Role Updating for Assignments

Jinwei Hu†, Yan Zhang†, Ruixuan Li†; and Zhengding Lu†
†Intelligent and Distributing Computing Laboratory, School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan 430074, China
‡Intelligent Systems Laboratory, School of Computing and Mathematics
University of Western Sydney, Sydney 1797, Australia
{jwhu,rxli,zdlu}@hust.edu.cn yan@scm.uws.edu.au

ABSTRACT
The role-based access control (RBAC) has significantly simplified the management of users and permissions in computing systems. In dynamic environments, systems are usually undergoing changes, whereas the associated user-role, role-role and role-permission relations need to be updated accordingly in order to reflect the systems’ evolutions. However, such updating process is generally complicated as the resulting system state is expected to meet necessary constraints. This paper presents an approach for assisting administrators with the update task: using this approach, it is possible to check, in an automatic way, whether a required update is achievable or not, and if so, a reference model will be produced. In light of this model, administrators could fulfill the changes to RBAC systems. We propose a formalization of the update approach, investigate its properties, and develop an updating algorithm based on model checking techniques. Our experimental results demonstrate the effectiveness of our approach.

Categories and Subject Descriptors

General Terms
Management, Security, Verification, Performance

Keywords
RBAC, Role Updating, Role Engineering, Model Checking

1. INTRODUCTION
Role-based access control (RBAC) is an effective mechanism for simplifying the administration of users and permissions in computing systems [11, 26]. In RBAC systems, users are associated with roles such as manager and employee, and a role in turn is defined as a set of permissions; users acquire permissions via roles. Essentially, an RBAC system manages three relations: user-role relation, role-role relation, and role-permission relation. Owing to RBAC’s advantages (e.g., being policy-neutral and the ability to support a wide range of access control requirements) it is widely supported in commercial operating systems and database systems.

However, the administration of large RBAC systems is a complex and challenging task. As organizations undergo changes, the definition of roles (in terms of permissions) may need updating to reflect those changes. We refer to the updating of user-role, role-role, and role-permission relations as role updating. Role updating is a key component of role maintenance in role life-cycle [16], and takes a great proportion of the total cost of role maintenance [22].

Despite many convenient RBAC administration models (e.g., [8, 18, 25]) at our disposition, role updating is generally difficult and error-prone, because usually the resulting state is expected to meet various constraints. For example, updating is supposed not to unexpectedly change users’ role set or permission set and thus hinder tasks being performed in the system; nor should it assign a student role to a user who is a professor in real-life, even though her permission set remains the same.

Given a high-level update request, the trial-and-error approach is work-intensive, inefficient and, when the request is not achievable at all, very frustrating. An updating strategy, if exists, may involve several administrative actions. The situation becomes worse as the RBAC systems grow large. Hence, a tool facilitating role updating is desired, as is also motivated by the following scenarios.

Repairing. Misconfigurations in access control systems can result in severe consequences [3]. As such, correcting misconfigurations is essential to systems’ usability and security. It consists of two parts: identifying misconfigurations and changing access control policies. While the former has been studied, how to change access control policies is still an open problem, even though it may be clear who should implement the changes. As stated in [3]: “How the user elects to modify the policy is orthogonal to our work; it could entail changing an access-control list on a server or ...”. Therefore, role updating might be needed under the circumstance that misconfigurations in RBAC systems are detected.

User-permission assignments. In task-based systems, each task is associated with a set of permissions [13].† The set of permissions should be assigned to a set of users so that the associated task could be accomplished. In systems where RBAC is deployed, the user-permission assignments are achieved via roles [2]. Considering the variety of tasks, it is likely that any combinations of roles fail to enable exactly the needed user-permission assignments for a task. In this case, to support the task, one may be willing to change the

†Permissions for tasks may be augmented with temporal constraints. One essential problem is to assign permissions.
current RBAC state, to some extent such as with the constraint that previous task assignments are not affected. Furthermore, the permissions required by users and tasks are not static. For one thing, modern enterprises and organizations need to adapt to constantly changing environments and requirements [22]. For another, it could happen that a group $U$ of users are performing a task with a set $P$ of permissions, and that it turns out that an extra permission $p_0 \notin P$ is a must at the late stage of the task; or the administrator may recognize that $P$ contains more privileges than strictly necessary for the task, she may try to revoke redundant permissions from $U$ [13].

To help system security officers (SSOs) understand and manage RBAC policies, various RBAC policy analysis tools have been invented [3, 14, 15, 19, 28, 31, 32, 36]. However, little effort has been devoted to answering what if the current RBAC state fails to meet a requirement. One may consider role updating as an unpleasant experience and thus avoid it. Also, it is because the updating is not straightforward, an automated assistant tool has some appeal.

In this paper, we investigate the role updating problem (RUP), with a bias on the update of role-permission relation, and provide an automated role update tool (Route). Note that Route is not designed to automatically change RBAC policies, but to, given a high-level update request, check whether the request is achievable and, if so, present a reference model, in light of which SSOs could implement changes to RBAC systems. Specifically, we make the following contributions:

- We formally define a basic version of RUP and a notion of update constraint schema, which is able to express a wide range of constraints on updates. We show the intractability of RUP.
- We present a set of reductions of RUP. The reductions cannot only be used by Route, but also simplify RUP itself.
- We devise an automated tool Route for RUP based on model checking techniques, for which there exist mature tools that have already been proven to work well in practice.
- Finally, we undertake a set of experiments, which illustrate the effectiveness and efficiency of Route, considering the use-pattern and frequency of role updating.

The remaining of the paper is organized as follows. Section 2 gives the preliminaries of the RBAC model used in this paper. A motivation example is presented in Section 3. In Section 4, we define RUP, show its complexity, and give the basic approach. The reductions of RUP are presented in Section 5. Experiment results are reported in Section 6. Related work is discussed in Section 7. We conclude in Section 8. In this paper, we write SSOs and administrators interchangeably; and a singleton set $\{s\}$ is sometimes written as $s$. Except for Theorem 2, proofs are given in [12].

2. PRELIMINARIES

The RBAC model used in this paper is defined as follows. An RBAC state $\gamma$ is a tuple $\langle \gamma.U, \gamma.R, \gamma.P, \gamma.U.A, \gamma.P.A \rangle$, where $\gamma.U$ is a set of users, $\gamma.R$ is a set of roles, $\gamma.P$ is a set of permissions, $\gamma.U.A \subseteq \gamma.U \times \gamma.R$ is user-role relation, and $\gamma.P.A \subseteq \gamma.R \times \gamma.P$ is role-permission relation.

Given a state $\gamma$, we define several convenient functions. Function $\text{users}_{\gamma}.: \gamma.U \mapsto \{u \in \gamma.U \mid (u, r) \in \gamma.U.A\}$; for $x \subseteq \gamma.R$, $\text{users}_{\gamma}.[x] = \bigcup_{r \in x} \{u \in \gamma.U \mid (u, r) \in \gamma.U.A\}$; for $x \subseteq \gamma.P$, $\text{users}_{\gamma}.[x] = \text{users}_{\gamma.}[r][x]$.

Function $\text{roles}_{\gamma}.: \gamma.U \mapsto \gamma.R \mapsto \gamma.R$ returns the set of roles that are related to users and permissions: for $x \subseteq \gamma.U$, $\text{roles}_{\gamma}.[x] = \bigcup_{u \in x} \{r \mid (u, r) \in \gamma.U.A\}$; for $x \subseteq \gamma.R$, $\text{roles}_{\gamma}.[x] = \bigcup_{r \in x} \{u \mid (u, r) \in \gamma.U.A\}$; for $x \subseteq \gamma.P$, $\text{roles}_{\gamma}.[p][x] = \bigcup_{r \mid r \in x} \{u \mid (u, r) \in \gamma.U.A\}$.

We present a set of reductions of $\gamma$, which are reported in Section 6. Related work is discussed in Section 7. We conclude in Section 8. In this paper, we write SSOs and administrators interchangeably; and a singleton set $\{s\}$ is sometimes written as $s$. Except for Theorem 2, proofs are given in [12].

3. A MOTIVATION EXAMPLE

Figure 1 shows an example RBAC state $\gamma_{ex}$. Suppose that an SSO needs to update $\gamma_{ex}$ in response to the following requests.

Repairing. Suppose that a misconfiguration that the user $u_2$ should have been able to perform $p_2$ is detected. The SSO is informed of this problem and asked to take measures to grant $u_2$ the permission $p_2$. The question is how the SSO may accomplish this, while keeping other users' role sets and permission sets unchanged? It seems overreacting to assign $p_2$ with $r_1$ or with $r_3$, because that would incidentally grant $p_2$ to $u_1$ or $u_3$.

Permission assignments. Suppose that a task $\tau$ requires the permissions $P_1 = \{p_5, p_6, p_9\}$. Currently in $\gamma_{ex}$, there does not exist a set of roles whose permission set is exactly $P_1$. If $\tau$ is important and all permissions in $P_1$ are indispensable, an update request, that $P_1$ be assigned via roles but with users' role sets and permission sets unchanged, may be issued in hope of enabling task $\tau$.

This type of requests could also happen in the context of inter-domain role mappings: a set of permissions (e.g. $\{p_5, p_6, p_9\}$) are to be shared via roles, but the underlying state could not enable the exact set of permissions $\{4, 9, 29, 30, 35, 38\}$.

In both cases, the SSO needs to either find a satisfactory update or determine the request is not satisfiable at all. However, even with this toy example, it is not straightforward to make decisions.

![Figure 1: A running example RBAC state $\gamma_{ex}$](image-url)
4. PROBLEM DEFINITION

4.1 Constraints on updates

Given the current state γ, consider an update request that a set \( P \) of permissions be assigned to users via roles. If new roles can be defined for \( P \), this question becomes trivial. Hence, we look at more difficult role updating problem in RBAC systems where no new roles can be created. We believe this is more common; because new roles should be introduced with discretion, but not merely in response to such permission assignment requests. For any state χ obtained after updating γ, we may want to impose this constraint:

\[
χ.R = γ.R.
\] (1)

The effects of an update should be confined to a certain set \( U \) of users. In other words, updates are supposed to be transparent to users outside \( U \) in the sense that their permission sets in χ remain as in γ. In this way, the tasks that are being performed in the system would not be interfered by updating. Put formally, the constraint is:

\[
\text{perms}_χ[u] = \text{perms}_γ[u], \quad \text{for all } u \in γ.U \setminus U.
\] (2)

Additionally, to make updates rational, the following constraint seems necessary.

\[
\text{roles}_χ[u] = \text{roles}_γ[u], \quad \text{for all } u \in γ.U \setminus U.
\] (3)

Constraint (3) keeps users’ role sets unchanged. This is motivated by the following observations: (1) User-role assignments are business-driven; users’ role memberships are determined by their attributes, jobs, titles, and etc. (2) From the user-experience viewpoint, a staff in a university may be reluctant to take a student role even though she is entitled to the same privileges. Further, a user often activates only a subset of her roles for a task; it would be obtrusive for a professor to activate both the student role and the secretary role to finish what she can do with a professor role before, had an update adjusted her role assignments.

Motivated by these observations, we define a notion of Update Constraint Schema, which generalizes constraints (1), (2), and (3).

Definition 1. (Update Constraint Schema - UCS) Given a state γ, a UCS of γ is a tuple \( π^γ = (π^γ.U, π^γ.th) \), where

1. \( π^γ.U ⊆ γ.U \), and
2. \( π^γ.th : π^γ.U \to 2^{π^γ.P} \) such that, for any \( u \in π^γ.U \),

\[
π^γ.th[u] ⊆ \text{perms}_γ[u] \quad (\text{th is short for threshold}).
\]

When γ is clear from context, we omit the superscript γ.

When SSOs are specifying \( π, π.U \) often contains those users for whom the SSOs are not responsible so that the SSOs have to ensure that the potential update does not affect those users (by setting \( π.th[u] = \text{perms}_γ[u] \), and/or those users whose permissions are designated by the SSOs and vary within a range (i.e., the lower bound is \( π.th[u] \) and the upper bound is \( \text{perms}_γ[u] \)). The set \( γ.U \setminus π.U \) is a set of users whose current role sets and permission sets in γ are neglected by updates; updates may change their role-assignments and permission-assignments.

Definition 2. (π-compatibility) Given γ, π, and an RBAC state χ \( \in \text{space}(γ) \), we say χ is π-compatible if the following conditions are met.

\[
\text{COND-U-R} \quad \text{for any } u \in π.U, \text{roles}_χ[u] = \text{roles}_γ[u], \quad \text{and}
\]

\[
\text{COND-U-P} \quad \text{for any } u \in π.U, \pi.th[u] ⊆ \text{perms}_γ[u] \subseteq \text{perms}_χ[u].
\]

Note that only RBAC states in space(γ) are considered. That means, we assume that no changes are made to the user set, role set, and permission set. Condition \( \text{COND-U-R} \) requires that the role assignments of any \( u \in π.U \) in \( χ \) remain as in γ, i.e., \( χ.UA ∩ (π.U × χ.R) = γ.UA ∩ (π.U × γ.R) \). This is exactly the same as (3). \( \text{COND-U-R} \) also indicates that we focus on the update of role-permission assignments. In accordance with \( \text{COND-U-P} \), any \( u \in π.U \) must satisfy: u’s permission set varies from \( π.th[u] \) to \( \text{perms}_χ[u] \), both inclusively. This is a generalization of (2). At least, by letting \( π.th[u] = \text{perms}_γ[u] \) so that users permission sets remain the same, SSOs guarantee that tasks assigned to users in π.U progress smoothly after update.

Definition 3. An update request to γ, \( \text{req}(γ, P, π, T) \), comprises the current RBAC state γ, a set of permissions \( P \subseteq γ.P \), a UCS π of γ, and a set of roles \( T \subseteq γ.R \).

Definition 4. Given \( Q = \text{req}(γ, P, π, T) \), we say χ is an update of \( Q \), if χ is π-compatible and there exists a set \( R \subseteq T \) such that \( \text{perms}_χ[R] = P \). Denote the set of updates of \( Q \) as \( \text{upd}(γ, P, π, T) \). We say γ is satisfiable, if \( \text{upd}(γ, P, π, T) \neq \emptyset \).

Definition 5. (Role Updating Problem - RUP) The RUP is to, given \( \text{req}(γ, P, π, T) \), check whether \( \text{upd}(γ, P, π, T) \neq \emptyset \) and, if so, to find one \( u \in \text{upd}(γ, P, π, T) \).

SSOs express update requests by specifying \( \text{req}(γ, P, π, T) \). If it turns out that \( \text{upd}(γ, P, π, T) = \emptyset \), that means the update request cannot be fulfilled without violating π-compatibility. Otherwise, suppose \( χ \in \text{upd}(γ, P, π, T) \) is found, SSOs can make changes to γ in light of χ.

Route might be used when the RBAC state is migrating to an ideal state suggested by role mining techniques; the ideal state may be fine-tuned using Route to adjust assignments. More importantly, Route is able to assist SSOs with administrative works, especially when assigning (or adjusting) permissions to users.

Satisfiability is defined based on π. One most important restriction that π puts is that, only a limited set of permissions may be assigned with a role. Given \( \text{req}(γ, P, π, T) \), \( r \in γ.R \) and \( R \subseteq γ.R \), let

\[
\text{max-perms}_γ(π, r) = \bigcup_{u \in \{\text{users} \mid r \in π.th[u]\}} \text{perms}_γ[u],
\]

and \( \text{max-perms}_γ(π, r) = R \subseteq \bigcup_{r \in \text{max-perms}_γ(π, r)} r \). If \( \gamma, π \) is clear from context, we omit the subscript \( (γ, π) \). max-perms[r] (resp., \( \text{max-perms}(R) \)) is the maximal set of permissions that \( r \) (resp., \( R \)) could possibly be assigned with in γ.

Proposition 1. Given \( \text{req}(γ, P, π, T) \), if \( χ \) is π-compatible, then, for all \( (r, p) \in χ.P.A \), \( p \in \text{max-perms}[r] \).

Supposing \( χ \in \text{upd}(γ, P, π, T) \), π-compatibility is to restrict the difference between γ and χ, which is measured as below.

Definition 6. Given γ and χ such that \( χ \in \text{space}(γ) \), the difference between γ and χ is defined as \( \text{diff}(γ, χ) = (γ.P.A \setminus χ.P.A) \cup (χ.P.A \setminus γ.P.A) \). Note that \( \text{diff}(χ, γ) = \text{diff}(γ, χ) \).

4.2 RUP is intractable

Theorem 2. Given an update request \( \text{req}(γ, P, π, T) \), it is NP-complete to decide its satisfiability.
PROOF. Since elements in $Q = \text{req}(\gamma, P, \pi, T)$ are finitely fixed and, for any $x \in \text{space}(\gamma)$, it takes polynomial time to check if $x \in \text{upd}(\gamma, P, \pi, T)$, a non-deterministic Turing machine can guess an update $\chi$ and verify if it $x \in \text{upd}(\gamma, P, \pi, T)$. Hence, the problem is in NP. To show its NP-hardness, we reduce the known NP-complete problem Monotone SAT to our problem.

Monotone SAT: Given a set $X$ of boolean variables and a collection $C$ of clauses over $X$ where each clause contains either only positive literals or only negative literals, is there a truth assignment of $X$ so that $\bigwedge C$ is true? We call a clause with only positive literals a positive clause, denoted as $c^+$ and otherwise a negative clause, denoted as $c^-$.

Given a monotone SAT instance, construct a $\text{req}(\gamma, P, \pi, T)$ as follows. First we construct $\gamma$. For each clause $c^+ \in C$, create a permission $p_{c^+}$; for each $c^- \in C$, create a permission $p_{c^-}$. Denote $P^+ = \{p_{c^+} \mid c^+ \in C\}$ and $P^- = \{p_{c^-} \mid c^- \in C\}$. Let $\gamma.P = P^+ \cup P^-$. For each $x \in X$, create a corresponding role $r_x$. Let $(r_x, p_{c^+}) \in \gamma.P.A$ if and only if $c^+$ contains the literal $x$ and $(r_x, p_{c^-}) \in \gamma.P.A$ if and only if $c^-$ contains the literal $\neg x$. For each $c^-$, create a user $u_{c^-}$ and let $(u_{c^-}, r_x) \in \gamma.U.A$ if and only if $c^-$ contains $\neg x$. For each $r_x$ create a user $u_x$ and let $(u_x, r_x) \in \gamma.U.A$. Let $P = P^+$. Now we configure $\pi$ by letting

- $\pi.U = \gamma.U$, $T = \gamma.R$,
- for each $u_{c^-}$. $\pi.th[u_{c^-}] = \text{perms}_{[u_{c^-}]}$, and
- for each $u_x$. $\pi.th[u_x] = \text{perms}_{[r_x]} \cap P$.

We now show that monotone SAT is satisfiable if and only if one can update $\text{req}(\gamma, P, \pi, T)$.

Suppose that $\tau$ is a truth assignment that makes $\bigwedge C$ true. Then an update of $\text{req}(\gamma, P, \pi, T)$ consists of: removing all $(r_x, p)$ where $\pi(x) = 1$ and $p \in P^-$. Note that $P = P^+$. Since $\bigwedge C$ is true, all $c^+$ is true. As a result, for any $p \in P$, there exists at least one $x$ such that $\pi(x) = 1$ and $(r_x, p) \in \gamma.P.A$. Then $(r_x, p) \in \gamma.P(A)$ and $\pi(x) = 1$ in the updating state is a role whose permission

set is exactly $P$. Since the monotone SAT formula is satisfied by $\tau$, each positive clause $c^+$ is true under $\tau$; for each $p \in P$, there must exist at least one $r_x$ such that $(r_x, p) \in \gamma.P.A$ and $\pi(x) = 1$; hence after removing all permissions of $r_x$ which are also outside $P$, $p$ can be assigned to users via $r_x$ in the update. $\pi$-compatibility is established as follows. For any user $u_x$ if any permission $p \in P^+$ belongs to $\text{perms}_{[u_x]}$, since the update does not involve permissions in $P^+$, $u_x$ still acquires that permission in the update. Similarly, conditions regarding any user $u_x$ are also satisfied. For any user $u_{c^-}$, consider any $p_{c^-} \in \text{perms}_{[u_{c^-}]}$. Since $c^-$ is true under $\tau$, there exists at least one $x$ in $c^-$

with $\pi(x) = 0$. From the construction of the update, the update does not affect $r_x$; therefore, it holds that $p_{c^-} \in \text{perms}_{[r_x]}$, namely, $u_{c^-}$ is still associated with $p_{c^-} \in r_x$.

On the other hand, suppose that $\chi \in \text{upd}(\gamma, P, \pi, T)$. Construct a truth assignment $\tau$ over $X$: $\tau(x) = 1$ if and only if $\text{perms}_{[r_x]} \subseteq P$, and otherwise, $\tau(x) = 0$. Then $\tau$ can make the formula true for the following reasons. Recall that $P = P^+$. Suppose that $c^+ = x_1 \lor \cdots \lor x_n$ is false under $\tau$; that is, $\tau(x_i) = 0$ for $1 \leq i \leq n$. In light of the definition of $\tau$, it holds that $\text{perms}_{[r_x]} \subseteq P$, for $1 \leq i \leq n$. Hence, from the fact that $\chi \in \text{upd}(\gamma, P, \pi, T)$, there exists $r \in \chi.R$ such that $p_{c^+} \notin \text{perms}_{[r]}$, $\text{perms}_{[r]} \subseteq P$, and $r \notin \{r_{x_1}, \cdots, r_n\}$. However, since $\chi$ is $\pi$-compatible, particularly the effects of users $u_x$, it must hold

that $\text{diff}(\chi, \gamma) \subseteq \gamma.R \times P^-$. But $p_{c^+} \notin \text{perms}_{[r]}$ because $r \notin \{r_{x_1}, \cdots, r_n\}$; as a result, $(r, p_{c^+}) \in \text{diff}(\chi, \gamma)$. Thus, we reach a contradiction. Suppose that $c^- = \neg x_1 \lor \cdots \lor \neg x_n$ is false under $\tau$; that is, $\tau(x_i) = 1$ for $1 \leq i \leq n$. Then according to the definition of $\tau$, $\text{perms}_{[r_x]} \subseteq P$ for $1 \leq i \leq n$. Since $u_{c^-}$ is only assigned with roles $(r_{x_1}, \cdots, r_n)$ and $p_{c^-} \notin P$, it holds that $p_{c^-} \notin \text{perms}_{[u_{c^-}]}$. However, $p_{c^-} \notin \text{perms}_{[u_{c^-}]}$, a contradiction with $\pi$-compatibility (in terms of $\text{COND-U-P}$ for $u_{c^-}$).

4.3 A model checking approach

Route leverages model checking techniques [6] for RUP. The basic idea is as follows. Given $\text{req}(\gamma, P, \pi, T)$, let $\phi$ denote the statement that a (witness) user could acquire $\mathcal{P}$ via roles in $T$; we ask if $\neg \phi$ is always true in all $\pi$-compatible reachable states from $\gamma$. If a positive answer is returned, that means one cannot fulfill $\text{req}(\gamma, P, \pi, T)$ without violating the constraints on updates specified by $\pi$. Otherwise, model checkers would generate a counterexample, from which Route constructs an update.

4.3.1 Formalization

The model checking techniques that Route uses works with computational tree logic (CTL) and Kripke structures [6]. We first introduce these notions and the Model Checking Problem (MCP) considered in this paper; then and translate RUP to MCP.

Let $AP$ be a set of atomic propositions. A Kripke structure $M$ is a tuple $(S, \sigma, L)$ where

1. $S$ is a finite set of states,$^4$
2. $\sigma$ is a binary relation on $S$ (i.e., $\sigma \subseteq S \times S$) which defines the transitions between states, and
3. $L : S \rightarrow 2^{AP}$ associates each state $s \in S$ with a set of propositions in $AP$.

A path in $M$ starting from $s$ is denoted as $H = \{s_0, s_1, \cdots, s_i, s_{i+1}, \cdots\}$, where $s_0 = s$ and $(s_i, s_{i+1}) \in \sigma$ holds for all $i \geq 0$.

The syntax of CTL is given as: Every atomic proposition $ap \in AP$ is a CTL formula; and if $\phi_1$ and $\phi_2$ are CTL formulas, then so are $\neg \phi_1, \phi_1 \land \phi_2, \phi_1 \lor \phi_2$, and $AG \phi_1$. The formula $AG \varphi$ means that on every computation path $\varphi$ holds at every state. The semantics of CTL formulas is usually defined with respect to a Kripke structure. Let $M = (S, \sigma, L)$ be a Kripke structure for CTL. Given any $s \in S$, denote a CTL formula $\varphi$ holds in $M$ at state $s$ as $(M, s) \models \varphi$. The relation $\models$ is defined by structural induction on CTL formulas:

- $(M, s) \models ap$ iff $ap \in L(s)$.
- $(M, s) \models \neg \varphi$ iff $(M, s) \not\models \varphi$.
- $(M, s) \models \varphi_1 \land \varphi_2$ iff $(M, s) \models \varphi_1$ and $(M, s) \models \varphi_2$.
- $(M, s) \models \varphi_1 \lor \varphi_2$ iff $(M, s) \models \varphi_1$ or $(M, s) \models \varphi_2$.
- $(M, s) \models AG \varphi$ iff for all paths $H = \{s_0, s_1, s_2, \cdots\}$, where $s_0 = s$ and for all $i \geq 0$, $(M, s_i) \models \varphi$.

An MCP can be abstracted as follows [5]. Given $(M, \phi, I)$, where $M$ is a Kripke structure, $\phi$ is a CTL formula, and $I \subseteq S$.

$^4$Note that these are not RBAC states.
$^3$This definition is not complete; but we only use the formula $AG \varphi_1$.\footnote{Note that these are not RBAC states.\footnote{This definition is not complete; but we only use the formula $AG \varphi_1$.}}
is a set of initial states, the problem is to determine whether
$I \subseteq \{s \in S \mid (M,s) \models \varphi\}$.
We now connect RUP with MCP. Given $Q = \text{req}(\gamma, P, \pi, T)$, we define a set of propositions $\text{prop}(Q) = Pr_1 \cup Pr_2$, where

- $Pr_1 = \{x \cdot w \cdot r \mid u \in \gamma.U \land r \in \gamma.R \cup \{x \cdot r \cdot p \mid r \in \gamma.R \land p \in \gamma.P\}$, and
- $Pr_2 = \{x \cdot w \cdot r \mid r \in T\}$.

The proposition $x \cdot r \cdot p$ is meant to represent the assignment of $p$ to $r$: $x \cdot r \cdot p = 1$ means $r$ is assigned with $p$, and otherwise not. The propositions in $Pr_1$ are used to model the user-role and role-permission assignments in RBAC states. The propositions in $Pr_2$ denote whether or not a witness user $wu$ is assigned with the roles in $T$.

Given $Q$, we construct a Kripke structure $M_Q = (S, \sigma, L)$ as follows.

- $S = \{s_A \mid A \subseteq \text{prop}(Q)\}$,
- $L$ is defined as $L(s_A) = \{ap \in \text{prop}(Q) \mid ap \in A\}$.

To define $\sigma$, we define a mapping $\gamma : S \rightarrow \text{space}(\gamma)$:

- $g(s_A) = \gamma_A = (\gamma_A.U, \gamma_A.R, \gamma_A.P, \gamma_A.UA, \gamma_A.PA)$ if and only if
  - $\gamma_A.U = \gamma.U$, $\gamma_A.R = \gamma.R$, $\gamma_A.P = \gamma.P$,
  - $(u, r) \in \gamma_A.UA$ if and only if $x \cdot u \cdot r \in A$,
  - $(r, p) \in \gamma_A.PA$ if and only if $x \cdot r \cdot p \in A$.

Then for any $s_A, s_B \in S, (s_A, s_B) \in \sigma$ if and only if both $g(s_A)$ and $g(s_B)$ are $\pi$-compatible.

To define the initial states of Kripke structure, for any RBAC state $\gamma', \ell$, let $Kstates(\gamma') = \{s_A \in S \mid g(s_A) = \gamma'\}$. We let $I_Q = Kstates(\gamma)$ so that the initial states correspond to the requested RBAC state $\gamma$.

Finally, we define $\phi$ of MCP ($M_Q, \phi, I_Q$), i.e., the property that we want to check. Assume that $T = \{r_1, \ldots, r_t\}$, $P = \{p_1, \ldots, p_m\}$ and $\gamma.P \setminus P = \{p_{m+1}, \ldots, p_n\}$. We let $\phi$ be $AG \neg (\phi_1 \land \phi_2)$, where

- $\phi_1 = X \cdot w \cdot u \cdot p_1 \land \cdots \land X \cdot w \cdot u \cdot p_m$,
- $\phi_2 = \neg (X \cdot w \cdot u \cdot p_{m+1} \land \cdots \land X \cdot w \cdot u \cdot p_n)$.

In turn, for $1 \leq \ell \leq n$, $X \cdot w \cdot u \cdot p_i$ is defined as

$\langle x \cdot w \cdot u \cdot r_1 \land x \cdot r_1 \cdot p_i \rangle \land \cdots \land \langle x \cdot w \cdot u \cdot r_2 \land x \cdot r_2 \cdot p_i \rangle \land \cdots \land \langle x \cdot w \cdot u \cdot r_{t} \land x \cdot r_{t} \cdot p_i \rangle \land \cdots \land \langle x \cdot w \cdot u \cdot r_{t} \land x \cdot r_{t} \cdot p_i \rangle$.

Intuitively, each $X \cdot w \cdot u \cdot p_i$ is testing if the witness user $wu$ has the permission $p_i$. Then $\phi_1$ models if (a) $wu$ can have all permissions in $P$, whereas $\phi_2$ is used to test if (b) $wu$ has no permission in $\gamma.P \setminus P$. As a result, $AG \neg (\phi_1 \land \phi_2)$ is asking if there is no reachable state from the initial states that satisfies both (a) and (b). Formally, we have the following result.

**Theorem 3.** $\text{upd}(\gamma, P, \pi, T) = \emptyset$ if and only if $I_Q \subseteq \{s \in S \mid (M_Q, s) \models AG\neg (\phi_1 \land \phi_2)\}$.

If $AG \neg (\phi_1 \land \phi_2)$ is checked to be true, then there is no reachable state satisfying the requirements (a) and (b); otherwise, NuSMV would generate a counterexample. The counterexample here is a state $s_A$; then the RBAC state $g(s_A)$ is a desired update for $Q$.

### 4.3.2 Implementation in NuSMV

Particularly, Route embeds the model checker NuSMV [5] to search for updates. NuSMV is a modern symbolic model checker, supporting various useful features such as the TRANS constraints. A TRANS constraint defines which next states that the current state may transit into. Namely, the model of the constraint is a set of current/next state pairs. Multiple TRANS constraints are treated as the conjunction of all TRANS constraints. We refer readers to [5] for details of NuSMV.

**Encoding states.** Recall the condition COND-U-R of $\pi$-compatibility, that users’ role assignments remain during updates; hence, we do not need to encode the user-role assignments. A set of boolean variables are defined to describe the role-permission assignments. According to Proposition 1, only a vector of variables for $\{r \mid \text{xmax-perms}[r]\}$ is defined for each $r \in \gamma.R$.

**Encoding $\pi$.** The UCS $\pi$ mainly defines the set of resulting states that $\gamma$ may evolve into. Recall that, for a $\pi$-compatible RBAC state $\chi$, the condition COND-U-P requires that $\pi.th[u] \subseteq \text{perms}[u] \subseteq \text{perms}[u]$. Since $\text{perms}[u] = \text{perms}[\text{roles}[u]]$ and $\text{roles}[u] = \text{roles}[u]$, this requirement actually puts restrictions on the variables for $\{r \mid \text{xmax-perms}[r]\}$, for each $r \in \text{roles}[u]$.

Route translates this requirement into TRANS constraints in NuSMV. For each $\pi.th[u] \subseteq \text{perms}[\text{roles}[u]] \subseteq \text{perms}[u]$, Route constructs two TRANS constraints. The $\pi.th[u] \subseteq \text{perms}[\text{roles}[u]]$ part requires that, for each $p \in \pi.th[u]$, there exists at least one $r \in \text{roles}[u]$ such that $(r, p) \in \chi.PA$. Thus, suppose that $\pi.th[u] = \{p_1, \ldots, p_s\}$ and roles[$u$] = $\{r_1, \ldots, r_m\}$, the following NuSMV constraint is needed.

**TRANS next** $(\langle x \cdot r_1 \cdot p_1 \cdot \ldots \cdot x \cdot r_s \cdot p_s \rangle \land \ldots \land \langle x \cdot r_m \cdot p_m \cdot \ldots \cdot x \cdot r_m \cdot p_m \rangle)$

For $\text{perms}[\text{roles}[u]] \subseteq \text{perms}[u]$ part, first observe that $\text{perms}[\text{roles}[u]] \subseteq \text{perms}[u]$ if and only if $\text{perms}[\text{roles}[u]] \cap \text{max-perms}[\text{roles}[u]] \cap \text{perms}[u] = \emptyset$. Hence, assuming $\text{max-perms}[\text{roles}[u]] \cap \text{perms}[u] = \{q_1, \ldots, q_t\}$, it is required that each $x \cdot r \cdot q_i$ be 0 constantly if $q_i \in \text{max-perms}[r_i]$, for $1 \leq i \leq m$ and $1 \leq j \leq t$.

The encoding of $AG \neg (\phi_1 \land \phi_2)$ is straightforward. Appendix A provides a counterexample for an example update.

### 4.4 Example usage of role updating

This section presents several example configurations of $\text{req}(\gamma, P, \pi, T)$ that SSOs can specify to achieve different update objectives, with respect to $\gamma_{ex}$ in Figure 1.

**Adjusting role sets and permission sets.** To repair $\gamma_{ex}$ so that $u_2$ can have $p_2$, the SSO could issue a request $Q_1 = \text{req}(\gamma_{ex}, P, \pi, T)$:

- $P = \text{perms}_{\text{ex}}[u_2] \cup \{p_2\} = \{p_1, p_2, p_3, p_4, p_5\}$,
- $\pi.U = \gamma_{ex}.U \setminus \{u_2\} = \{u_3, u_4\}$, and for any $u \in \pi.U, \pi.th[u] = \text{perms}_{\text{ex}}[u]$, and
- $T = \gamma_{ex}.R = \{r_1, r_2, r_3, r_4, r_5, r_6\}$.

Running with $Q_1$, Route would suggest a sequence of actions: $s_1 = \langle \text{revok}(p_6, r_2); \text{assign}(r_7, u_2); \text{assign}(r_5, u_2); \text{assign}(r_3, u_1) \rangle$, where $wu$ is the witness user not belonging to $\gamma$. Then the SSO can follow $s_1$ to make changes to $\gamma_{ex}$: revoice $p_6$ from $r_2$ and associate $\{r_1, r_2, r_3\}$ with $u_2$.

For another example, suppose that the SSO wants to shrink $u_3$’s permission set to $\{p_1, p_2, p_2\}$ and revoke $r_3$ from $u_3$. Then $Q_2 = \text{req}(\gamma_{ex}, P, \pi, T)$ models this request:
\[ P = \{p_1, p_2, p_3\}, \]
\[ \pi U = \gamma_{ax}U \cup \{u_3\} = \{u_1, u_2, u_4\} \]
\[ \psi th[u] = perms_{\gamma_{ax}}[u], \]
\[ T = \{\text{roles}_{\gamma_{ax}}[u_3] \backslash \{r_3\} = \{r_4, r_5\}. \]

Route would return a sequence of actions: \( s_2 = (\text{assign}(p_5, r_3); \text{revoke}(p_6, r_4); \text{revoke}(p_4, r_3); \text{assign}(p_6, r_6); \text{assign}(r_4, r_5); \).

If the SSO requests to remove \( r_3 \) from \( u_2 \) (i.e., assign only \( r_1 \) with \( u_2 \)) but retain \( u_2 \)'s current permission set: \( \{p_1, p_3, p_4, p_5\}. \) Route would report that the request is not achievable.

**Permission assignments.** To update for enabling the exact set of permissions \( P_U = \{p_5, p_6, p_7\} \), the SSO could set \( Q_3 = \text{req}(\gamma_{ax}, \psi, \pi, T) \) as follows:

\[ \pi U = \gamma_{ax}U \cup \{u_3\} = \{u_1, u_2, u_3, u_4\} \]
\[ \psi th[u] = perms_{\gamma_{ax}}[u], \]
\[ T = \{r_3 \cup \{r_4, r_5, r_6\}. \]

Given \( Q_3 \), Route returns the sequence of administrative actions: \( s_3 = (\text{assign}(p_5, r_3); \text{assign}(p_6, r_6); \text{assign}(r_4, r_5); \).

### 5. REDUCTIONS

Section 4.3 describes the idea of applying model checking techniques to RUP. However, the execution of the NuSMV program directly transformed from \( \text{req}(\gamma, P, \pi, T) \) quickly leads to state explosion (and memory crash). In this section, we present a set of reductions for RUP. These reductions are not exclusive: Route but could also benefit other approaches to RUP.

Consider two requests \( Q_1 = \text{req}(\gamma_1, P, \pi_1, T) \) and \( Q_2 = \text{req}(\gamma_2, P, \pi_2, T) \), which share the same \( P \) and \( T \). If it holds that \( \text{upd}(\gamma_1, P, \pi_1, T) \neq \emptyset \) if and only if \( \text{upd}(\gamma_2, P, \pi_2, T) \neq \emptyset \), then \( Q_1 \) is satisfiable if and only if so is \( Q_2 \). On the other hand, if \( \text{upd}(\gamma_1, P, \pi_1, T) \subseteq \text{upd}(\gamma_2, P, \pi_2, T) \), then when we find an update \( \chi \) for \( Q_1 \), we also obtain an update for \( Q_2 \). Put together, we have the following definition.

**Definition 7.** Given a request \( Q = \text{req}(\gamma, P, \pi, T) \) and a set of requests \( Q_{\text{set}} = \{\text{req}(\gamma_1, P, \pi_1, T), \ldots, \text{req}(\gamma_m, P, \pi_m, T), \} \), we say \( Q \leftarrow Q_{\text{set}} \) if the following two conditions are satisfied.

1. \( \bigcup_{1 \leq i \leq m} \text{upd}(\gamma_i, P, \pi_i, T) \neq \emptyset \) if and only if \( \text{upd}(\gamma, P, \pi, T) \neq \emptyset. \)

2. \( \bigcup_{1 \leq i \leq m} \text{upd}(\gamma_i, P, \pi_i, T) \subseteq \text{upd}(\gamma, P, \pi, T). \)

It can be seen that if \( Q \leftarrow Q_{\text{set}} \), then we can work with the set \( Q_{\text{set}} \) instead of \( Q \); we need to find a \( Q_{\text{set}} \) that is easier to tackle.

### 5.1 Reduction on core

**Observation 1.** For \( \pi \)-compatibility, changes can only happen around roles that are related to the permissions in \( P \).

**Definition 8.** Given a state \( \gamma \) and \( R \subseteq \gamma, R \), we say \( \gamma_{[R]} \) is a filtered state of \( \gamma \) by \( R \), where \( \gamma_{[R]} \), \( \gamma_{[R]} \) is \( \text{roles}_{\gamma_{[R]}}, \text{users}_{\gamma_{[R]}}, \text{perms}_{\gamma_{[R]}}, \text{UCS}_{\gamma_{[R]}}, \text{core}_{\gamma_{[R]}}, \text{core}_{\gamma_{[R]}} \).

Given \( \text{req}(\gamma, P, \pi, T) \), denote the state \( \gamma_{\text{core}_{\gamma}(P, \pi)} \) as \( \text{core}_{\gamma}. \)

While \( \gamma \) is filtered with respect to \( \text{core}_{\gamma}(P, \pi) \), a new UCS on updates of \( \text{core}_{\gamma} \) is to be constructed as well in a way that those updates to \( \text{core}_{\gamma} \) can be seen as updates to \( \gamma \). Not surprisingly, this new UCS stems from \( \pi \).

**Definition 9.** (Refinement of \( \pi \)) Given \( \text{req}(\gamma, P, \pi, T) \) and a state \( \gamma' \) obtained by filtering \( \gamma \) with some role set, define a UCS \( \pi_{\gamma'} \) by letting:

1. \( \pi_{\gamma'} \cdot U = \pi \cdot U \cup \gamma', U \), and
2. for any \( u \in \pi_{\gamma'} \cdot U \), \( \pi_{\gamma'} \cdot th[u] = \pi \cdot th[u] \) \( \backslash \text{roles}_{\gamma'}[u] \).

The UCS \( \pi_{\gamma'} \) is a refinement of \( \pi \) to \( \gamma' \). \( \pi_{\gamma'} \) only allows changes to \( \gamma' \), which is a filtered state of \( \gamma \). Hence, all updates allowed by \( \pi_{\gamma'} \) would not influence users in \( \pi_{\gamma'} \cdot U \cdot \gamma' \); the first clause refines the user set. Since \( \pi_{\gamma'} \) only allows changes to roles in \( \gamma' \), the permission set of any role in \( \gamma', R \cdot \gamma' \) would not be changed by any update of \( \gamma' \) and, therefore, nor is the permission set of any role in \( \text{roles}_{\gamma'} \), \( \theta = \text{roles}_{\gamma'}[u] \cdot \gamma' \). For each \( u \in \pi_{\gamma'} \cdot U \).

Let \( Q_{\text{core}} = \text{req}(\pi_{\text{core}_{\gamma}}, P, \pi_{\text{core}_{\gamma}}, T) \). Proposition 4 corresponds to Observation 1.

**Proposition 4.** \( Q \leftarrow Q_{\text{core}}. \)

### 5.2 Decomposition

**Observation 2.** It is sometimes useful to decompose \( Q = \text{req}(\gamma, P, \pi, T) \) sub-problems that can be solved separately.

A collection \( C_{\text{de}} \subseteq 2 \cdot \gamma \cdot R \) is a decomposition of \( Q \) if

1. for any \( S, S' \in C_{\text{de}} \), \( P \subseteq \text{perms}_{\gamma}[S] \), and either \( S = S' \) or \( S \nsubseteq S' \),

2. for any \( R \subseteq \gamma \cdot R \), if \( R \not\in C_{\text{de}} \) then \( \forall S \not\in C_{\text{de}} \), either \( S \not\in C_{\text{de}} \).

Given \( S \in C_{\text{de}} \), let \( S = \text{roles}_{\gamma}, \text{users}_{\gamma}[S] \) and denote the state \( \gamma_{[S]} \) as \( \text{de}_{\gamma}(\gamma) \). One may decompose \( Q \) into a set of requests:

\[ Q_{\text{de}} = \{\text{req}(\text{de}_{\gamma}(\gamma), P, \pi_{\text{de}_{\gamma}(\gamma)}), T \mid S \in C_{\text{de}} \}. \]

**Proposition 5.**\( Q \leftarrow Q_{\text{de}}. \)

Proposition 5 does not reduce the complexity of RUP theoretically. It is likely that there exists \( S \in C_{\text{de}} \) such that \( S = \text{core}_{\gamma}(P, \pi) \) — in this case, we may still have to work with \( Q \). However, the benefits lie in practice. For one thing, when \( \text{upd}(\gamma, P, \pi, T) \neq \emptyset \), we may figure out one \( \chi \in \text{upd}(\text{de}_{\gamma}(\gamma), P, \pi_{\text{de}_{\gamma}(\gamma)}, T) \) with smaller \( S \), which could be more efficient than working with \( Q \); for another, the incurred changes could be more restricted, for \( S \subseteq \text{roles}_{\gamma} \).

**Definition 8.** Given a state \( S \subseteq \gamma \cdot R \), we say \( S \subseteq \gamma \cdot R \) is a filtered state of \( \gamma \) by \( R \), where \( \gamma_{[R]} \), \( \gamma_{[R]} \) is \( \text{roles}_{\gamma_{[R]}}, \text{users}_{\gamma_{[R]}}, \text{perms}_{\gamma_{[R]}}, \text{UCS}_{\gamma_{[R]}}, \text{core}_{\gamma_{[R]}}, \text{core}_{\gamma_{[R]}} \).

Given \( \text{req}(\gamma, P, \pi, T) \), denote the state \( \gamma_{\text{core}_{\gamma}(P, \pi)} \) as \( \text{core}_{\gamma}. \)
\( c_{\text{de}}^{\text{approx}} \) is a decomposition of \( Q \). \( c_{\text{de}}^{\text{approx}} \) features that each \( M \in \mathcal{M}_{\text{de}} \) is a minimal role set whose permission set contains \( \mathcal{P} \). Hence, evaluating \( \text{req}(\text{de}_M(\gamma), \mathcal{P}, \pi(\text{de}_M(\gamma)); T) \) for \( M \in c_{\text{de}}^{\text{approx}} \) might be easier. Unfortunately, computing \( c_{\text{de}}^{\text{approx}} \) is NP-hard.

In our prototype of \( \text{Route} \), priority is given to the linear computing runtime of a decomposition \( c_{\text{de}}^{\text{approx}} \), while keeping each member as minimal as possible. Practical systems may use specialized algorithms for \( c_{\text{de}}^{\text{approx}} \), which is beyond the scope of this paper.

5.3 Removing ignorable roles

**Observation 3.** For a role \( r \) such that \( \text{perms}_r[r] \cap \mathcal{P} = \emptyset \), if its permission assignments could not be changed due to \( \pi \), \( r \) would not help with \( \text{req}(\gamma, \mathcal{P}, \pi, T) \) and thus can be ignored by updates.

Given \( r \in \gamma.R \), we say \( r \) is ignorable if \( r \not\in \text{roles}_{\mathcal{P}}(\gamma) \) and \( \text{perms}_r[r] = \text{max-perms}[r] \). That is, \( r \) is ignorable if it cannot accept any permission assignments other than those that are already assigned with it in \( \gamma \).

Denote the set of non-ignorable roles as \( R_{\text{non}} \) and the state obtained by filtering \( \gamma \) with \( R_{\text{non}} \) (i.e., \( \gamma|_{R_{\text{non}}} \)) as \( \gamma|_{\mathcal{P}} \). Let \( Q = \text{req}(\gamma, \mathcal{P}, \pi, T) \) and \( Q_{\text{non}} = \text{upd}(\gamma|_{\mathcal{P}}, \mathcal{P}, \pi|_{\mathcal{P}}; T) \). Proposition 6 corresponds to Observation 3.

**Proposition 6.** \( Q \hookrightarrow Q_{\text{non}} \)

5.4 Propagating requested permissions

**Observation 4.** If complying with \( \pi \), we may associate permissions in \( \mathcal{P} \) with as many roles as possible. In addition, there exists an update that does not remove any assignment \((r,p)\) such that \( p \in \mathcal{P} \), should there exist an arbitrary update.

Given \( \chi \in \text{upd}(\gamma, \mathcal{P}, \pi, T) \), \( p \in \mathcal{P} \), and \( r \in \gamma.R \) such that \( \text{perms}_r[r] \not\subseteq \mathcal{P} \) and \( (r,p) \notin \chi.PA \), if allowed by \( \pi \), making \( (r,p) \in \chi.PA \) would retain \( \chi \) being an update. Namely, we can propagate permissions in \( \mathcal{P} \) among roles and only change the role-permission assignments whose permissions reside outside \( \mathcal{P} \).

Define a state \( \text{pt}_\pi(\gamma) \) by letting

\[
\begin{align*}
\text{pt}_\pi(\gamma).U &= \gamma.U, \quad \text{pt}_\pi(\gamma).R = \gamma.R, \quad \text{pt}_\pi(\gamma).P = \gamma.P, \\
\text{pt}_\pi(\gamma).U.A &= \gamma.U.A, \quad \text{pt}_\pi(\gamma).R.A &= \gamma.R.A, \quad \text{pt}_\pi(\gamma).P.A &= \gamma.P.A \cup \{(r,p) \mid r \in \gamma.R \land p \in (\mathcal{P} \cap \text{max-perms}[r])\}.
\end{align*}
\]

**Proposition 7.** \( \text{req}(\gamma, \mathcal{P}, \pi, T) \hookrightarrow \text{req}(\text{pt}_\pi(\gamma), \mathcal{P}, \pi, T) \)

**Proposition 8.** If \( \chi_1 \in \text{upd}(\text{pt}_\pi(\gamma), \mathcal{P}, \pi, T) \), then there exists \( \chi_2 \in \text{upd}(\text{pt}_\pi(\gamma), \mathcal{P}, \pi, T) \) such that \( \text{diff}(\text{pt}_\pi(\gamma), \chi_2) \cap \{(r,p) \mid p \in \mathcal{P}\} = \emptyset \).

Proposition 8 means that we can just fix the assignment \((r,p)\) constantly, if \( p \in \mathcal{P} \). Together with Proposition 7, we would not miss an update if there exists one.

6. EXPERIMENTS

Figure 2 depicts the prototype of \( \text{Route} \). The interface receives SSO’s update requests and forwards them to the back-end. Algorithm 1 presents the pseudo code of the back-end. Function TransNuSMV translates the request into NuSMV programs; it explores some further simplifications of the problem. The basic intuition is to reduce the number of variables in NuSMV programs in accordance with \( \pi \)-compatibility. TransNuSMV also groups together those permissions that can be treated likewise (i.e., either revoke all of them from one role or assign all of them with one role) and retain only one of them.

**Figure 2: Route prototype.**

![Figure 2: Route prototype.](image-url)

**Algorithm 1: Algorithm of the back-end.**

**Input:** \( \text{req}(\gamma, \mathcal{P}, \pi, T) \)

**Output:** either \( \text{upd}(\gamma, \mathcal{P}, \pi, T) = \emptyset \) or \( \chi \in \text{upd}(\gamma, \mathcal{P}, \pi, T) \).

1. begin
2. Compute \( \text{pts}_{\gamma, \pi}(\mathcal{P} \cup \text{core-\gamma}) \), and the set \( \mathcal{P}_{\text{de}}^{\text{approx}} \);
3. foreach \( S \in \mathcal{P}_{\text{de}}^{\text{approx}} \) do
4. Apply the reductions of “decomposition”, “Removing ignorable roles” and “Propagating requested permissions” to \( \text{pts}_{\gamma, \pi}(\mathcal{P} \cup \text{core-\gamma}) \) in sequence;
5. Denote the obtained state and UCS as \( \gamma' \) and \( \pi' \), respectively;
6. TransNuSMV \( (\text{req}(\gamma', \mathcal{P}, \pi', T);) \);
7. Execute the resulting NuSMV program;
8. if a counterexample is returned then
9. Construct \( \chi \) according to the counterexample;
10. return \( \text{diff}(\chi, \gamma) \);
11. return \( \text{upd}(\gamma, \mathcal{P}, \pi, T) = \emptyset \);
12. end

6.1 Experiments

**Experiment Cases.** The prototype of \( \text{Route} \) is implemented in JAVA. A request \( Q_{\text{exp}} = \text{req}(\gamma, \mathcal{P}, \pi, T) \) is randomly generated. In all tests, we let \( T = \gamma.R \) and set \( \pi \) by setting \( \pi.U = \gamma.U \) and, for all \( u \in \pi.U, \pi.th[u] = \text{perms}_r[u] \). The reductions are performed in sequence as shown in Algorithm 1. For each \( S \in \mathcal{P}_{\text{de}}^{\text{approx}} \), we generate a file for the corresponding NuSMV program (denoted as \( \text{progs}(S) \)) and a file for NuSMV commands. Then for each NuSMV program, we fork a thread, which executes the NuSMV program via NuSMV’s batch mode. The thread is killed if the execution of the NuSMV program exceeds 12 hours. Unlike Algorithm 1, the tests would not finish until all NuSMV programs are generated, executed and returned, even though an update has been found. We record the time for processing each \( S \) denoted as \( \text{time}(S) \), including both the averaged shared preparation time and its own model checking time. Experiments were performed on a machine with an Intel(R) Core(TM)2 CPU T5500 @ 1.66GHz, and with 2GB of RAM running Microsoft Windows XP Home Edition Service Pack 3.

**Synthetic Data Generation.** To generate requests, we adapt data-generation algorithms from [34, 37], which is parameterized by the number of users (\( \text{noU} \)), the number of roles (\( \text{noR} \)), the number of permissions (\( \text{noP} \)), the maximum number of roles (\( \text{noUR} \)), each user may be assigned with, the maximum number of permissions (\( \text{noRP} \)) each role could possibly be assigned to, and the number of requested permissions (\( \text{noRQs} \)). The relation \( \gamma.UA \) (resp. \( \gamma.PA \)) is generated by assigning each user (resp. each role) a num-
ber $k$ of roles (resp. permissions) where $k$ is randomly chosen and uniformly distributed between 1 and noUR (resp. noRP). Note that in all tests, no two users have the same role set and no two roles have the same permission set. $\mathcal{P}$ is determined by randomly chosen a number noReqps of permissions from $\gamma, \mathcal{P}$.

**Time Metric.** Given the set of programs of $\text{prog}(\mathcal{C}_\text{approx}^\text{de})$, let $\text{prog}_j$ be the set of programs that are checked false, $\text{prog}$, the set of programs that are checked true, and $\text{prog}_j$, the set of other programs that could not finish within time limits or that lead to memory crash. Fortunately, in all our test cases, we did get answers (i.e., either $\text{prog}(\mathcal{C}_\text{approx}^\text{de}) = \text{prog}$, and all programs are finished in time or there exist some programs that return with a counterexample prior to time limit). Even though timeouts happen in several cases, there was another $\text{prog}(S)$ whose NuSMV program is efficiently checked false and thus a counterexample is generated, which reveals an update.

The time records shown in Figure 3 are computed as follows.

$$ \text{time} = \max\{\text{time}(S) \mid S \in \text{prog}_j\} + \sum_{S \in \text{prog}_j} \text{time}(S). \quad (4) $$

For example, suppose $\mathcal{C}_\text{approx}^\text{de} = \{S_1, S_2, S_3, S_4, S_5\}$ where $\text{prog}(S_1)$ and $\text{prog}(S_5)$ are checked true, checking $\text{prog}(S_2)$ times out, and $\text{prog}(S_3)$ and $\text{prog}(S_4)$ are checked false and counterexamples are generated. Then the computing time is: $\text{time}(S_1) + \text{time}(S_5) + \max\{\text{time}(S_2), \text{time}(S_3), \text{time}(S_4)\}$. It is arguably reasonable to ignore the time of $\text{prog}$. Take $\text{time}(S_2)$ for instance. In practice, one can easily compare the number of variables in $\text{prog}(S_1)$, $\text{prog}(S_2)$, $\text{prog}(S_3)$, $\text{prog}(S_4)$, and $\text{prog}(S_5)$. Since NuSMV’s performance highly depends on the number of variables, one can schedule programs’ executions in increasing order by the number of variables and thus put $\text{prog}(S_2)$ at the end of queue. When, for example, $\text{prog}(S_3)$ returns a counterexample, then an update is found and there is no need to execute $\text{prog}(S_2)$.\(^6\) In this case, decomposition appears useful. Hence, we suspect that $\text{time}$ in (4) is possibly longest time taken by Route to evaluate $\text{req}(\gamma, \mathcal{P}, \pi, T)$. Since the data set is randomly created, for each configuration of parameters, we ran the test 5 times. The time in Figure 3 is averaged over the 5 runs.

**Results.** Figure 3 reports the experiment results. The time is in minute and log-scale in Figure 3(e), but is in second and linear in others. Note that for each run, a new instance $Q_{\text{exp}}$ was generated each time a configuration was tested. In Figure 3(a), we generated $Q_{\text{exp}}$ by fixing “noU=500 noP=2000 noUR=3 noRP=150 noReqps=500” but varying noRp. The peak (about 17 minutes) was reached at “noReqps=500-600”. The use of “propagating requested permission” reduction saved Route from setting more and more variables, with the increase of noReqps. This explains why the time starts to drop at 700.

Figure 3(d) shows the case where “noU=1500 noR=500 noP=2000 noUR=3 noReqps=500” were fixed and Figure 3(e) the case where “noU=1500 noR=500 noP=2000 noRP=150 noReqps=500” were fixed. While Route dealt quite well with large noRP, the performance of Route with respect to large noUR was relatively poor. One reason might be that, the larger noUR is, the more NuSMV programs were created and checked; and the reductions also took notable time. Even though reductions were performed and many role-permission assignments were set constants according to $\pi$, there were still many variables, for $[\text{roles}, [u]]$ could be quite large (>, 5); since constraints are put on the permission assignments of roles in $\text{roles}, [u]$, the large number of $[\text{roles}, [u]]$ prevents more variables from being fixed.

In real-world large-scale RBAC systems, even though noUR could be large (>, 5), we expect that only a small portion of users have a number noUR of roles and that the number of roles that are under an SSO’s control will be small. Hence, we conjecture Route will be able to handle update requests in these RBAC systems.

7. RELATED WORK

**RBAC policy analysis and repair.** Many RBAC policy analysis tools (RPATs) are invented to help administrators understand and manage RBAC policies e.g., [3, 14, 19, 31, 32, 36], to name a few. Most (safety) analysis problems in literature basically can be stated as: given the current state $\gamma$, a query $q$, and a state-change rule $\varphi$, can $\gamma$ be taken a state $\gamma'$ where $q$ evaluates true? If this is the case, one may argue that the steps taking $\gamma$ to $\gamma'$ may also be reported to SSOs so that they can follow to make $\gamma'$. However, as the objectives are different, we believe this report could hardly be sufficient for RUP. As remarked in [14], “...q typically encodes an unsafe situation that should never occur:...” Hence, RPATs explore every possible sequence of actions, as long as they are allowed by $\varphi$, to test if there is such a $\gamma'$ where $q$ is true; consequently, RPATs do not care what the resulting states look like. On the contrary, Route seeks for a resulting state with expected assignments. In addition, most RPATs focus on user-role assignments; although it is argued that role-permission relation is a dual of user-role relation and might be treated likewise, role-permission relation also deserves its own attention [22], especially in terms of role updating. Finally, Route goes beyond by enabling various constraints on updates, with which SSOs specify requirements on updates.

**Role engineering.** Recently, much research effort has been devoted to role engineering [10, 7, 20, 33, 34, 37]. Basically speaking, existing role engineering tools (eREts) consume a predefined user-permission assignments and output a set of user-role assignments and a set of role-permission assignments, taking into account some optimization objectives (e.g., a minimal set of roles) and possibly other concerns such as roles’ business meanings, semantics, users’ attributes. Taxonomically, Route can be viewed as a role engineering tool, as RUP also deals with these assignments. However, role updating works when RBAC states have been defined and possibly deployed, whereas eREts usually define roles from scratch. The focuses are also different. Role updating aims to answer SSOs’ question whether an update is achievable with respect to update

\(^6\)In practice, one can run all programs in parallel, and stop them when one of them returns a counterexample or all programs return with an answer “true”.
constraints and, if any, to generate one. On the contrary, eRETs put more emphasis on how to define an appropriate set of roles. In the view of role life cycle, Route is for role maintenance, while eRETs help role design. Thus, one may consider Route as a complement of eRETs; Route can be used to fine-tune the ideal state generated by eRETs.

Model checking in RBAC. Jha et al. [14] presented a transformation from an RBAC policy analysis instance to a NuSMV program by setting states correspond to user-role assignments and transitions correspond to administrative rules. However, their transformation does not fit for Route. The reason is that it is more intuative and convenient to encode constraints on updates (i.e., π) in NuSMV states rather than in the transitions. This reflects the difference between administrative rules and constraints on updates: administrative rules specify what transitions can be made from each state, whereas constraints on updates put restrictions on the resulting states. Schaad et al. [27] applied model checking techniques to automated analysis of delegation and revocation functionalities, with an emphasis on static and dynamic separation of duty properties. They do not consider the role-permission assignments. Reith et al. [24] applied model checking techniques to the policy analysis of a language RTD, which can be viewed as a generation of RBAC models considered in this paper. It is unclear how to use their algorithms to tackle RUP though. There are some other works, which applied model checking techniques to RBAC or its variants, such as [1, 21]; but, to our knowledge, Route is the first to use them for RUP.

RBAC updating. Ni et al. [22] studied the role adjustment problem (RAP) in the context of role-based provisioning based on machine-learning algorithms. Though similar, RUP differs from the RAP in several aspects. First, customized constraints on updates are enforced in role updating, whereas it is unclear if these constraints could be supported in RAP. Second, the role updating is request-driven, whereas RAP is a learning process. Specially, SSOs submit a specific update objective to Route, which tries to find the expected update. On the contrary, RAP is supplied by SSOs with provisioning data and output a set of mappings from roles to (new) entitlements. Hence, RAP and Route are both assistant tools for SSOs but with different usage and orientation. Fisler et al. [15] presented a tool to investigate the semantic difference of two RBAC policies (in XACML) and the properties of the difference. They do not consider how to make a different desired state from the current one. Ray [23] studied the problem of real-time update of access control policies, in the context of a database system. The focus was put on the transaction properties. However, RBAC models have important features that deserve consideration when updating.

8. CONCLUSION AND FUTURE WORK

We have studied the RUP problem, presented a set of reduction for RUP, and proposed a role updating tool Route based on model checking techniques. Experiments confirm the effectiveness and efficiency of Route. There are several avenues for future work. Two additional components, role hierarchies and separation of duty (SoD) policies, are also useful in RBAC systems. Their presence complicates the problem. Role hierarchies are important for RBAC systems, as they further mitigate the burden of security administration and maintenance. In the case of SoD policies, enforcing SoD policies is difficult by itself [17]. The interaction between updating and SoD policies poses new challenges. Existing works often assume that role-permission relation is fixed, when considering SoD policies. However, this assumption does not hold from the viewpoint of role updating. Another interesting problem is to update RBAC systems when administrative rules are in position to regulate SSOs’ actions.

Acknowledgment

This work is supported by National Natural Science Foundation of China under Grant 60873225, 60773191, 70771043, National High Technology Research and Development Program of China under Grant 2007AA01Z403, and Natural Science Foundation of Hubei Province under Grant 2009CDB298. This project is supported in
part by an Australian Research Council (ARC) Discovery Projects Grant (DP0988396). We thank the anonymous reviewers for their helpful comments.

9. REFERENCES


