Declarative Infrastructure Configuration Synthesis and Debugging[1]

Sanjai Narain, Gary Levin and Vikram Kaul, Telcordia Technologies, Inc.

Sharad Malik, Princeton University

Abstract

There is a large conceptual gap between end-to-end infrastructure requirements and detailed component configuration implementing those requirements. Today, this gap is manually bridged so large numbers of configuration errors are made. Their adverse effects on infrastructure security, availability, and cost of ownership are well documented. This paper presents ConfigAssure to help automatically bridge the above gap. It proposes solutions to four fundamental problems: specification, configuration synthesis, configuration error diagnosis, and configuration error repair. Central to ConfigAssure is a Requirement Solver. It takes as input a configuration database containing variables, and a requirement as a first-order logic constraint in finite domains. The Solver tries to compute as output, values for variables that make the requirement true of the database when instantiated with these values. If unable to do so, it computes a proof of unsolvability. The Requirement Solver is used in different ways to solve the above problems.

The Requirement Solver is implemented with Kodkod, a SAT-based model finder for first-order logic. While any requirement can be directly encoded in Kodkod, parts of it can often be solved much more efficiently by non model-finding methods using information available in the configuration database. Solving these parts and simplifying can yield a reduced constraint that truly requires the power of model-finding. To implement this plan, a quantifier-free form, QFF, is defined. A QFF is a Boolean combination of simple arithmetic constraints on integers. A requirement is specified by defining a partial evaluator that transforms it into an equivalent QFF. This QFF is efficiently solved by Kodkod. The partial evaluator is implemented in Prolog. ConfigAssure is shown to be natural and scalable in the context of a realistic, secure and fault-tolerant datacenter.

[1] This material is based upon work supported by Air Force Rome Laboratories (AFRL) under contract FA8750-07-C-0030, funded by Dr. Carl Landwehr. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Air Force Rome Laboratories.

This paper will appear in *Journal of Network and Systems Management, Special Issue on Security Configuration*, eds., Ehab Al-Shaer, Charles Kalmanek and Felix Wu, 2008.

1 Introduction

There is a large conceptual gap between end-to-end infrastructure requirements and detailed component configurations implementing those requirements. Today, this gap is manually bridged. This causes large numbers of configuration errors whose adverse effects on infrastructure security, availability, and cost of ownership are well documented [7, 8, 9, 10]. This paper presents ConfigAssure to help automatically bridge this gap.

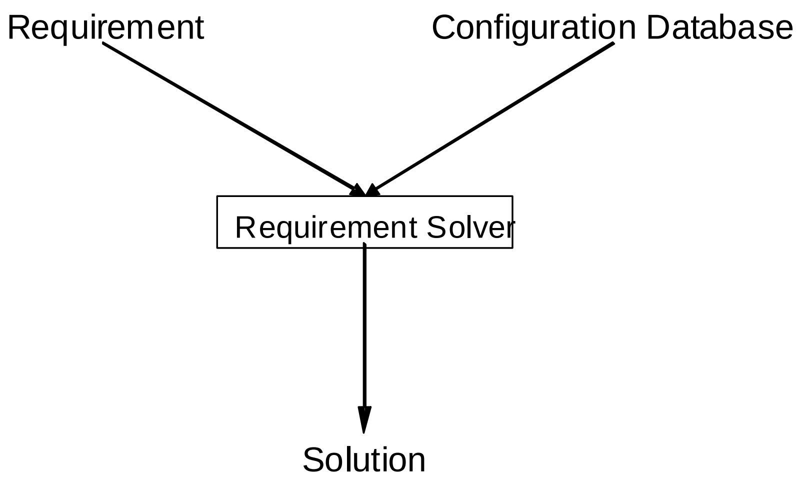


Figure 1. Requirement Solver

Central to ConfigAssure is a Requirement Solver as shown in Figure 1. This takes as input a configuration database, and a requirement on that database. Tuples in the configuration database can contain configuration variables. The Solver tries to compute values for these variables such that when these are replaced by their values, the resulting instantiated database satisfies the requirement. If unsuccessful, the Solver also computes a proof of unsatisfiability. The Requirement Solver is used to solve the following fundamental problems for bridging the gap between requirements and configurations:

- **Specification:** All requirements are expressed as constraints on configurations. A constraint is a unifying concept. Requirements on security, functionality, performance and reliability can all be naturally regarded as constraints. Configuration information is conveniently represented with a database. The meaning of requirements is defined by a partial evaluator that computes an equivalent quantifier-free form.
- **Configuration synthesis:** The declarative nature of the Requirement Solver provides the very important advantage of compositionality. Given requirements A and B, if the Solver computes a solution to $A \wedge B$ the solution is guaranteed to satisfy both A and B. Compositionality is not guaranteed with procedural approaches. If B is enforced after A is, then A may well become false. The configurations for B may overwrite those for A.
- **Configuration error diagnosis:** If a requirement is unsolvable, the Solver computes a proof of unsolvability as a set of primitive constraints that is also unsolvable. If it contains a constraint of the form $x=c$ where x is a configuration variable and c a constant, then $x=c$ is a useful root cause. It pinpoints which configuration parameter's value is contributing to unsolvability.
- **Configuration error repair:** If a constraint $x=c$ occurs in the proof of unsolvability, and also in the original requirement, then removing this constraint from the requirement is a good heuristic

for restoring solvability. Repair is a hard problem since changing a configuration to satisfy one requirement may violate others. Thus, the change must simultaneously satisfy all requirements, not just the violated one.

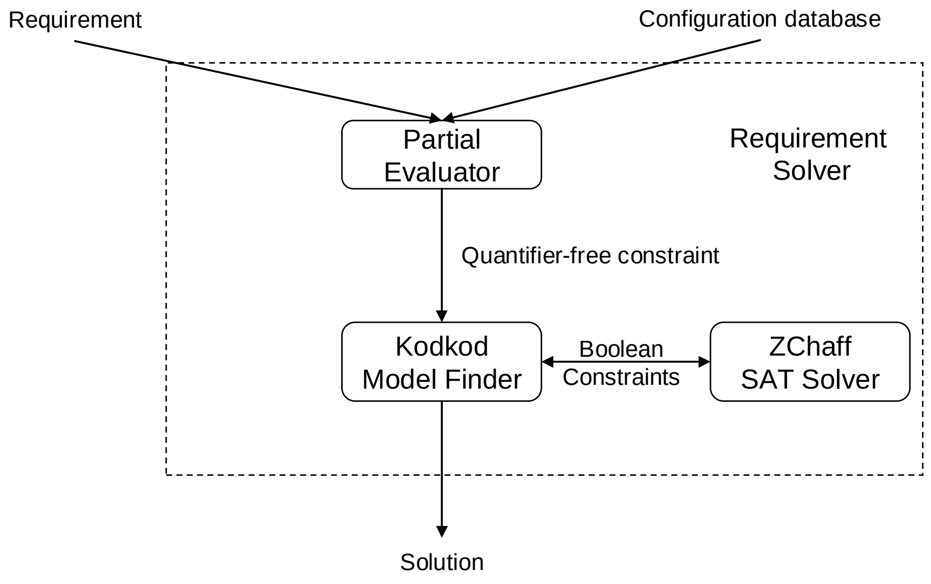


Figure 2. Internal Architecture of Requirement Solver

As shown in Figure 2, the Requirement Solver is implemented with Kodkod, a model-finder [3, 4]. Kodkod allows specification of first-order logic constraints in finite domains, transforms these into Boolean ones, solves these with SAT (satisfiability) solvers, then reflects results back into a model of first-order logic. If unable to do so, it outputs a proof of unsolvability. Modern SAT solvers such as ZChaff [2, 16] can solve millions of Boolean constraints in millions of Boolean variables (or output proofs of their unsolvability) in seconds.

Kodkod serves a critical role as an expressive, efficient and scalable front-end to a SAT solver. It directly accepts QFFs as input. It provides solutions or proofs of unsolvability in easy to manipulate data structures. Its optimizations, such as logarithmic encoding of integer variables, greatly reduce sizes of generated Boolean constraints.

However, while any requirement can be directly encoded and solved in Kodkod, many parts of it can often be solved much more efficiently by specialized constraint solvers, database engines or algorithms that use partial information available in the configuration database. Solving these parts and simplifying can yield a reduced constraint that truly requires the power of model-finding. A partial evaluation stage where a requirement is transformed into such a reduced constraint can scale up model-finding, and hence the Requirement Solver, to handle problems of realistic size.

ConfigAssure implements this plan by defining a quantifier-free form, QFF, consisting of Boolean combinations of simple arithmetic constraints on integers. A requirement is specified by defining a partial evaluator for it that transforms it into an equivalent QFF. A guiding principle for keeping the size of the QFF small is to disallow any constraint in it that can be evaluated via non model-finding methods. The QFF is submitted to Kodkod for solution. The Partial Evaluator in Figure 2 is the specification of all requirements of interest in a domain. It is currently implemented in Prolog [5,18]. QFFs offer several advantages:

- Their high-level nature simplifies the design of the partial evaluator.

- They can be efficiently solved by Kodkod.
- Their proofs of unsolvability simplify diagnosis and repair algorithms.

Section 2 outlines the design of a realistic, secure and fault-tolerant datacenter. It is the context for illustrating ConfigAssure. Section 3 describes the design of the Requirement Solver. Section 4 presents the partial evaluator for several representative requirements. Section 5 describes the application of ConfigAssure to solve fundamental configuration problems for the datacenter. Section 6 discusses ConfigAssure's performance for the datacenter and a related example. Section 7 discusses relationship with previous work. Section 8 presents a summary and outlines directions for future work.

2 A Secure, Fault-Tolerant Data Center

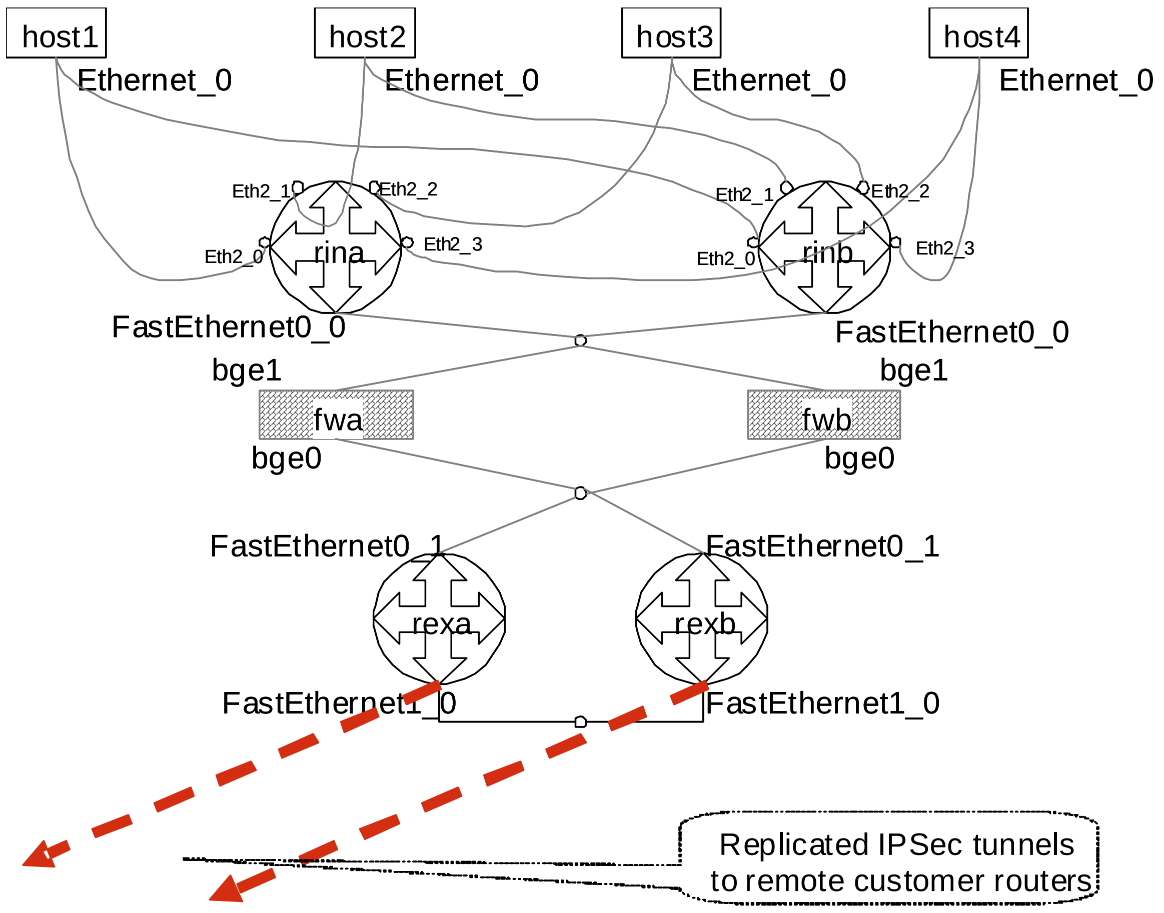


Figure 3. Secure, Fault-Tolerant Data Center

ConfigAssure will be illustrated in the context of a realistic datacenter as shown in Figure 3. This datacenter operates a shared wireless service backend for tens of external customers. It has to satisfy stringent requirements on security, performance and fault-tolerance. Hosts host1,..,host4 are the servers. Packets from these are statically routed via rina/rinb through firewalls fwa and fwb and out of gateway routers rexa/rexb. They then travel via IPSec tunnels to the external customer sites. Firewalls permit only predefined packet flows defined by source, destination and ports and protocols at source and destination. IPSec tunnels encrypt sensitive flows. The hot standby routing protocol, HSRP, run by external interfaces of rexa and rexb provides a virtual IP address to the outside world. If one router fails, the other takes over the virtual IP address to restore IP connectivity to the outside world. Requirements that capture the above datacenter design are:

IP Addressing

- All addresses are within a definite address range.
- All addresses in a subnet are unique.
- No two distinct subnets overlap.

Routing

- If an IPSec tunnel protects a packet, there is a static route that directs it into the tunnel.

Fault tolerance

- External interfaces of external routers participate in an HSRP group
- If an IPSec tunnel originates at an interface participating in an HSRP group, then the IPSec tunnel is replicated at all interfaces participating in that group.

Security

- Every packet flow to a customer site is protected via an IPSec tunnel at both external routers.
- Only negotiated packet flows to and from customer sites are permitted through firewalls.
- Rule sets on all firewalls are identical.

To implement these requirements, system components such as hosts, routers and firewalls need to be configured. Some configuration parameters are:

- IP addresses and masks of interfaces.
- Static route destinations and next hops on routers.
- IPSec tunnel source, destination, encryption and hash algorithms, keys, and packet flows to be encrypted. Tunnels are configured on routers.
- HSRP groups, interfaces participating in these groups and virtual address for each group. HSRP is configured on sets of interfaces.
- Firewall rules specifying what packet flows are permitted or denied. Firewall rules are configured on firewalls and routers.

A typical router's configuration file can contain anywhere from a hundred to several thousand configuration commands. Ensuring that these are consistent with end-to-end requirements is hard, so large numbers of configuration errors are made. These can cause loss of connectivity, security breaches, single points of failure and performance degradation.

A configuration error actually caused a single point of failure in the datacenter, in spite of redundant resources in place. It arose from an interesting dependence between security and fault-tolerance. If the active HSRP router fails, then the backup router assumes the virtual IP address. However, this is not sufficient to restore IPSec connectivity with customer sites. Every IPSec tunnel originating from the active router must also be replicated at the backup router. This is the reason for the second fault-tolerance requirement. This requirement was incorrectly implemented for one customer site, so that site was at risk of losing service to the datacenter.

3 Requirement Solver Design

This section presents a precise definition of the Requirement Solver then shows how it is used to solve fundamental configuration problems.

3.1 Definitions

Primitive Basis. We assume an enumerably infinite set of configuration variables, and function symbols, predicate symbols and configuration database symbols of zero or more arguments. Zero-argument function symbols are also called scalars. The set of scalars includes integers.

A Requirement Solver for a particular configuration application will contain only a finite number of variables and symbols but all drawn from the above set.

Terms. Every configuration variable and scalar is defined to be a term. If $x_1,..,x_k$ are terms and F is a k-argument function symbol then $F(x_1,..,x_k)$ is a term. An example of a term is the list [host1-h1, rexa-rA1, rexb-rB1] containing three host-interface pairs. Here [A, B, C] is an abbreviation for the term |(A, |(B, |(C, []))) where | is a list constructor symbol and [] is the symbol for the empty list. Also, A-B is an abbreviation for –(A, B) where – is a pair constructor symbol. – is different from the symbol for subtraction, and meanings are distinguished from context.

Configuration Database. Where P is a database symbol of k-arguments and $x_1,..,x_k$ are either scalars or configuration variables then $P(x_1,..,x_k)$ is a database tuple. A finite set of database tuples is a configuration database, for example:

```
ipAddress(host1, h1, '121.96.41.1', 24).
ipAddress(rexa, rA1, '121.96.41.2', 24).
hsrp(rexa, rA1, int(1), int(2)).
hsrp(rexb, rB1, int(3), int(4)).
ipSecTunnel('121.96.41.1', '192.168.1.2', 3des, sha, xxx, profile)
flow(rexa, profile, '1.1.1.1', 80, '2.2.2.2', 80, tcp)
staticRoute(host1, '192.168.1.2', '255.255.255.255', '121.96.41.3').
```

The first tuple states that the IP address of interface h1 on host1 is '121.96.41.1' with a 24 bit mask. An n-bit mask is a bit sequence of n ones followed by 32-n zeros. Similarly, for the second. The third states that interface rA1 of rexa participates in an HSRP group int(1) with virtual IP address int(2). Similarly, for the fourth. Here int(1), int(2), int(3), int(4) are configuration variables. The fifth tuple states that there is an IPSec tunnel between addresses in the first two fields with encryption algorithm 3des, hash algorithm sha, preshared key xxx. The packets this tunnel protects are specified by the flow profile. The sixth tuple defines profile on rexa as the set of packets with source address, source port, destination address, destination port, and protocol as specified in the third through the seventh fields, respectively. The last tuple states that on host1 there is a static route to the address range specified by the second and third fields via the next hop in the last field.

The syntactic convention followed *in a configuration database* is that all compound terms, such as int(1), int(2) are configuration variables.

For exposition purposes, IP addresses are shown in dotted quad notation but in the implementation, they are represented by integers. For example, '121.96.41.1' is represented by 2036345089=121*256^3 + 96*256^2 + 41*256 +1.

Requirements. If $x_1,..,x_k$ are terms and R is a k-argument predicate symbol then $R(x_1,..,x_k)$ is a requirement. If R1 and R2 are requirements then so are not(R1), and(R1, R2), or(R1, R2), implies(R1, R2). Requirements don't have explicit quantifiers but they can be given a meaning that is equivalent to a first-order logic formula that does. A configuration variable-free requirement is intended to be true or false of a configuration variable-free database. We use "constraints" and "requirements" interchangeably. Examples of requirements are:

- all_physical_addresses_distinct. For a database, this means that no two physical IP addresses in the database are equal. This requirement has implicit universal quantification.
- address_space(121.96.0.0, 16). For a database, this means that all addresses in that database are in the range 121.96.0.0 to 121.96.255.255.
- subnet([host1-h1, rexa-rA1, rexb-rB1]). For a database, this means that the database contains tuples defining IP addresses and masks for interfaces h1, rA1, rB1 on devices host1, rexa, rexb respectively. Furthermore, these interfaces are on same subnet in that their network identifiers are identical. The network identifier of an interface is obtained by bitwise-anding its IP address and mask.
- hsrp_subnet([rexa-rA1, rexb-rB1]). For a database, this means that the database contains tuples defining HSRP group identifiers and virtual IP addresses for interfaces rA1, rB1 on devices rexa, rexb respectively. Furthermore, these interfaces have the identical group identifier and virtual address, and this virtual address and physical addresses of interfaces are in the same subnet.
- contained('121.96.41.1', 16, '121.96.41.2', 24). This means that the network identifier 121.96.41.1/16 contains the subnet range '121.96.41.2'/24. A/M is the network identifier formed from bitwise-anding the IP address A and mask M.

Quantifier-Free Forms. A quantifier-free form, QFF, is a constraint formed from configuration variables, integers, function symbols for addition, subtraction and bitwise operations, and predicate symbols =, <, >, >=, =<. An example of a QFF is:

```
and(mask(1)<mask(2),
    bitwiseand(addr(1), 4294967295<<(32-mask(1)))
     = bitwiseand(addr(2), 4294967295<<(32-mask(1))))
```

This is an expansion of the constraint contained(addr(1), mask(1), addr(2), mask(2)). The binary representation of 4294967295 is a sequence of 32 1s. It is left shifted to construct the actual mask.

Partial Evaluator. The meaning of requirements in the context of a database is defined by the Partial Evaluator. The evaluator transforms a requirement into an equivalent QFF. Let σ be an assignment of variables to integers $\{<x_1=v_1>,..,<x_k=v_k>\}$, each x_i a configuration variable and each v_i an integer. Let Req be a requirement, DB a database and C a QFF. Let Reqσ be the result of replacing each variable in Req by its value in σ. Similarly, for DBσ and Cσ. Now, eval(Req, DB, C) means that for any assignment σ, Reqσ is true of DBσ iff Cσ. In practice, DB is implicit and we define the predicate eval(Req, C).

The Partial Evaluator for a configuration application is the definition of the eval predicate for all requirements and databases of interest in that application. It is implemented in Prolog.

A central guiding principle in the implementation of eval(Req, DB, C) is that C should not contain a constraint that can be evaluated by non-model finding methods. Kodkod will still evaluate such a constraint but incur needless overhead. For example, and(2+3>4, mask(1)=24) should be reduced to mask(1)=24 before being submitted to Kodkod.

Kodkod Interface. This interface is defined by a Prolog predicate solve(Q, Result) where Q is a QFF. If Q is solvable, Result is a term solvable:σ where σ is a assignment of configuration variables to integers satisfying Q. If Q is unsolvable, Result is a term unsolvable:P where P is a proof of unsolvability. This proof is a list of QFFs whose conjunction is itself unsolvable. An example of P is [mask(0)=16, not(mask(0)=16)]. Usually, this conjunction is much smaller than Q. The implementation of solve calls Kodkod methods for initializing Kodkod, setting up the QFF in Java, solving it and finally importing a solution or a proof of unsolvability back into Prolog.

3.2 Solving Fundamental Configuration Problems

The above framework greatly simplifies the solution of fundamental configuration problems. To **specify a requirement**, define the partial evaluator (eval) for it. One need not define eval for each requirement but rather for requirement templates such as subnet(L) above. For a particular application domain a Requirement Library is envisioned that contains procedures to efficiently solve fundamental classes of requirements in that domain. A user can compose these to define complex requirements. As this library grows, the task of specifying new requirements will become simpler.

For **configuration synthesis**, given a requirement R and a configuration database DB, to find a variable valuation σ such that Rσ is true of DBσ, use the following Prolog query:

```
eval(R, DB, Q), solve(Q, solvable: X)
```

If the query succeeds, the Prolog variable X will be bound to σ.

For **configuration error diagnosis**, let DB be a configuration database containing variables $x_1,\ldots,x_k$. Let Relaxable be the conjunction of the primitive constraints $x_1=v_1,..,x_k=v_k$ where each x_i is a configuration variable and each v_i is an integer. Relaxable is used not only to specify initial values for a set of variables but also to indicate that these values can be relaxed if necessary. Suppose for some Req, DB and Q, eval(Req, DB, Q) but and(Q, Relaxable) is unsolvable. Then the Prolog query:

```
eval(Req, DB, Q), solve(and(Q, Relaxable), unsolvable:Proof)
```

will succeed, binding the Prolog variable Proof to a list of QFFs whose conjunction is also unsolvable. If a constraint $x_i=v_i$ is a member of Proof then it becomes a useful root cause of the unsolvability of and(Q, Relaxable). If such a constraint cannot be found, the algorithm halts.

For **configuration error repair**, to find an alternative value of x_i, remove $x_i=v_i$ from Relaxable to create Relaxable' and try to solve and(Q, Relaxable'). If solve succeeds, it will find a new value of x_i, in effect repairing the incorrect value v_i it was set to. If not, compute a new proof of unsolvability and repeat. If such a constraint cannot be found, the algorithm halts.

4 Partial Evaluator Examples

This section describes the partial evaluator for a representative set of requirements. The partial evaluator is implemented in Prolog.

4.1 All Physical IP Addresses Distinct

The partial evaluator for the requirement all_physical_addresses_distinct finds the set of all address fields in all ipAddress tuples in the configuration database, then computes the QFF for all of these to be distinct. Let U and A be two distinct fields. If any of these is a configuration variable then one cannot evaluate whether U and A are unequal so not(U=A) is included in the QFF. Otherwise, not(U=A) is evaluated by Prolog. If true, then it is not included in the QFF. If false, then the QFF false is output signifying that all_physical_addresses_distinct cannot be true for the given database. This plan illustrates ConfigAssure's guiding principle that if a constraint can be evaluated by non model-finding methods then it not be included in the QFF. The plan is implemented by the following Prolog rules:

```
eval(all_physical_addresses_distinct, C):-,
    findall(X, H^I^M^ipAddress(H, I, X, M), S),
    eval(no_duplicates(S), C).

eval(no_duplicates([]), true).
eval(no_duplicates([U|V]), and(D, E)):-
    eval(no_duplicates(V), D),
    eval(non_member(U, V), E).

eval(non_member(U, []), true).
eval(non_member(U, [A|B]), and(C, D)):-
    check([not(U=A)], C),
    eval(non_member(U, B),D).
```

The first rule states that the QFF for all_physical_addresses_distinct is C provided S is the set of all IP addresses in the configuration database, and C is the QFF for no_duplicates(S). The second states that if S is empty, this QFF is true. The third computes two QFFs, one for the tail of S and another for the requirement that the head of S not be a member of the tail of S. Finally, it returns the conjunction of the two QFFs. The fourth and fifth rules compute the QFF for an address not being a member of a list. The check procedure binds C to not(U=A) if one of U and A is a configuration variable, and to true if both U and A are distinct integers. If U and A are equal, check fails. This failure causes eval to return false via a default rule, not shown here. The Prolog query eval(all_ip_addresses_distinct, C) for the database:

```
ipAddress(rexa, ha, '121.96.41.1', 24).
ipAddress(rexb, hb, ‘121.96.41.2’, 24).
ipAddress(rexc, hc, addr(1),  24).
```

binds C to the QFF and(not(addr(1)='121.96.41.1'), not(addr(1)='121.96.41.2')). Note that not('121.96.41.1'='121.96.41.2') does not appear in the QFF because it is already evaluated to be true. In general, if there were k interfaces and addresses of k-1 interfaces are known, then the QFF would contain only k-1 constraints. It would *not* contain k*(k-1)/2 inequalities from a straightforward interpretation of all_physical_addresses_distinct.

4.2 All Interfaces In Same Subnet

The partial evaluator for a more complex requirement, subnet(L) where L is a list of host-interface pairs computes the QFF for the subnet identifier of each interface to be equal. It does so via the contained(A, M, B, N) requirement template defined earlier. It is implemented by the following Prolog rules:

```
eval(subnet([]), true).
eval(subnet([_]), true).
eval(subnet([H-I, H1-I1|Rest]), C):-
    ipAddress(H, I, A, M),
    ipAddress(H1, I1, A1, M1),
    check([contained(A, M, A1, M1)], C1),
    check([M=M1], C2),
    eval(subnet([H1-I1 | Rest]), CR),
    simplify(and(C1, and(C2, CR)), C).
```

The first two rules state that if there is just zero or one interface in L, the QFF is true. The last rule states if there is more than one interface, then the QFF is C where:

- C1 is the QFF for the network identifier of the first interface in L containing the network identifier for the second.
- C2 is the QFF for the masks of the first and second interfaces being equal
- CR is the QFF for all interfaces in L, except the first one, being on the same subnet
- C is the Boolean simplification of and(C1, and(C2, CR)).

The first two calls in the body retrieve addresses A, A1 and masks M, M1 from the configuration database. The third call in the body evaluates the containment constraint for A, M, A1, M1. If these are not all constants, then the constraint cannot be evaluated and C1 is bound to (an unevaluated version) of the constraint itself. If the constraint can be evaluated, then C1 is bound to its value, true or false. Similarly, for the fourth call in the body. These calls to check again illustrate the ConfigAssure guiding principle that if a constraint can be evaluated through non model-finding means it should not become part of the final QFF. If C1 or C2 are bound to true or false, the call to simplify will filter these away leading to an even more reduced QFF. The definition of check is:

```
check([], true).
check([contained(A,M,A1,M1) | Z], Rest):-
    forall(member(X, [A, M, A1, M1]), number(X)),
    M>=M1,
    chk_subnet_id(A, M, N),
    chk_subnet_id(A1,M, N),
    check(Z, Rest).
check([U=V | Z], Rest):-
   atomic(U),
   atomic(V),
   U=V,
   check(Z, Rest).

chk_subnet_id(A, Mask, N):-
    two_to_the_thirty_two_minus_one(X),
    Z is X<<Mask,
    N is Z /\ A.
```

The first rule states that if the list of constraints to be checked is empty, then return the QFF true. The second rule states that if arguments of the containment constraint are numbers and the containing mask M is greater than or equal to the contained mask M1, evaluate it via the two calls to check_subnet_id. If the result is true, discard this constraint (since it is already true) and return the result of checking the rest of the list. The third rule is analogous to the second but for equality. The fourth rule defines chk_subnet_id for computing the network identifier given an address and a mask (the latter expressed as the number of zeros in the conventional representation of a mask). The sequence of 32 1s is left shifted by Mask and then bitwise-anded with A to return the network identifier N.

Where R is the requirement subnet([rexa-ha, rexb-hb, rexc-hc]) and DB is the database:

```
ipAddress(rexa, ha, '121.96.41.1', 24).
ipAddress(rexb, hb, ‘121.96.41.2’, 24).
ipAddress(rexc, hc, '121.96.41.3', mask(1)).
```

the Prolog query eval(R, C) will bind C to the QFF:

```
and(contained(‘121.96.41.1’, 24, ‘121.96.41.3’, mask(1)), mask(1)=24).
```

Note that C does not contain any constraint about rexa-ha and rexb-hb belonging to the same subnet. It has already been evaluated to be true.

4.3 All Interfaces Participating in HSRP Group

We now present the partial evaluator for a cross-protocol requirement hsrp_subnet(L) where L is a list of host-interface pairs participating in the same HSRP group. It captures the dependence between HSRP and IP addressing configurations. This dependence is that the virtual IP address of an interface must be in the subnet of that interface's physical IP address. The Prolog rules implementing it are similar to those for subnet(L):

```
eval(hsrp_subnet([]), true).
eval(hsrp_subnet([H-I]), true).
eval(hsrp_subnet([H1-I1, H2-I2|Rest]), and(C, CRest)):-
    hsrp(H1, I1, G1, V1),
    hsrp(H2, I2, G2, V2),
    ipAddress(H1, I1, A1, M1),
    check([contained(A1, M1, V1, 32)], C1),
    check([contained(A1, M1, V2, 32)], C2),
    check([G1=G2, V1=V2], C3),
    andEach([C1, C2, C3], C),
    eval(hsrp_subnet([H2-I2|Rest]), CRest).
```

4.4 Security And Fault-Tolerance Requirement

We now present the partial evaluator for another cross-protocol requirement whose QFF can contain an implication. This requirement from Section 2 is "If an IPSec tunnel originates at an interface participating in an HSRP group, then the IPSec tunnel is replicated at all interfaces participating in that group". Its partial evaluator enumerates all hsrp-ipsec lists of the form [Source1, Source2, D, EA, HA, K, F] where Source1 and Source2 are addresses participating in an HSRP group and D, EA, HA, K, F are parameters of an IPSec tunnel, respectively, destination, encryption algorithm, hash algorithm, key and protection profile. For each such list, the evaluator checks that if there is a tunnel originating from Source1 with the tunnel parameters then there is also a tunnel originating from Source2 with the same parameters. If any field in the list is a configuration variable, an implication is added to the QFF. Otherwise, the list is evaluated for conformance to the requirement. If conforming, true is added to the QFF. If false, eval is made to return false. This requirement is named all_ipsec_cloned_at_hsrp. Its partial evaluator is implemented by the following Prolog rules:

```
eval(all_ipsec_cloned_at_hsrp, C):-
   findall(C, T^(ipsec_hsrp_tuple(T), eval(ipsec_cloned(T), C)), S),
   andEach(S, C).

ipsec_hsrp_tuple([Source1, Source2, D, EA, HA, K, F]):-
   ipSecTunnel(_, D, EA, HA, K, F),
   hsrp_address_pair(Source1, Source2).

eval(ipsec_cloned([Source1, Source2, D, EA, HA, K, F]), implies(C1, C2)):-,
   eval(ipSecTunnel(Source1, D, EA, HA, K, F), C1),
   eval(ipSecTunnel(Source2, D, EA, HA, K, F), C2).
```

The first rule computes the set of all hsrp-ipsec lists, computes the QFF for each satisfying the ipsec_cloned requirement and returns a conjunction of the QFFs. The second rule computes an hsrp-ipsec list. The third rule evaluates the requirement for such a list.

4.5 Top-Level Datacenter Requirement

Definitions of eval for above requirements and those for many others are combined into that for the top-level requirement, datacenter as follows:

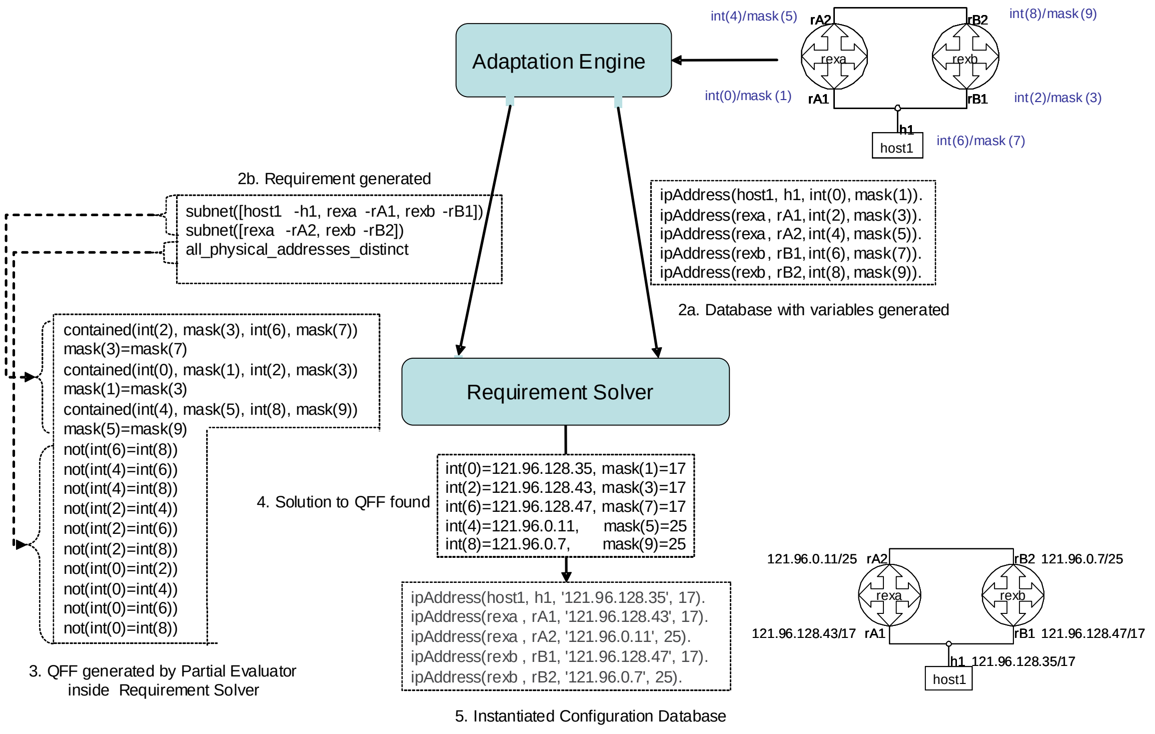
```
eval(datacenter, QFF):-
    eval(addressing, CA),
    eval(routing, CR),
    eval(fault_tolerance, CFT),
    eval(security, CS)
    andEach([CA, CR, CFT, CS], QFF).
```

Here addressing, routing, fault_tolerance, and security implement the requirements in Section 2. The definition of eval for addressing calls those for all_physical_addresses_distinct and subnet(L), and that for fault_tolerance calls that for all_ipsec_cloned_at_hsrp.

5 Examples of Use of ConfigAssure

We now illustrate how fundamental configuration problems are solved by ConfigAssure in the context of the above datacenter. ConfigAssure also contains an auxiliary system called the Adaptation Engine that responds to external events by generating the two inputs to the Requirement Solver: a requirement and a configuration database.

5.1 Configuration Synthesis



```
subnet([host1  -h1, rexa  -rA1, rexb  -rB1])
subnet([rexa  -rA2, rexb  -rB2])
all_physical_addresses_distinct
```

```
ipAddress(host1, h1, int(0), mask(1)).
ipAddress(rexa, rA1, int(2), mask(3)).
ipAddress(rexa, rA2, int(4), mask(5)).
ipAddress(rexb, rB1, int(6), mask(7)).
ipAddress(rexb, rB2, int(8), mask(9)).
```

```
contained(int(2), mask(3), int(6), mask(7))
mask(3)=mask(7)
contained(int(0), mask(1), int(2), mask(3))
mask(1)=mask(3)
contained(int(4), mask(5), int(8), mask(9))
mask(5)=mask(9)
not(int(6)=int(8))
not(int(4)=int(6))
not(int(4)=int(8))
not(int(2)=int(4))
not(int(2)=int(6))
not(int(2)=int(8))
not(int(0)=int(2))
not(int(0)=int(4))
not(int(0)=int(6))
not(int(0)=int(8))
```

```
int(0)=121.96.128.35, mask(1)=17
int(2)=121.96.128.43, mask(3)=17
int(6)=121.96.128.47, mask(7)=17
int(4)=121.96.0.11,    mask(5)=25
int(8)=121.96.0.7,     mask(9)=25
```

```
ipAddress(host1, h1, '121.96.128.35', 17).
ipAddress(rexa , rA1, '121.96.128.43', 17).
ipAddress(rexa , rA2, '121.96.0.11', 25).
ipAddress(rexb , rB1, '121.96.128.47', 17).
ipAddress(rexb , rB2, '121.96.0.7', 25).
```

Figure 4. Configuration Synthesis

In Figure 4, in step 1 at upper right, the Adaptation Engine receives a request to set up addressing for the network as shown. This has two routers and a host. The lower subnet connects interfaces rA1 on router rexA, rB1 on router rexb, and h1 on host1. The upper subnet connects rA2 and rB2. The configuration database is currently empty. This represents a greenfield deployment.

In step 2a, the Adaptation Engine generates a configuration database consisting of five tuples, each representing the IP address and mask of an interface.

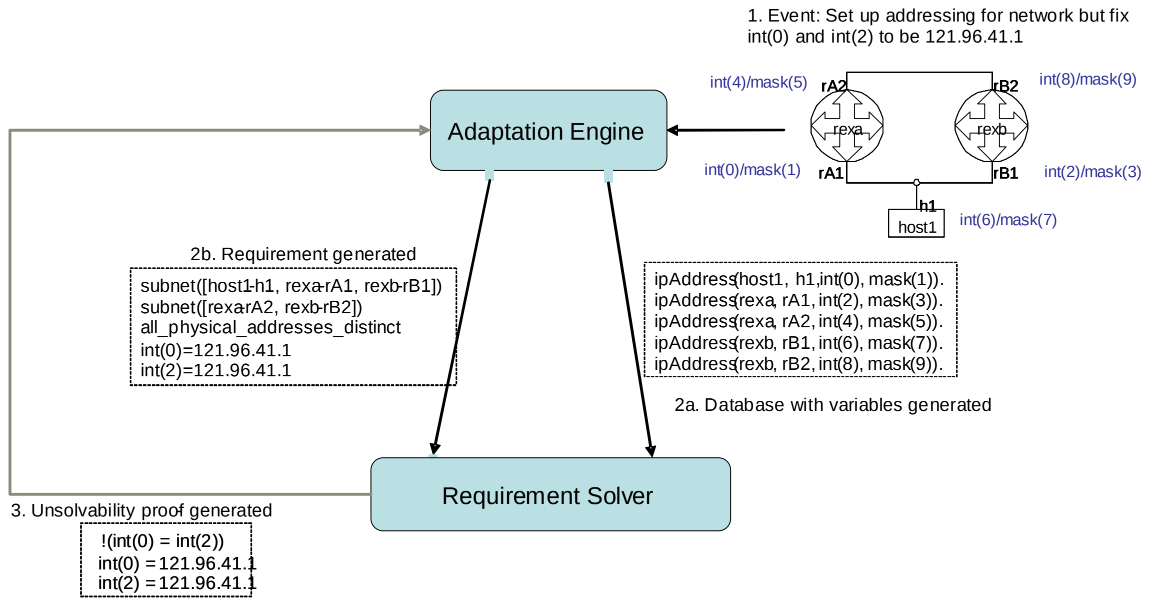
In step 2b, the Engine also generates three requirements. The first states that three interfaces in the lower subnet are to be on the same subnet. This means their network IDs must be identical. The second states the same for the interfaces on the upper subnet. The last states that all addresses must be distinct.

In step 3, when the configuration database with variables and requirements are submitted to the Requirement Solver, it generates the QFF on the left. The first 6 constraints are lower-level representations of the first two constraints in Step 2b.

The last 10 constraints in Step 3 are the QFF for the third requirement in Step 2b. These are the (5*4)/2 pairs of IP address inequalities for the five interfaces.

In step 4, the Requirement Solver produces a solution as shown. When the configuration database with variables is instantiated with this solution, the configuration database at the bottom is computed. Addressing is set up as shown on the lower right.

5.2 Configuration Error Diagnosis


1. Event: Set up addressing for network but fix int(0) and int(2) to be 121.96.41.1
Adaptation Engine
int(4)/mask(5)
rA2
rB2
int(8)/mask(9)
rexa
rexb
int(0)/mask(1)
rA1
rB1
int(2)/mask(3)
h1
host1
int(6)/mask(7)
2b. Requirement generated
2a. Database with variables generated
Requirement Solver
3. Unsolvability proof generated

```
subnet([host1-h1, rexa-rA1, rexb-rB1])
subnet([rexa-rA2, rexb-rB2])
all_physical_addresses_distinct
int(0)=121.96.41.1
int(2)=121.96.41.1
```

```
ipAddress(host1, h1,int(0), mask(1)).
ipAddress(rexa, rA1, int(2), mask(3)).
ipAddress(rexa, rA2, int(4), mask(5)).
ipAddress(rexb, rB1, int(6), mask(7)).
ipAddress(rexb, rB2, int(8), mask(9)).
```

```
!(int(0) = int(2))
int(0) =121.96.41.1
int(2) =121.96.41.1
```

Figure 5. Configuration Error Diagnosis

Figure 5 shows how to do configuration error diagnosis via analysis of proof of unsolvability. In Step 1, the Adaptation Engine receives a request to set up addressing for the same network but where rA1 and rA2 have already been configured with the same IP address, 121.96.41.1. In steps 2a and 2b, the Engine generates the same configuration database and Requirement as before. However, it also strengthens Requirement with the constraints int(0)=121.96.41.1=int(2). The Requirement Solver transforms Requirement into a QFF but is unable to solve it. The requirement all_physical_addresses_distinct cannot be satisfied.

In step 3, it produces the root cause as a set of three conditions that cannot be satisfied. It is important to note that this set is much smaller than the 18 constraints in the full QFF. In particular, only the two relevant variables appear, not the other eight irrelevant ones.

The root cause is a good basis for repairing configuration errors as shown next.

5.3 Configuration Error Repair

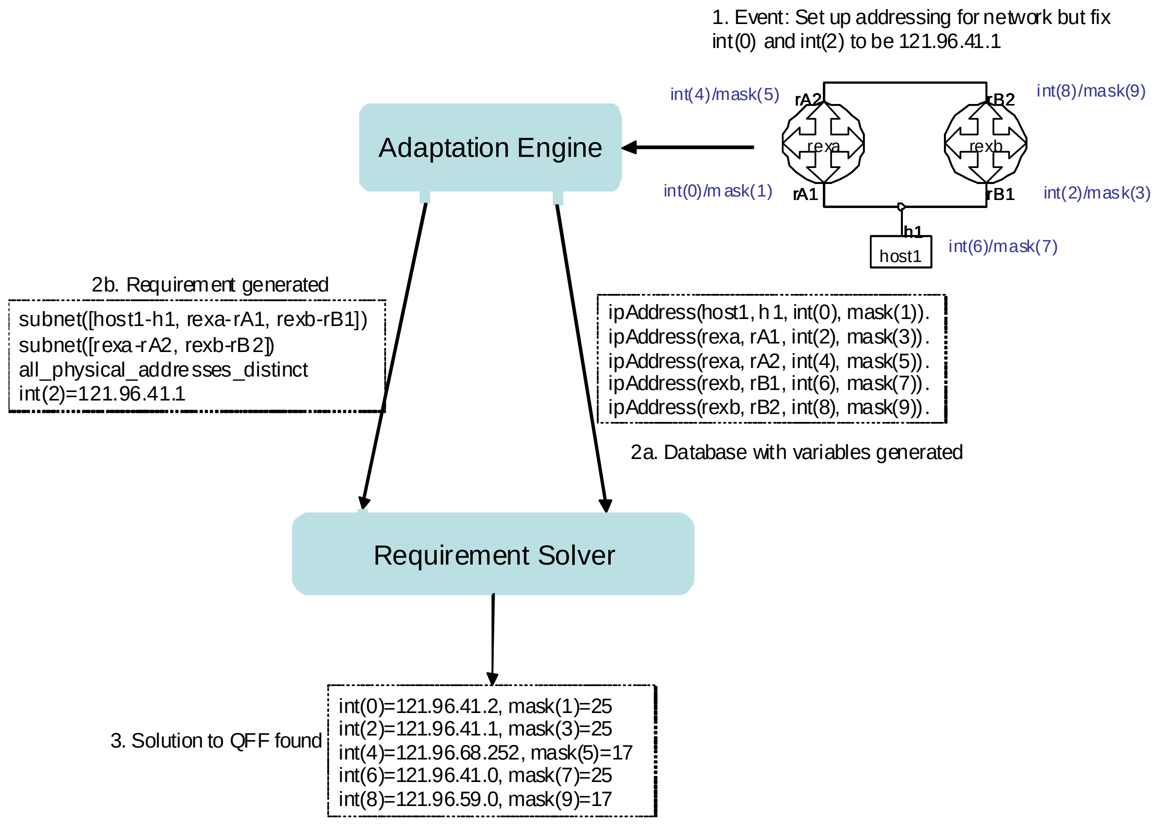

1. Event: Set up addressing for network but fix int(0) and int(2) to be 121.96.41.1
int(4)/mask(5)
rA2
rB2
int(8)/mask(9)
rexa
rexb
int(0)/mask(1)
rA1
rB1
int(2)/mask(3)
h1
host1
int(6)/mask(7)
Adaptation Engine
2b. Requirement generated
subnet([host1-h1, rexa-rA1, rexb-rB1])
subnet([rexa-rA2, rexb-rB2])
all_physical_addresses_distinct
int(2)=121.96.41.1
ipAddress(host1, h1, int(0), mask(1)).
ipAddress(rexa, rA1, int(2), mask(3)).
ipAddress(rexa, rA2, int(4), mask(5)).
ipAddress(rexb, rB1, int(6), mask(7)).
ipAddress(rexb, rB2, int(8), mask(9)).
2a. Database with variables generated
Requirement Solver
3. Solution to QFF found
int(0)=121.96.41.2, mask(1)=25
int(2)=121.96.41.1, mask(3)=25
int(4)=121.96.68.252, mask(5)=17
int(6)=121.96.41.0, mask(7)=25
int(8)=121.96.59.0, mask(9)=17

Figure 6. Configuration Error Repair

In Figure 6, the proof of unsolvability is fed back to the Adaptation Engine. In it, the Engine finds two constraints of the form x=c where x is a variable and c is a constant. These are int(0)=121.96.41.1 and int(2)=121.96.41.1. The Engine removes one of these constraints and produces a new Requirement as shown in Step 2b. The Solver now computes a solution as shown.

5.4 Defense Hardening

One method of increasing an adversary's effort required for his success is to increase the number of conditions he needs to satisfy to succeed. For example, if packets from h1 travel through rexa then in order to deny service to h1, an adversary could gain access to rexa and shut it down. To make it harder for the adversary to do so, one can make the routers participate in a fault-tolerance protocol such as Hot Standby Router Protocol, HSRP. Then, rexA and rexB would offer a virtual address to host1. All packets from host1 would be directed towards this virtual address. If rexA is shut down, rexB will take over the virtual IP address so host1's packets will continue to be routed out.

This idea is easy to implement in ConfigAssure. If the infrastructure needs to satisfy requirement R, and the defense hardening condition is S, find a solution to $R \wedge S$. To harden defense against denial-of-service, the new requirement generated by the Adaptation Engine contains an extra constraint to add HSRP on the lower subnet:

```
subnet([host1-h1, rexa-rA1, rexb-rB1])
subnet([rexa-rA2, rexb-rB2])
hsrp_subnet([rexa-rA1, rexb-rB1]).
```

Assuming one wants to keep an existing address assignment and only compute HSRP configurations, a configuration database with variables is:

```
hsrp(rexa, rA1, int(1), int(2)).
hsrp(rexb, rB1, int(3), int(4)).
ipAddress(host1, h1, '121.96.0.106', 25).
ipAddress(rexa, rA1, '121.96.0.72', 25).
ipAddress(rexb, rB1, '121.96.0.21', 25).
ipAddress(rexa, rA2, '121.96.93.148', 17).
ipAddress(rexb, rB2, '121.96.31.182', 17).
```

The first tuple states that interface rA1 belongs to the HSRP group int(1) and its virtual address is int(2). Similarly, for rB1. The Requirement Solver computes the conjunction of the following QFFs:

```
contained('121.96.0.72', 25, int(2), 32)
contained('121.96.0.72', 25, int(4), 32)
int(1)=int(3)
int(2)=int(4)
```

The first constraint states that the subnet int(2)/32 must be contained in '121.96.0.72/25'. This means that the virtual address of rA1 must be on the subnet of rA1. Similarly, for the second. The next two state that the groups and virtual addresses of both interfaces must be equal.

It is important to note that even though the requirements subnet([host1-h1, rexa-rA1, rexb-rB1]) and subnet([rexa-rA2, rexb-rB2]) are input to the Requirement Solver, the QFF contains no constraints relating to addressing. This is because the Solver, via the check predicate, used the addressing information in the configuration database to evaluate these constraints to true. *The size of the QFF is independent of the number of subnet requirements and that of ipAddress tuples as long as these tuples are consistent with* subnet *requirements.* This again illustrates ConfigAssure's guiding principle that any constraint that can be evaluated by non model-finding methods not be included in the QFF. When this QFF is solved by Kodkod, one obtains the fully instantiated configuration database:

```
hsrp(rexa, rA1, 0, '121.96.0.1').
hsrp(rexb, rB1, 0, '121.96.0.1').
ipAddress(host1, h1, '121.96.0.106', 25).
ipAddress(rexa, rA1, '121.96.0.72', 25).
ipAddress(rexb, rB1, '121.96.0.21', 25).
ipAddress(rexa, rA2, '121.96.93.148', 17).
ipAddress(rexb, rB2, '121.96.31.182', 17).
```

6 Implementation Notes and Performance Evaluation

6.1 Implementation Notes

The configuration database is conveniently implemented with a Prolog database. The eval and solve predicates are also implemented in Prolog. We use the stable, public-domain SWI-Prolog implementation [5]. Prolog calls Kodkod via the Java to Prolog interface called JPL [6]. QFFs are preprocessed in Prolog before being submitted to Kodkod. Solutions or proofs of unsolvability returned from Kodkod are postprocessed into Prolog data structures.

The reason for restricting values of configuration variables to integers is Kodkod's logarithmic encoding of integers. If there are n elements in a set then one only needs $\log_2 n$ bits to encode an element of that set. For example, while each IPv4 address takes 32 bits to represent, each mask just takes 5 bits if a mask is represented by the number of 1s in it. For 1000 addresses and 1000 masks, without logarithmic encoding, 64K bits = 1000*32+1000*32 are needed. With logarithmic encoding, just 37K bits = 1000*32+1000*5 are needed. This leads to drastic reduction in the size of generated Boolean constraints and SAT solving time.

6.2 Performance Evaluation for Datacenter Configuration Synthesis

All requirements in Section 2 were encoded in ConfigAssure and associated configurations synthesized. For the largest case, configurations were synthesized in about 2.5 minutes. The performance is summarized in Table 1 below.

Cust	Gen QFF sec	Setup K2 sec	K2QFF sec	K2 Trans sec	K2 Solve sec	p	cnf	primary	|QFF|	#Vars	|DB|
1	0	1	0.03	0.375	0.297	28406	79933	2798	967	111	63
10	0.11	1.31	0.11	0.906	1.844	127046	188023	7694	4855	291	180
20	0.3	2	0.16	2	0.953	359006	308123	13134	12215	491	310
30	0.61	3.58	0.27	3.688	12.015	719766	428223	18574	22775	691	440
40	1.05	6.23	0.36	6.375	27.063	1209326	548323	24014	36535	891	570
50	1.53	10.59	0.53	8.703	84.422	1827686	668423	29454	53495	1091	700
60	2.25	15.53	0.64	12.031	98.281	2574846	788523	34894	73655	1291	830
70	3.09	23.33	1	15.797	90.547	3450806	908623	40334	97015	1491	960

Table 1. Performance Evaluation For Datacenter Configuration Synthesis

Columns are as follows:

- Cust: Number of datacenter customer sites
- Gen QFF: Time in seconds to generate the QFF from requirements
- Setup K2: Time in seconds to initialize Kodkod
- K2QFF: Time in seconds to set up QFF in Kodkod
- K2 trans: Time in seconds for Kodkod to translate QFF into Boolean constraint
- K2 Solve: Time in seconds for ZChaff to solve Boolean constraint
- p: Number of Boolean variable occurrences in the CNF
- cnf: Number of clauses in the CNF
- primary: The number of distinct Boolean variables in the CNF
- |QFF|: Number of constraints in the QFF

- #Vars: Number of configuration variables in the QFF
- |DB|: Number of tuples in the configuration database.

6.3 Performance Evaluation for Address Assignment

For a separate network, we enforced just the IP addressing requirements of Section 2 together with requirements of the form subnet(L) specifying which interfaces belonged to the same subnet. The configuration database only had ipAddress(Host, Interface, Address, Mask) tuples with Address and Mask as variables. The results are summarized in Table 2.

Subnets	Gen QFF sec	Setup K2 sec	K2QFF sec	K2 Trans sec	K2 Solve sec	p	cnf	primary	|QFF|	#Vars	|DB|
50	0.03	1.09	0.08	1.843	1.907	216062	584046	2664	2589	143	72
100	0.16	2.31	0.16	7.968	43.875	856376	2323146	6290	10281	339	170
150	0.36	5.41	0.25	18.735	750.187	1915317	5204372	9472	22932	511	256
200	0.64	10.89	0.44	NA	NA	NA	NA	NA	40602	683	342

Table 2. Performance Evaluation For Greenfield Address Assignment

The largest network that could be addressed contained 150 subnets and 256 interfaces, as shown in the columns on the extreme left and right. The total time taken was 12.9 minutes. For 200 subnets, the Kodkod process did not terminate even after two hours. In all cases, the time taken to compute the QFF in Prolog was a small fraction of the overall time. The number of constraints grows as the square of the number of subnets because of the requirement that no two subnets overlap.

However, the performance is far superior for the common real-world scenario where most of the configuration is fixed and only a relatively small part of it needs reconfiguration. We emulated this scenario by starting with a fully instantiated configuration database but changing the IP address and mask for interfaces in 50 subnets to variables. As Table 2 shows, even with 463 subnets containing 830 interfaces, values of variables were found in 33 seconds. This illustrates the benefit of the partial evaluator. It determined that of the 463 subnets, 413 were already consistent with requirements, therefore did not include constraints for these in the QFF submitted to Kodkod. Thus, it drastically reduced the size of the QFF.

Subnets	Gen QFF sec	Setup K2 sec	K2QFF sec	K2 Trans sec	K2 Solve sec	p	cnf	primary	|QFF|	#Vars	|DB|
50	0.05	1.39	0.06	3.16	2.05	216,062	584,046	2,664	2,589	143	72
100	0.19	1.58	0.14	2.95	1.94	227,212	601,946	2,664	7,589	143	170
150	0.42	2.27	0.20	3.84	3.02	237,262	617,096	2,664	12,589	143	256
200	0.72	3.22	0.23	4.83	1.97	247,262	632,096	2,664	17,589	143	342
300	1.77	5.95	0.36	6.50	3.73	267,312	662,246	2,664	27,589	143	518
463	4.13	12.91	0.53	11.58	3.89	299,912	711,146	2,664	43,889	143	830

Table 3. Performance Evaluation For Incremental Address Assignment

7 Relationship With Previous Work

Kodkod itself contains a partial evaluator for constraints expressible in its first-order relational logic. It allows one to specify partial information about a relation as a set of tuples in that relation. It uses this information to ensure that a constraint that can be evaluated from partial information is not represented in the Boolean constraint that it eventually generates. Thus, the size of Boolean constraints is greatly reduced. However, often there are more efficient ways of evaluating a constraint than Kodkod's algorithms for its general relational logic. For example, while integer constraints are evaluated with a small number of machine operations in a programming language, they require a much large number of operations in Kodkod. This is because Kodkod represents integers as bitsets. ConfigAssure performs constraint evaluation outside of Kodkod and therefore its speed of partial evaluation is much higher. Of course, ConfigAssure's QFFs, although adequate for configuration requirements, are less expressive than Kodkod's constraints. For example, values of ConfigAssure variables can only be integers whereas those of Kodkod's object attributes can be arbitrary relations. Finally, we are investigating how to have QFFs benefit from Kodkod's partial evaluation. For example, the constraint `x=1` could be specified as Kodkod partial information rather than explicitly as it is now.

The concept of a Requirement Solver was first proposed in [11]. It was shown how fundamental configuration management problems could be formalized in Alloy [1]. Alloy is a first-order logic model finder but with a simple syntax and user interface. However, the Alloy formalization of the Requirement Solver did not contain a partial-evaluation phase. Neither did that implementation of Alloy have Kodkod's partial evaluation optimization. Thus, the Solver did not scale to networks of realistic size.

As mentioned in Section 1, the declarative nature of the Requirement Solver provides the important advantage of compositionality. This is critical for resolving the inherent tension between security and functionality. Security is about preventing bad behavior whereas functionality is about enabling good. Incorrect resolution of this tension can either allow an adversary access to services he shouldn't be able to, or equally undesirably, disallow a legitimate user from accessing services he should be able to. By representing both security and functionality requirements as constraints, and solving these, the tension is automatically resolved where possible. Procedural approaches for resolving the above tension do not scale beyond a few rules. Avoiding conflicts between rules for enforcing security and functionality is tantamount to replicating the capabilities of powerful model-finders and SAT solvers.

A closely related system is MulVAL [13]. It expresses requirements for adversary's success in Datalog, a subset of Prolog. If checks whether such a requirement is true and if so, analyzes the associated proof to compute what configurations to change at minimum cost to block the adversary's success. It does so by creating an attack graph, transforming it into a Boolean constraint and solving it with a min-cost SAT solver [17]. However, MulVAL does not synthesize configurations from high-level requirements. An idea that seems worthwhile exploring is combining the two methods of identifying which configurations to change. The first is MulVAL's method of analyzing a Datalog proof. The second is ConfigAssure's method of analyzing the proof of unsolvability.

The configuration error diagnosis problem solved by ConfigAssure is somewhat different from that solved by other systems such as [14] and [15]. These check whether a requirement is true of a fully instantiated configuration database, i.e. one *without* configuration variables. If not, they try to output a counterexample. For such a database, ConfigAssure's `eval` will return true or false but no counterexample. For its diagnosis to be interesting, the configuration database needs to contain at least one variable. If configuration parameters whose values could contribute to falsehood can be identified in advance, these can be replaced by variables in the database. The fact that these parameters are fixed to their current values is emulated by adding equality constraints as described in Section 3.2. Then, ConfigAssure would identify which variable is responsible for falsehood. The set of configurations whose values are to be made into variables does not have to be precisely identified.

ConfigAssure can be combined with other diagnosis systems to automate counterexample generation and repair as follows: replace a counterexample to a requirement with variables in the database; use ConfigAssure to compute new values so that *all* end-to-end requirements become simultaneously true, not just the violated ones.

Finally, a number of prior efforts share the same vision of top down network design as this work. Much of the early efforts in this direction focused on optimizing the performance of particular protocols, e.g., tuning of OSPF weights for traffic engineering purposes [21]. A recent work [22] took a comprehensive look at enterprise network design and investigated ways to systematically derive device-level VLAN and packet filter configuration parameters from network-wide operational requirements.

8 Summary And Future Directions

This paper identified fundamental configuration problems that need to be solved to automatically bridge the gap between end-to-end requirements and configurations. These are specification, configuration synthesis, configuration error diagnosis and configuration error repair. The paper presented a system called ConfigAssure for solving these problems. Central to ConfigAssure is a Requirement Solver that takes as input a requirement and a configuration database with variables. It outputs values of variables that make the requirement true of the database, when instantiated with these values. If unable to do so, it outputs a proof of unsolvability. The Requirement Solver is used in different ways to solve the above problems. The Requirement Solver exploits the power of Kodkod, a SAT-based model-finder. A new method of encoding requirements allows scalability to problems of realistic scale. The idea is to transform a requirement into an equivalent QFF that truly requires the power of model-finding. A QFF is a Boolean combination of simple arithmetic constraints on integers.

The novelty of ConfigAssure is its overall framework consisting of the following ideas: that a database can contain configuration variables, that a requirement can be naturally specified via a partial evaluator that transforms it into an equivalent QFF, that a QFF is efficiently solved by Kodkod, and that a QFF simplifies diagnosis and repair algorithms.

ConfigAssure is illustrated in the context of a realistic, secure and fault-tolerant datacenter. Based on performance evaluation for this datacenter, ConfigAssure seems viable for solving configuration problems of realistic scale. ConfigAssure is under trial at a major enterprise to assist in rearchitecting its network.

While it seems feasible to have ConfigAssure generate “greenfield” configurations for infrastructure of realistic size, it is particularly well-suited for incremental configuration even for large configuration databases. Typically, most of the configuration in an infrastructure is fixed. When a change occurs e.g., a new site is added, only a small number of configuration parameters need to change to reenforce end-to-end requirements. The partial evaluator will generate only those constraints that are related to these parameters. It will evaluate away the requirements for all configurations that cannot change.

Besides the two directions to investigate outlined in the previous section, there are several others. The first is the creation of a Requirement Library for specific domains. Requirements in it will be efficiently implemented with algorithms that exploit strengths of Kodkod and SAT solvers. Users can compose these to create complex requirements. A good heuristic for identifying these is to identify logical structures associated with different protocols and distributed algorithms that are set up via configuration [20]. The requirements in Section 3.1 are examples of such a library for IP networks.

The second direction to investigate is choice of which constraint to remove from the proof of unsolvability in the repair algorithm. Different choices may have different impact on efficiency and cost.

The third direction to investigate is the use of Satisfiability Modulo Theories solvers e.g., [19] for solving QFFs. An SMT constraint is a Boolean constraint in which some variables are replaced by constraints in a theory. A QFF is an SMT constraint in which the theory is simple arithmetic. The theory is kept simple to allow efficient solution by pure SAT solvers underlying Kodkod. SMT solvers may allow efficient solution of a richer class of QFFs e.g., with linear arithmetic constraints.

The last direction to investigate is safe reconfiguration planning. Even if the final infrastructure configuration is known, in general, all components cannot be concurrently reconfigured. Thus, one needs to compute the order in which to reconfigure so that in the transition, security breaches do not arise and mission critical services are not disrupted. While this problem is expressible as a SAT problem [12], how to scale it up to realistic size seems to be an open question. We are investigating easier but useful versions of this problem.

Acknowledgements

We thank Professor Daniel Jackson, Emina Torlak and Felix Chang at MIT, Professor Simon Ou at Kansas State University, Professor Ehab Al-Shaer at DePaul University and Professor Geoffrey Xie at Naval Post Graduate School for very helpful comments on this paper.

Biographies

Sanjai Narain is a Senior Research Scientist at Telcordia Technologies, Inc. His current research is in principles of secure, survivable infrastructure design. His formal training is in mathematical logic and programming languages. He received a B.Tech. in Electrical Engineering from Indian Institute of Technology, New Delhi, in 1979, an M.S. in Computer Science from Syracuse University in 1981, and a Ph.D. in Computer Science from UCLA in 1988.

Gary Levin is a Senior Research Scientist at Telcordia Technologies. His expertise is in mathematical logic, programming languages and design of secure, survivable infrastructure. He received a B.S. in Computer Science from University of Delaware in 1975 and a Ph.D. in Computer Science from Cornell University in 1980.

Vikram Kaul is a Senior Research Scientist at Telcordia Technologies. His expertise is in software development and design of secure, survivable infrastructure. He received a B.E. (Hons) in Electrical & Electronics Engineering from Birla Institute of Technology and Science, Pilani in 1997 and an M.S. from The Wireless Information Networks Laboratory at Rutgers University in 2000.

Sharad Malik is George Van Ness Lothrop Professor of Electrical Engineering at Princeton University. His current research is in electronic design automation. The ZChaff SAT Solver developed by his team is widely used in research and industry. He received a B. Tech. in Electrical Engineering from Indian Institute of Technology, New Delhi, in 1985, and an M.S. and Ph.D. in Computer Science from University of California, Berkeley in 1987 and 1990 respectively.

References

1. Alloy: http://alloy.mit.edu/
2. ZChaff: http://www.princeton.edu/~chaff/
3. Kodkod: http://web.mit.edu/emina/www/kodkod.html
4. E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. *Tools and Algorithms for Construction and Analysis of Systems (TACAS '07)*. Braga, Portugal, March 2007. (PDF) (slides)
5. SWI-Prolog: http://www.swi-prolog.org/
6. SWI-Prolog-JPL: http://www.swi-prolog.org/packages/jpl/prolog_api/overview.html
7. Security and business continuity solutions from British Telecom. http://www.btglobalservices.com/business/global/en/products/docs/28154_219475secur_bro_single.pdf
8. B. Lampson. *Computer Security in Real World*. Annual Computer Security Applications Conference, 2000. http://research.microsoft.com/Lampson/64-SecurityInRealWorld/Acrobat.pdf
9. D. Oppenheimer, A. Ganapathy, D. Patterson. *Why do Internet Services Fail and What Can Be Done About It?* 4th USENIX Symposium on Internet Technologies and Systems, 2003. http://roc.cs.berkeley.edu/papers/usits03.pdf
10. John Schwartz. *Who Needs Hackers?* New York Times, September 12, 2007. http://www.nytimes.com/2007/09/12/technology/techspecial/12threat.html.
11. Sanjai Narain. *Network Configuration Management via Model-Finding. Proceedings of USENIX Large Installation System Administration (LISA) Conference*, San Diego, CA, 2005.
12. B. Selman, H. Kautz, "Planning As Satisfiability," *Proceedings of ECAI-92*, http://www.cs.cornell.edu/selman/papers/pdf/92.ecai.satplan.pdf.
13. J. Homer, Xinming Ou, Miles McQueen. From attack graphs to automated configuration management --- an iterative approach Technical Report 2008-1, Computer Science Department, Kansas State University.
14. Ehab Al-Shaer and Hazem Hamed . Modeling and Management of Firewall Policies. *IEEE Transactions on Network and Service Management*, Volume 1-1, April 2004.
15. Telcordia IPAssure. http://www.argreenhouse.com/papers/narain/TelcordiaIPAssure.pdf.
16. Y.Mahajan, Z. Fu, Sharad Malik. Zchaff2004, An Efficient SAT Solver. *Proceedings of 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2004.
17. Z. Fu, Sharad Malik. Solving the Minimum-Cost Satisfiability Problem using Branch and Bound Search. *Proceedings of IEEE/ACM International Conference on Computer-Aided Design ICCAD*, 2006
18. Bratko, Ivan. *Prolog Programming for Artificial Intelligence*. Addison-Wesley International Computer Science Series, 1990
19. Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515-518. Springer, July 2004. Boston, Massachusetts.

20. Sanjai Narain, Thanh Cheng, Brian Coan, Vikram Kaul, Kirthika Parmeswaran, William Stephens. Building Autonomic Systems via Configuration. *Proceedings of AMS Autonomic Computing Workshop*, Seattle, WA, 2003.

21. A. Feldmann and J. Rexford. IP network Configuration for Intradomain Traffic Engineering. *IEEE Network Magazine*, Sept. 2001

22. Y. Sung, S. Rao, G. Xie, and D. Maltz. Systematic Configurator Design for Enterprise Networks. Technical Report, TR ECE 08-07, Department of ECE, Purdue University, May 2008.