Motivating Constraint Solving For Networking

Lecture 3

CS 598D, Spring 2010
Princeton University

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

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

Today

- We outline other features of Prolog:
  - Unification, cut and negation as failure
- Discuss a problem for which Prolog is not appropriate
- Show how that problem can be solved with constraint solvers such as Kodkod (via SAT) and SMT
- The problem is that of firewall verification

Unification

It is Prolog’s parameter passing Mechanism

It was invented by J. A. Robinson in his resolution inference procedure of 1965

A substitution σ is a *unifier* for expressions φ and ψ if and only if $\varphi\sigma=\psi\sigma$.

$$p(X,Y)\{X\leftarrow a,Y\leftarrow b,V\leftarrow b\}=p(a,b)$$
$$p(a,V)\{X\leftarrow a,Y\leftarrow b,V\leftarrow b\}=p(a,b)$$

The two expressions below are not unifiable

$$p(X,X)$$
$$p(a,b)$$

Non-Uniqueness of Unification

Unifier 1:

$$p(X,Y)\{X \leftarrow a, Y \leftarrow b, V \leftarrow b\}=p(a,b)$$
$$p(a,V)\{X \leftarrow a, Y \leftarrow b, V \leftarrow b\}=p(a,b)$$

Unifier 2:

$$p(X,Y)\{X \leftarrow a, Y \leftarrow V\}=p(a,V)$$
$$p(a,V)\{X \leftarrow a, Y \leftarrow V\}=p(a,V)$$

A substitution σ is a *most general unifier* (*mgu*) of two expressions if and only if it is as general as or more general than any other unifier.

If two expressions are unifiable, then they have an mgu that is unique up to variable permutation.

$$p(X,Y)\{X \leftarrow a, Y \leftarrow V\}=p(a,V)$$
$$p(a,V)\{X \leftarrow a, Y \leftarrow V\}=p(a,V)$$

$$p(X,Y)\{X \leftarrow a, V \leftarrow Y\}=p(a,Y)$$
$$p(a,V)\{X \leftarrow a, V \leftarrow Y\}=p(a,Y)$$

Prolog’s Cut (!) Operator

Without Cut

```
insert(X,[],[X]).
insert(X, [Y|Sorted], [X,Y|Sorted]) :-
      X=<Y.
insert(X, [Y|Sorted], [Y|Sorted1]) :-
      X > Y,
      insert(X, Sorted, Sorted1).
```

With Cut

```
insert(X,[],[X]).
insert(X, [Y|Sorted], [X,Y|Sorted]) :-
      X=<Y, !.
insert(X, [Y|Sorted], [Y|Sorted1]) :-
      ~~X > Y,~~
      insert(X, Sorted, Sorted1).
```

Application of Cut: Negation As Failure

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

With the database

```
p(1).
p(2).
```

The query

```
?- p(X), not(X=1)
```

will return X=2

<u>This is a powerful and well-used feature</u>

But, this is not true negation. The query

```
?-member(X, [1,2,3]), not(p(X))
```

succeeds as expected, but the equivalent

?-not(p(X)), member(X, [1,2,3]) fails!

Firewall Policy Verification

- A firewall is a sequence of rules, each with the following IP packet fields:
  - SAL: lower bound of source address
  - SAU: upper bound of source address
  - DAL: lower bound of destination address
  - DAU: upper bound of destination address
  - Permission: permit or deny
- Real firewalls contain additional fields for source and destination ports and protocols
- Example
  - 1, 3, 5, 7, deny
  - 9, 11, 13, 15, permit
- Rules are tried from top to bottom.
  - A rule may permit a packet but if an earlier rule denies it, packet is denied
  - If no rules apply, the packet is denied

Firewall Verification Questions

- Permission: Is an IP packet permitted by a firewall?
- Inclusion: Is every packet permitted by a firewall permitted by another? If not, produce a counter example
- Equivalence: Given two firewalls, do they permit exactly the same packets?
- Redundancy: Does removing a rule change the set of permitted packets?

Specifying "permitted" in Prolog

Specifying firewalls as Prolog facts

```
fw(fw1, [[1, 3, 5, 7, deny],
         [2, 2, 6, 6, permit],
         [9, 11, 13, 15, permit]]).
```

Specifying permitted relation: A packet with source address SA and destination address DA is permitted by firewall FW

```
permitted(FW, SA, DA):-
   fw(FW, Rules),
   permitted_by_rules(SA, DA, Rules).

permitted_by_rules(SA, DA, [Rule | Rest]):-
   matches_rule(SA, DA, Rule),
   Rule=[_,_,_,_,permit].

permitted_by_rules(SA, DA, [Rule | Rest]):-
   \+matches_rule(SA, DA, Rule),
   permitted_by_rules(SA, DA, Rest).

matches_rule(SA, DA, [SAL, SAU, DAL, DAU, _]):-
   SAL=<SA, SA=<SAU, DAL=<DA, DA=<DAU.
```

?- permitted(fw1, 11, 15) → true

?- permitted(fw1, 2, 6) → false

Even though 2, 6 is permitted by second rule, it matches the first, so it is denied

?-permitted(fw1, 100, 100) → false

100, 100 matches no rules

?-permitted(fw1, S, D).

ERROR: =</2: Arguments are not sufficiently instantiated

Definition of permitted assumes S and D are already known. To compute S and D use a generator of integers.

```
an_address_pair(Lower, Upper, S, D):-
      between(Lower, Upper, S),
      between(Lower, Upper, D).
```

Specifying Inclusion in Prolog

To specify that SA, DA in the range Lower, Upper, is permitted by FW1 but not FW2, use

```
permitted_by_first_but_not_second(FW1, FW2, Lower, Upper, SA, DA):-
    an_address_pair(Lower, Upper, SA, DA),
    permitted(FW1, SA, DA),
    \+permitted(FW2, SA, DA).
```

To check inclusion of FW1 in FW2 for addresses between Lower and Upper, check that

```
permitted_by_first_but_not_second(FW1, FW2, Lower, Upper, SA, DA) fails
```

For firewalls

```
fw(fw1, [[1, 3, 5, 7, deny], [2, 2, 6, 6, permit], [9, 11, 13, 15, permit]]).
fw(fw3, [[1, 25, 1, 25, permit]]).

permitted_by_first_but_not_second(fw1, fw3, 1, 100, SA, DA) fails but
permitted_by_first_but_not_second(fw3, fw1, 1, 100, SA, DA) has many solutions e.g., SA=1, DA=1
```

Using Constraints For Firewall Verification

- But if we want to check inclusion for all addresses between 0 and 2^{32}-1, then
  - permitted_by_first_but_not_second(fw1, fw3, 0, 2^{32}-1, SA, DA) will iterate over 2^{64} possibilities
- Is there another approach?
- Represent a firewall F as a constraint permitted(F) on variables src, dest
- Then:
  - src, dst is permitted by F provided permitted(F) is true for src, dst
  - F1 is included in F2 if there is no solution to the constraint permitted(F1) $\wedge \neg$permitted(F2)
  - F1 and F2 are equivalent if F1 and F2 include each other
  - To check whether a rule in F is redundant, delete it and check equivalence of the remainder and F
- This is a general way of reasoning about access-control policies

Representing Firewall As Constraint

The firewall

```
fw(fw4, [[1, 3, 5, 7, deny], [9, 11, 13, 15, permit]]).
```

is represented by the constraint at right.

```
or[
  and[
    and([1<=src, src<=3, 5<=dst, dst<=7])
    false
  ]
  and[
    not [
      and([1<=src, src<=3, 5<=dst, dst<=7])
    ]
    or[
      and[
        and([9<=src, src<=11, 13<=dst, dst<=15])
      ]
      and[
        not [
          and([9<=src, src<=11, 13<=dst, dst<=15])
        ]
        false
      ]
    ]
  ]
]
```

Constraint Generation Algorithm: Linear in Number of Rules

The constraint on src, dst for the empty list of firewall rules is false

The constraint on src, dst for [Rule | Rest] is a disjunct of:
    the constraint that src,dst match Rule's address range and that Rule's permission is permit
    the constraint that src,dst do not match Rule's address range and the constraint on src, dst for Rest

The Prolog program is:

```
eval(matchesAddr([SAL, SAU, DAL, DAU,Perm]), C):-
    and_each([SAL<=src, src<=SAU, DAL<=dst, dst<=DAU], C).

eval(permitting([_,_,_,_,permit]), true).
eval(permitting([_,_,_,_,deny]), false).

eval(permitted_by_rules([]), false).
eval(permitted_by_rules([Rule | Rest]), C) :-
    eval(matchesAddr(Rule), MatchesThisRule),
    eval(permitting(Rule), Action),
    C=or(and(MatchesThisRule, Action),
          and(not(MatchesThisRule), PermittedRest)),
    eval(permitted_by_rules(Rest), PermittedRest).

eval(permitted(FW), C):-
    fw(FW, Rules),
    eval(permitted_by_rules(Rules), C).
```

Verifying Firewall Inclusion With Yices

For the firewalls

```
fw(fw1, [[1, 3, 5, 7, deny], [2,2,6,6, permit], [9, 11, 13, 15, permit]]).
fw(fw5, [[1, 1000000, 1, 1000000, permit]])
```

The query:

```
?-eval(permitted(fw1), C1),
  eval(permitted(fw5), C2),
  solve_smt(and(C1, not(C2)), Result)
```

returns Result=unsat in about 1 second

The query:

```
?-eval(permitted(fw5), C1),
  eval(permitted(fw1), C2),
  solve_smt(and(C1, not(C2)), Result).
```

returns Result = [src=12, dst=16] in about 1 second

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 (a, m) contains that represented by (b, n)

- It is a good intermediary between Alloy and Boolean.

- It is adequate for networking since most configuration variables are addresses

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

- Can also use SMT solvers to solve QFFs. These also have other advantages.

Next Lecture

- How to specify complex requirements with QFFs and solve the following problems:
  - Configuration synthesis
  - Configuration error diagnosis
  - Configuration error repair; at minimum cost
  - Reconfiguration planning