUniquePref(A, Prefs) :- not member(A, Prefs).

broadcast(A,B) :-

name(B, Z),
name(A,Y),
reverse(Z, [53,53,50 | RevPrefix]),
reverse(RevPrefix, Prefix),
append(Prefix, _, Y).