Constraint Solving For Network Configuration

Lecture 4

CS 598D, Spring 2010
Princeton University

Sanjai Narain
narain@research.telcordia.com
908 337 3636

Note On Negation-As-Failure

```
not(F):-F,!,fail.
not(F).
```

This is a powerful and well-used feature

But, this is not true negation. The query

```
?-member(X, [1,2]), not(X=1)
```

succeeds with X=2 but the equivalent

```
?-not(X=1), member(X, [1,2])
```

fails

- Constraint solvers handle true negation correctly

The story so far

- We have seen how Prolog can be used:
  - To analyze ad hoc configuration language files
  - To evaluate whether requirements are true of configurations
  - As a metalevel language to convert to other forms such as Graphviz dot files
- We motivated the need for constraint solvers for firewall verification
  - Used Prolog as a metalevel language to *automatically* generate constraints and exploit the power of modern constraint solvers

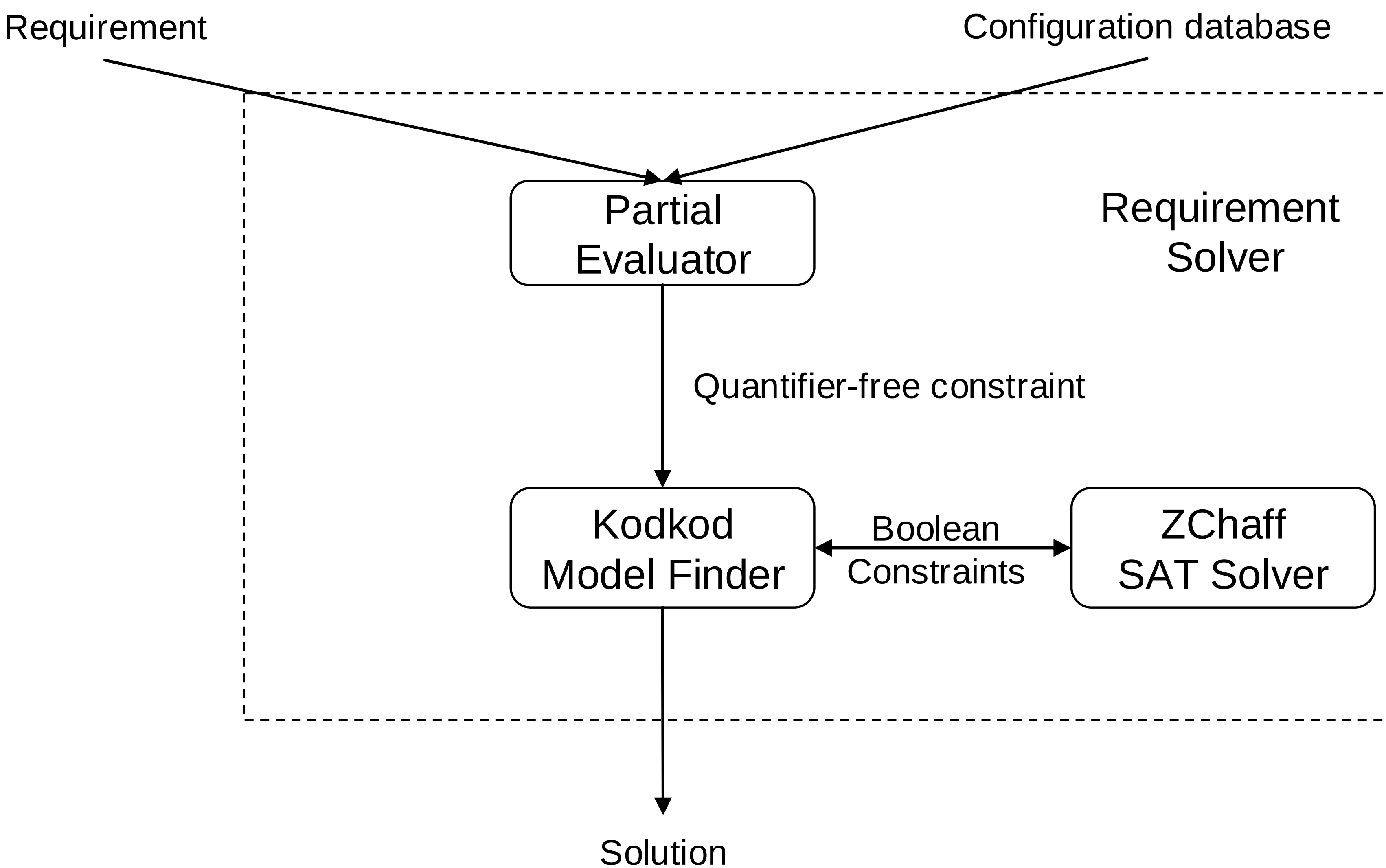
Today

- Discuss the use of constraint solvers for solving configuration problems:
  - Synthesis
  - Diagnosis
  - Repair
  - Repair at minimum cost

- Again use Prolog as a metalevel language to generate constraints and call a constraint solver

- Reconfiguration planning will be discussed later in semester

ConfigAssure System Architecture



What Is the Constraint Language?

- Arithmetic quantifier-free form (QFF)

- A QFF = Boolean combination of:
  - x op y
  - contained(a, m, b, n)

  where x, y, a, m, b, n are integer variables or constants and op is =,<,>,<=,>= and

  contained(a, m, b, n) means the address range represented by the IP address a and mask m contains that represented by address b and mask n

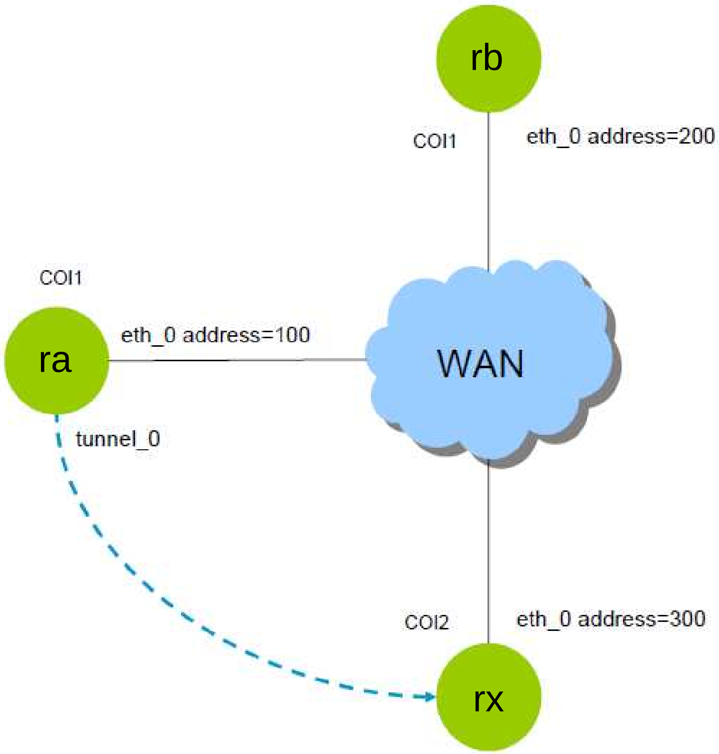
- It is a good intermediary between full first-order logic and Boolean. Adequate for networking since most configuration variables are addresses

- It simplifies design of algorithms for configuration error diagnosis, repair and reconfiguration planning

- It is efficiently compiled into Boolean by Kodkod, the Java API underlying Alloy

- It is directly solved by SMT solvers. These solvers also have other advantages.

Prolog Specification of VPN Requirements




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

Specification

```
good:-gre_connectivity(ra, rb).
bad:-gre_tunnel(ra, rx).
bad:-route_available(ra, rx).

gre_connectivity(RX, RY):-
   gre_tunnel(RX, RY),
   route_available(RX, RY).

gre_tunnel(RX, RY):-
   gre(RX, _, _, RemoteAddr),
   ipAddress(RY, _, RemoteAddr, _).

route_available(RX, RY):-
   static_route(RX, Dest, _, _),
   ipAddress(RY, _, RemotePhysical, _),
   Dest=RemotePhysical.
```

Evaluating Requirements

```
?- good.
false
?- bad.
true
```

With this specification, Prolog will not tell you new configurations such that good $\wedge$ not(bad)

Solving Configuration Problems With Constraint Solver

- Define a constraint Req on configuration variables x_1 .. x_k such that (good $\wedge$ not(bad))
- For synthesis: solve Req and take the result
- Let InitVal be the constraint ($x_1=c_1 \wedge .. \wedge x_k=c_k$) where $c_1,\ldots,c_k$ are current values of variables
- For diagnosis: solve (Req $\wedge$ InitVal). Since Req is false for InitVal, solver will return an unsat-core. Any constraint x=c in it is a root cause
- For repair: from InitVal, delete a constraint x=c in unsat-core and reattempt solution to (Req $\wedge$ InitVal)
- For repair with cost under T:
  - Let the cost of changing x_i from c_i to a new value be σ_i.
  - Define new variable cx_i representing the cost of changing x_i
  - Add the constraint (if $x_i=c_i$ then $cx_i = 0$ else $cx_i = \sigma_i$) to Req
  - Let TotalCost = $cx_1 + .. + cx_k$
  - Solve (Req $\wedge$ TotalCost<T)
- Use binary search over [0, T] to find repaired configuration at minimum cost

How To Compute Req and InitVal?

Configuration Database With Values Replaced By Configuration (not Prolog) Variables

static_route(ra, **dest**, **mask**, 400).
gre(ra, tunnel_0, **gre_a_local**, **gre_a_remote**).
ipAddress(ra, eth_0, **ra_addr**, 0).
ipAddress(rb, eth_0, **rb_addr**, 0).
ipAddress(rx, eth_0, **rx_addr**, 0).

```
eval(initVal, Cond):-
        Cond=and_each(
                [dest=300,
                 mask=0,
                 gre_a_local=100,
                 gre_a_remote=300,
                 ra_addr=100,
                 rb_addr=200,
                 rx_addr=300])
```

Metalevel Version of Specification

eval(X, Y) means Y is the QFF representation of requirement X

```
eval(good, Cond):-
   eval(gre_connectivity(ra, rb), Cond).

eval(gre_connectivity(X, Y), and(C1, C2)):-
   eval(gre_tunnel(X, Y), C1),
   eval(route_available(X, Y), C2).

eval(gre_tunnel(RX, RY), and(LocalAddr=Addrx,
        RemoteAddr=Addry)):-
   gre(RX, _, LocalAddr, RemoteAddr),
   ipAddress(RX, _, Addrx, _),
   ipAddress(RY, _, Addry, _).

eval(route_available(RX, RY), Dest=RemotePhysical):-
   static_route(RX, Dest, Mask, _),
   ipAddress(RY, _, RemotePhysical, _).
```

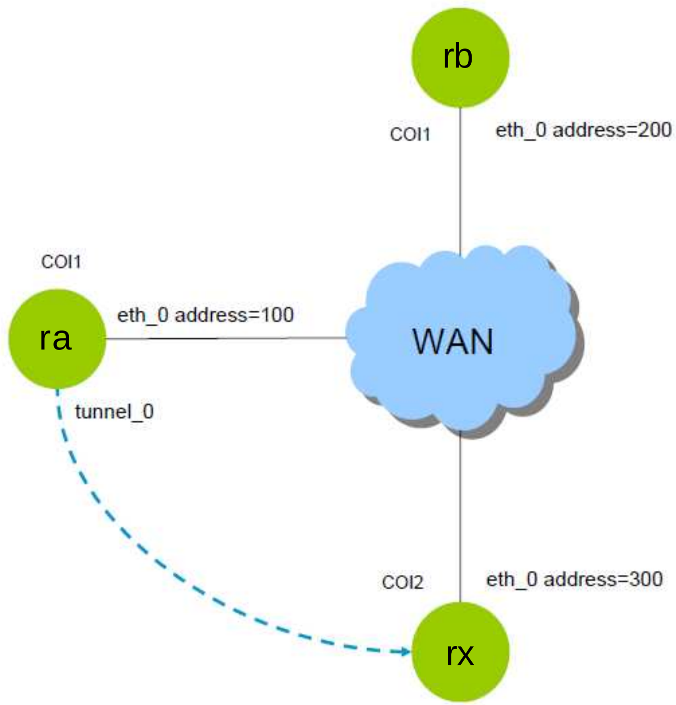
Synthesis

```
?- eval(and(good, not(bad)), C)

C=
      and[
       gre_a_local=ra_addr
       gre_a_remote=rb_addr
       dest=rb_addr
       not [
        or[
         and[
          gre_a_local=ra_addr
          gre_a_remote=rx_addr
         ]
         dest=rx_addr
        ]
       ]
      ]
```

```
?- solve(and(good, not(bad)), C).

C = [
     ra_addr=1,
     rb_addr=2,
     rx_addr=3,
     gre_a_local=1,
     gre_a_remote=2,
     dest=2]
```



Diagnosis And Repair

```
?- solve(and(initVal, and(good, not(bad))), C)

Unsat:
C  =  [gre_a_remote=rb_addr,
       gre_a_remote=300,
       rb_addr=200] .

initVal1 = initVal \ {gre_a_remote=300}

?-solve(and(initVal1, and(good, not(bad))), C).

Unsat:
C=[dest=rb_addr, dest=300, rb_addr=200] .

initVal2 = initVal1 \ {dest=300}
```

```
?- solve(and(initVal2, and(good, not(bad))), C).

Sat:
C=   [ra_addr=100,
      rb_addr=200,
      rx_addr=300,
      gre_a_local=100,
      gre_a_remote=200,
      dest=200,
      mask=0]
```

Repair At Minimum Cost

The cost of changing dest is 4 and 1 for all other variables

```
eval(topReq(MaxCost), C):-
   eval(good, G),
   eval(bad, B),
   eval(addr_unique, AU),
   add_costs([dest, mask, gre_a_local, gre_a_remote, ra_addr, rb_addr, rx_addr], TotalCost),
   and_each([G, not(B), AU, CostC, TotalCost<MaxCost], C),
   cost_constraints([dest,  mask, gre_a_local,  gre_a_remote, ra_addr,  rb_addr,   rx_addr], CostC).
```

	Initial	MaxCost=10	MaxCost=5	MaxCost=3	MaxCost=2
dest	300	200	300	300	unsat
gre_a_local	100	100	301	100	
gre_a_remote	300	200	300	300	
ra_addr	100	100	301	100	
rb_addr	200	200	300	300	
rx_addr	300	300	302	301	

QFF For topReq(10)
Note "implies" constraints at end constraining cost of change

```
and[
 gre_a_local=ra_addr
 gre_a_remote=rb_addr
 dest=rb_addr
 not [
  or[
   and[
    gre_a_local=ra_addr
    gre_a_remote=rx_addr
   ]
   dest=rx_addr
  ]
 ]
 not [
  ra_addr=rb_addr
 ]
 not [
  rb_addr=rx_addr
 ]
 not [
  rx_addr=ra_addr
 ]
 implies(dest=300, cdest=0)
 implies(not(dest=300), cdest=4)
 implies(mask=0, cmask=0)
 implies(not(mask=0), cmask=1)
 implies(gre_a_local=100, cgre_a_local=0)
 implies(not(gre_a_local=100), cgre_a_local=1)
 implies(gre_a_remote=300, cgre_a_remote=0)
 implies(not(gre_a_remote=300), cgre_a_remote=1)
 implies(ra_addr=100, cra_addr=0)
 implies(not(ra_addr=100), cra_addr=1)
 implies(rb_addr=200, crb_addr=0)
 implies(not(rb_addr=200), crb_addr=1)
 implies(rx_addr=300, crx_addr=0)
 implies(not(rx_addr=300), crx_addr=1)
 cdest+ (cmask+ (cgre_a_local+ (cgre_a_remote+ (cra_addr+ (crb_addr+ (crx_addr+0))))))<10
```

Next Lecture

- Building partial evaluation into eval to reduce the size of generated QFF
- Solving the variable-reference problem: how to systematically refer to thousands of variables?
- Projects on configuration