LPOP2018 XSB Position Paper

David S. Warren
Department of Computer Science
Stony Brook University, New York, USA

May 30, 2018

1 Introduction

In this position paper we first describe a classic logic programming approach to the solution of (a portion) of the challenge problem involving RBAC. We use the XSB Tabled Prolog Language and system [3], with ideas from Transaction Logic [1]. Then we discuss efficiency and scalability issues for this implementation. Finally we discuss issues that involve what would be required to use such an implementation in a real-world application requiring RBAC functionality.

2 RBAC Challenge Problem in XSB

We describe our solution to the challenge problem. We use a module, `prolog_db`, that was recently added to the XSB system that allows a Prolog database (i.e., a set of clauses) to be represented as a ground term, which we call a PDB. A number of operations are provided to access and update PDB's, the salient ones here being a) `assert_in_db(+Clause,+PDB0,-PDB)`, which adds a clause to a PDB to generate a new PDB, b) `retract_in_db(+Clause,+PDB0,-PDB)`, which deletes a clause from a PDB to generate a new PDB, and c) `call_in_db(?Goal,+PDB)`, which calls a goal in a given PDB, returning instances that are true in the given PDB. For this RBAC application the clauses in a PDB will always be ground facts. We use the Prolog Definite Clause Grammar (DCG) notation for writing these programs, since it supports a convenient notation for writing rules that define state transformations. The (implicit) state is always a PDB.

The description of the RBAC challenge problem is given at `https://drive.google.com/file/d/1q9W15kI624TI6pEbh2IMPDw6X5_5MiW7/view`. The XSB program for the RBAC challenge problem (minus the two `MinRoleAssignment` functions in the Administrative component) is provided in the Appendix.

This is a relatively straightforward specification (and implementation) of the problem in classical logic programming. Since the RBAC database is represented explicitly as a term in Prolog, general Prolog backtracking restores earlier database states. So this makes post conditions, such as in `create_ssdSet`, trivial to implement: do the operation, and then check the post condition; if it fails, the system automatically backtracks to restore the initial database state.

Note also that the tabling is correct even as the PDBs change, since the appropriate PDBs, which are implicit in the DCG notation, are arguments to the tabled predicates. One might want to abolish the tables periodically if space becomes an problem.

2.1 Performance Issues

This implementation should be quite efficient as an XSB program. A PDB represents a set of clauses. The `prolog_db` module uses a trie data structure to store a PDB, with a variant of a radix tree at each branch point in the trie. This makes the representation canonical, in that a given set of clauses is represented by the same term, regardless of the sequence of asserts and retracts (_in_db) that generates that set. So all updates and accesses are done in log time. Also, the terms representing PDBs are ground and so can use "interned terms", also sometimes known as hash-consing, which are implemented in XSB [2]. Thus the terms are copied to a global store and uniquely stored; i.e. all common subterms are shared. Then the Prolog code passes around what are essentially pointers to tries in the global store. With this representation tabling involves only the constant-time copying of a "pointer" into and out of a table. Also equality comparison of two PDBs is simply a comparison of their pointers. The `GetRoles(Shortest)Plan` functions do an exhaustive search for plans, which in some cases could be expensive, but the tabling does provide help. The two `MinRoleAssignments` functions are not implemented because they seem to require constraint solving, which is not XSB's strength. An exhaustive search could be implemented directly in XSB but would be uninteresting.

2.2 Interfacing with the System Environment

So this seems to us to be a reasonably elegant solution to the formal RBAC problem as specified in (all but two functions of) the challenge. However, there is the question of how this code might really be used in a much larger system in which access control is only a small component. As described in the previous subsection, we don't think that the performance and scalability of the execution of the RBAC operations would present a problem. The more difficult issues, we believe, involve data persistence and concurrent usage. There are various potential solutions, but no single obvious one (at least to us.) And the potential solution seem to require procedural, more than logical, thinking and programming.

References

[1] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Comput. Sci.*, 133:205–265, October 1994.

[2] D. S. Warren. Interning ground terms in XSB. In *Proceedings of CICLOPS 2013*, August 2013. In conjunction with ICLP'2013.

[3] D. S. Warren, T. Swift, and K. F. Sagonas. The XSB programmer's manual, Version 2.7.1. Technical report, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York, 11794-4400, Mar 2007. The XSB System is available from xsb.sourceforge.net, and the system and manual is continually updated.

A RBAC Implementation in XSB

```
:- import assert_in_db/3, retractall_in_db/3, call_in_db/2, size_db/2, new_dbi/1
    from prolog_db.

%% rename update operations for clarity (and brevity)
add(Fact,DB0,DB) :- assert_in_db(Fact,DB0,DB).
del(Fact,DB0,DB) :- retractall_in_db(Fact,DB0,DB).

%% CORE RBAC
%% Make relation lookups into identity transactions (convenience)
%%  i.e., they return the exact database they receive.
users(User,D,D) :- call_in_db(users(User),D).
roles(Role,D,D) :- call_in_db(roles(Role),D).
perms(Perm,D,D) :- call_in_db(perms(Perm),D).
ur(User,Role,D,D) :- call_in_db(ur(User,Role),D).
pr(Perm,Role,D,D) :- call_in_db(pr(Perm,Role),D).

%% update functions
%% to add and delete users...
addUser(User) --> \+ users(User), add(users(User)).
deleteUser(User) --> users(User), \+ ur(User,_), del(users(User)).

%% to add and delete roles...
addRole(Role) --> \+ roles(Role), add(roles(Role)).
deleteRole(Role) --> roles(Role), \+ ur(_,Role), \+ pr(_,Role), del(roles(Role)).

%% to add and delete permissions...
addPerm(Perm) --> \+ perms(Perm), add(perms(Perm)).
deletePerm(Perm) --> perms(Perm), \+ pr(Perm,_), del(perms(Perm)).

%% to add and delete users in roles
addUR(User,Role) --> users(User), roles(Role), \+ ur(User,Role), add(ur(User,Role)).
deleteUR(User,Role) --> ur(User,Role), del(ur(User,Role)).

%% to add and delete permissions that roles have.
addPR(Perm,Role) --> perms(Perm), roles(Role), \+ pr(Perm,Role), add(pr(Perm,Role)).
deletePR(Perm,Role) --> pr(Perm,Role), del(pr(Perm,Role)).

%% simple rename as required
assignedRoles(User,Role) --> ur(User,Role).

%% define user permissions by joining user-role and role-permission
userPermissions(User,Perm) --> ur(User,Role), pr(Perm,Role).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% HIERARCHICAL RBAC (additions)

%% immediate subclass relation to identity transaction
%% (would be more efficient in other order..)
rh(RoleAsc,RoleDsc,D,D) :- call_in_db(rh(RoleAsc,RoleDsc),D).

%% update functions
%% add an inheritance fact if no loop is generated
addInheritance(RoleAsc,RoleDsc) -->
    roles(RoleAsc),roles(RoleDsc),
    add(rh(RoleAsc,RoleDsc)),
    \+ trans(RoleAsc,RoleDsc).

%% define transitive closure for inheritance, and loop checking
%% note that the db is a (hidden) parameter, so this is correct over updates
:- table trans/4.
trans(Dsc,Dsc) --> [].
trans(Dsc,Asc) --> trans(Dsc,Par),rh(Asc,Par).

%% remove an inheritance fact.
deleteInheritance(RoleAsc,RoleDsc) -->
    rh(RoleAsc,RoleDsc),
    del(rh(RoleAsc,RoleDsc)).

%% add rule to include inheritance when determining authorized roles
authorizedRoles(User,Role) -->
    trans(Role,ARole),
    assignedRoles(User,ARole).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% SSD

%% again make lookup operations into identity transactions, for convenience.
ssdCount(Name,Cnt,D,D) :- call_in_db(ssdCount(Name,Cnt),D).
ssdRole(Name,Role,D,D) :- call_in_db(ssdRole(Name,Role),D).

%% add ssdRole and check consistency
addSsdRoleMember(Name,Role) -->
    roles(Role),
    add(ssdRole(Name,Role)),
    ssdConsistent(Name).

%% delete ssdRole
deleteSsdRoleMember(Name,Role) -->
    ssdRole(Name,Role),
    del(ssdRole(Name,Role)).

%% set SSD cardinality
setSsdSetCardinality(Name,Cnt) -->
    (ssdCount(Name,OCnt)            % if already has a cardinality
     ->({OCnt \== Cnt}             % if cardinality is changed
         -> del(ssdCount(Name,OCnt)),
            add(ssdCount(Name,Cnt)),
            ssdConsistent(Name)     % check consistency after update
         ;  []                      % no change, no need to check consistency
        )
     ; add(ssdCount(Name,Cnt)),    % create initial cardinality
ssdConsistent(Name)         % check consistency
    ).

%% ssd set ops
deleteSsdSet(Name) -->
    ssdCount(Name,_),
    del(ssdCount(Name,_)),
    del(ssdRole(Name,_)).

create_ssdSet(Name,RoleList,Cnt) -->
    \+ ssdCount(Name,_),
    add(ssdCount(Name,Cnt)),
    addSsdRoleMembers(Name,RoleList),
    ssdConsistent(Name).

%% add role members
addSsdRoleMembers(_,[]).
addSsdRoleMembers(Name,[Role|Roles]) :-
    roles(Role),
    add(ssdRole(Name,Role)),
    addSsdRoleMembers(Name,Roles).

%% define consistency.
ssdConsistent(Name) --> ssdConsistentAssigned(Name).

%% or if include authorized, would use the following clause:
%% ssdConsistent(Name) --> ssdConsistentAuthorized(Name).

%% consistent if not inconsistent
ssdConsistentAssigned(Name) -->
    \+ ssdInconsistentAssigned(Name).

%% inconsistency is more easily defined
ssdInconsistentAssigned(Name) -->
    ssdCount(Name,Cnt),
    ssdAssignedRoleCnt(Name,_User,UCnt),
    {UCnt > Cnt}.

%% aggregation predicate for counting roles
sum(A,B,C) :- C is A + B.

%% aggregate using sum (admittedly, a bit awkward...)
:- table ssdAssignedRoleCnt(_,_,fold(sum/3,0),_,_).
ssdAssignedRoleCnt(Name,User,1) -->
    assignedRoles(User,Role),
    ssdRole(Name,Role).

%% same as above but using authorized as opposed to assigned.
ssdConsistentAuthorized(Name) -->
    \+ ssdInconsistentAuthorized(Name).

ssdInconsistentAuthorized(Name) -->
    ssdCount(Name,Cnt),
    ssdAuthorizedRoleCnt(Name,_User,UCnt),
    {UCnt > Cnt}.

:- table ssdAuthorizedRoleCnt(_,_,fold(sum/3,0),_,_).
ssdAuthorizedRoleCnt(Name,User,1) -->
    authorizedRoles(User,Role),
    ssdRole(Name,Role).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Administrative RBAC (Planning functions)

%% Exhaustive search for a plan
:- table getRolesPlan/6.
getRolesPlan(User,Roles,Acts,PlanSet) -->
    (hasAllRoles(User,Roles)   % at the goal, so
     ->{new_dbi(PlanSet)}     % no actions needed
     ; {member(Act,Acts)},    % for each action
        call(Act),             % perform it
        getRolesPlan(User,Roles,Acts,PlanSet0),  % search from new DB
        {assert_in_db(Act,PlanSet0,PlanSet)}  % succeeded, add this act
    ).

%% check goal state
hasAllRoles(_,[]) --> [].
hasAllRoles(User,[Role|Roles]) -->
    authorizedRoles(User,Role),
    hasAllRoles(User,Roles).

%% Search for shortest plan:
%% Same as above, but only keep plans with fewest actions
:- table getRolesShortestPlan(_,_,_,lattice(smaller_plan/3),_,_).
getRolesShortestPlan(User,Roles,Acts,PlanSet) -->
    (hasAllRoles(User,Roles)
     ->{new_dbi(PlanSet)}
     ; {member(Act,Acts)},
        call(Act),
        getRolesShortestPlan(User,Roles,Acts,PlanSet0),
        {assert_in_db(Act,PlanSet0,PlanSet)}
    ).

%% Plan2 is the smaller of Plan0 and Plan1
smaller_plan(Plan0,Plan1,Plan2) :-
    size_db(Plan0,N0),
    size_db(Plan1,N1),
    (N0 =< N1
     ->Plan2 = Plan0
     ; Plan2 = Plan1
    ).
```