A temporal-logic extension of role-based access control covering dynamic separation of duties

Till Mossakowski
BISS, Dept. of Computer Science
University of Bremen
P.O. Box 330 440, 28334 Bremen, Germany
till@tzi.de

Michael Drouineaud
BISS, Dept. of Computer Science
University of Bremen
P.O. Box 330 440, 28334 Bremen, Germany
mdruid@tzi.de

Karsten Sohr
BISS, Dept. of Computer Science
University of Bremen
P.O. Box 330 440, 28334 Bremen, Germany
sohr@tzi.de

Abstract

Security policies play an important role in today's computer systems. We show some severe limitations of the widespread standard role-based access control (RBAC) model, namely that object-based dynamic separation of duty as introduced by Nash and Poland cannot be expressed with it. We suggest to overcome these limitations by extending the RBAC model with an execution history. The natural next step is then to add temporal logic for the specification of execution orders. We show that with this, object-based dynamic separation of duty, as well as other policies, can be adequately specified.

1. The RBAC model/Separation of duty

Role-based access control (RBAC) [18, 1, 2, 9] has received considerable attention as a promising alternative to traditional discretionary and mandatory access control. Moreover, an extensive field study by the National Institute of Standards and Technology (NIST) pointed out that in practice permissions are assigned to users according to their roles/functions in the organization [8]. The explicit representation of roles greatly simplifies the security management and makes possible to use security principles like separation of duty and least privilege [18]. In the following, we give an overview of RBAC96, a widespread RBAC model introduced by Sandhu et al. [18]. In Figures 1 and 2 the entity sets of RBAC96 and the relationships between them are shown.

A further important concept of the advanced RBAC96 models are *constraints* on the relations *assigned_to*, *auth*, *active_in*, etc. With help of these constraints, separation of duty (SOD) can be enforced. SOD is a well-known principle that prevents fraud and error by requiring that at least two persons are required to complete a task. SOD is often applied in everyday life, e.g., a paper submitted to a conference typically is required to be reviewed by three referees who must be different from the author. There are several attempts to express SOD constraints in the computer security world, specifically in the area of banking like the Clark-Wilson model [6] or Sandhu's Transaction Control Expressions [17]. Usually, it can be differentiated between static and dynamic SOD. Static SOD means that a user is not permitted to perform certain steps of a task. By contrast, in dynamic SOD a user may carry out those steps, but only if he has not done/does not certain other steps of the task. Thus, dynamic SOD is more flexible than static SOD and hence better satisfies real-world requirements.

Usually, RBAC is expressed in set-theoretic notation. When using formal methods, one needs to express RBAC in some definite formalism (like the Z formalism used in [5]). We here formalize RBAC in many-sorted first-order logic, which also is a well-studied and tool-supported formalism [14]. In the specification given in Fig. 3, the axiom states that a session may activate a role only if its user is assigned to the role.

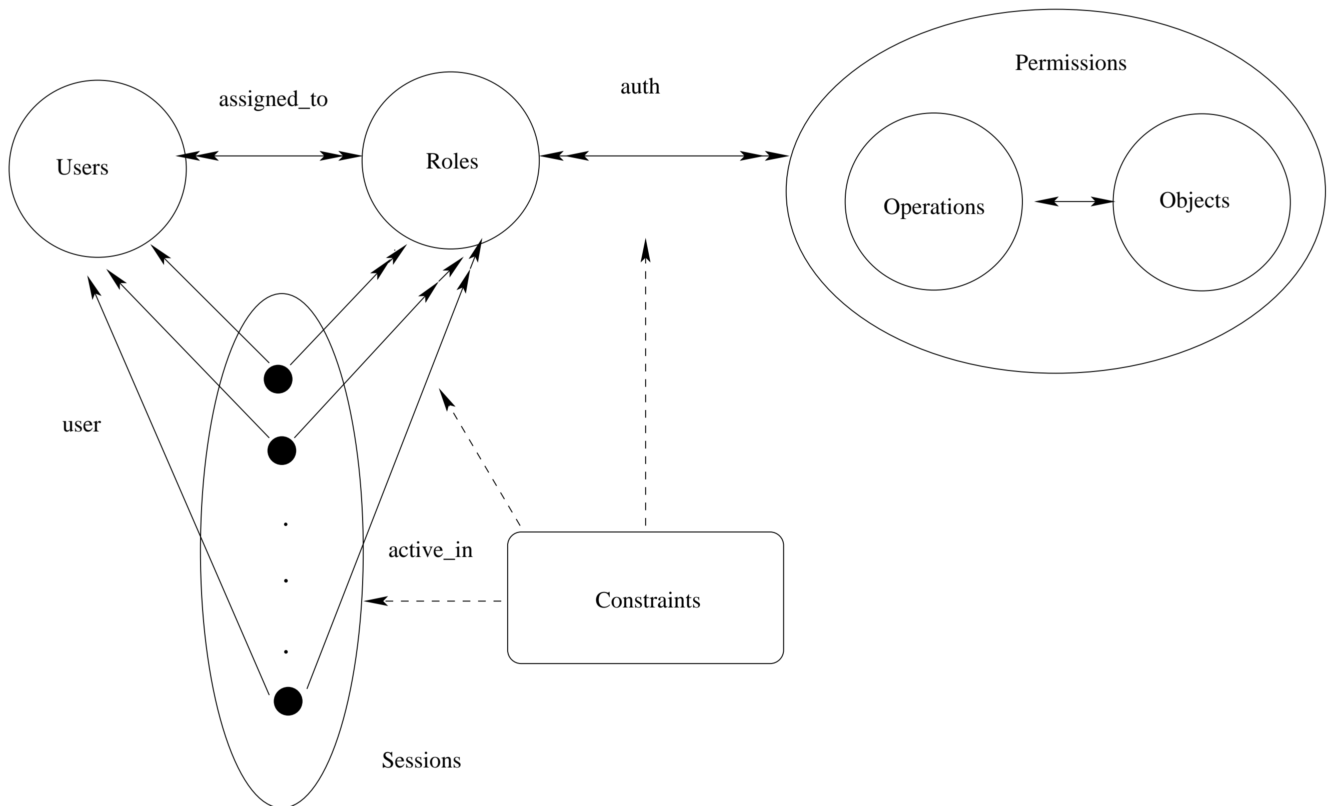

Figure 1. The components of RBAC96

Our name	[10]	[9], [1]	Meaning
Users	Users	Users	human being or an autonomous agent
Sessions	Subjects	Sessions	entities through which a user acts
Roles	Roles	Roles	job function or job title
Operations	Operations	Operations	(permitted or non-permitted) actions on objects
Objects	Objects	Objects	passive entities
user	subject_user	user	unique user of a session
auth	auth	PA	authorization of roles (to execute operations)
assigned_to	role_members	UA	static assignment between users and roles
active_in	subject_roles	roles	dynamic activation of roles in sessions

Figure 2. Table of terminology.

spec RBAC =
sorts $Users, Sessions, Roles, Operations, Objects$
op $user : Sessions \to Users$;
preds $__assigned_to__ : Users \times Roles$;
$auth : Roles \times Operations \times Objects$;
$__active_in__ : Roles \times Sessions$;
forall $r : Roles;\ s : Sessions$
• $r\ active_in\ s \Rightarrow user(s)\ assigned_to\ r$
end

Figure 3. RBAC, formalized within first-order logic.

2. Limitations of the RBAC model

Practical applications of RBAC often need dynamic SOD, see Simon and Zurko [19, 20] and Nash and Poland [15]. We cite an example of a purchasing department which processes invoices for received goods from [15]:

> *Let us assume that there are three types of transaction involved in processing an invoice:*
>
> 1. *Recording the arrival of an invoice and the information recorded on it;*
> 2. *Verifying that the goods in question have been received and that the details of price etc. match those agreed with the supplier;*
> 3. *Authorizing the payment of the supplier.*
>
> *We can also postulate three distinct types of user:*
>
> 1. *Data entry clerks, who enter the information on invoices into the system;*
> 2. *Purchasing officers, who verify the validity of invoices;*
> 3. *Supervisors, who authorize payments.*
>
> *If we allocate fixed users to fixed types of transaction, then data entry clerks, and only data entry clerks, can record the arrival of invoices; purchasing officers, and only purchasing officers, can perform verification transactions; and supervisors, and only supervisors, can authorize payments. (...) This is not what usually happens in the real world.*
>
> *Although some roles are restricted to certain transactions (for example, a data entry clerk is never permitted to authorize payments), it would be normal for an otherwise unoccupied purchasing officer to enter details of invoices if there is a backlog of unprocessed incoming paper, or for a supervisor to handle the verification of the details of a particularly complex invoice. What* never *happens is that the* same *purchasing officer verifies an invoice that he entered into the system, or the* same *supervisor who verifies an invoice is permitted to authorize payment against it.*

Now [18, 2, 9, 16] claim that RBAC covers dynamic separation of duties. [1] cites [15] when formulating his security policies using RBAC, but for the object-based case (which is needed here), he only covers *static* SOD. [7] even explicitly introduces an example like the above one, with a formalization using RBAC. However, the latter is inadequate since too static. Indeed, an important observation not having been made in the literature so far is the following:

Claim For the Nash and Poland example, dynamic separation of duties cannot be expressed in the standard RBAC model. This is even not possible if the role structure may be refined, as long as the number of roles is smaller than the number of permissions (a practical reasonable assumption).

The claim can be seen as follows. We refer to the above practical example, assuming that there are three operations (enter, verify, authorize) and three roles (clerk, officer, supervisor), such that the clerk role is authorized to execute the enter operation on any object, while the officer (resp. supervisor) role additionally is authorized to execute the verify (resp. verify and authorize) operations. Hence, supervisors are authorized to execute (potentially) any operation on any object. The intended security policy is: No user should be authorized to execute actually more than one operation on a given object. Now since authorization is connected to roles, not users, and roles can be activated in sessions, this has to be rephrased as: no user should activate roles r_1, r_2 in sessions s_1, s_2 such that r_1 and r_2 can execute two distinct operations on some object. This is formalized as

$$\begin{array}{l}\neg\exists r_1, r_2 : Roles;\ s_1, s_2 : Sessions;\\ \quad op_1, op_2 : Operations;\ obj : Objects\,.\\ \quad\quad op_1 \neq op_2 \wedge r_1\ active_in\ s_1 \wedge r_2\ active_in\ s_2\\ \quad\quad\quad \wedge\ user(s_1) = user(s_2)\\ \quad\quad\quad \wedge\ auth(r_1, op_1, obj) \wedge auth(r_2, op_2, obj)\end{array}$$

But for the above example, this means that no user could ever activate the supervisor role in a session s, since this role (taken for both r_1 and r_2) would authorize him to execute all three possible operations (taking $s = s_1 = s_2$).

Now assume that the role structure is refined somehow. Let u be some supervisor user. Since u is free to execute (potentially, not actually) any operation on any object, for each object obj, there must be three roles $r_{obj,enter}$,

$r_{obj,verify}$ and $r_{obj,authorize}$, being assigned to u and authorized to execute the three operations. Since the number of roles is smaller than the number of permissions, there must be $(obj_1, op_1) \neq (obj_2, op_2)$ with $r_{obj_1,op_1} = r_{obj_2,op_2}$. If $obj_1 = obj_2$, then $op_1 \neq op_2$, and u may execute two distinct operations on $obj_1 = obj_2$. If $obj_1 \neq obj_2$, choose op_0 with $op_0 \neq op_1$. Now u is free to execute (actually, not only potentially) op_0 on obj_1 and op_2 on obj_2, and therefore may activate both r_{obj_1,op_0} and r_{obj_2,op_2}. But since the latter is equal to r_{obj_1,op_1}, u may execute both op_0 and op_1 on obj_1. Hence, in both cases, the security policy is violated.

The deeper insight is that the concept of session, while introducing a certain degree of dynamics (opposed to a formulation using the entirely static $assigned_to$ predicate), is still too weak to express the distinction between the potential to execute an operation and the fact that an operation is actually executed. Indeed, some further dynamic concept (beyond the notion of session) is needed to model the latter. In fact, [17] stresses the importance of history. Gligor et al. [10] take this point seriously and formalize history via traces of states, but end up with rather complex formulas explicitly talking about states. [4] also introduce temporal aspects into RBAC. Their model has considerably complexity, because it talks about various forms of time intervals and about events triggering the activation of roles. While this is an important topic, our main topic, namely separation of duties, is not treated explicitly.

3. Extended RBAC

We now introduce *extended* RBAC as a rather *simple*, first-order based extension of the RBAC model that allows expressing realistic dynamic separation of duties policies. Surprisingly, this can be done without introducing a notion of state. We overcome the abovementioned limitations of RBAC by adding, besides the sessions, a further dynamic component tracking the *actually executed operations*. Extended RBAC will thus allow a correct specification of *dynamic* separation of duties while still being as concise as RBAC (and thus avoiding the complexity of the formalization in [10]).

The new component of extended RBAC is the predicate $exec$ tracing the operations performed: $exec(s, op, obj)$ means that session s executes (or has executed, or will execute) operation op on object obj. Of course, since FOL is a static logic, the $exec$ predicate does not depend on time, but it just collects all the executions performed during a session. (Temporal aspects will be covered in Section 5 below).

The axiom states that a session s may execute an operation (on an object) only if some role r that is authorized to perform the operation is active in s.

Fig. 5 contains definitions of some derived notions that

spec EXTENDEDRBAC = RBAC **then**
  pred $exec : Sessions \times Operations \times Objects$
  forall $s : Sessions;\ op : Operations;\ obj : Objects$
  • $exec(s, op, obj) \Rightarrow$
    $\exists r : Roles\ .\ r\ active_in\ s$
      $\wedge auth(r, op, obj)$
end

Figure 4. Extended RBAC, formalized within first-order logic.

spec RICHRBAC = RBAC **then**
  preds $__active_for__ : Roles \times Users;$
    $exec : Users \times Operations \times Objects$
  forall $r : Roles;\ u : Users;$
    $op : Operations;\ obj : Objects$
  • $r\ active_for\ u \Leftrightarrow$
    $\exists s : Sessions\ .\ user(s) = u$
      $\wedge r\ active_in\ s$
  • $exec(u, op, obj) \Leftrightarrow$
    $\exists s : Sessions\ .\ user(s) = u$
      $\wedge exec(s, op, obj)$
end

Figure 5. Derived predicates for extended RBAC.

will lead to more succinct formulations of security policies in some cases. If needed, the derived notions can be eliminated by replacing them with their definitions.

4. Separation of duty in extended RBAC

We now specify a bunch of security policies, following the formalizations in [1] and [10]. We overcome the problems of [1] who uses ordinary RBAC and thus fails to specify truly dynamic RBAC (as argued above). While the formalizations in [10] do not have this problem, they add considerable complexity since they explicitly talk about states and execution history. We here remain in a stateless world, but still can capture most of the policies of [10] quite adequately. Still, states may be needed when the order of execution is important. This will be subject of Section 5.

We follow [10] in relativizing all the security policies w.r.t. a given set $RoleSet$ of roles (regarded as conflicting roles), and given sets $OpSet$ and $ObjSet$ of operations and objects, resp. We freely will use these as sorts for variables. This can be understood as syntactic sugar for membership in a predicate on the sorts $Roles$, $Operations$ or $Objects$, respectively. An alternative (though more complicated) way is to use subsorts, e.g. as in CASL [3].

Static Separation of Duties (SSoD) At most one conflicting role can be assigned to a given user.

$$\begin{array}{l}\textbf{pred } SSoD \Leftrightarrow \\ [\forall r_1, r_2 : RoleSet;\ u : Users\ .\ r_1 \neq r_2 \Rightarrow \\ \qquad \neg(u\ assigned_to\ r_1 \wedge u\ assigned_to\ r_2)]\end{array}$$

It is also possible to express this equivalently in a positive way:

$$\begin{array}{l}\textbf{pred } SSoD \Leftrightarrow \\ [\forall r_1, r_2 : RoleSet;\ u : Users\ . \\ u\ assigned_to\ r_1 \wedge u\ assigned_to\ r_2 \Rightarrow r_1 = r_2]\end{array}$$

Nevertheless, in the sequel we will stick to the negative way, since this conforms with [10].

Simple Dynamic Separation of Duties (SDSoD) Two distinct conflicting roles cannot be activated for one and the same user (although they can be assigned to one and the same user).

$$\begin{array}{l}\textbf{pred } DSoD \Leftrightarrow \\ [\forall r_1, r_2 : RoleSet;\ u : Users\ .\ r_1 \neq r_2 \Rightarrow \\ \qquad \neg(r_1\ active_for\ u \wedge r_2\ active_for\ u)]\end{array}$$

Object-based Dynamic Separation of Duty (ObjDSoD) For a given object, a user may perform at most one operation on it.

Note that while the policies discussed so far can also be formalized in ordinary RBAC, the present policy cannot: it inherently needs the execution predicate (cf. the discussion in section 2).

$$\begin{array}{l}\textbf{pred } ObjDSoD \Leftrightarrow \\ [\forall u : Users;\ op, op' : OpSet;\ obj : Object\ . \\ \quad op \neq op' \wedge exec(u, op, obj) \Rightarrow \\ \qquad \neg exec(u, op', obj)]\end{array}$$

This also is the policy needed for the invoice example mentioned in section 2.

Operational Dynamic Separation of Duty (OpDSoD) The roles *active for* (and not just assigned to) a user are not allowed to perform all the operations of $OpSet$ (if more than one), regardless of the target object. Again, we have found a first-order equivalent to the second-order description in [10].

$$\begin{array}{l}\textbf{pred } ROpSSoD \Leftrightarrow \\ [(\exists op_1, op_2 : OpSet\ .\ op_1 \neq op_2) \Rightarrow \\ \ \forall u : Users\ .\ \exists op : OpSet\ . \\ \quad \forall r : RoleSet;\ obj : ObjSet\ . \\ \qquad \neg(r\ active_for\ u \wedge auth(r, op, obj))\ ]\end{array}$$

History-Based Dynamic Separation of Duty (HDSoD) One and the same user cannot perform all the operations in $OpSet$ on the same object of $ObjSet$. This is another policy that cannot be expressed in RBAC without the execution predicate.

$$\begin{array}{l}\textbf{pred } HDSoD \Leftrightarrow \\ [(\exists op_1, op_2 : OpSet\ .\ op_1 \neq op_2) \Rightarrow \\ \ \forall u : Users;\ r : RoleSet;\ obj : ObjSet\ . \\ \qquad \exists op : OpSet\ .\ \neg exec(u, op, obj)\ ]\end{array}$$

5. Temporal-logic RBAC

The approach of the previous section has two shortcomings. Firstly, it is not possible to talk about the order of executions. For example, a supervisor should authorize payment only *after* the invoice has been entered and verified. Secondly, separation of duties is only formulated in terms of *actual executions*, but not in terms of *authorization*. That is, actual execution of operations is not only constrained by authorization, but also by the security policy. The approach

in [10] has the same drawback. We propose instead to let the policy influence the authorization directly, and the actual execution only indirectly via the authorization.

We now introduce *temporal* RBAC as a formalism that combines RBAC with states. This allows for talking about things such as the execution history or order of executions, as in [10], while still being much simpler than [10]. Temporal-logic RBAC is based on temporal first-order logic, a logic that has been intensively studied in the literature [12, 13] and comes with standard tools, e.g. [11].

A temporal first-order signature consists of a set of sorts, a set of function symbols and a set of predicate symbols (each symbol coming with a string of argument sorts and, for function symbols, a result sort). Function and predicate symbols are partitioned into *rigid* and *flexible* symbols: the former do not change over time, while the latter may vary. Models live over discrete time, indexed by the natural numbers as time steps. They interpret the sorts with (time-independent) carrier sets, rigid function and predicate symbols with time-independent functions and predicates of appropriate types, and flexible function and predicate symbols with families of functions and predicates, where the families are indexed by natural numbers.

Sentences are the usual first-order sentences built from equations, predicate applications and logical connectives and quantifiers $\forall$, $\exists$. Additionally, we have the modalities $\Box$ (always in the future), $\Diamond$ (sometimes in the future) and $\bigcirc$ (in the next step). The corresponding past modalities are $\boxminus$, $\diamondminus$ and $\ominus$. Satisfaction is defined inductively for a given time step, where the modalities allow referring to other time steps. A sentence is satisfied in a model if it is satisfied in the time step zero.

In Fig. 6, we extend the specification of Fig. 3 to the temporal case. The function $user$ and the static predicate $auth$ (applicable to roles) are rigid (i.e. do not depend on the state), while the dynamic predicate $auth$ (applicable to users), as well as the predicates $active_in$ and $exec$ are flexible (i.e. do depend on the state). Hence, $exec$ now traces the operations performed: $exec(s, op, obj)$ means that session s executes operation op on object obj in the present (implicit) state. The flexible $auth$ for users expresses which operation the user is authorized to perform; indeed, this depends both on the static authorizations of the user's roles as well as on dynamic security policies such as SOD. The axioms of the specification constrain the user behaviour as expressed by the predicates $active_in$ and $exec$: their holding must imply the holding of appropriate role assignment and authorization predicates. Also, the flexible authorization predicate $auth$ is constrained, and it is expected that domain-specific security policies like SOD, when added to the specification, constrain $auth$ even further (cf. the next section). Once this is done, it might be useful to add a non-monotonic "closed world assumption" stating that

spec TEMPORALRBAC =
  sorts $Users, Sessions, Roles, Operations, Objects$
  rigid op $user : Sessions \to Users;$
  rigid pred $auth : Roles \times Operations \times Objects$
  flexible preds $__assigned_to__ : Users \times Roles;$
    $auth : Users \times Operations \times Objects;$
    $__active_in__ : Roles \times Sessions;$
    $exec : Sessions \times Operations \times Objects$
  forall $r : Roles;\ s : Sessions;$
    $op : Operations;\ obj : Objects$
  - $\Box((\Diamond r\ active_in\ s) \Rightarrow user(s)\ assigned_to\ r)$
  - $\Box(auth(u, op, obj) \Rightarrow \exists s : Sessions;\ r : Roles\ .\ user(s) = u \land r\ active_in\ s \land auth(r, op, obj))$
  - $\Box(exec(s, op, obj) \Rightarrow auth(user(s), op, obj))$

end

spec RICHTEMPORALRBAC = TEMPORALRBAC
  then
  flexible preds $__active_for__ : Roles \times Users;$
    $exec : Users \times Operations \times Objects$
  forall $r : Roles;\ u : Users;$
    $op : Operations;\ obj : Objects$
  - $\Box(r\ active_for\ u \Leftrightarrow \exists s : Sessions\ .\ user(s) = u \land r\ active_in\ s)$
  - $\Box(exec(u, op, obj) \Leftrightarrow \exists s : Sessions\ .\ user(s) = u \land exec(s, op, obj))$

end

Figure 6. Temporal-logic RBAC, formalized within temporal first-order logic.

$auth$ holds as much as possible without sacrificing the constraints. While this in general would go beyond temporal first-order logic, for specific policies, it does not.

6. Separation of duty in Temporal-logic RBAC

We can now reformulate some security policies from Section 4 for the state-based case. Besides inserting the appropriate temporal modalities, we also have replaced $exec$ with $auth$ at several places.

Dynamic Separation of Duties (DSoD)

$$\mathbf{pred}\ DSoD \Leftrightarrow$$
$$[\forall r_1, r_2 : RoleSet;\ u : Users\ .\ r_1 \neq r_2 \Rightarrow$$
$$\Box\neg(r_1\ active_for\ u \wedge r_2\ active_for\ u)]$$

Object-based Dynamic Separation of Duty (ObjDSoD)
We now can reformulate Object-based Dynamic Separation of Duty (ObjDSoD) in terms of authorization, not just execution:

$$\mathbf{pred}\ ObjDSoD \Leftrightarrow$$
$$[\forall u : Users;\ op, op' : OpSet;\ obj : Object\ .$$
$$\Box(op \neq op' \wedge exec(u, op, obj) \Rightarrow$$
$$\Box\neg auth(u, op', obj))]$$

Operational Dynamic Separation of Duty (OpDSoD)

$$\mathbf{pred}\ ROpSSoD \Leftrightarrow$$
$$[(\exists op_1, op_2 : OpSet\ .\ op_1 \neq op_2) \Rightarrow$$
$$\forall u : Users\ .\ \exists op : OpSet\ .$$
$$\forall r : RoleSet;\ obj : ObjSet\ .$$
$$\Box\ \neg(r\ active_for\ u \wedge auth(r, op, obj))\ ]$$

History-Based Dynamic Separation of Duty (HDSoD)

$$\mathbf{pred}\ HDSoD \Leftrightarrow$$
$$[(\forall u : Users;\ obj : ObjSet;\ op : OpSet\ .$$
$$\Box(\ (\forall op' : OpSet\ .\ op' \neq op \Rightarrow$$
$$\Diamond\!\!\!-\, exec(u, op', obj))$$
$$\Rightarrow \neg auth(u, op, obj)\ )\ ]$$

7. The Nash and Poland Example in Temporal-logic RBAC

The invoice example can now be refined such that a certain order of execution of the operations is imposed (cf. Fig. 7).

The axioms state that a *verify* operation can only happen after an *enter* operation, and an *authorize* operation can only happen after a *verify* operation. Thus we have enforced ObjDSoD and the consecutive execution of the given operations in the desired chronological order (enter, verify, authorize) by the means of our logic. From this example we see that sometimes the greater expressiveness of temporal-logic RBAC is needed.

spec INVOICE = TEMPORALRBAC **then**
  rigid ops $Clerk, Officer, Supervisor : Roles;$
    $enter, verify, authorize : Operations$
  forall $u : Users;\ op_1, op_2 : Operations;$
    $invoice : Object$
  - $auth(Clerk, enter, invoice)$
  - $auth(Officer, enter, invoice)$
  - $auth(Supervisor, enter, invoice)$
  - $auth(Officer, verify, invoice)$
  - $auth(Supervisor, verify, invoice)$
  - $auth(Supervisor, authorize, invoice)$
  - $\Box(op_1 \neq op_2 \wedge exec(u, op_1, invoice) \Rightarrow \Box\neg auth(u, op_2, invoice))$
  - $\Box(auth(u, verify, invoice) \Rightarrow \exists s : Sessions\ .\ \circleddash\Diamond exec(s, enter, invoice))$
  - $\Box(auth(u, authorize, invoice) \Rightarrow \exists s : Sessions\ .\ \circleddash\Diamond exec(s, verify, invoice))$

end

Figure 7. The invoice example with execution orders specified.

8. Conclusion

We have argued that the standard formalisms for role-based access control (RBAC) are not adequate to express realistic dynamic separation of duty (SOD) policies. The formalisms proposed in the literature [10] that can handle this properly add considerable complexity to the RBAC model. We have shown how to eat the cake and have it, too: with a rather simple extension, many realistic SOD policies can be expressed. In a first step, we have added an execution predicate to the RBAC model. This uses standard many-sorted first-order logic as background formalism, easing the formal verification of systems using standard tools. However, for realistic practical examples, often the order of executions is crucial. Here, temporal logic can be used. We show how to formulate dynamic separation of duties with temporal logic and illustrate this with an example coming from a standard practical application. We claim that our formalization is the first one that is both adequate to the problem domain and simple in the technical details.

This simplicity should also ease the use of standard tools, as e.g. the Stanford Temporal Prover STeP ([11]). We intend to use this tool for a task that surprisingly has not been tackled so far to our knowledge: the verification of algorithms against dynamic separation of duty security policies.

References

[1] G.-J. Ahn. *The RCL 2000 language for specifying role-based authorization constraints*. PhD thesis, George Mason University, Fairfax, Virginia, 1999.

[2] G.-J. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207–226, Nov. 2000.

[3] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: the Common Algebraic Specification Language. *Theoret. Comput. Sci.*, 286:153–196, 2002.

[4] E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3):191–223, Aug. 2001.

[5] A. D. Brucker, F. Rittinger, and B. Wolff. The CVS-server case study: A formalized security architecture. In D. Haneberg, G. Schellhorn, and W. Reif, editors, *FM-TOOLS 2002*, number 2002–11 in Technical Report, pages 47–52. Augsburg, July 2002.

[6] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[7] C. Eckert. *IT-Sicherheit: Konzepte, Verfahren, Protokolle*. R. Oldenbourg Verlag, 2001.

[8] D. Ferraiolo, D. Gilbert, and N. Lynch. An examination of federal and commercial access control policy needs. In *Proc. of the NIST-NCSC Nat. (U.S.) Comp. Security Conference*, pages 107–116, 1993.

[9] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramoli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[10] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *1998 IEEE Symposium on Security and Privacy (SSP '98)*, pages 172–185, Washington - Brussels - Tokyo, May 1998. IEEE.

[11] Z. Manna, N. Bjørner, A. Browne, E. Chang, M. Colón, L. de Alfaro, H. Devarajan, A. Kapur, J. Lee, H. Sipma, and T. E. Uribe. STeP: The stanford temporal prover. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 793–794. Springer-Verlag, 1995.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer-Verlag, 1992.

[13] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[14] K. Meinke and J. V. T. (eds.). *Many-sorted logic and its applications*. Wiley, Chichester, 1993.

[15] M. J. Nash and K. R. Poland. Some conundrums concerning separation of duty. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 201–207, 1990.

[16] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security*, 2(1):3–33, Feb. 1999.

[17] R. Sandhu. Transaction control expressions for separation of duties. *Fourth Aerospace Computer Security Applications Conference, Orlando*, pages 282–286, 1988.

[18] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb. 1996.

[19] R. Simon and M. Zurko. Separation of duty in role-based environments. In *10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 183–194, Washington - Brussels - Tokyo, June 1997. IEEE.

[20] M. E. Zurko, R. Simon, and T. Sanfilippo. A user-centered, modular authorization service built on an RBAC foundation. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 57–71, Oakland, CA, May 1999. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.