The RBAC challenge in the KB-paradigm

Marc Denecker Jo Devriendt
Department Computer Science, KU Leuven
firstname.lastname@cs.kuleuven.be

June 25, 2018

1 Introduction

The RBAC challenge paper of the LPOP workshop [6] describes a dynamic system for role based access control. In this dynamic system, new users, roles and permissions are added dynamically, or existing ones are deleted. Users are assigned new roles or are stripped of them; roles are assigned new permissions or stripped of them. Roles are organized in hierarchies that may change over time. Users can pose queries. Optimized configurations of role assignments may have to be computed. Plans must be searched to realize goal configurations; selected plans must be executed. The goal of the challenge is to build a software system that implements this dynamic system and various functionalities in it.

In this position paper, we take a theoretical perspective on the problem. The questions we asked ourselves initially were of the following kind: how much of the RBAC domain can be *formally specified* in the logic FO(.) (First Order logic extended with inductive/recursive definitions and aggregates) [3]? How much of the RBAC system can be *analyzed* on the basis of the formal specification? How much of the functionalities of the RBAC system can be *executed* on the basis of the formal specification? What *forms of inference are needed* for that? Some of these questions pertain to the fundamental goals of the scientific domain of Declarative Knowledge Representation. Simple as the RBAC challenge is, we didnot know the answers at the start of this project and some questions remained unanswered at the end.

Our exercise fits in the context of what we called the *Knowledge Base Paradigm* [4, 9]. It is the idea that all problem solving is based on domain knowledge, but domain knowledge itself is inherently independent of the computational problem; formal specifications of it can be reused to solve a range of problems in the application domain. The goal of this experiment is to test this idea in the RBAC challenge: to build one formal specification, one knowledge base (or, well, as few as possible), and to reuse them in various inference problems to provide a maximal range of functionalities. We investigated how much can be *implemented/prototyped with the knowledge base system IDP* [2].

Our exercise is theoretical in the sense that we ignore the main metric in computational logic and Knowledge Representation research: efficiency/scalability. Nevertheless, we found the exercise interesting and thought provoking and we hope others will think the same. As such, we hope this paper provides some material for discussion for the LPOP workshop. New questions that we hope can be addresed during the workshop are: *where is the expertise to derive software systems from formal specifications?*, *what are the best formal specification languages for domains such as RBAC and others?*, *what are the leading systems and technologies to achieve these goals?*, *what further research is needed?*

2 KR: building a formal specification of RBAC

The vocabulary An essential step in KR is the design of the formal vocabulary and its informal interpretation. It should be designed to express the relevant concepts of the application domain, at the right level of abstraction. In this experiment, this step was trivial since all important concepts are explicitly stated in the RBAC challenge paper [6]. The vocabulary is available in the appendix C as *vocabulary V_RBAC*.

The theory We expressed the RBAC dynamic domain in the logic FO(Types,ID,Agg), as much as conveniently possible. This is First Order logic, extended with inductive/recursive definitions and agregates [7]. Below, we denote this logic as FO(.). Since FO(.) is not a temporal logic, the dynamics of the domain needs to be explicated in the vocabulary and the theory. For this we use the methodology of Linear Time Calculus (LTC) [1], a simplified version of the event calculus which uses the natural numbers as a diskrete time line. The LTC methodology introduces some fixed overhead: explicit time arguments for action and fluent (=state) symbols; frame axioms for all fluent symbols $f(ArgTypes, Time)$, expressed in terms of 3 auxiliary predicates per fluent:

- $INIT\_f(ArgTypes)$ to express the initial state of f;
- $C\_f(ArgTypes, Time)$ to express when f is caused to be true.
- $CN\_f(ArgTypes, Time)$ to express when f is caused to be false. The prefix "CN" stands for "Causes Not".

Also, standard actions of adding and deleting fluent atoms need to be specified. More than 80% of the specification is *boiler plate* overhead which in a special purpose dynamic specification language could and should be avoided.

The main components of the theory are:

- recursive definitions of fluents expressing inertia and how actions influence fluents. It contains inertia rules such as:

$$\forall x, t : USERS(x, Next(t)) \leftarrow USERS(x, t) \wedge \neg CN\_USERS(x, t).$$

and causal rules such as:

$$\forall x,t : CN\_USERS(x,t) \leftarrow Delete\_USERS(x,t).$$

which expresses that the delete action causes $\neg USERS(x)$ to become true;

- action preconditions; e.g., to express that new user-role relations may be added only for active users and roles:

$$\forall x,r,t : Add\_UR(u,r,t) \Rightarrow \neg UR(u,r,t) \wedge USERS(u,t) \wedge ROLES(r,t).$$

- definitions of several derived concepts: e.g., $UserPermission$; HUR which relates users with all the roles in the role hierarchy that they possess;
- concurrency axioms constraining simultaneous execution of actions.

A full theory is specified in Appendix C.

Transactions that were not specified There are also transactions of the RBAC software system that are not and could not (conveniently) be formalized *in* the theory. All transactions that take a *logic expression* as input were not formally specified: the operations of querying, planning, plan execution and optimisation take expressions as input. E.g., the query operation takes as input a query *expression* and returns as output the *value* of this expression in the current state. E.g., this expression could be a set-expression, and its value a set. The problem is that FO(.) lacks the expressivity to (*conveniently*) express a function from expressions to their value in the underlying structure. To express this in the logic, meta-facilities are required and they are not available in FO(.). The same argument holds for planning (the goal) and for optimisation (the cost function).

The absence of specification of these transactions in the formal theory of RBAC is a striking gap in the theory. However, it does not mean that these transactions cannot be *executed* using logical inference methods. This is discussed in the next section. But what it entails is, for instance, that any formal analysis of the RBAC theory, e.g., for proving invariants, is partial: it does not take into account the missing transactions.

The essence of the specification The declarative information in the RBAC challenge is quite limited. The essential information is in the definitions and in the invariants. The definitions are of the transitive closure $TransHR$ of the role hierarchy, the user roles HUR in the hierarchy, and the user permissions $UserPermission$. This amounts to:

```
/* Definitions */
    { ! x[Role], y[Role], t[Time] : TransRH(x,y,t) <- RH(x,y,t).
      ! x, y, t : TransRH(x,y,t) <- ?z: RH(x,z,t) & TransRH(z,y,t).
    }
    {  ! u[User], r[Role], t[Time]: HUR(u,r,t) <-  UR(u,r,t).
       ! u, r, t: HUR(u,r,t) <- ?r1: UR(u,r1,t) & TransRH(r,r1,t).
    }

    { !u[User], p[Perm], t[Time]: UserPermission(u,p,t) <-
                                     ?r: HUR(u,r,t)) & PR(p,r,t).
    }

/* Invariants */
    ! u, r, t: UR(u,r,t) => USERS(u,t) & ROLES(r,t).

    ! p, r, t: PR(p,r,t) => PERMS(p,t) & ROLES(r,t).

    ! r, r1, t: RH(r,r1,t) => ROLES(r,t) & ROLES(r1,t).
    ! t: ~? r: TransRH(r,r,t).

    ! ssd, r, t: SSD_ROLES(ssd,r,t) => ROLES(r,t).

    ! u, ssd[SSD], t:
        #{r[Role] : HUR(u,r,t) & SSD_ROLES(ssd,r,t)} =< SSD_Card(ssd,t).
```

This theory (±) is presented in Appendix A. All the rest of the theory is boiler-place and can be generated automatically. One obtains a theory as in appendix C. Or, in a suitable dynamic logic derived from FO(.), the extra rules and assertions would be implicit in the semantics of the language. In particular, for each fluent, the frame axioms, the add and delete actions and their preconditions are similar in all cases and, in a dynamic logic, should be implicit. All functionalities specified in the RBAC challenge paper can be derived by generic inference on the completed theory. We discuss how in the next section.

Remark Regarding the suitability of our logic for real world applications, if we forget about the boiler plate which evidently must be eliminated, our specification is simple to understand, compact, and contains no redundancy: every aspect that was formalized needs to be formalized.

A feature of a formal specification in LTC is that it is state-oriented: transactions are atomic actions. The same is true in many dynamic specification languages. One problem that we see with, e.g., the optimization transaction or the planning transactions in RBAC is that, in practice, these operations are not atomic but they are processes involving user interaction. At the very least, the user needs to make a selection out of the possible reconfigurations or plans. Thus, a specification language may need to support the concept of process. Standard imperative or object oriented languages are strongly process oriented. Surely, this has sometimes its disadvantages as well. E.g., such process descriptions often impose unnecessary constraints on the order of execution of actions. So, one question that rises here is the issue of how to formally specify processes, whether or when state-oriented versus process-oriented is best, and how to combine the best of worlds.

3 Tasks

Analysis: Verification of invariants The theory `T_RBAC_Pre` in Appendix C specifies action preconditions. E.g., that a new edge (r_1, r_2) can be added to the role hierarchy RH only if r_2 is not higher in the hierarchy than r_1. This is to avoid that cycles are created in the hierarchy.

The RBAC challenge paper [6] mentions a set of invariants of RBAC. E.g., roles assigned to users need to be in the set $ROLES$ of active roles, and can be assigned only to elements of the set $USERS$ of active users. Others are implicit, e.g., there are no cycles in the role hierarchy. They are described above. They are the elements of the theory `invariant` in the appendix C.

One analysis task for the specification is to check if `T_RBAC_Pre` entails `invariant`. This is a deduction problem over the inductively defined set of natural numbers ($Time$). Using a standard technique, this problem can (often) be reduced to determining unsatisfiability of the following theory:

$$\texttt{T\_RBAC\_Pre\_bs} + \texttt{invariant(0)} + \neg\texttt{invariant(1)}$$

Here, `T_RBAC_Pre_bs` is the *bistate theory*, the part of theory `T_RBAC_Pre` expressing the relationship between two successive states, named $0, 1$; $invariant(0)$ and $\neg invariant(1)$ express that the invariants are satisfied at 0 and not at 1. If this theory is unsatisfiable, then any process starting from an initial state satisfying $invariant(0)$ preserves the invariants.

The theory `T_RBAC_Pre` contains an inductive definition of `TransRH` that cannot be expressed in predicate logic. To the best of our knowledge, there are currently no theorem provers for predicate logic augmented with inductive definitions[1]. The IDP system supports a light weight version: it can verify the satisfiability in the context of fixed finite domain. It is nevertheless useful. That is, the *small scope hypothesis* works fine in *many* cases: errors in the specification often do emerge in small domains.

We performed this analysis with IDP. At first, we assumed a *no concurrency* axiom, excluding the presence of multiple simultaneous actions. The analysis brought a few forgotten preconditions to the surface: namely that no element of $USERS, ROLES, PERMS$ may be be deleted when still in use in UR, PR, RH, SSD_Roles.

In a second step, we analyzed concurrent execution of actions (dropping *no concurrency*). The analysis showed that with concurrency, all action preconditions need to be strengthened. So that, e.g., it is not possible to simultaneously add a role to user u and delete u of $USERS$. The action preconditions become quite complex then. However, (1) the action preconditions can be computed automatically from the invariants by the principle of **regression** [8]; (2) by adding

[1] Entailment of predicate logic with inductive definitions is not decidable, not even semi-decidable.

the invariants to the theory, combinations of actions that violate invariants can be detected by satisfiability checking. Thus, if the goal of an action precondition is merely to safeguard the invariants, there is no need for it: a suitable transaction engine will be able to accept or reject a proposed transaction on the basis of an LTC theory including the invariants. Thus, we can greatly simplify the theory. The appendix A contains the theory from which all boiler-plate was removed. It is the input of the IDP-solution.

We observe that not every action precondition serves to protect an invariant. E.g., an action precondition for the operation of adding x to a fluent is that x is not an element already of the fluent. This action precondition is not related to an invariant.

Executing updates for RBAC The RBAC challenge specifies a dynamic transactional system with persistent data and updates through add and delete operations. We here describe how, in theory, the updates could be derived from the formal specification.

For simplicity, we assume that times and dates are associated with natural numbers. E.g., 3:35pm on 18/7/2018 is associated with the total number of seconds that has passed since 0:00am of 1/1/1980.

Given an LTC theory T, we define a *state theory at time* $n \in \mathbb{N}$ as a theory consisting of the following components:

- the theory T, extended with
- equation $now = n$, where now is a logical constant informally interpreted as the current time;
- an exhaustive description of the initial values of all fluents;
- an exhaustive description of the set of actions that occur at time points $t \leq n$ (i.e., in the past of now). E.g., in FO(.), the actions could be described by:

  $$\left\{\begin{array}{l} Add\_USERS(u,t) \leftarrow Future\_Add\_Users(u,t) \wedge t > now. \\ Add\_USERS(Jim,''2/1/2018, 10:31am''). \\ Add\_USERS(Sarah,''2/1/2018, 15:02pm''). \\ \ldots \langle \text{ set of add operations to } USERS \text{ in the past of } now \rangle \end{array}\right\}$$

  This definition expresses a *local closed world assumption* on Add_USERS, for the past of now. Here, the predicate $Future\_Add\_Users$ represents the unknown future Add_USERS transactions.

An evolution of the RBAC software system corresponds to an evolution of state theories. At each time point n, the state of the software system corresponds to a state theory at n. This state theory represents the *epistemic state* of the application: what it knows and does not know. With an update at time n, a new state theory corresponds which is obtained by extending the previous one with actions at time n. E.g., if Dave is added as a user on 18/6/2018 12am, the above definition is extended with $Add\_USERS(Dave,''18/6/2018, 12am'')$.. With time, the state theory accumulates more information about the world, in particular about the past of *now*. At no point in time, the state theory knows the future of *now*. There is a monotonicity property: the class of models/possible worlds of state theories decreases monotonically with time until, at infinity, it becomes *categorical* and has only one model: the history as it happened. Since the class of possible worlds decreases, the knowledge increases. There is one aspect that is non-monotonic though: with time, the value of *now* changes and this is a non-monotonic change. I.e., with time the application changes its mind about what is the current time. Even when nothing happens for a while, the application accumulates extra knowledge: that no change happened. Of course, the meta-operations (e.g., the queries and planning operations) are not and cannot be registered in the state theory.

Conceptually, to verify if the action preconditions and concurrency axioms are satisfied by a proposed update, the update is inserted in the state theory and the theory is verified for satisfiability. If the state theory is satisfiable, the update is accepted and the state theory is stored. Otherwise, the update is rejected and the state theory remains unchanged (except for the new value of *now*).

For a practical implementation of the above theoretical procedure, many optimizations are possible and necessary. For example, the satisfiability of the action preconditions of an update at time $now = n$ can be computed using the *current state structure*: the state at time *now*. Implementation-wise, it makes sense to explicitly store this structure. If the transaction is accepted, the current state structure can be *progressed* to the new current state [5, 1]. The current state structure is useful as well to answer what will probably be the bulk of the queries, namely queries about the current state.

Solving current state queries and temporal queries A state theory at time n determines two structures: the *current state structure* I_{Cur}, expressing all fluents and actions at time $now = n$, and the *past state structure* I_{Past}, expressing all fluents and actions for the interval $[0, n]$. The current state structure is a structure of the *single state vocabulary*: this is the vocabulary from which time is projected out from fluents and actions.

Queries over the current state can be expressed as a set expression, or a formula or a function term in the single state vocabulary. An example is:

$$\{u[User] : \#\{p[Perm] : UserPermission(u, p)\} \geq 4\}$$

It expresses the set of users that have at least 4 permissions in the current state.

Temporal queries generalize current state queries. E.g., this current state query can be expressed as the temporal query:

$$\{u[User] : \#\{p[Perm] : UserPermission(u, p, now)\} \geq 4\}$$

Temporal queries over the past can be expressed as expressions of the same sort over the original vocabulary. E.g., the following query is whether there is a user that once had a permission to “write”, lost it and then regained it:

$$\exists u : \exists t1, t2, t3 : t1 < t2 < t3 < now \wedge UserPermission(u, Write, t1) \wedge \\ \neg UserPermission(u, Write, t2) \wedge UserPermission(u, Write, t3)$$

As explained above, the query operations cannot be formally specified in the description of the dynamic system. They can be expressed on the (procedural) meta-level and such queries can be solved by IDP in the suitable structure. Querying does not change the state and hence, it trivially preserves all invariants.

Planning and plan execution For this problem of the RBAC challenge, the goal is to compute a series of updates to transform the current state into a goal state satisfying a formula $\Psi[t]$. In a next phase, if the user accepts the computed plan, the plan has to be executed.

The planning inference problem takes as input the current state theory and the goal formula $\exists t : t \geq now \wedge \Psi[t]$. Its output is representable as an exhaustive enumeration of add and delete actions in some interval $[now, t\_end]$ so that the state theory extended with it is satisfiable and entails $\Psi[t\_end]$.

In practice, this problem can be solved using iterated model expansion. This is a well known approach in SAT for planning and in answer set programming. The search is for a model of the current state theory augmented $\Psi[now + N]$. N is incremented until a model is found.

At this theoretical level, “execution” of the plan boils down to add the actions in the time interval $[now, now + N]$ in this model to the state theory. In reality, there is much more to do. E.g., execution of plans with actions that change the external world have to be monitored since they may fail. Here we will ignore this problem.

As explained above, formally specifying the planning transaction in the description of the dynamic system requires meta-facilities in the underlying logic. This does not prevent us from specifying the transaction at the procedural meta-level, using o.a. a call to a planning inference engine.

Optimizing the configuration The last problem of the RBAC challenge considered here is to reconfigure the base relations UR, PR, RH: determine minimal values for these relations such that all users maintain exactly the same permissions as in the current state.

This problem can be specified as the following inference problem. It takes as input the definition of $UserPermission/2$, the current state structure projected on the symbols $USERS, ROLES, PERMS$ and $UserPermission/2$, and finally a cost function specifying that the sum of the cardinality of the relations UR, PR, RH is minimal. The output is a value for UR, P, RH in a model that minimizes the cost function. This is an application of *optimization inference*.

We observe that the input of the problem contains both a value and an (inductive) definition for $UserPermission$. Thus, the model generator needs to find values for the parameters of this definition such that the value determined by the definition corresponds to the given value.

The final step is to bring the database in the optimized state. This problem can be reduced to the application of the planning and plan execution procedure.

4 Implementation in IDP

We implemented a prototype system in IDP. It supports base versions of all the requested functionalities of the challenge. The system has a persistent state represented as a state theory. Implemented operations are: verification of invariants, temporal queries, updates of base relations, planning, execution of chosen plans, optimization, and planning and executing a choosen optimization. Our explicit goal was to narrowly implement the theoretical ideas, that is, to characterize and implement a maximum of functionality and flexibility on the basis of a minimal, purely declarative specification. The input of the system is the formal specification provided in the appendix A. All "boiler plate" is automatically generated from it. A trace file is presented in appendix B. None of the optimizations proposed above were implemented. The system handles only toy examples. Nevertheless, we expect that with a limited effort, it should be possible to build a system from off-the-shelve tools that can handle small applications. The system is available at `bitbucket.org/krr/rbac`.

5 Conclusion

The contributions of this paper are more in the scientific questions that we pose than in the complexity of the solutions that were offered. The goal of this experiment was the following: to check to what extend the RBAC software system could be implemented by generic inference on a knowledge base/formal specification. To this aim, we have evaluated the instances for RBAC of some fundamental questions of KR: what parts of the dynamic system can be formally specified, what forms of inference are needed to implement the functionalities of RBAC.

We have seen gaps in the expressivity of the logic (which occur in many, if not all current dynamic logics), namely to express complex transactions that take arguments of type *Expression* as input. That does not mean that for executing them, logic based systems are of no use. The contrary is true. However, it certainly means that the full RBAC system cannot be formally analyzed, e.g., proving invariants. It also excludes that the RBAC system as a whole can be run by uniformally applying a fixed form of inference on the specification.

All parts of the RBAC challenge can be "implemented" by inference on the formal specification(s). In all of this, the same very limited set of propositions are used time and again: the definitions of the main concepts ($UserPermission$, HUR, $TransRH$), concurrency axioms, invariants. Beside this, other declarative entities such as queries and goals and cost functions need to be expressed depending of the problem at hand.

There are important functionalities that were not considered. E.g., verification of temporal logic properties. E.g., *revision inference* to erase erroneous facts. For example, assume that in 2011, a non-existing user was accidentally added and this is discovered in 2018.

References

[1] Bart Bogaerts, Joachim Jansen, Maurice Bruynooghe, Broes De Cat, Joost Vennekens, and Marc Denecker. Simulating dynamic systems using linear time calculus theories. *TPLP*, 14(4–5):477–492, 7 2014.

[2] Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312v2, 2016.

[3] Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, 9(2):14:1–14:52, April 2008.

[4] Marc Denecker and Joost Vennekens. Building a knowledge base system for an integration of logic programming and classical logic. In María García de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *LNCS*, pages 71–76. Springer, 2008.

[5] Fangzhen Lin and Raymond Reiter. How to progress a database. *Artif. Intell.*, 92(1-2):131–167, 1997.

[6] Yanhong A. Liu. Role-based access control as a programming challenge. Logic and Practice of Programming workshop, July 18 2018.

[7] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *TPLP*, 7(3):301–353, 2007.

[8] Raymond Reiter. *Artificial intelligence and mathematical theory of computation*, chapter The frame problem in situation the calculus: A simple solution (sometimes) and a completeness result for goal regression, pages 359–380. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

[9] Pieter Van Hertum, Ingmar Dasseville, Gerda Janssens, and Marc Denecker. The KB paradigm and its application to interactive configuration. *TPLP*, 17(1):91–117, 2017.

A Input RBAC specification (without boiler plate)

```
vocabulary V{
    extern vocabulary LTCvoc

    TransRH(Role,Role,Time)
    HUR(User,Role,Time)
    UserPermission(User,Perm,Time)
}

theory Invariants:V{
    /* No concurrency axiom  */


    /* Invariants */
    !ssd,r,t: SSD_ROLES(ssd,r,t) => ROLES(r,t).

    ! u[User],ssd[SSD],t, c: SSD_Card(ssd,c,t) =>
    #{r[Role] : HUR(u,r,t) & SSD_ROLES(ssd,r,t)} =< c.
}

theory UserPermission:V{
    { ! x,y,t : TransRH(x,y,t) <- RH(x,y,t).
        ! x,y,t : TransRH(x,y,t) <- ?z: RH(x,z,t) & TransRH(z,y,t).
    }

    {  ! u,r,t: HUR(u,r,t) <-  UR(u,r,t).
        ! u,r,t: HUR(u,r,t) <- ?rr: UR(u,rr,t) & TransRH(r,rr,t).
    }

    /* UserPermission : definition:  */
    { !u,p,t: UserPermission(u,p,t) <- ?r: HUR(u,r,t) & PR(p,r,t).
    }

    /* Invariants */
    ! u,r,t: UR(u,r,t) => USERS(u,t) & ROLES(r,t).

    ! p,r,t: PR(p,r,t) => PERMS(p,t) & ROLES(r,t).

    ! r,rr,t: RH(r,rr,t) => ROLES(r,t) & ROLES(rr,t).
    !t: ~?r: TransRH(r,r,t).
}

structure S:V {
    Time = {0..20}
    now = procedure readNow
}

term ObjPlan:V{
    maxT
}

term ObjOptimize:V{
    #{u[User] r[Role] : UR(u,r,now)} +
      #{p[Perm] r[Role] : PR(p,r,now)} + #{r[Role] rr[Role] : RH(r,rr,now)}
}
```

```
vocabulary OptimizeProjection{
    extern vocabulary Types

    extern V::USERS/2
    extern V::PERMS/2
    extern V::ROLES/2
    extern V::now/0:1
    extern V::UserPermission/3
}

vocabulary OptimizeGoal{
    extern vocabulary Types

    extern V::UR/3
    extern V::PR/3
    extern V::RH/3

}
```

B A trace

```
$./reset.sh
$./query.sh "{ x : USERS(x,now)}"
{  }
$./query.sh "{ x : x=now}"
{ 1 }
$./query.sh "{ x : x=now}"
{ 2 }
$./update.sh "Add_USERS(u1). Add_USERS(u2). Add_USERS(u3).
Add_ROLES(r1). Add_ROLES(r2). Add_ROLES(r3).
Add_PERMS(read). Add_PERMS(write). Add_PERMS(modify).
Add_UR(u2,r2). Add_UR(u3,r3).
Add_PR(write,r1). Add_PR(read,r2). Add_PR(modify,r3).
Add_RH(r1,r2). Add_RH(r2,r3)."
Update succesful.
$./query.sh "{ x p : UserPermission(x,p,now)}"
{ u2,read; u2,write; u3,modify; u3,read; u3,write }
$./update.sh "Add_USERS(u1)."
Warning: given update violates preconditions and is aborted.
$./update.sh "Delete_USERS(u1)."
Update succesful.
$./update.sh "Add_USERS(u1)."
Update succesful.
$./update.sh " Add_RH(r3,r3)."
Warning: given update violates preconditions and is aborted.
$./update.sh " Add_RH(r3,r2)."
Warning: given update violates preconditions and is aborted.
$./query.sh "{ x p : TransRH(x,p,now)}"
{ r1,r2; r1,r3; r2,r3 }
$./plan.sh "/*all users have all permissions; only one action per time point */
! x[User], p1[Perm]: UserPermission(x,p1,maxT)  &
 ! t: t>= now => #{u:Add_USERS(u,t)}+
    #{u:Add_ROLES(u,t)}+
    #{u:Add_PERMS(u,t)}+
    #{u, r:Add_UR(u,r,t)}+
    #{p, r :Add_PR(p,r,t)}+
    #{r,rr :Add_RH(r,rr,t)}+
    #{s,r :Add_SSD_ROLES(s,r,t)}+
    #{s,c :Add_SSD_Card(s,c,t)}+
    #{u:Delete_USERS(u,t)}+
    #{u:Delete_ROLES(u,t)}+
    #{u:Delete_PERMS(u,t)}+
    #{u, r:Delete_UR(u,r,t)}+
    #{p, r :Delete_PR(p,r,t)}+
    #{r,rr :Delete_RH(r,rr,t)}+
    #{s,r :Delete_SSD_ROLES(s,r,t)}+
    #{s,c :Delete_SSD_Card(s,c,t)}=<1. "
A correct plan is:
1) Add_PR(modify,r2).
2) Add_UR(u1,r3).
Commit this plan? (y/n/q) y
$./optimize.sh
Optimization:
PR(modify,r2). PR(read,r2). PR(write,r2).
UR(u1,r2). UR(u2,r2). UR(u3,r2).

Proposed plan:
1) Add_PR(write,r2). Delete_PR(modify,r3).  Delete_PR(write,r1).
Delete_RH(r1,r2). Delete_RH(r2,r3). Add_UR(u3,r2).
Delete_UR(u3,r3).
Commit this plan? (y/n/q) y
$
```

C RBAC with action preconditions and without concurrency

```
LTCvocabulary V_RBAC{
    type Time isa int
    Start:Time
    partial Next(Time):Time

    type User
    type Role
    type Perm

    USERS(User,Time)
    Add_USERS(User,Time)
    Delete_USERS(User,Time)

    Init_USERS(User)
    C_USERS(User,Time)
    CN_USERS(User,Time)

    UsedUSERS(User,Time)
    UsedROLES(Role,Time)
    UsedPERMS(Perm,Time)

    ROLES(Role,Time)
    Add_ROLES(Role,Time)
    Delete_ROLES(Role,Time)
```

```
Init_ROLES(Role)
C_ROLES(Role,Time)
CN_ROLES(Role,Time)

PERMS(Perm,Time)
Add_PERMS(Perm,Time)
Delete_PERMS(Perm,Time)

Init_PERMS(Perm)
C_PERMS(Perm,Time)
CN_PERMS(Perm,Time)

UR(User,Role,Time)
Add_UR(User,Role,Time)
Delete_UR(User,Role,Time)

Init_UR(User,Role)
C_UR(User,Role,Time)
CN_UR(User,Role,Time)

PR(Perm,Role,Time)
Add_PR(Perm,Role,Time)
Delete_PR(Perm,Role,Time)

Init_PR(Perm,Role)
C_PR(Perm,Role,Time)
CN_PR(Perm,Role,Time)

UserPermission(User,Perm,Time)

/* hierarchical */
RH(Role,Role,Time)
Add_RH(Role,Role,Time)
Delete_RH(Role,Role,Time)

Init_RH(Role,Role)
C_RH(Role,Role,Time)
CN_RH(Role,Role,Time)

TransRH(Role,Role,Time)
HUR(User,Role,Time)  /* AuthorizedRole */
HUserPermission(User,Perm,Time)

/*SSD*/

type SSD
type NrRoles isa int
SSD_ROLES(SSD,Role,Time)
Add_SSD_ROLES(SSD,Role,Time)
Delete_SSD_ROLES(SSD,Role,Time)

Init_SSD_ROLES(SSD,Role)
C_SSD_ROLES(SSD,Role,Time)
CN_SSD_ROLES(SSD,Role,Time)

SSD_Card(SSD,Time):NrRoles
Set_SSD_Card(SSD,NrRoles,Time)
```

```
    Init_SSD_Card(SSD,NrRoles)
    C_SSD_Card(SSD,NrRoles,Time)

}

theory T_RBAC_Pre: V_RBAC{

    { ! u,t:UsedUSERS(u,t)<- ?r:UR(u,r,t).}

    { ! r,t:UsedROLES(r,t)<- ?u:UR(u,r,t).
       ! r,t:UsedROLES(r,t)<- ?p:PR(p,r,t).
        ! r,t:UsedROLES(r,t)<- ?r1:RH(r,r1,t).
        ! r,t:UsedROLES(r,t)<- ?r1:RH(r1,r,t).
        ! r,t:UsedROLES(r,t)<-? ssd: SSD_ROLES(ssd,r,t). }

    { ! p, t :UsedPERMS(p,t)<- ?r:PR(p,r,t).}

 /*RBAC pure */
    { !u: USERS(u,Start) <- Init_USERS(u).
      !u, t: USERS(u,Next(t)) <- C_USERS(u,t).
      !u, t: USERS(u,Next(t)) <- USERS(u,t) & ~CN_USERS(u,t).

      !u,t: C_USERS(u,t) <- Add_USERS(u,t).
      !u,t: CN_USERS(u,t) <- Delete_USERS(u,t).
    }

    ! x,t: Add_USERS(x,t) => ~USERS(x,t).
    ! x,t: Delete_USERS(x,t) => USERS(x,t) & ~ UsedUSERS(x,t).

    { !r: ROLES(r,Start) <- Init_ROLES(r).
      !r, t: ROLES(r,Next(t)) <- C_ROLES(r,t).
      !r, t: ROLES(r,Next(t)) <- ROLES(r,t) & ~CN_ROLES(r,t).

      !r,t: C_ROLES(r,t) <- Add_ROLES(r,t).
      !r,t: CN_ROLES(r,t) <- Delete_ROLES(r,t).
    }

    ! x,t: Add_ROLES(x,t) => ~ROLES(x,t).
    ! x,t: Delete_ROLES(x,t) => ROLES(x,t) & ~ UsedROLES(x,t).


    { !p: PERMS(p,Start) <- Init_PERMS(p).
      !p, t: PERMS(p,Next(t)) <- C_PERMS(p,t).
      !p,t: PERMS(p,Next(t)) <- PERMS(p,t) & ~CN_PERMS(p,t).

      !p,t: C_PERMS(p,t) <- Add_PERMS(p,t).
      !p,t: CN_PERMS(p,t) <- Delete_PERMS(p,t).
    }

    ! x,t: Add_PERMS(x,t) => ~PERMS(x,t).
    ! x,t: Delete_PERMS(x,t) => PERMS(x,t)& ~ UsedPERMS(x,t).

    { !u,r: UR(u,r,Start) <- Init_UR(u,r).
      !u, r, t: UR(u,r,Next(t)) <- C_UR(u,r,t).
      !u,r,t: UR(u,r,Next(t)) <- UR(u,r,t) & ~CN_UR(u,r,t).
      !u,r,t: C_UR(u,r,t) <- Add_UR(u,r,t).
      !u,r,t: CN_UR(u,r,t) <- Delete_UR(u,r,t).
    }

    ! u,r,t: Add_UR(u,r,t) => ~UR(u,r,t).
    ! u,r,t: Add_UR(u,r,t) => USERS(u,t) & ROLES(r,t).
    ! u,r,t: Delete_UR(u,r,t) => UR(u,r,t).

    { !p,r: PR(p,r,Start) <- Init_PR(p,r).
      !p, r, t: PR(p,r,Next(t)) <- C_PR(p,r,t).
      !p,r,t: PR(p,r,Next(t)) <- PR(p,r,t) & ~CN_PR(p,r,t).

      !p,r,t: C_PR(p,r,t)<-Add_PR(p,r,t).
      !p,r,t: CN_PR(p,r,t)<-Delete_PR(p,r,t).
    }

    ! p,r,t: Add_PR(p,r,t) => ~PR(p,r,t).
    ! p,r,t: Add_PR(p,r,t) => PERMS(p,t) & ROLES(r,t).
    ! p,r,t: Delete_PR(p,r,t) => PR(p,r,t).

    { !u,p,t: UserPermission(u,p,t) <- ?r: UR(u,r,t) & PR(p,r,t).
    }

/* Hierarchical */

    { !r,r1: RH(r,r1,Start) <- Init_RH(r,r1).
      !r,r1, t: RH(r,r1,Next(t)) <- C_RH(r,r1,t).
      !r,r1, t: RH(r,r1,Next(t)) <- RH(r,r1,t) &  ~CN_RH(r,r1,t).

      !r,r1,t: C_RH(r,r1,t)<-Add_RH(r,r1,t).
      !r,r1,t: CN_RH(r,r1,t)<-Delete_RH(r,r1,t).
    }

    ! r,r1,t: Add_RH(r,r1,t) => ~RH(r,r1,t).
    ! r,r1,t: Add_RH(r,r1,t) => ROLES(r,t) & ROLES(r1,t) & r~=r1.
    ! r,r1,t: Add_RH(r,r1,t) => ~TransRH(r1,r,t).
    ! r,r1,t: Delete_RH(r,r1,t) => RH(r,r1,t).

    { ! x,t : TransRH(x,y,t) <- RH(x,y,t).
      ! x,y,t : TransRH(x,y,t) <- ?z: RH(x,z,t) & TransRH(z,y,t).
    }

    { ! u,r,t: HUR(u,r,t) <- ?r1: UR(u,r1,t) & TransRH(r,r1,t).
    }

    { !u,p,t: HUserPermission(u,p,t)<- ?r: (UR(u,r,t) | HUR(u,r,t)) & PR(p,r,t).
    }

/* SSD */
    { !ssd,r: SSD_ROLES(ssd,r,Start) <- Init_SSD_ROLES(ssd,r).
      !ssd,r, t: SSD_ROLES(ssd,r,Next(t)) <- C_SSD_ROLES(ssd,r,t).
      !ssd,r,t: SSD_ROLES(ssd,r,Next(t)) <- SSD_ROLES(ssd,r,t) & ~CN_SSD_ROLES(ssd,r,t).

      !ssd,r,t: C_SSD_ROLES(ssd,r,t) <- Add_SSD_ROLES(ssd,r,t).
      !ssd,r,t: CN_SSD_ROLES(ssd,r,t) <- Delete_SSD_ROLES(ssd,r,t).
    }
    ! ssd,r,t: Add_SSD_ROLES(ssd,r,t) => ~SSD_ROLES(ssd,r,t).
    ! ssd,r,t: Delete_SSD_ROLES(ssd,r,t) => SSD_ROLES(ssd,r,t).

    { !ssd,c: SSD_Card(ssd,Start)=c <- Init_SSD_Card(ssd,c).
      !ssd,c, t: SSD_Card(ssd,Next(t))=c <- C_SSD_Card(ssd,c,t).
      !t,ssd: SSD_Card(ssd,Next(t))=SSD_Card(ssd,t) <- ~? c: C_SSD_Card(ssd,c,t).

      !t,ssd,c: C_SSD_Card(ssd,c,t)<- Set_SSD_Card(ssd,c,t).
    }

    ! u[User],ssd[SSD],t: #{r[Role] : HUR(u,r,t) & SSD_ROLES(ssd,r,t)} =< SSD_Card(ssd,t).

 //No concurrency
   ! t: #{u:Add_USERS(u,t)}+
 #{u:Add_ROLES(u,t)}+
 #{u:Add_PERMS(u,t)}+
 #{u, r:Add_UR(u,r,t)}+
 #{p, r :Add_PR(p,r,t)}+
 #{r,r1 :Add_RH(r,r1,t)}+
 #{s,r :Add_SSD_ROLES(s,r,t)}+
 #{s,c :Set_SSD_Card(s,c,t)}+
 #{u:Delete_USERS(u,t)}+
 #{u:Delete_ROLES(u,t)}+
 #{u:Delete_PERMS(u,t)}+
 #{u, r:Delete_UR(u,r,t)}+
 #{p, r :Delete_PR(p,r,t)}+
 #{r,r1 :Delete_RH(r,r1,t)}+
 #{s,r :Delete_SSD_ROLES(s,r,t)}=<1.

}

theory invariant: V_RBAC{
    ! u,r,t: UR(u,r,t) => USERS(u,t) & ROLES(r,t).
    ! p,r,t: PR(p,r,t) => PERMS(p,t) & ROLES(r,t).

    ! r,r1,t: RH(r,r1,t) => ROLES(r,t) & ROLES(r1,t).
    !t: ~?r: TransRH(r,r,t).

    !ssd,r,t: SSD_ROLES(ssd,r,t) => ROLES(r,t).

    ! t[Time]: !u[User],ssd[SSD]:
              #{r[Role] : HUR(u,r,t) & SSD_ROLES(ssd,r,t)}
                                               =< SSD_Card(ssd,t).
}
```