Cybersecurity

RESEARCH Open Access

# On the use of Max-SAT and PDDL in RBAC maintenance

Marco Benedetti[†] and Marco Mori[*†]

**Abstract**

Role-Based Access Control (RBAC) policies are at the core of Cybersecurity as they ease the enforcement of basic security principles, e.g., *Least Privilege* and *Separation of Duties*. As ICT systems and business processes evolve, RBAC policies have to be updated to prevent unauthorised access to resources by capturing errors and misalignments between the current policy and reality. However, such update process is a human-intensive activity and it is expected to meet specific constraints. This paper proposes a semi-automatic RBAC maintenance process to fix and refine an RBAC state when “exceptions” and “violations” are detected. Exceptions are permissions some users realise they miss that are instrumental to their job and should be granted as soon as possible, while violations are permissions that have to be revoked since they are no longer required by their current owners. We propose a formalisation for the maintenance process which fixes single and multiple exceptions and violations by balancing two conflicting objectives, i.e., (i) optimising the current RBAC state, and (ii) reducing the transition cost. Our approach is based on a Max-SAT formalisation of the constraint-based optimisation problem, and on PDDL planning to define the transition strategy with minimum cost. Our implementation relies on incomplete Max-SAT solvers and satisficing PDDL planners which provide approximations of optimal solutions. Experiments along with a comparative evaluation show good performance on real-world benchmarks.

**Keywords:** RBAC maintenance, Max-SAT, PDDL planner, RBAC evolution

## Introduction

Granting access based on direct assignments of permissions to users is highly inefficient for large-scale organisations, where complex business processes and numerous human resources are involved. Thus, many organisations now adopt RBAC (Role Based Access Control) mechanisms which simplify the management of permissions by defining *roles*, which include the permissions required to execute certain tasks (see Ferraiolo et al. (2001)). Users gain the permissions which are included in at least one of the roles assigned to them. In case users are moved to different tasks or if tasks change, it is sufficient to operate at the role level without modifying several Permission-to-User assignments. By posing conditions on roles and their assignments it is also easy to enforce basic security principles such as *Least Privilege* and *Separation of Duties*.

The management of roles throughout their entire lifecycle is a complex and challenging task aiming at designing, implementing, testing, and maintaining roles (Wachs [2014]; Iverson [2015]; Kern et al. 2002). While role mining (Mitra et al. 2016) supports the discovery of roles from scratch, in many cases as organisations undergo changes, it is necessary to *maintain* the current set of roles. Various causes may invalidate the currently deployed roles: Roles may have to be revised to support either new (i.e., exceptions) or missing (i.e., violations) User-to-Permission assignments following technological or business changes; or, a (possibly hand-made) RBAC state may contain errors, which need to be first discovered and then incorporated/subtracted into/from the current state (after validation by a security administrator).

While maintaining an RBAC state, the company has to consider conflicting objectives. It certainly benefits from a *simplification* of the current Permission-to-Role and Role-to-User assignments, i.e., of the so called *RBAC state*. For a security administrator it is indubitably more efficient to deal with a reduced number of roles which are *simple* in terms of permissions and assignments than having a higher number of roles or a high number of Permission-to-Role assignments (Colantonio et al. 2008; Molloy et al.

*Correspondence: marco.mori@bancaditalia.it
†Marco Benedetti and Marco Mori contributed equally to this work.
Bank of Italy, ICT Department, Centro Donato Menichella, Roma, Italy

2008). Nevertheless, any *uninformed* simplification of a given RBAC state may heavily modify the current assignments, thus disrupting organisational processes and/or separation of duty constraints (Vaidya et al. 2008), not to mention the muscle memory of any human actor involved in managing and assigning roles. Losing *similarity* to the initial roles may be acceptable if the organisation is very small or at a preliminary stage of RBAC adoption; not so much in case of a large company with its own well-defined, perhaps "cherished" working set of roles, which have been costly defined and negotiated. These are conflicting objectives, so a trade-off has to be found to balance them out. For an optimised state to be adopted, it is also necessary to determine the course of actions to reconfigure the input set of roles. These are typically time-consuming human activities whose complexity grows with the number of required interventions to the target access control system.

In the context of role maintenance, we have identified two challenging tasks for security administrators: (i) determining one or a set of candidate target RBAC states that fix all outstanding issues, and (ii) defining the simplest course of actions to implement the required changes. Our previous work (Benedetti and Mori [2018]) defined a new *parametric* RBAC maintenance process to determine one or a set of candidate target RBAC states starting from a missing permission to be granted (i.e., an exception). When performing maintenance, the in-place RBAC state is provided as input to the algorithm (together with the permission to be granted) and an adjusted model is produced as output, which accommodates the new requirements by "patching" the original state. When starting from a clean slate, a fully compliant RBAC state is grown out of the empty state by a sequence of patches. In this paper we extend the maintenance process in (Benedetti and Mori [2018]) by considering also violations to the access control policy (i.e., permissions to be revoked) along with the capability of including multiple exceptions and violations in one single problem instance. In addition, this paper formalises the automated planning of actions required to adopt an optimised RBAC state when it has been discovered and presents a comparative analysis with a few representative state-of-the-art maintenance approaches.

As a key feature, the proposed algorithm aims at balancing two possibly conflicting metrics: *similarity* and *simplicity*. Similarity is a measure of how different the patched state is from the pre-patch state: the closer the two states the less disrupting the update. Simplicity is a measure of how close the patched state is to the unconstrained, *optimal* RBAC state: the more a patch strives for optimality, the larger the potential impact on the organisation. As we will see, in our approach the RBAC administrator is provided with means to strike the best balance between these two objectives.Finally, the course of actions to adopt the target state is evaluated according to its length.

Our maintenance process generates Max-SAT (Johnson 1973) and PDDL - Planning Domain Definition Language (Ghallab et al. 2004) instances and solves them via publicly-available, state-of-the-art solvers and planners. Several well known datasets are adapted to this new incremental setting and are used to establish the practicality of our solution in real-world cases.

The rest of this paper is organised as follows. "RBAC maintenance via Max-SAT" section frames the problem and presents a working example. The "Formalisation" section defines the Max-SAT and the PDDL planning formalisation and introduces the metrics we use to evaluate the quality of an RBAC state and of the corresponding reconfiguration strategy. "Validation" section describes the datasets we use in the experiments and presents the results of our experimental evaluation along with a comparative analysis versus a few representative state-of-the-art role mining algorithms. "Related work" section discusses related works. "Conclusion and future work" section concludes the paper and proposes directions for future work.

## RBAC Maintenance process

### Preliminaries

*Role mining* is a preliminary step to enable the adoption of an RBAC model. It can be executed bottom-up or top-down.

The bottom-up approach takes as input an existing set of direct Permission-to-User assignments. Then, mining roles consists in defining a collection of roles (sets of permissions) and then assigning one or more roles to each user. The permissions each user is granted by any of his role(s) have to be exactly the same Permissions-to-User assignments he had before roles were introduced.

Formally, starting from a binary matrix $UPA \in \{0,1\}^{m,n}$ representing the assignments of $n$ permissions to $m$ users, role mining factorizes it into a Role-to-User matrix $UA \in \{0,1\}^{m,k}$ and a Permission-to-User matrix $PA \in \{0,1\}^{k,n}$, for some positive integer $k$ (number of roles), such that $UA \otimes PA = UPA$, where the boolean matrix multiplication operator $\otimes$ is defined as $UPA_{ij} = \bigvee_{l=1}^{k}(UA_{il} \wedge PA_{lj})$.

Top-down approaches—employed as an alternative or in combination with bottom-up mining—define roles starting from an analysis of the business processes to support. Top-down roles are typically hand-crafted by business stakeholders after an analysis of the organisation processes and of the permissions users require to execute them. Hand-made roles, though validated and closely inspected, may of course contain (several) errors.

Whether the synthesis is top-down or bottom-up, once an *RBAC state* is in place, a *role maintenance* process is required to adapt the state to intervening organisational changes (employees move, new systems are deployed, etc.) or to fix errors that become apparent during actual usage.

## Maintenance of an RBAC state

Errors show up as either Permission-to-User assignments that are not implied by the current state but should be, or as unnecessary permissions granted to users. Figure 1 describes the maintenance process starting from the detection of missing and extra permissions. The logs captured by the *monitoring* systems record (among the other things) events corresponding to missing permissions. These can be directly forwarded to the validation phases. The security administrator comes into play at this stage by *validating* prohibited accesses; of particular interest are false positives, i.e., permissions a user misses but should be allowed to have, which we call *exceptions*.

Candidates for revocation, which we call *violations*, are detected through a *discovering* phase which collects all permissions a specific user exercises over a period of time (during which he carries out his complete set of job functions), and then looks for granted permissions that were never used. At this stage, checking the differences among users sharing the same job functions can help too. In any case, the security administrator has to validate each violation before enacting the maintenance process.

The administrator may also directly specify further exceptions and violations by generalising the case at hand.

Starting from a list of validated exceptions $\mathbb{E}$ and violations $\mathbb{V}$ ($x \in \mathbb{E} \cup \mathbb{V}$, with $x : p \to u$) to be incorporated into the *current RBAC state* $S_0$, the *RBAC optimisation* phase generates a *target RBAC state* with the following properties:

(a) it is semantically equivalent to $S_0$, aside from accommodating each $x \in \mathbb{E} \cup \mathbb{V}$, i.e., $u$ now has permission $p$ (if $x \in \mathbb{E}$), and $u$ no longer has permission $p$ (if $x \in \mathbb{V}$);
(b) among all states that satisfy (a), it is either (i) the *most similar* to $S_0$, or (ii) the *simplest possible* state, or (iii) any state *in between* these two extremes.

"RBAC optimisation" section formalises the meaning of "most similar", "simplest possible", and "in between". The RBAC optimisation is applied as new exceptions/violations arrive. If necessary, several exceptions and violations may be dealt with at once, in a single step. The degree of freedom at (b) is exploited by the security administrator to strike the best balance between evolving the state towards simplicity and preserving the status quo, according to criteria external to the algorithm.

Once the target state is determined, the *plan synthesis* phase takes place to detect the best possible course of actions to reconfigure the input RBAC state. These actions represent the *maintenance plan* which has to be implemented into the target access control system. The admissible actions to reconfigure an RBAC state are described in "Plan synthesis" section.

## Scope of application

Role maintenance may also act as a *fixing* or *mining* tool, depending on how it is applied; indeed, the state fed into it can be:

| | |
|---|---|
| Fix | A hand-made, top-down RBAC state, which despite possibly containing (several) errors cannot be radically overhauled because it has been agreed on by several stakeholders; here, role maintenance is basically used to gradually shepherd the hand-made model to the reality of business processes; |
| Tune | An in-place, fully satisfactory RBAC state which needs to undergo maintenance to accommodate exogenous events, such as updates in business processes, deployment/replacement of information systems, relocation of employees, etc.; |
| Mine | An empty state granting no permission to anyone; in this case, role maintenance is used as an incremental role mining procedure whereby the administrator generates a well-behaved and well-structured RBAC state by monitoring missing permissions on-line and judiciously granting them. |

We focus on the "tune" use case in the rest of the paper.

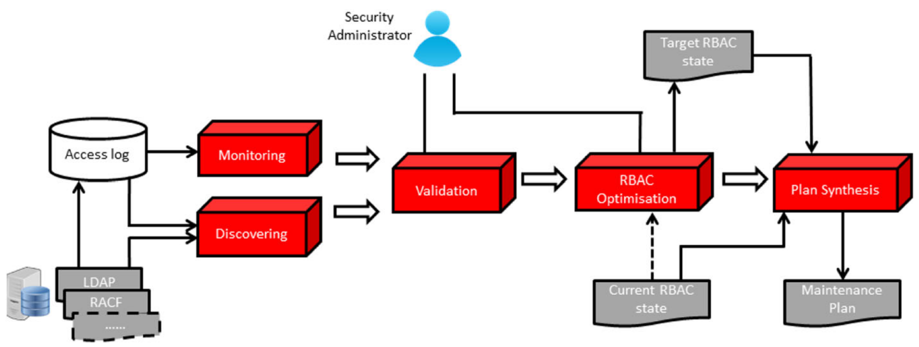


**Fig. 1** RBAC Maintenance process

### A working example

Let us consider a consulting firm with dozens of employees. They use two applications that store data into a company database. The first application supports the *publishing* activity of the company. The second application is used for *marketing* purposes, specifically in the definition of campaigns for mainstream media. An independent in-house *quality assurance* process ensures the compliance of the actions taken through the publishing and the marketing platform with all relevant guidelines and regulations. Employees also use a *general*-purpose Internet service and an in-house e-mail server, through which they *communicate* with institutional stakeholders. An Intranet is available for internal communication and to access *business* applications. The *HR* department manages a database with employees' data. A *web site* describes the services offered by the company on the web. The application server and the *databases* are managed by different administrators.

An RBAC solution is in place at this company: Roles have been crafted so that each employee is enabled to perform only the task(s) he is involved in. The set of (business and technical) roles are: *publishing*, *marketing*, *quality assurance*, *general communication*, *business communication*, *HR management*, *server administration* and *database administration*. As usual, each role is associated with the set of permissions required to carry out the corresponding task(s). For example, the marketing (publishing) role entails the permission to access the marketing (publishing) application, in addition to the permission to access the shared business database. The business communication role grants the permission to use the mail server and the Intranet, while the general communication role gives access to the Internet. Finally, the database administration role is required to operate on the business and the HR databases.

Let us suppose the RBAC manager at this company is now faced with these urgent requests: An employee is about to join the marketing division and needs the corresponding permissions; at the same time, another employee who is already in that division just realised he lacks some much needed permissions to operate; another employee from the marketing division is required to (also) work in the publishing process for a while. Finally, yet another employee is to be temporarily granted permissions to help the database administrator, but only on a non-critical dataset.

The RBAC administrator would like to be supported in the process of re-configuring the RBAC state. In particular, he would like to automatically obtain good answers to questions such as:

- *"How to modify the current RBAC state to implement the necessary variations and nothing more?"*
- *"Is it possible to operate on the Permission-to-Role and/or Role-to-User assignments without adding new roles?"*
- *"Would it be better to just create new custom roles?"*
- *"Is this an occasion to merge some existing roles into less and more clear-cut business figures?"*
- *"Once I'm at it, can I simplify the current role set while keeping the solution stable w.r.t. to the desired RBAC state?"*
- *"Once a target RBAC state is determined, which is the minimum possible set of applicable reconfigurations to Permission-to-Role and/or Role-to-User assignments which implement it?"*

## Formalisation

We use the language of propositional logic to represent *RBAC optimisation* instances. The question we ask is not simply about the satisfiability of a logic statement by some propositional model (SAT), because we need to also express preferences between conflicting objectives (similarity and simplicity of the output RBAC state). It turns out a slightly different, "optimising" framework perfectly suited to our needs is Max-SAT, which considers all possible truth assignments to the input formula and picks the best one according to a fitness metric defined in the Max-SAT language itself ("SAT and Max-SAT", "Hard constraints", "Soft constraints" and "RBAC optimisation" sections).

As far as the *plan synthesis* problem is concerned, we use the language of domain-dependent automated planning by (Ghallab et al. 2004). This framework supports the definition of optimisation problems which consist in finding the course of RBAC actions to transform the input into the target RBAC state while minimising the number of required actions as fitness metric ("Plan synthesis" section). By solving such a planning problem, one finds the best (shortest) sequence of actions to reach the desired RBAC state.

### SAT and Max-SAT

A SAT problem (Cook 1971) is solved by assigning a truth value in *{True, False}* to each Boolean variable that appears in the input propositional formula in such a way that the formula as a whole evaluates to *True*. If at least one such assignment exists, the formula is *satisfiable* and the satisfying assignment is called a *model*; otherwise, the formula is *unsatisfiable* (inconsistent) and it has no model. This problem is intractable (NP-complete) in general, yet several highly efficient solvers exist that in practice solve real-world problems with millions of variables in reasonable time.

Given any $UPA$ matrix, its factorisation into $UA \otimes PA = UPA$ for some number of roles $k$ can be directly expressed in SAT as:

$$\bigwedge_{ij}\left[UPA_{ij} \leftrightarrow \vee_{l=1}^{k}(UA_{il} \wedge PA_{lj})\right] \qquad (1)$$

Equation (1) is a SAT instance with $k(m+n)$ variables. Without loss of generality, most SAT solvers require input formulas in Conjunctive Normal Form (CNF), i.e., a conjunction of clauses, each clause being a disjunction of literals. By either reworking the formula (possibly enlarging its size) or adding auxiliary variables, it is possible to rewrite any non-CNF problem like (1) in CNF.

In SAT, all clauses have to be satisfied for the formula to be declared satisfiable. The *Maximum Satisfiability problem* (Max-SAT) is a variant of SAT that relaxes this premise. The goal of Max-SAT is to find an assignment that makes true the *largest possible subset* of clauses: some may remain unsatisfied. *Partial* Max-SAT is another variant in which some clauses are declared *hard* and must be satisfied no matter what, as in SAT, while others are declared *soft*: As many of them as possible must be satisfied, à la Max-SAT. Yet another variant is *Weighted* Max-SAT, that generalises Max-SAT by associating a positive real or integer weight to each input clause: An assignment that *maximizes the sum of the weights* of satisfied clauses, rather than just their number, is sought (clauses left unsatisfied contribute nothing to the sum).

Finally, the *Weighted Partial* Max-SAT (WPMS) language combines both features: It finds an assignment that satisfies all hard clauses while maximizing the cumulative weight of satisfied soft clauses. We employ WPMS here (and call it simply Max-SAT).

## Hard constraints

Hard clauses are well suited to represent invariants that must hold on any RBAC state we may possibly output, and in particular to capture condition (a) from "Maintenance of an RBAC state" section.

The input RBAC state is represented by two boolean matrices $UA^0$ and $PA^0$ such that $UA^0 \otimes PA^0 = UPA^0$. $UA^0$ and $PA^0$ are respectively a $m \times k^0$ and a $k^0 \times n$ boolean matrices having non-empty roles, i.e.,

$$\forall_{t=1,..,k^0}\left(\bigvee_{i=1,..,m} UA^0_{i,t} = True\right) \wedge \left(\bigvee_{j=1,..,n} PA^0_{t,j} = True\right)$$

We consider exceptions $e \in \mathbb{E}$ and violations $v \in \mathbb{V}$. An exception $e : p \rightarrow u$ is meant to grant permission $p$ to user $u$, assuming it was $UPA^0_{u,p} = False$. A violation $v : p \rightarrow u$ is meant to revoke permission $p$ from user $u$, assuming it was $UPA^0_{u,p} = True$. We define $EXC(\mathbb{E})$ as the $m \times n$ boolean matrix that is *True* at position $(u,p)$ if $e : p \rightarrow u \in \mathbb{E}$ and *False* otherwise, and $VIOL(\mathbb{V})$ as the $m \times n$ boolean matrix that is *True* at position $(u,p)$ if $v : p \rightarrow u \in \mathbb{V}$ and *False* otherwise. We then need $k(m+n)$ variables to represent the unknown elements of the updated Role-to-User ($UA$) and Permissions-to-Role ($PA$) matrices, where $m$, $k$, and $n$ are the number of users, roles, and permissions.

We have to find two boolean matrices $UA$ and $PA$ such that:

$$UA \otimes PA = UPA^0 \oplus (EXC(\mathbb{E}) + VIOL(\mathbb{V})) = UPA \quad (2)$$

where $\oplus$ is the element-by-element exclusive-or operator, while $+$ is the element-by-element or operator. In terms of the variables defining UA and PA, for each user $i = 1,..,m$ and for each permission $j = 1,...,n$, this can be written as:

$$\bigwedge_{i,j} F_{i,j}, \text{ where } F_{i,j} \coloneqq \left(\vee_{t\in[1,k]} UA_{i,t} \wedge PA_{t,j}\right)$$
$$= UPA^0_{i,j} \oplus (EXC_{i,j}(\mathbb{E}) + VIOL_{i,j}(\mathbb{V}))$$

The formulas $F_{i,j}$ are not in CNF, thus we convert them into CNF. Two different cases have to be taken into account:

$$\begin{pmatrix} \dots & \dots & \dots \\ UA_{i_0,1} & \dots & UA_{i_0,k} \\ \vdots & \vdots & \vdots \\ UA_{i_1,1} & \dots & UA_{i_1,k} \\ \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} \dots & PA_{1,j_0} & \dots & PA_{1,j_1} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & PA_{k,j_0} & \dots & PA_{k,j_1} & \dots \end{pmatrix} =$$

$$\begin{array}{c|ccccc} & & j_0 & & j_1 & \\ \hline & \dots & \dots & \dots & \dots & \dots \\ i_0 & \dots & False & \dots & \dots & \dots \\ & \vdots & \vdots & \vdots & \vdots & \vdots \\ i_1 & \dots & \dots & \dots & True & \dots \\ & \dots & \dots & \dots & \dots & \dots \end{array}$$

If $UPA_{i,j}$ is *False*, such as at index $(i_0, j_0)$ above, we have:

$$\left[\bigvee_{t\in[1,k]} (UA_{i,t} \wedge PA_{t,j})\right] = False$$

which can be rewritten as a set of clauses by negating both sides:

$$hasnt_{i,j}(UA, PA) :=\ \bigwedge_{t\in[1,k]} (\neg UA_{i,t} \vee \neg PA_{t,j}) \qquad (3)$$

If $UPA_{i,j}$ is *True*, such as at index $(i_1, j_1)$, we have:

$$\left[\bigvee_{t\in[1,k]} (UA_{i,t} \wedge PA_{t,j})\right] = True$$

Here we apply the Tseytin transformation (Tseitin 1983) by inserting $k$ auxiliary variables $\{aux_1, \dots, aux_k\}$, one for each disjunct:

$$\begin{cases} \bigvee_{t\in[1,k]} aux_t \\ \bigwedge_{t\in[1,k]} aux_t \leftrightarrow (UA_{i,t} \wedge PA_{t,j}) \end{cases}$$

which is equivalent to the CNF form:

$$has_{i,j}(UA,PA):=\begin{cases}\bigvee_{t\in[1,k]} aux_t\\ \bigwedge_{t\in[1,k]}(\neg aux_t \vee UA_{i,t}) \wedge (\neg aux_t \vee PA_{t,j})\\ \bigwedge_{t\in[1,k]}(aux_t \vee \neg UA_{i,t} \vee \neg PA_{t,j})\end{cases} \tag{4}$$

Overall, the hard clauses that guarantee we are both incorporating the exceptions in $\mathbb{E}$ and removing the violations in $\mathbb{V}$ are:

$$exc\_viol(UA,PA):=\bigwedge_{i,j|UPA_{i,j}} has_{i,j}(UA,PA) \wedge \bigwedge_{i,j|\neg UPA_{i,j}} hasnt_{i,j}(UA,PA) \tag{5}$$

## Soft constraints

We use soft clauses to express constraints that we know are in trade-off and cannot always be satisfied simultaneously. In particular, we want to capture the conditions (i) and (ii) at point (b) in "Maintenance of an RBAC state" section.

**Identity.** To express the condition that the output RBAC state $(UA, PA)$ must be identical to the input state $(UA^0, PA^0)$ we write:

$$\begin{cases}\bigwedge_{i\in[1,m],j\in[1,k]} UA_{i,j} \leftrightarrow UA^0_{i,j}\\ \bigwedge_{i\in[1,k],j\in[1,n]} PA_{i,j} \leftrightarrow PA^0_{i,j}\end{cases}$$

which in CNF is:

$$eq(UA,PA):=\begin{cases}\bigwedge_{i,j}\left(UA_{i,j}\vee\neg UA^0_{i,j}\right)\wedge\left(\neg UA_{i,j}\vee UA^0_{i,j}\right)\\ \bigwedge_{i,j}\left(PA_{i,j}\vee\neg PA^0_{i,j}\right)\wedge\left(\neg PA_{i,j}\vee PA^0_{i,j}\right)\end{cases} \tag{6}$$

Note that $exc\_viol(UA,PA) \wedge eq(UA,PA)$ is unsatisfiable by construction in the traditional SAT sense.

**Sparsification.** To express the condition that the output RBAC state $(UA, PA)$ must be as "simple" as possible we need to adopt some notion of simplicity. The two simplifying conditions we consider are: *(i)* a reduction in the number of Roles-to-User and/or Permission-to-Role assignments ("sparsification"), and *(ii)* a reduction in the number of roles (see next section, "Contraction").

According to (i), the sparsest possible RBAC state would be:

$$sparse(UA,PA):=\begin{cases}\bigwedge_{i\in[1,m],j\in[1,k],UA^0_{i,j}=1}\neg UA_{i,j}\\ \bigwedge_{i\in[1,k],j\in[1,n],PA^0_{i,j}=1}\neg PA_{i,j}\end{cases} \tag{7}$$

**Contraction.** Sometimes, incorporating an exception or removing a violation makes it possible to satisfy Eq. 2 with one less role than those present in $(UA^0, PA^0)$. The soft constraint $sparse(UA, PA)$ is not sufficient to prefer a reduction in the number of roles over similarly-sized but sparse unassignments from $UA$. The fact that role $j$ is no longer in use in the target RBAC state is represented by a column of 0's in the Role-to-User assignment matrix ($UA$); so the condition is:

$$contr(UA):=\bigwedge_{j\in[1,k]} unused_j(UA) \tag{8}$$

where $\{unused_j\}$ are auxiliary variables telling whether role $j$ is assigned to someone ($unused_j = False$) or not ($unused_j = True$):

$$unused_{contr}(UA):=\bigwedge_{j\in[1,k]}\left[unused_j(UA)\leftrightarrow \wedge_{t\in[1,m]}\neg UA_{t,j}\right] \tag{9}$$

**Expansion.** Dually, it is possible that the only way to incorporate an exception or removing a violation is by adding one role to the state, i.e., one column to $UA$ and one row to $PA$. In a propositional encoding, the only way to accommodate for a possible expansion of the matrices we handle is to "reserve space" in advance for the additional row and column. This means we actually work with an RBAC state made by a matrix $UA^+$ with $k + |\mathbb{E}| + |\mathbb{V}|$ columns and a matrix $PA^+$ with $k + |\mathbb{E}| + |\mathbb{V}|$ rows. The condition that these additional roles stay unused is:

$$noexp(UA):=\bigwedge_{j\in[k+1,k+|\mathbb{E}|+|\mathbb{V}|]} unused_j \tag{10}$$

where $unused_j$ is an auxiliary variable telling whether the corresponding additional role remains unused:

$$unused_{exp}(UA):=\bigwedge_{j\in[k+1,k+|\mathbb{E}|+|\mathbb{V}|]}\left[unused_j\leftrightarrow\wedge_{i\in[1,m]}\neg UA^+_{i,j}\right] \tag{11}$$

$$unused(UA):=unused_{contr}(UA)\wedge unused_{exp}(UA) \tag{12}$$

In an encoding where both contraction and expansion constraints are added, we enforce a coherent interplay between the two opposite effects by saying they must not happen simultaneously:

$$mutex(UA):=noexp(UA)\vee\neg leastcontr(UA) \tag{13}$$

where *leastcontr* tells weather at least a role is no longer in use:

$$leastcontr(UA):=\bigvee_{j\in[1,k]} unused_j(UA)$$

## RBAC optimisation

We have all the pieces to present the complete encoding of the RBAC optmisation problem as a Weighted Partial Max-SAT instance. First, let us list the hard constraints (HC) that any feasible solution must comply with. In addition to those from "Hard constraints" section, all the auxiliary contraints from "Soft constraints" section are hard, because auxiliary propositional variables cannot cause inconsistencies on their own, yet they must always be assigned a consistent value:

$$HC(UA,PA): = exc\_viol\left(UA^{+},PA^{+}\right) \wedge unused(UA) \wedge mutex(UA) \quad (14)$$

Soft constraints (SC) are used to express and balance two potentially conflicting objectives of the RBAC maintenance procedure:

1. maximizing the *similarity* between the target and the origin RBAC state; in this sense, the higher the weight we assign to clauses in (6), the better;
2. maximizing the *simplicity* of the target RBAC state (independently of the origin RBAC state); in this sense, the higher the weight we assign to clauses in (7), (8), and (10), the better.

Let us assume the weight of a given clause $C$ is a real value[1] $w \in [0,1]$, noted $w : C$. The notation $w : (C_1 \wedge \ldots \wedge C_n)$ is a shorthand for $w/n : C_1 \wedge \ldots \wedge w/n : C_n$, while $w_1 : (w2 : C)$ is interpreted as $(w_1 \cdot w_2) : C$. The three components (7), (8), and (10) of the "simplicity" objective are merged into one weighted set of clauses, noted $SIMP_{k^+,k^-}(UP,PA)$ and defined as:

$$\nu : \left[k^{-}: contr(UA,PA) \wedge sparse(UA,PA) \wedge k^{+}: noexp(UA,PA)\right] \quad (15)$$

Here, $k^- \geq 0$ is a parameter meant to quantify how much we reward the elimination of a role versus to the sparsification of the matrix; $k^+ \geq 0$ measures our adversion to introduce a new role if not strictly necessary. $\nu : = \left(1+k^- + k^+\right)^{-1}$ is a normalization factor meant to ensure the weights of the three components sum up to 1. We assume $k^- > k^+ > 1$, i.e., we value the possibility to expunge a role more than the removal of an equivalent number of assignments; adding roles is a last resort.

Finally, let $\beta \in [0,1]$ be a balancing parameter that measures the extent to which the RBAC administrator is interested in a simplified ($\beta$ closer to 1) VS a stable ($\beta$ closer to 0) RBAC state; we pose:

$$SC_{\beta,k^-,k^+}(UA,PA): = (1-\beta): eq(UA,PA) \wedge \beta : SIMP_{k^+,k^-}(UA,PA) \quad (16)$$

Given (i) the input RBAC state $\left(UA^0,PA^0\right)$, (ii) the exceptions list $\mathbb{E}$ to incorporate and the violations to remove in $\mathbb{V}$, and (iii) some specific values for the parameters $\beta, k^+$, and $k^-$, we submit to a Max-SAT solver the hard clauses (14) plus the soft clauses (16). The Max-SAT solver returns a *fixed* RBAC state $\left(UA^+,PA^+\right)$, i.e., a truth assignment to all the $(k+|\mathbb{E}|\cdot|\mathbb{V}|)\cdot(m+n)$ variables that define $UA^+$ and $PA^+$. Note that the encoding is satisfiable by construction, so we always obtain a fixed state.

**Plan synthesis**

The Plan synthesis phase is activated after the *RBAC optimisation* in order to determine the plan of actions a security administrator has to carry out to deploy the target RBAC state. The point here is that the Max-SAT solver reasoned on the initial and target RBAC states independently of the course of actions eventually required to transform the former into the latter. The problem of finding such plan, possibly the "best" or "cheapest" one according to some formal metrics, is not trivial.

We formalise the problem as a Planning problem within the PDDL framework (McDermott et al. 1998), one of the reference languages for the planning community. In PDDL, several concepts have to be formalised to produce workable instances: *objects* are the entities of interest in the world; *predicates* are the properties of the objects we are interested in, and are either true or false at a certain point; *actions* change the state of the world by changing the truth value of predicates; they have *preconditions* that must be met before considering them for execution; the set of predicates that are true at the beginning is called *initial state*; the set of predicates we want to be true in the final state is called the *goal* state.

In our formalisation we consider *user*, *permission*, and *role* object types and two predicates: $userRole(u,r)$ (true iff role $r$ is assigned to user $u$) and $rolePermission(r,p)$ (true iff the permission $p$ is entailed by role $r$). These two predicates represent the state of $UA$ and $PA$. The *initial* and the *goal* state are defined as the conjunction of predicates which represent the assignments in the initial and final Permission-to-Role and Role-to-User RBAC states (see Eqs. 17–18 where $\mathbb{N}$ represents the set of potentially added roles, $|\mathbb{N}| \leq |\mathbb{E}| + |\mathbb{V}|$).

$$initial\left(UA^0,PA^0\right): = \begin{cases} \bigwedge_{i\in[1,m],j\in[1,k],UA^0_{i,j}=1} userRole(u_i,r_j) \\ \bigwedge_{i\in[1,m],j\in[1,k],UA^0_{i,j}=0} \neg userRole(u_i,r_j) \\ \bigwedge_{i\in[1,k],j\in[1,n],PA^0_{i,j}=1} rolePermission(r_i,p_j) \\ \bigwedge_{i\in[1,k],j\in[1,n],PA^0_{i,j}=0} \neg rolePermission(r_i,p_j) \\ \bigwedge_{i\in[1,m],j\in[k+1,k+|\mathbb{N}|]} \neg userRole(u_i,r_j) \\ \bigwedge_{i\in[k+1,k+|\mathbb{N}|],j\in[1,n]} \neg rolePermission(r_i,p_j) \end{cases} \quad (17)$$

$$goal\left(UA^+,PA^+\right): = \begin{cases} \bigwedge_{i\in[1,m],j\in[1,k+|\mathbb{N}|],UA^+_{i,j}=1} userRole(u_i,r_j) \\ \bigwedge_{i\in[1,m],j\in[1,k+|\mathbb{N}|],UA^+_{i,j}=0} \neg userRole(u_i,r_j) \\ \bigwedge_{i\in[1,k+|\mathbb{N}|],j\in[1,n],PA^+_{i,j}=1} rolePermission(r_i,p_j) \\ \bigwedge_{i\in[1,k+|\mathbb{N}|],j\in[1,n],PA^+_{i,j}=0} \neg rolePermission(r_i,p_j) \end{cases} \quad (18)$$

*Actions*, whose formalisation depends on the target RBAC system, are defined in terms of input parameters, preconditions, and effects. The set of action templates the planner is allowed to instantiate have to reflect what the RBAC management system at hand exposes to the administrator. In Listing 1–10 we present a fairly basic formalisation of a few core actions common to most RBAC management systems. Listing 1 defines in PDDL the action meant to assign a role $r$ to a user $u$. User $u$ and role $r$ are input parameters to the action (they will be instantiated to specific values by the planner); the preconditions states that the predicate $userRole(u, r)$ has to be false for the rule to fire, i.e., that $r$ is not already assigned to the user. The effect section states that the predicate $userRole(u, r)$ becomes true, i.e., that role $r$ is assigned to $u$ as a consequence of the action. Similarly, Listing 2, 3 and 4 show how to remove a role from a user and assign/remove a permission to/from a role. Listing 5 shows how to remove a role $r$ from all users by posing to false $userRole(u, r)$ for each user $u$ for which this predicate holds. Similarly, Listing 6 adjusts the predicates to invalidate all the role assignments for an input user. We also make it possible to adjust the predicates in order to remove a single permission from each role (see Listing 7) and to strip a single role from all its permissions (see Listing 8). Listing 9 erases the entire input RBAC state by posing all predicates $userRole(u, r)$ and $rolePermission(r, p)$ to *False*. Finally, Listing 10 replaces the assignments of a permission from one role to another by inverting the validity of the corresponding *rolePermission* predicates.

**Listing 1** Assigning a role to a user

```
(:action assignRoleToUser
:parameters (?u - user ?r - role)
:precondition (not (userRole ?u ?r))
:effect (userRole ?u ?r)
)
```

**Listing 2** Removing a role from a user

```
(:action removeRoleFromUser
:parameters (?u - user ?r - role)
:precondition (userRole ?u ?r)
:effect (not (userRole ?u ?r))
)
```

**Listing 3** Assigning a permission to a role

```
(:action assignPermissionToRole
:parameters (?r - role ?p - permission)
:precondition (not (rolePermission ?r ?p))
:effect (rolePermission ?r ?p)
)
```

**Listing 4** Removing a permission from a role

```
(:action removePermissionFromRole
:parameters (?r - role ?p - permission)
:precondition ( rolePermission ?r ?p)
:effect (not (rolePermission ?r ?p))
)
```

**Listing 5** Removing a role from all users

```
(:action removeRoleFromAllUsers
:parameters (?r - role)
:effect (forall (?u - user)
(when (userRole ?u ?r)
(not (userRole ?u ?r))))
)
```

**Listing 6** Removing all roles from a user

```
(:action removeAllRolesFromUser
:parameters (?u - user)
:effect (forall (?r - role)
(when (userRole ?u ?r)
(not (userRole ?u ?r))))
)
```

**Listing 7** Removing a permission from all roles

```
(:action removePermissionFromAllRoles
:parameters (?p - permission)
:effect (forall (?r - role)
(when (rolePermission ?r ?p)
(not (rolePermission ?r ?p))))
)
```

**Listing 8** Removing all permission from a role

```
(:action removeAllPermissionsFromRole
:parameters (?r - role)
:effect (forall (?p - permission)
(when (rolePermission ?r ?p)
(not (rolePermission ?r ?p))))
)
```

**Listing 9** Delete all assignments

```
(:action deleteUAPA
:parameters ()
:effect (and (forall (?u - user)
(forall (?r - role)
(when (userRole ?u ?r)
(not (userRole ?u ?r)))))
(forall (?p - permission)
(forall (?r - role)
(when (rolePermission ?r ?p)
(not (rolePermission ?r ?p))))))
)
```

**Listing 10** Swap a permissions from two roles

```
(:action swapPermissionFromRoleToRole
:parameters (?r1 - role ?r2 - role ?p1
- permission)
:precondition (and (not (rolePermission
 ?r1 ?p1))
(rolePermission ?r2 ?p1))
:effect (and (not (rolePermission ?r2 ?p1))
(rolePermission ?r1 ?p1))
)
```

The set of admissible actions (Listing 1–10), along with the initial (Eq. 17) and goal (Eq. 18) RBAC state are issued to a state-of-the-art PDDL planner to synthesise the maintenance plan. In our formalisation all actions have cost 1, thus the optimising planner (that seeks the cheapest plan) looks for the shortest plan. Certain trivial plans of actions may be considered as baselines. In particular: A full rewrite plan consists in erasing the input RBAC state (through *deleteUAPA* action) and then adding the assignments to achieve the target RBAC state (actions in Listing 1 and 3). The cost (length) of such plan can be easily computed as:

$$baseline_{rewrite} :=1+\left|UA^{+}\right|+\left|PA^{+}\right| \tag{19}$$

A second trivial plan is *diff*, which consists in adjusting the differences in terms of assignments occurring between the target and input matrices (actions in listings 1–4) whose cost is:

$$baseline_{diff} :=\left|UA^{+}-UA^{0}\right|+\left|PA^{+}-PA^{0}\right| \tag{20}$$

We expect the PDDL planner to find a plan with fewer actions than either baseline.

## Solutions and their quality

To assess the quality of $UA^{+}, PA^{+}$ (both per se and w.r.t. $UA^{0}, PA^{0}$) we need some synthetic indicators for the two dimensions we are after: simplicity, similarity and maintainability.

**Similarity** is computed according to the metric defined in Vaidya et al. (2008), which is based on the Jaccard coefficient.

*Role-to-Role similarity.* The similarity between two roles $r_1$ and $r_2$ granting permissions $P_1$ and $P_2$ respectively is defined as:

$$sim_{1-1}(r_1,r_2) := \frac{|P_1 \cap P_2|}{|P_1 \cup P_2|}$$

*Role-to-RoleSet similarity.* The similarity between a role $r$ and a set of roles $R$ is defined as:

$$sim_{1-N}(r,R) := max_{r_x \in R} sim_{1-1}(r,r_x)$$

*RoleSet-to-RoleSet similarity.* The similarity between a set of roles $R_1$ and a set of roles $R_2$ is defined as:

$$sim_{N-N}(R_1,R_2) := avg_{r \in R_1} sim_{1-N}(r,R_2)$$

*Similarity.* Building on the former definition, we pose the similarity of the target RBAC state (with roles $R$) to the source RBAC state (with roles $R_0$) as:

$$sim(R_0,R) := \frac{sim_{N-N}(R_0,R)+sim_{N-N}(R,R_0)}{2} \tag{21}$$

This similarity function is always between 0 and 1; in particular, the value 1 is obtained iff $R \equiv R_0$.

**Simplicity** is defined in different ways in the relevant literature; metrics taken into account to define it include, but are not limited to, the number of roles, assignments, erroneous permission granted, relations among roles, permission assigned directly, and any combination of these metrics. We start from a definition of "absolute complexity" that grows with the number of roles and the number of assignments in *UA* and *PA*:

$$comp(UA,PA) := (|UA|+|PA|)+k^{-} \cdot |R| \tag{22}$$

where $R$ is the set of roles, $k^{-}$ is the parameter to reward elimination of a role versus the the sparsification of the matrixes and *UA* and *PA* are the assignments of permissions and roles[2]. A *relative* complexity is then computed as the ratio between the complexity of the target state and the complexity of the "trivial" admissible state; in such state, each user is assigned one and only one custom role which entails exactly the set of permissions that user has in *UPA*. In this configuration, the *UA* matrix becomes the identity matrix, there are as many roles as users, and all the knowledge about permissions is in *PA*. The absolute complexity of such trivial state is thus $(|UPA|+|U|)+k^{-} \cdot |U|$.

The (relative) simplicity[3] of a state is then:

$$spt=1-\frac{|UA|+|PA|+k^{-} \cdot |R|}{|UPA|+|U|+k^{-} \cdot |U|} \tag{23}$$

## Validation

In this section we validate our maintenance process *(i)* at a *small-scale* in "Maintenance of our working example" section, where we apply it to our motivating example from "A working example" section; *(ii)* at a *larger scale* in "Experimental evaluation" section, where we present experimental results showing its viability in real-word cases.

### Maintenance of our working example

Table 1 formalises (a small version of) the example from "A working example" section. Table 2 show the corresponding RBAC state as Permission-to-Role and Role-to-User assignments.

**Table 1** Permission-to-User assignment matrix (*UPA*) with legend

| U\P | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 | p9 | p10 | p11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| u1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| u2 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| u3 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | $0[1]^3$ | 0 | 0 | 0 |
| u4 | 1 | 1 | 0 | 1 | 0 | 1 | $0[1]^1$ | 1 | 0 | 0 | 0 |
| u5 | 1 | 1 | 0 | 1 | 0 | 1 | $0[1]^2$ | 1 | 0 | 0 | 0 |
| u6 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | $0[1]^4$ | 0 |
| u7 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| u8 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| u9 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| u10 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| u11 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

| Permission | Meaning |
|---|---|
| p1 | Internet Access |
| p2 | Mail Access |
| p3 | HR DB Access |
| p4 | Business DB Access |
| p5 | High Perf. Comp. Resource Access |
| p6 | Intranet Access |
| p7 | Marketing Application Access |
| p8 | Publishing Application Access |
| p9 | HR DB Admin |
| p10 | Business DB Admin |
| p11 | Application Server Admin |

We now imagine that four exceptions to such initial RBAC state—listed in Table 3—are captured by the monitoring system or directly specified by the RBAC administrator. We have to incorporate them.

$e_1$**:** According to the first exception, it is necessary to augment the permissions of user *u*4 by granting him access to the marketing application (permission *p7*). The user already has access to the publishing application and to the business database. Different RBAC states result from different values of the balance parameter $\beta$. For example, if (i) the administrator prefers not to alter the current state too much and he submits the value $\beta = 0.1$ to the algorithm, the exception is incorporated by simply assigning role *marketingFunct* to *u*4 (action 1 in Listing 11). The variation over the current RBAC state is minimal ($sim = 1$) and the complexity of the state is almost unchanged at $opt = 0.252$ (same number of roles and the addition of a single assignment); in case (ii) the administrator values simplicity more and sets the value $\beta = 0.5$, the algorithm again answers by assigning role *marketingFunct* to *u*4, but then carries out further simplifying and adjustments to the input state.

Namely, the *busComm* role is augmented with the Internet access permission (action 2 in Listing 11) thus making it possible to reduce the assignments of role *genComm* to just *u*6 and *u*7 through the sequence of actions 3, 4 and 5 in Listing 11. Simplicity raises to $opt = 0.309$ at the price of lowering the similarity to $sim = 0.958$. Let us suppose in what follows that the RBAC administrator picks option (ii).

$e_2$: It is now required that the marketing application (*p7*) is made accessible to user *u*5 too. Assuming (i) a high interest in similarity ($\beta = 0.1$), the algorithm assigns role *marketingFunct* to *u*5 (action 6 in Listing 11), thus

**Table 2** Permission-to-Role matrix *PA* (top) and User-to-Role matrix $UA^T$ (bottom)

| R\P | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 | p9 | p10 | p11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| bussComm | $0[1]^1$ | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| genComm | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| marketingFunct | 0 | 0 | 0 | 1 | 0 | 0 | 1 | $0[1]^3$ | 0 | 0 | 0 |
| publishingFunct | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| HRManagement | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| QualityAssurance | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| DBAdmin | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| ServerAdmin | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| BussDBAdmin | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[1]^4$ | $[0]^4$ |
| R\U | u1 | u2 | u3 | u4 | u5 | u6 | u7 | u8 | u9 | u10 | u11 |
| bussComm | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| genComm | $1[0]^1$ | $1[0]^1$ | $1[0]^1$ | $1[0]^1$ | $1[0]^1$ | 1 | 1 | $1[0]^1$ | $1[0]^1$ | $1[0]^1$ | $1[0]^1$ |
| marketingFunct | 1 | 1 | 1 | $0[1]^1$ | $0[1]^2$ | 0 | 0 | 0 | 0 | 0 | 0 |
| publishingFunct | $1[0]^3$ | $1[0]^3$ | $0[0]^3$ | $1[0]^3$ | $1[0]^3$ | $0[0]^3$ | $0[0]^3$ | $0[0]^3$ | $0[0]^3$ | $0[0]^3$ | $0[0]^3$ |
| HRManagement | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| QualityAssurance | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| DBAdmin | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| ServerAdmin | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| BussDBAdmin | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[1]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ | $[0]^4$ |

**Table 3** List of exceptions to incorporate in the working example

| Exception | $\beta$ | $\lvert UPA\rvert$ | RBAC state |  | *opt* | *sim* |
|---|---|---|---|---|---|---|
|  |  |  | $\lvert R\rvert$ | $\lvert UA\rvert + \lvert PA\rvert$ |  |  |
| - | - | 50 | 8 | 47 | 0.254 | - |
| $e_1 : p7 \rightarrow u4$ | 0.5 | 51 | 8 | 40 | 0.309 | 0.958 |
| $e_2 : p7 \rightarrow u5$ | 0.1 | 52 | 8 | 41 | 0.307 | 1 |
| $e_3 : p8 \rightarrow u3$ | 0.3 | 53 | 7 | 36 | 0.397 | 0.935 |
| $e_4 : p10 \rightarrow u6$ | 0.1 | 54 | 8 | 38 | 0.338 | 0.958 |

In Tables 1 and 2, we note *old*[ *new*]$^i$ an assignment that is changed from value *old* to value *new* by applying exception $e^i$

maintaining a stable state ($sim = 1$) and essentially the same simplicity ($opt = 0.307$). Alternatively, (ii) the assignment $\beta = 0.5$ causes the algorithm to extend the role *publishingFunct*—which is already assigned to $u5$—with the missing permission *p7*. This is possible since no user is granted $p8$ but not *p7* anymore. Thus, the new *publishingFunct* role enables both business applications, while *marketingFunct* gives access to the marketing application only, and is assigned to $u3$. This new RBAC state has a lower similarity at $sim = 0.958$, but only a slightly improved simplicity at 0.329. Since the optimization is marginal and it comes with a damage to similarity, the administrator picks option (i) over option (ii).

$e_3$: User $u3$ needs to access the publishing application (permission *p7*). It turns out that in the current state all users granted $p8$ should also have *p7*. It follows that the distinction between *marketingFunct* and *publishingFunct* makes no longer sense. It is thus not surprising that with even moderate importance assigned to simplicity ($\beta = 0.3$), the algorithm answers by joining such two roles and reducing the state complexity to $opt = 0.397$, at the price of some state variation ($sim = 0.935$). In particular, role *publishingFunct* is erased (action 7 in Listing 11) and role *marketingFunct*, already assigned to $u3$, is augmented with *p8*, thus enabling the access to both business applications (action 8 in Listing 11)

$e_4$: User *u6* is assigned the responsibility of managing the business DB (permission *p10*). A role enabling the management of all the databases (business and HR, $p10$ and $p9$) already exists: *DBAdmin*. However, *u6* is to be granted access to the business database only; it may be necessary to augment the role set. By applying the role-maintenance algorithm with $\beta = 0.1$, we are indeed returned a target RBAC state with a new role (we call it *BussDBAdmin*) assigned to *u6* and including permission *p10* only. The planning actions to achieve this target state are the last two in the overall maintenance plan of Listing 11. The complexity of the RBAC state is augmented since a new role is introduced ($opt = 0.338$); there is a minimal variation from the initial state ($sim = 0.958$).

**Listing 11** Maintenance Plan

```
1 assignRoleToUser(u4,marketingFunct)
2 assignPermissionToRole(p1,bussComm)
3 removingRoleFromAllUser(genComm)
4 assignRoleToUser(u6,genComm)
5 assignRoleToUser(u7,genComm)
6 assignRoleToUser(u5,marketingFunct)
7 removingRoleFromAllUser(publishing
Funct)
8 assignPermissionToRole(marketingFunct,
p8)
9 assignRoleToUser(BussDBAdmin,u6)
10 assignPermissionToRole(BussDBAdmin,
p10)
```

## Experimental evaluation

To experiment at a larger scale we exercise our algorithms on the semi-synthetic datasets described in "Synthetic datasets: single violations/exceptions" and "Synthetic dataset: Multiple violations/exceptions and Planning" sections. Different Max-SAT solvers are compared in "Choosing a Max-SAT solver" section to select the one that best suits our needs.

We experiment with the inclusion of single exceptions and violations ("Experimental Results: single exceptions/violations" section) and we compared such performance with SOTA maintenance algorithms ("Comparative evaluation" section). We also experiment with the inclusion of queues of exceptions and violations ("Multiple exceptions/violations" section) and with planning the minimum course of actions required to reach the optimised RBAC state ("Planning evaluation" section).

Experimental results have been performed on a 20-core Intel CPU with 138GB of memory and, unless noted otherwise, they have been obtained by setting $k^+ = k^- = 1$. All the results and datasets can be downloaded from the web site Mori and Benedetti [2019].

### *Synthetic datasets: single violations/exceptions*

In order to experiment with single exceptions/violations, we require datasets that include *(i)* some initial RBAC state defined in terms of a set of roles and their assignment to users, and *(ii)* a list of exceptions and violations: couples of Permission-to-User assignments that are missing and couples that are in excess in such RBAC state. To the best of our knowledge, no public dataset provides such information. Most datasets only consist of a binary matrix of Permission-to-User assignments, with no associated RBAC state. Moreover, exception histories are never included.

In order to synthesize a benchmark for the maintenance problem, we start with four existing datasets: our example from "Maintenance of our working example" section, named *SmallComp* here, plus three classical problems of increasing dimension used in the role-mining literature, called *Domino*, *University*, and *Firewall1*. All these instances are defined in terms of their User-to-Permission matrix. Starting from one of these UPA matrices, we generate (purely additive) maintenance instances as follows:

1. Given a positive integer $k$, we randomly select $k$ user permissions in $UPA$ and remove them, thus obtaining $UPA^0$. A random order is assigned to such permissions to obtain the list of exceptions which we ask our algorithm to incorporate;
2. Given $k$, we randomly select $k$ user permissions from $UPA^0$ without removing them from the matrix. This list consists of the violations we ask our algorithm to remove from $UPA^0$;
3. We synthesize a complete RBAC state out of $UPA^0$ via one of the role mining algorithms available in the literature. Given that we aim at an arbitrary (not necessarily optimal) initial state, we adopt *Fast Miner*, a heuristic procedure which returns a sub-optimal set of roles[4]. The corresponding $UA^0$ matrix is then iteratively generated, and the resulting RBAC state is used as initial state for the maintenance algorithm.

The resulting benchmark is described in Table 4.

### *Synthetic dataset: Multiple violations/exceptions and Planning*

We need a dataset to prove the ability of our algorithm to include sets of exceptions and violations in one single step. Typically, security administrators have to include exceptions and violations which are related to one another: Either they manipulate different permissions of the same user, or they alter the same permission for different users. Taking this perspective into account, we generate different maintenance instances as follows:

1. We implement a Markov chain model to generate the list of exceptions and violations. According to this model, a first exception/violation $(u, p)$ is randomly generated. Then, two equally probable options exist. Either the model chooses one among the not-yet selected permissions of the same users $u$ or, alternatively, one among any of the other users with the same permission $p$. Once one option is picked, the same process is repeated with probability 0.8, while with probability 0.2 the process re-starts with a novel exception/violation $(u', p')$. This process is repeated until $k$ exceptions/violations are generated. Finally, the list of exceptions are removed from $UPA$, thus obtaining $UPA^0$.
2. A complete RBAC state is defined from $UPA^0$ as described in the case of single exceptions/violations.

**Table 4** Datasets in our benchmark: single exceptions and violations

| | | | | RBAC state | |
|---|---|---|---|---|---|
| Dataset | #U | #P | Density | $\|R^0\|$ | $\|UA^0\| + \|PA^0\|$ |
| SmallComp | 11 | 11 | 0.207 | 13 | 65 |
| Domino | 79 | 231 | 0.039 | 71 | 1803 |
| University | 493 | 56 | 0.143 | 71 | 8769 |
| Firewall1 | 365 | 709 | 0.123 | 580 | 99713 |
| | | Max-SAT encoding (single exceptions) | | | |
| Dataset | #Excs | $\#V$ | $\#C$ | $\#C_h$ | $\#C_s$ |
| SmallComp | 12 | 6.8602 | 2.9943 | 2.6293 | 3.6502 |
| Domino | 19 | 7.3664 | 1.4476 | 1.4236 | 2.3894 |
| University | 10 | 3.2375 | 2.6436 | 2.5966 | 4.7824 |
| Firewall1 | 32 | 1.9177 | 1.8838 | 1.8878 | 6.2355 |
| | | Max-SAT encoding (single violations) | | | |
| Dataset | #Viols | $\#V$ | $\#C$ | $\#C_h$ | $\#C_s$ |
| SmallComp | 12 | 6.5802 | 2.9363 | 2.5713 | 3.6502 |
| Domino | 19 | 7.3514 | 1.4476 | 1.4236 | 2.3894 |
| University | 10 | 3.2365 | 2.6436 | 2.5956 | 4.7824 |
| Firewall1 | 32 | 1.9177 | 1.8838 | 1.8778 | 6.2355 |

*#U*, *#P*, and *Density* are the number of users, permissions, and the percentage of assignments in the $UPA^0$ matrix, respectively, after the removal of *#Excs* exceptions. $|R^0|$ and $|UA^0| + |PA^0|$ are the number of roles and assignments in the initial RBAC state. $\#V$, $\#C$, $\#C_h$, and $\#C_s$ are the number of variables, clauses, hard clauses, and soft clauses in the Max-SAT encoding (exception and violation cases), respectively

We also carry out experiments aimed at proving the ability of our planning phase to generate good maintenance plans. The dataset for these tests are generated by adopting the same Markov chain model as above. Our intention here is to evaluate the plan required to fix the exceptions/violations without considering possible optimisations to the input state. Thus, we use as input a set of RBAC states already optimised by applying our algorithm with $\beta = 1$.
Table 5 describes the benchmarks generated for the multiple exceptions/violations case, and for the planning phase.

### *Choosing a Max-SAT solver*

Max-SAT solvers are either complete or incomplete. Complete solvers always identify the optimal solution (if one exists), given enough time. Incomplete solvers determine an approximate solution, with no guarantee on how distant it is from the optimum. In practice, on satisfiable non-random instances, they often produce good approximations, and quickly. Furthermore, some solvers work as anytime algorithms, i.e., given any timeout as input, they return the best solution they could possibly find (if any) within the assigned timeout.

*Complete solvers.* Solving large instances of the role-maintenance problem may be unfeasible in practice for complete solvers. To check if this is the case, we chose a few of the best performers at the Max-SAT 2016 international competition Then, we exercise these promising solvers on the datasets from "Synthetic datasets: single violations/exceptions" section, imposing a time limit of 1 h. Results are in the following table.

| Solver | SmallComp | Domino | University | Firewall1 |
|---|---|---|---|---|
| Maximo | $\beta \leq 0.5$ | $\beta = 0$ | $\beta = 0$ | $\beta = 0$ |
| MaxHS | $\beta \leq 0.4$ | $\beta = 0$ | $\beta = 0$ | - |
| LMHS | $\beta \leq 0.3$ | $\beta = 0$ | $\beta = 0$ | - |
| Ahmaxsat | $\beta \leq 0.25$ | - | - | - |

The table shows the maximum value of $\beta$ for which solvers are able to incorporate all the exceptions. The symbol "—" means the solver failed (timeout or memout) on one or more instances.

The *SmallComp* dataset is the only one for which we obtain some results across the entire panel of solvers. Even with such dataset of small instances, as $\beta$ grows most solvers quickly stop responding within the alloted timeout. For example, even the best solver of *SmallComp* (i.e., Maximo) fails instances as $\beta > 0.5$.

As expected, the larger the instances in the benchmark, the sooner complete solvers stop responding. More surprising is how quickly performance deteriorates: By the time we try to solve *Firewall1*, even the best solver doesn't return solutions unless $\beta = 0$.

We conclude that it is not feasible to employ complete solvers to tackle real-world instances of our RBAC maintenance encoding, except perhaps for very small values of $\beta$.

*Incomplete solvers.* Let us shift our attention to state-of-the-art incomplete solvers, as represented by the best performers at Max-SAT 2016. We tested: *Dsat*, *CCLS2015*, *CCEHC*, *OptiRiss*, *Dist*, *WPM3*.

CCEHC (Luo et al. 2017) stands out here, because it is the only solver that: (1) showed a strong performance at Max-SAT 2016, (2) accepts command-line options to tune its solving behavior for industrial benchmarks, and (3) works as an anytime algorithm. Let us first measure how CCEHC behaves in a case clearly beyond reach for complete solvers. We generate 90 maintenance instances of increasing size from *Firewall1* by selecting more and more of its users (i.e., rows); each instance is associated with a single exception to incorporate and generates a Max-SAT encoding of growing size. We ask CCEHC to incorporate the exception leaning strongly towards on optimized, simpler RBAC state ($\beta = 0.8$). Figure 2 shows the minimum timeout needed to obtain a feasible solution for these inputs as a function of their size.

While the minimum timeout grows exponentially, performances over instances in a real-world size range are acceptable; for example, it is possible to obtain a solution in less than one hour for a 337.2*MB* formula that encodes the problem of incorporating one exception into an RBAC state with 165 users and 709 permissions.

But how good are these solutions? To provide an estimate we compare them to optimal solutions returned by complete solvers. As we have shown, *Firewall1* is completely out of reach, so we resort to *SmallComp*. Given that hard constraints are always satisfied by feasible solutions from both complete and incomplete solvers, we focus on the ability of the incomplete solver to satisfy soft constraints. In particular, we compute the average weight of satisfied soft constraints over the total sum of input weights for the 12 different exceptions in the dataset. In Figure 3 we report this metric measured after CCEHC has worked for $t = 2sec$ and then for $t = 180sec$. We include the same metric computed on optimal solutions by complete solvers, which we could obtain for $\beta \leq 0.5$ (1 h timeout).

As expected, the complete solver outputs (slightly) better answers across the line, independently of $\beta$, but CCEHC is not far. These results, though quite comforting, have to be taken with a grain of salt because the small instances in *SmallComp* may not be representative of the general behavior of CCEHC.

**Table 5** Datasets in our benchmark: multiple exceptions and violations

| | | Max-SAT encoding - multiple exceptions/violations | | | |
|---|---|---|---|---|---|
| Dataset | #Runs | #$V$ | #$C$ | #$C_h$ | #$C_s$ |
| SmallComp | 10 | 1.2464 | 4.9284 | 4.4514 | 4.7743 |
| Domino | 10 | 5.6837 | 1.1167 | 1.1007 | 1.6575 |
| University | 10 | 1.7636 | 1.4357 | 1.4137 | 2.1685 |
| | | PDDL encoding | | | |
| Dataset | #Runs | #Atoms | #Init_State | #Goal_State | #Dim(byte) |
| SmallComp | 10 | 9.0223 | 1.1773 | 1.1703 | 8.0004 |
| Domino | 10 | 1.2246 | 4.8134 | 4.8124 | 3.7486 |

#$V$, #$C$, #$C_h$, and #$C_s$ are the number of variables, clauses, hard clauses, and soft clauses in the Max-SAT encoding to include multiple exceptions and violations. #Runs is the number of instance, #Atoms, #Init_State, #Goal_State and #Dim(byte) are respectively the number of atomic prepositions, predicates of the init and goal state and the size of the PDDL instance

## Experimental Results: single exceptions/violations

Incremental Max-SAT solvers seem capable of providing good approximate solutions within reasonable time on real-world instances. In the rest of the experiments we use them (namely, CCEHC) to explore the behavior of our encoding and we set $k^- = 7$ and $k^+ = 2$ to prefer role reduction to matrix simplification.

*Impact of $\beta$.* The first thing we assess experimentally is the impact the value of $\beta$ has on the structure of the fixed RBAC state when performing additions of exceptions and removals of violations. To this end, starting from the input state (see Table 4) we perform maintenance using the

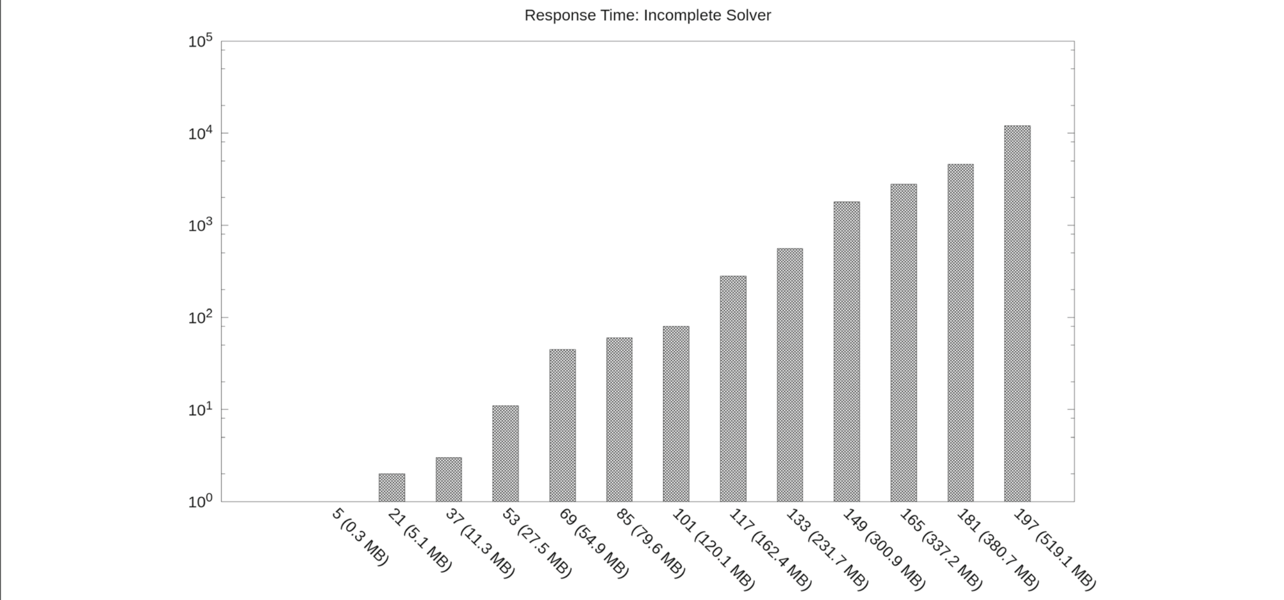


**Fig. 2** Minimum time to compute feasible solutions to Firewall1 (y axis, secs) as a function of the number of users (x axis). Along the x axis we also note the size of the corresponding CNF encoding in Megabytes

input list of exceptions and violations with different $\beta$ settings and then we collect the average results. For the inclusion of each single exception and violation, we set the timeout to 600, 500, and 800 seconds for *smallcomp*, *domino*, and *university*, respectively.

Figure 4 shows that–for all datasets–similarity decreases (and simplicity increases) almost monotonically as $\beta$ grows. Similarity and simplicity curves coming from the addition of exceptions are only slightly different from the results of violation removals (see dashed trends). Figure 5 shows how the corresponding number of roles and assignments decrease on average by increasing the preference towards optimized solutions ($\beta$ close to 1). Conversely, if the encoding leans towards preserving the original RBAC state ($\beta$ close to 0) these numbers stay close to the input values. This holds for both addition of exceptions and removal of violations. None of the instances in the largest of the four datasets could be solved within 1 h:

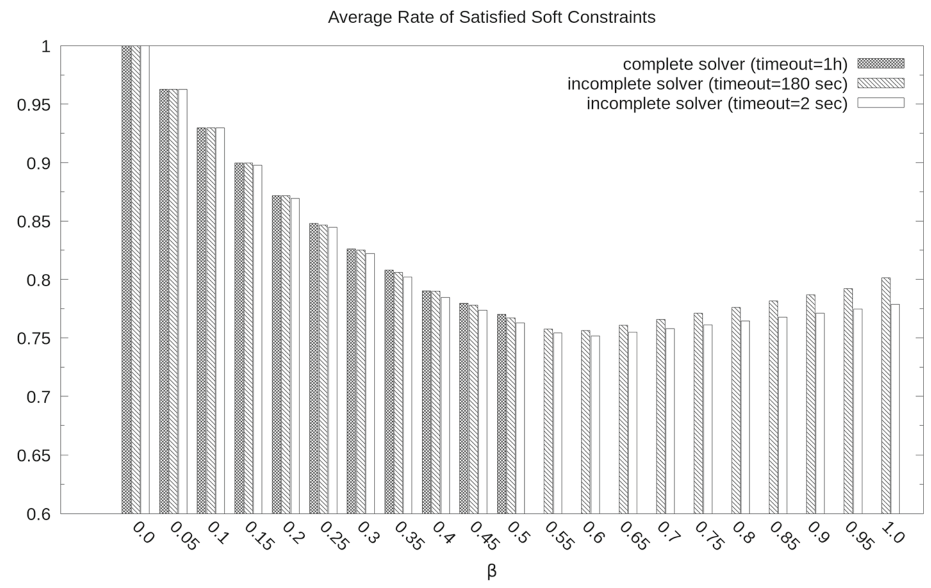
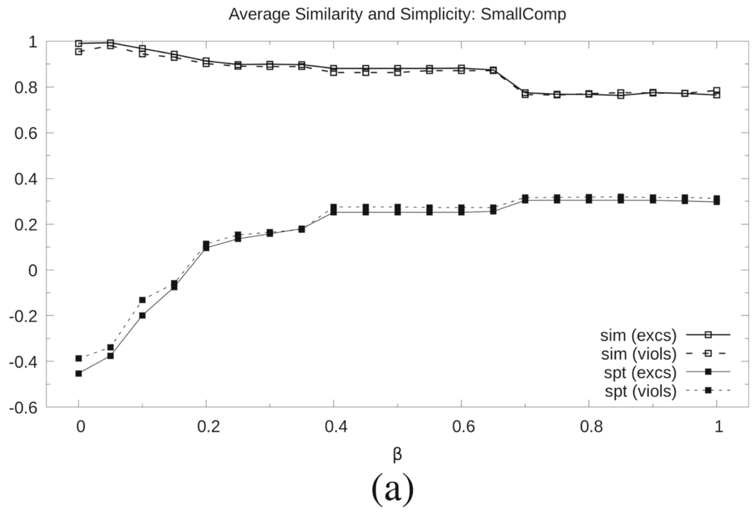
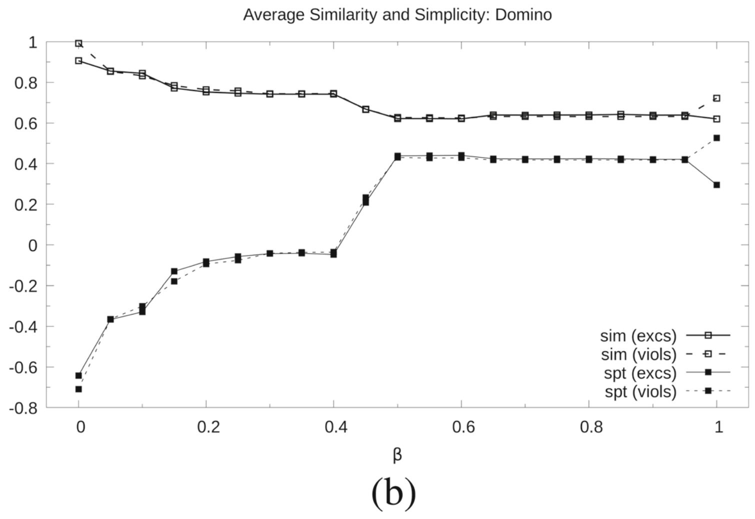


**Fig. 3** Average percentage of satisfied soft clauses (y axis) as a function of the balance $\beta$ (x axis) in the *SmallComp* dataset





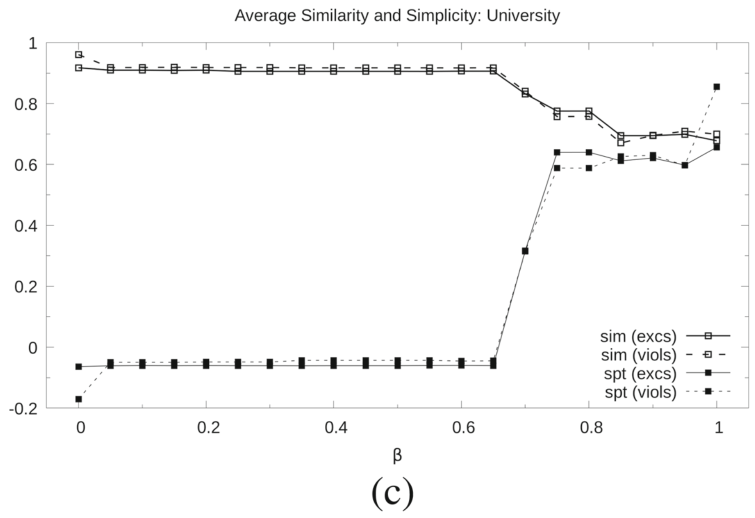
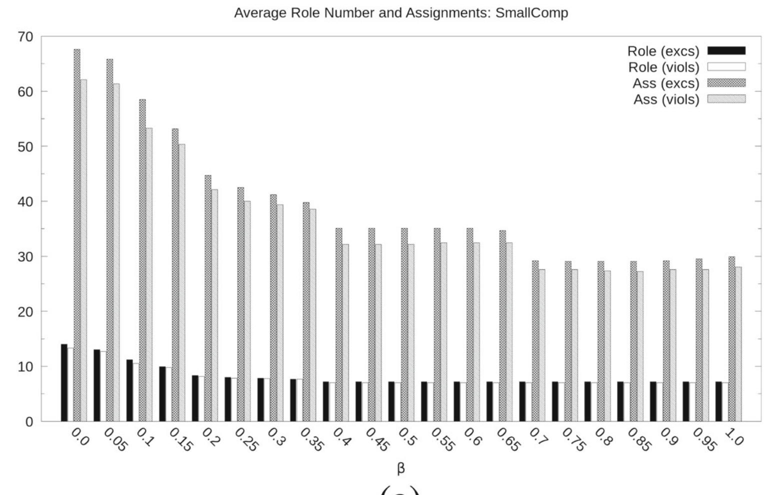
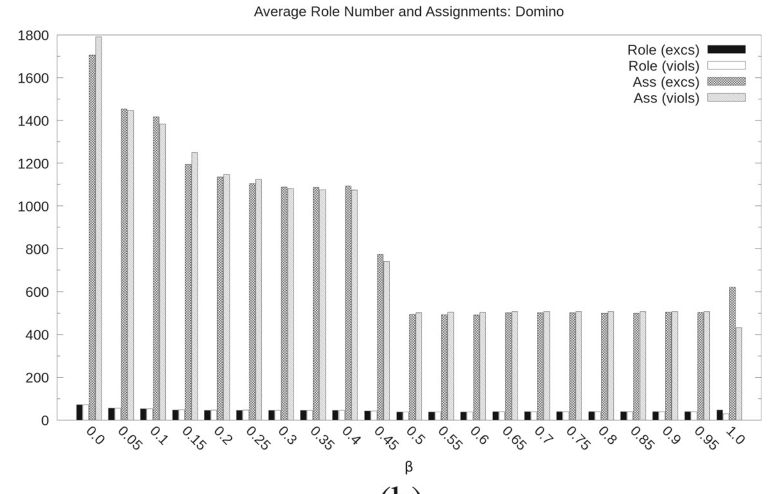


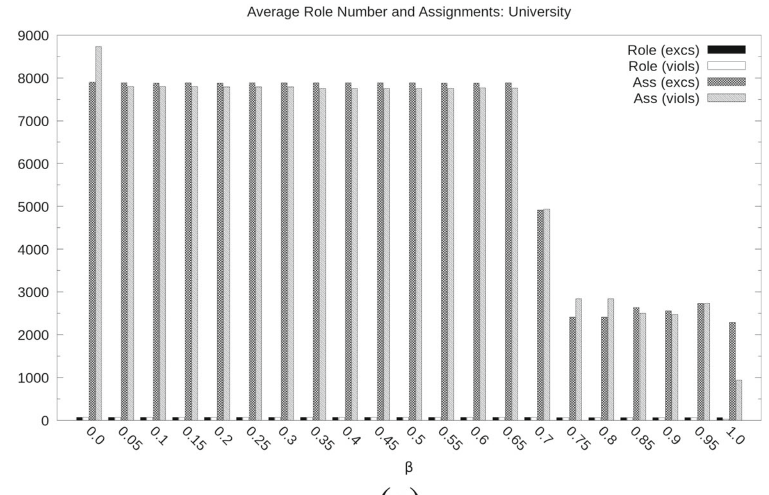
**Fig. 4** Average similarity and simplicity (*y* axis) as a function of the balance $\beta$ (*x* axis) for single exceptions and violations to (**a**) *SmallComp*, (**b**) *Domino* and (**c**) *University* dataset. Twenty-one values of $\beta \in [0, 1]$ are sampled, at regular intervals (step 0.05)



(a)



(b)



(c)

**Fig. 5** Average number of roles and assignments after incorporating single exceptions and violations (y axis) as a function of the balance $\beta$ (x axis) to (**a**) *SmallComp*, (**b**) *Domino* and (**c**) *University* dataset

Firewall1 thus constitute a *hard* problem at present (see Conclusions). Overall, by tuning $\beta$, it seems actually possible to steer the quality of the solution towards similarity or simplicity during maintenance.

*Impact of timeout.* We show how the output RBAC state (simplicity, similarity) changes by granting more time to the solver, at different balance points. Figure 6a and b show the results for timeouts in 10s–600s on the dataset *Domino* (representative of the entire benchmark). Simplicity increases for $\beta \geq 0.25$ (we are optimizing the state as a side effect of the maintenance) while it drifts towards lower values when $\beta < 0.25$ (the price we pay to avoid reworking the RBAC state too much as exceptions arrive). Conversely, similarity improves for low values of $\beta$ while it remains almost stable for $\beta \geq 0.25$. It follows that if the interest towards simplification is high, it always pays to allot more reasoning time to the solver. Conversely, if the interest is in preserving the input state, increasing the reasoning time pays for $\beta < 0.25$, where similarity increases at the expense of simplicity.

*The order of exceptions.* In these experiments, exceptions are incorporated sequentially, one-by-one, as they show up (though bulk incorporation is also possible, as already noted). We are interested in understanding how much the order in which they manifest affects the quality of the eventual state. In principle, the number of roles can either increase (small values of $\beta$) or decrease (large $\beta$). To confirm this, we select the *Domino* dataset and pick a string of 6 exceptions to be incorporated. We generate all their permutations (720) as incorporation sequences. For each sequence, we record the final number of roles, assignments, simplicity and similarity to the initial state (that has 73 roles). Figure 7 shows the distribution generated by 715 paths (each solvable in 60

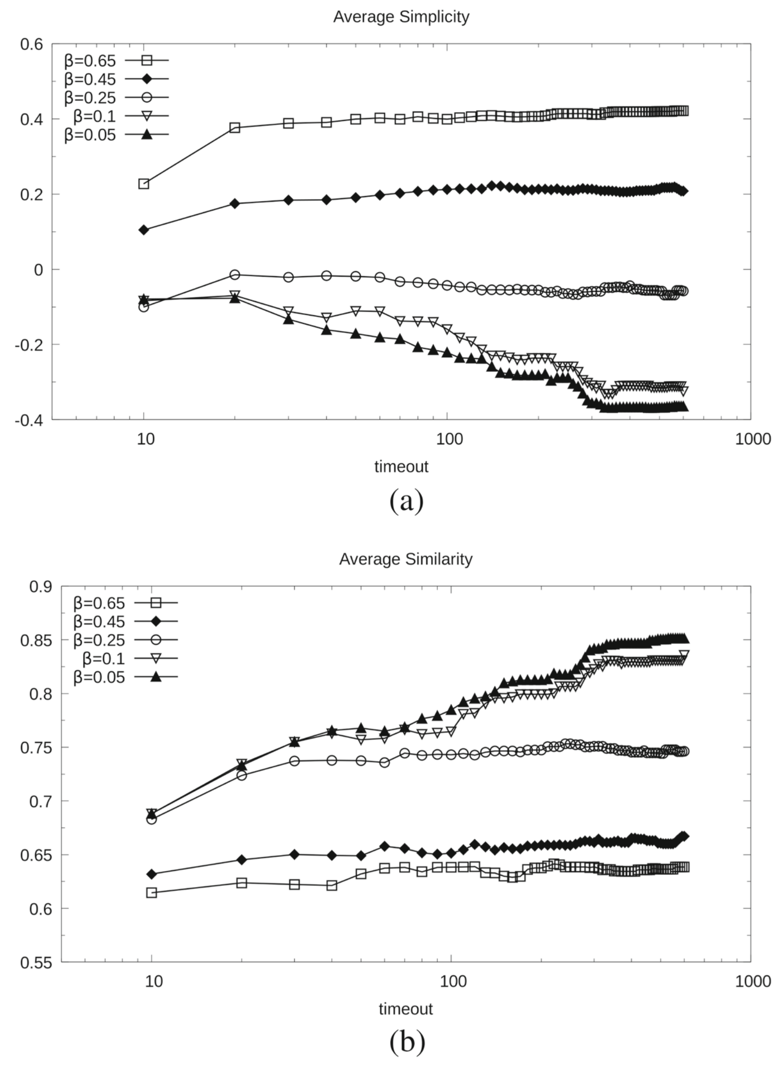


**Fig. 6** Average simplicity (**a**) and similarity (**b**) in *Domino* (y axis) as a function of the timeout (x axis, secs) at difference balance points ($\beta$)

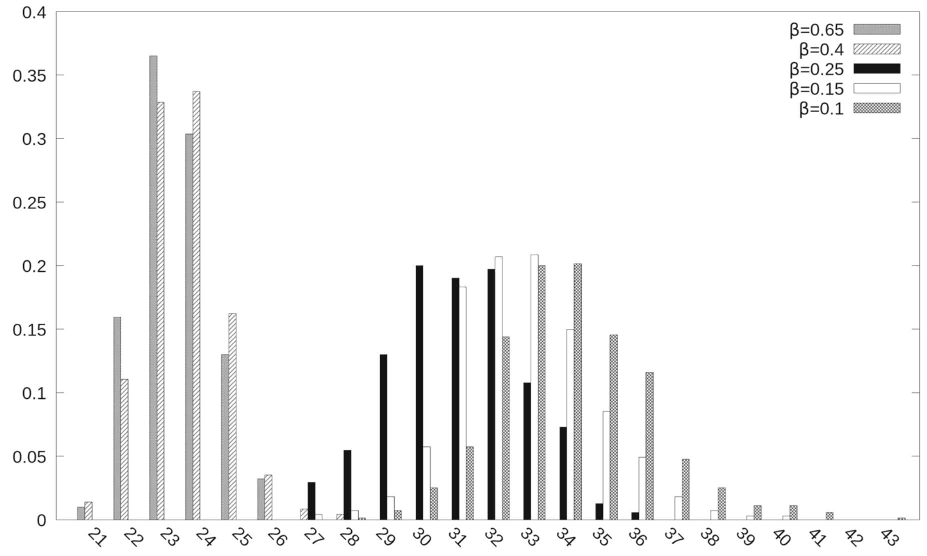


**Fig. 7** Distribution of the number of roles in *Domino* (x axis) after incorporating 6 exceptions in all possible orderings

seconds) with four different $\beta$ configurations. While by construction the final states are equivalent in terms of Permission-to-User assignments, the average value and the variance of the distributions are profoundly impacted by $\beta$; the variance in particular widens as $\beta$ goes close to 0; for example, with $\beta = 0.1$ the number of roles ranges from 28 to 43, corresponding respectively to a 61.6% to 41.1% reduction with respect to initial RBAC state.

This behavior suggests that—independently of the order in which exceptions arrive—by leveraging $\beta$ the RBAC administrator can direct the state over time to become simpler, to accumulate "clutter," or to remain stable in terms of number of roles.

### *Comparative evaluation*

We carry out a comparative evaluation to assess how our approach compares to other state-of-the-art role mining algorithms. We chose two algorithms of different nature: (i) Fast Miner (FM), a classical role mining algorithm which mines roles by neglecting the input RBAC state; and (ii) Minimal Perturbation (MP), which considers the input state while determining the target roles.

Fast Miner is a heuristic procedure which prioritizes a set of eligible roles which are generated from (i) distinct user permission sets and (ii) their intersections. For each eligible role $r$, FM measures how many users have exactly the same set of permissions in $r$ and how many users are assigned a superset of $r$. A parameter exists to balance the interest on (i) over (ii). Minimal Perturbation applies FM to generate a candidate set of roles which is then evaluated according to their coverage of the input UPA matrix, and based on their similarity to the input RBAC state. A real parameter is adopted to tune the interest on coverage versus similarity (see "Related work" section for further details).

We run these algorithms and our approach (RBAC Maintenance via Max-SAT - RMS) over our set of benchmarks to compare the "quality" (in terms of simplicity and similarity) of their output solutions.

All three algorithms take a real parameter in $0.0 - 1.0$ as input. Such parameters control the nature of eligible roles (FM), the bias toward the input set of roles (MP), and the bias toward maintaining the input RBAC state (RMS). We tried different values for all parameters and all algorithms sampling the $0.0-1.0$ intervals uniformly with step 0.1. We use as input a set of randomly selected exceptions from the input *UPA* matrices, and we generate the corresponding input RBAC states using *Pair Count*[5]. Once maintenance is performed with the different algorithms, we record the averages simplicity and similarity obtained by each of them, and a 95% confidence interval. Figure 8 shows the average similarity and simplicity obtained for *SmallComp* (a), *Domino* (b) and *University* (b) datasets. On all input datasets, RMS solutions move considerably in the similarity-simplicity plane as $\beta$ changes, whereas FM and MP solutions stay approximately in the same position, even as the input parameters change.

In *SmallComp* (Fig. 8a) RMS produces solutions which dominate all FM outputs. With high values of $\beta$, RMS produces solutions which are simpler but less stable than FM. MP produces solutions which dominate the ones produced by RMS with $\beta \in [0.2 - 0.3]$, but RMS can either improve similarity ($\beta \in [0.1 - 0.2]$) to the detriment of simplicity, or it may improve simplicity ($\beta \in [0.4 - 1]$) at the expenses of similarity. In *Domino* (Fig. 8b) RMS

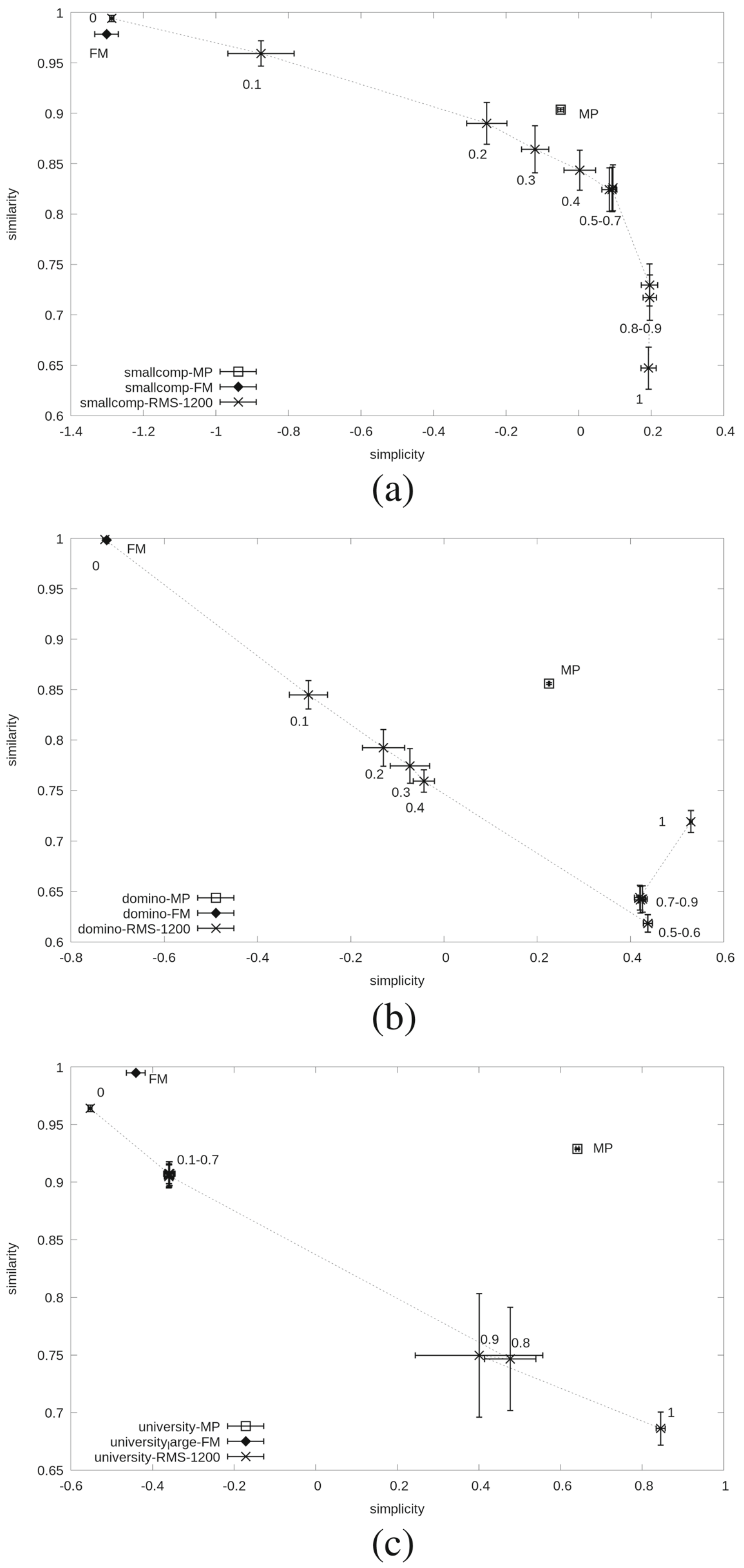


**Fig. 8** Solutions obtained for the datasets (**a**) *SmallComp*, (**b**) *Domino*, and (**c**) *University* by using FM, MP, and RMS (timeout=1200 sec) to perform RBAC maintenance tasks. For RMS, different values of the balancing parameter are plotted (sampled at regular intervals over 0.0 — 1.0 with step 0.1). Solutions are positioned according to their average simplicity (x axis) and similarity (y axis)

achieves solutions of the same quality as FM (with $\beta = 0$) and simpler solutions for higher values of $\beta$. MP dominates RMS solutions for values of $\beta \in [0.1 - 0.4]$, but RMS can still produce higher similarity ($\beta = 0$) and higher simplicity ($\beta \in [0.5 - 1]$).

In *University*, FM produces the highest possible level of similarity. In turn, RMS achieves better simplicity than FM ($\beta \in [0.1 - 1]$) to the detriment of similarity. MP dominates all RMS solutions with $\beta \in [0.1 - 0.8]$ but it cannot attain the maximum absolute similarity ($\beta = 0$) or simplicity ($\beta = 1$) of RMS.

Overall, RMS is able to produce results that are better than MP, either for similarity or simplicity, at the extreme values of $\beta$; however its solutions are dominated by MP's outputs for intermediate values of $\beta$. Crucially, only RMS is able to move the output solution dramatically across the simplicity-similarity plane, according to the user input.

### Multiple exceptions/violations

Multiple exceptions and/or violations can be taken care of in one single problem instance. Starting from the benchmarks described in "Synthetic dataset:Multiple violations/exceptions and Planning" section, we measure the average similarity and simplicity of the solutions resulting from including all the 10 different exceptions and violations at once. We set 600, 500 and 1600 seconds as timeouts for *smallComp*, *domino*, and *university*, respectively.

Figure 9 shows how the same behaviour obtained for the inclusion of single exceptions/violations (see Fig. 4) holds also for the inclusion of multiple exceptions/violations. Independently on the selected dataset, if the interest is to preserve the input state (low $\beta$ values), it is possible to maintain an high level of similarity at the price of having a low simplicity. Conversely, if we are interested towards RBAC optimisation (high $\beta$ values) we can improve simplicity at the price of changing the input state.

It is worth noting that for the experiments concerning *smallComp* and *domino*, we were able to apply the same timeouts as for the inclusion of single exceptions/violations. For *university*, we had to double the timeout to have the solver work with 20 exceptions/violations combined into one instance.

### Planning evaluation

We carry out a set of experiments to evaluate the quality of the maintenance plans created by the planning phase. We experimented with optimal and approximated planners. Optimal planners provide the best possible solution but it proved impractical to use them on our RBAC instances. *Satisficing* planners provide approximated (non-optimal) solutions, but they were able to provide good solutions in reasonable time. We selected *Fast-downward* (Seipp and Röger 2018) as a planner since it won the *satisficing* track at the IPC - International Planning Competition 2018 and it works as anytime algorithm with good performance on our instances.

We selected the *smallComp* and *domino* benchmarks described in "Synthetic dataset:Multiple violations/exceptions and Planning" section and we set different RBAC optimisation and planning timeouts. We respectively set 600, 500 seconds for RBAC optimisation and 30, 480 minutes for planning.

Figure 1 shows the number of actions obtained with our planning phase depending on $\beta$ across the 10 different instances in the *smallComp* and *domino* datasets. These results are compared with two baselines: *rewrite*, which erases the input RBAC state to build the target state from scratch; *diff*, which adjusts the differences between target and input states. The figure shows how our planning process outperforms the baselines both on average and if we consider the best possible result across the 10 runs. The latter is computed by measuring the improvements of our planner with respect to the *diff* baseline; we did not consider *rewrite* to evaluate the best possible results since it only outperforms *diff* when the initial state is not to be taken into account (high values of $\beta$).

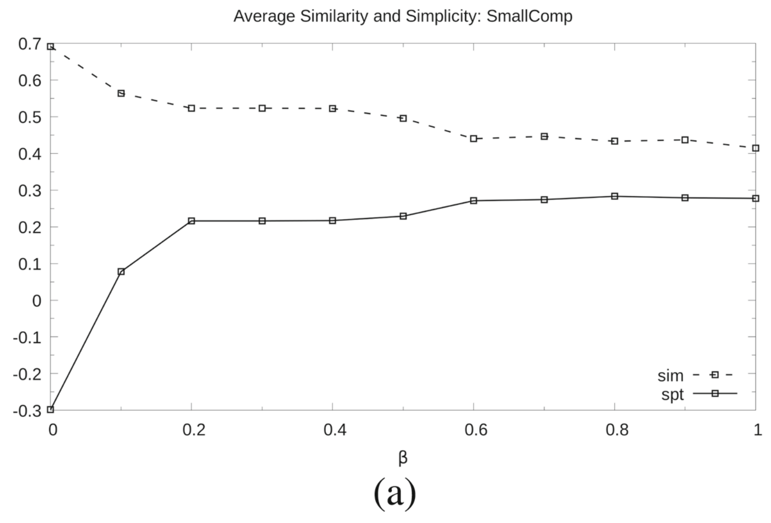
As shown in Fig. 10a (*SmallComp*), our planner outperforms on averages both baselines: it is able to create a plan which is shorter than the differences between matrices (*diff*) and shorter with respect to the set of actions to rebuild the target RBAC state from scratch (*rewrite*). On averages, with $\beta \leq 0.65$ we have an improvement which varies from 21 to 28% across the 10 different runs. As $\beta$ reaches 0.7, the behaviour of our planner becomes basically the same as the baseline *rewrite*, and improvements are no longer possible. For the best case, we can reduce the number of actions to $\beta = 0.65$ and the gain raises from 29 to 53% according to $\beta$. We observe a similar behaviour for *domino* (see Fig. 10(b)): Our planner can shorten the list of actions both on average and in the best case, up to $\beta = 0.35$. Here we measure an improvement which falls into the range from 8 to 37% on average and 63% at maximum. The curves resulting from both datasets show how the planning of actions to reconfigure the input RBAC state is convenient only if we did not change drastically the input state (low $\beta$ values). On the contrary, when the interest is on optimisation (high $\beta$ values), it is worth to discard the input solution and re-build the target state from scratch.

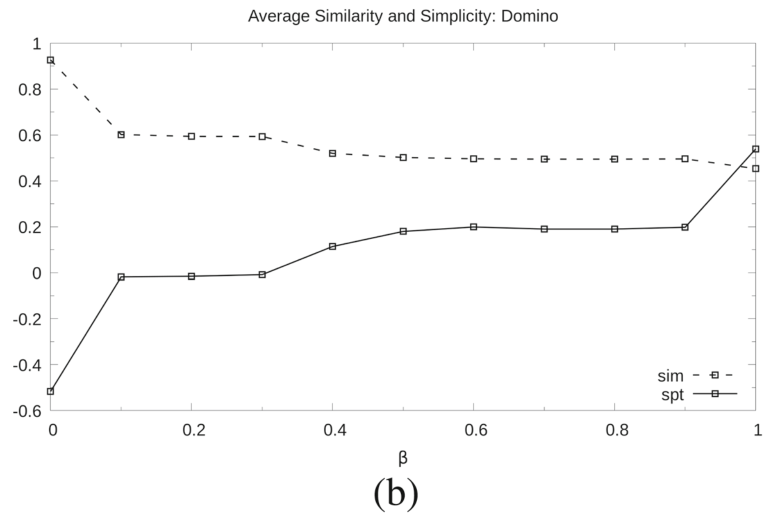
## Related work

In the literature, RBAC optimisation and Plan Synthesis problems have been considered separately.

### RBAC Optimisation

The literature on Role Mining and RBAC state tuning is quite extensive. The works most closely related to this





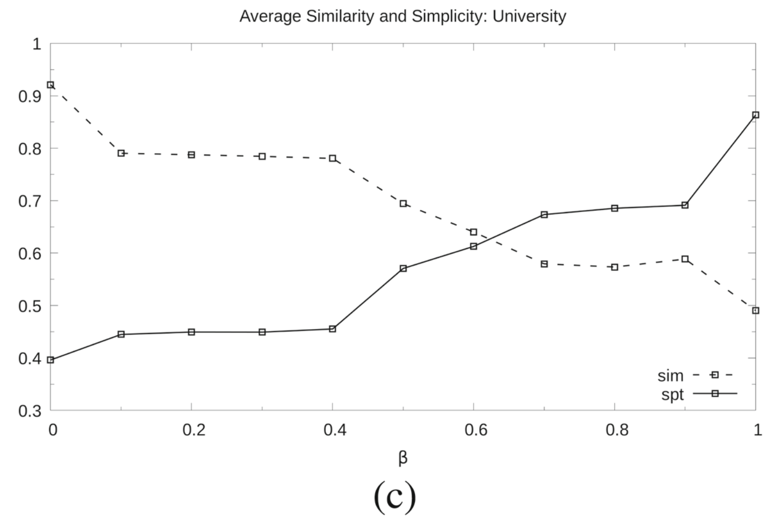
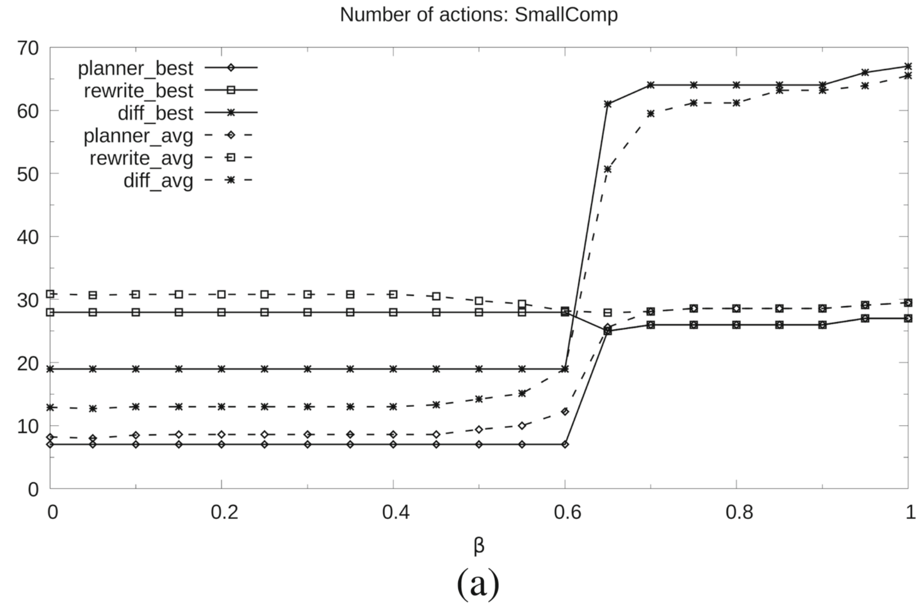


**Fig. 9** Average similarity and simplicity (*y* axis) as a function of the balance $\beta$ (*x* axis) for multiple exceptions and violations to (**a**) *SmallComp*, (**b**) *Domino* and (**c**) *University* dataset. Twenty-one values of $\beta \in [0, 1]$ are sampled, at regular intervals (step 0.05)



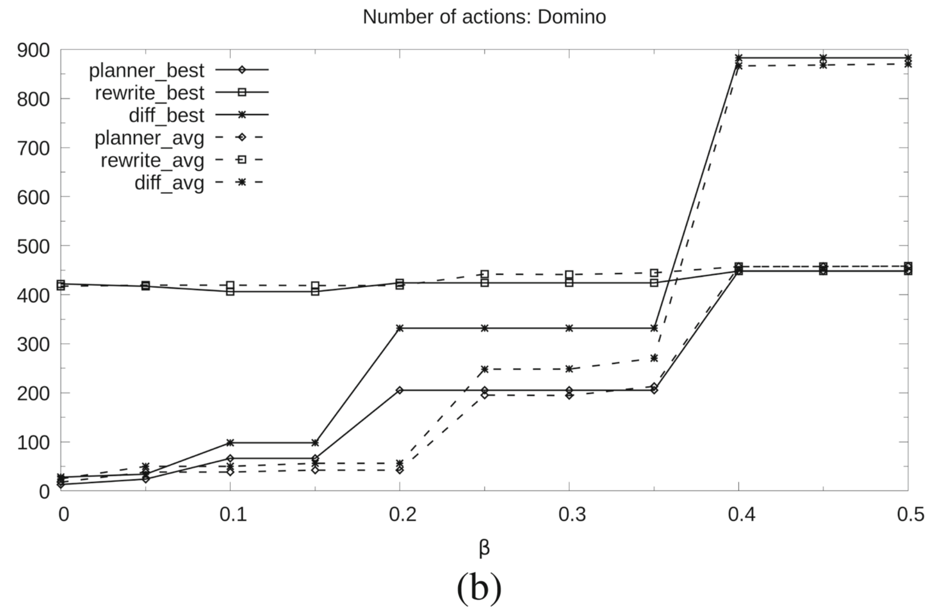


**Fig. 10** As shown in Fig. 10 **(a)** (*SmallComp*), our planner ouperforms on average both baselines: It is able to create a plan which is shorter than the difference between matrices (*diff*) and shorter than the list of actions necessary to rebuild the target RBAC state from scratch (*rewrite*). On average, with $\beta \leq 0.65$ we have an improvement

paper are (Jafarian et al. 2015; Xia et al. 2014; Zhang et al. 2013; Vaidya et al. 2008). These approaches are similar to ours in that they aim at reconfiguring an existing set of roles with the goal of simplifying them.

In Xia et al. (2014) the simplification procedure works as follows: A candidate set of new roles are iteratively generated starting from the trivial role set (one role per permission) and combining them in several ways (pairwise, etc.). The candidate set is kept consistent with the input RBAC state, i.e., each old role must be representable as the union of some new roles. Individual candidates with the lowest "management cost" are preferred, where the management cost is the sum of a fixed component (the same for all roles) plus a variable component which depends on the role cardinality and granularity. A parameter is introduced to balance their relative importance.

The approach presented in Vaidya et al. (2008) more directly balances the quality of the new RBAC state against its similarity to the current one. It first applies a canonical role mining algorithm (Fast Miner) to generate a starting set of candidates. Then it iteratively evaluates roles according to a fitness function and selects the best ones until a solution is obtained that covers all the Permission-to-User assignments of the input RBAC state. The fitness function is a weighted sum of a measure of similarity to the original roles and a measure of coverage of the input Permission-to-User matrix. This algorithm has been extended to work with role hierarchies (Guo et al. 2008; Takabi and Joshi 2010).

A different perspective is adopted in Zhang et al. (2013). This procedure analyzes logs of actual permission utilizations over time and exploits this data to inform the next RBAC state. There are two phases: The first phase applies a variant of the subset enumeration technique of Fast Miner to iteratively generate a set of roles, which are then sorted according to a fitness function. The fitness function is where the traces of real permission usage come into play: It aims at balancing similarity to the original roles with "homogeneity" to actual permission utilization. At each iteration, top ranked roles which cover the input Permission-to-User matrix are selected; this is repeated until fixpoint (two subsequent steps result in the same set of roles) or until a maximum number of iterations is reached. The second phase then assigns roles to users based on a heuristic process which aims to contain redundancy.

A first difference between our approach and (Xia et al. 2014; Vaidya et al. 2008) is that those procedures ignore how the simplified Permission-to-Role matrix they output will impact (in terms of administration costs) the correspondingly adjusted Permission-to-User assignments. This is evident because at no point those procedures take into account the UA matrix or how complex it would be for the administrator to edit it. Furthermore, in Xia et al. (2014) the management cost of a set of roles is defined by only looking at roles per se, and not at the distance from the original. Overall, the link between the input matrices and the output matrices is very indirect: Both couples represent the same Permission-to-User relations, but the difficulty for an administrator to transform $UA^0$ into $UA$ and $PA^0$ into $PA$ is not modeled. Conversely, our encoding explicitly captures the difference of $UA$ from $UA^0$ and $PA$ from $PA^0$, so it relates more directly to the amount of work an administrator will have to do.

Another major difference from (Xia et al. 2014; Vaidya et al. 2008; Zhang et al. 2013) is that our approach captures the RBAC optimization problem declaratively within a logic formalism, and neatly separates the declaration of the constraints and objective function from the search for an optimal solution. This provides several benefits and additional guarantees versus employing custom multi-phase heuristics. One advantage is that future breakthroughs from the community developing complete (and incomplete) Max-SAT solvers will result in an improved RBAC maintenance process without any change to our encoding. Another advantage is that the mechanism controlling the trade-off between quality and similarity provides strong guarantees about the output RBAC state; for example, for $\beta = 0$ we are sure that the output RBAC state is the least variation to the input that is capable of accommodating the intervening exception, whereas for $\beta = 1$ we have the guarantee that the output state is optimal w.r.t. a certain metric and that the input state has been completely ignored[6].

A logic-based formalization of the RBAC state similar to ours is presented in Jafarian et al. (2015). In that case, the Satisfiability Modulo Theory (SMT) framework is adopted; the theory used to expand the expressive power of SAT is Integer Linear Programming (ILP), employed to capture certain boolean-unfriendly quality metrics. The paper shows how to exploit SMT to solve several role mining variants, including one that generates an RBAC state optimal in terms of a combination of WSC (Molloy et al. 2008) and similarity to an input RBAC state. To this end, a heuristic technique is presented to iteratively generate role sets, which are evaluated according to a fitness function that balances the conflicting WSC-vs-similarity objectives. The complete RBAC state is thus taken into account, and configurations that disrupt the Role-to-User assignments are penalized. Other variants consider the permission usage or the RBAC hierarchy as optimisation metrics, resulting in a hybrid role mining approach.

The major difference between this approach and ours (beyond the definitions of certain quality metrics) is that we embrace the inherent nature of the role maintenance process as an *optimisation procedure* by using a logic framework (Max-SAT) meant to optimize an objective function, whereas Satisfiability Modulo Theory (SMT) is a decision procedure meant to prove the consistency of statements. As a result, while the entire problem is captured by a single Max-SAT instance and the reasoning/searching/optimisation stage is decoupled and offloaded to an external solver, in Jafarian et al. (2015) a long sequence of SMT problems is generated in the context of a complex, custom algorithm to achieve the trade-off between quality and similarity. Another difference is that instead of generating candidate role sets and testing them against the original RBAC state to assess their similarity, we directly embed the original RBAC state as-is in the encoding together with penalties for diverging from it.

Another key difference from previous approaches is in when and how the RBAC reconfiguration is supposed to happen. Previous algorithms are presented as offline procedures; administrators are supposed to first let the complexity of the RBAC state increase, perhaps by creating several ad-hoc new roles to quickly fix exceptions. Then, from time to time, a procedure is applied to simplify everything. Large discontinuities are acceptable at these "reconfiguration points". Conversely, the approach presented here can be leveraged to continuously introduce small optimizing changes: Applied on-line, in an exception-driven way, it steers the trajectory of the RBAC state towards simplicity without any costly or stakeholder-adverse update.

### Plan synthesis

Many approaches exist to evaluate the differences among RBAC models. In (Fisler et al. 2005) a tool-supported approach is presented to analyse differences and match a source and a target RBAC state defined via XACML. This approach does not explicitly support migrations through the definition of a plan but it aims at identifying if the combination of policies causes violation of access control properties. In (Ni et al. 2009) an approach is presented to formalise RBAC policies which can then be compared according to policy similarity measures. This approach is meant to be used to discover which one, in a set of policies, is the most similar to the currently deployed one. Differences can be reported in a human-readable format and then used as the basis for the definition of migration strategies. In (Saenko and Kotenko 2016) a genetic algorithm is presented to determine the minimum variations to Role-to-User and Permission-to-Role assignment matrices to include variations to the Permission-to-User assignments.

None of the above approaches (Fisler et al. 2005; Ni et al. 2009; Saenko and Kotenko 2016) produces a maintenance plan as a result.

In (Hu et al. 2010) a role update, request-driven approach is presented to support administrators in specifying variations to permission assignments. The latter are formalised within a single instance and then solved via model-checking techniques. The results consist in a sequence of assign and revoke operations working on Permission-to-Role and Role-to-Permission assignments. In (Baumgrass and Strembeck 2013) a model comparison approach is presented to identify differences between two RBAC states. Based on these differences, a set of migration rules is derived to determine which relationships have to be added/removed/changed. The approaches in (Hu et al. 2010) and (Baumgrass and Strembeck 2013) determine the minimum set of operations necessary to reconfigure the input state into the target state. However, these approaches only reason in terms of assignment/de-assignment operations to $UA^0$ and $PA^0$, similarly to our baseline fix strategy *diff*. It results from our experiments that our approach is able to produce plans shorter than *diff*, hence shorter than anything produced by the above techniques. This is possible by virtue of our additional operators (such as: erase entire rows or columns; swap Permission-to-Role assignments, etc.) which are supported by current RBAC technologies and correctly captured by our PDDL formalisation.

## Conclusion and future work

We presented a novel RBAC maintenance approach: It produces an optimised plan of actions meant to reconfigure the RBAC state after certain exceptions/violations show up. At the same time, it aims at improving the quality of the RBAC state by simplifying it. Our method is based upon generating Max-SAT/PDDL instances and solving them via state-of-the-art solvers/planners.

We are motivated to pursue RBAC maintenance by actual issues at our company. Seldom there is the time and opportunity to perform large, impactful RBAC updates; conversely, errors show up on a daily basis, and if any push towards optimization can be exerted during maintenance, the RBAC state may actually start to converge to an (almost) optimal version of itself.

Directions for future work are as follows. Large datasets (e.g., *Firewall1*) still require too much time to solve; improvements to the encodings and/or the solving and planning stage are of the essence here. We submitted our datasets to the organizers of the Max-SAT competition to provide them with challenging real-world instances on which to compare solvers. We plan to extend the RBAC model to include hierarchies of roles which have a great potential on easing administration activities and to include further constraints among roles as supported by the RBAC standard.

## Endnotes

[1] Some Max-SAT solvers only accept positive integer values as weights; we can map real values onto integers that are equivalent to the effect of the optimization we perform as $weight_{int} = \lceil \delta * weight_{real} \rceil$ where $\delta$ is the minimum absolute difference between the sum of the weights of any two disjoint subsets of the real weights mentioned in the formula.

[2] According to the formula, we penalize the growth of roles linearly (according to $k^-$) compared to the role and permission assignments.

[3] Another possibility here would be to measure the absolute complexity via (22) as a percentage of the complexity of the initial state. We prefer measure (23) because it is independent of the unknown quality of the initial state (which we'll synthesize via an approximate miner) and stays in the range $[0, 1)$ with a clear meaning at the extremes.

[4] We choose Fast Miner over the alternatives–which we tried–because it quickly returns solutions of good quality to medium and large problems. In preliminary experiments done with other miners, we observed different initial absolute values but quite similar trends (as a function of beta and of the timeout) in all the experiments.

[5] We did not chose FM to avoid favoring it in the evaluation. Pair Count is a heuristic procedure similar to FM except that it prioritizes eligible roles considering how many pairs of users share each candidate role, and not a superset of it as in FM.

[6] It is worth noticing that while the *encoding* we propose offers these guarantees, we may lose them by solving the resulting problems with *incomplete* solvers.

**Acknowledgments**
The authors thank Dr. Chuan Luo for supplying us with an improved and configurable version of the solver *CCEHC*; Dr. Carlos Ansótegui for the solvers Dsat and WPM3; Dr. Shaowei Cai for *Dist*. We thank Dr. Jendrik Seipp for helping us in configuring *Fast-downward* planner. We thank the original creators of the role-mining datasets and Ian M. Molloy for providing us with the corresponding set of standardized *domino, university* and *firewall1* matrices. Several colleagues from the Bank of Italy offered help: A special thanks goes to Domenico Pansini (security administrator) for providing us with examples and insights on role maintenance in a real-world setting; Oliver Giudice, Giuseppe Galano, and Roberto Favaroni (ART team) offered a great deal of support in setting up and configuring the computational infrastructure we used for our experiments; Arturo Baldo was the first to suggest the opportunity to investigate formal approaches to role maintenance. Finally, the anonymous referees helped us to produce a better and more readable contribution.

**Funding**
Not applicable.

**Availability of data and materials**
All the results and datasets can be downloaded from the web site (Mori and Benedetti [2019]).

**Authors' contributions**
MM and MB both co-worked on the model definition, on the setting/carrying out of the experiments and on the manuscript writing. Both authors read and approved the final manuscript.

**Competing interests**
The authors declare that they have no competing interests.

## Publisher's Note

Received: 6 September 2018 Accepted: 5 May 2019
Published online: 01 July 2019

**References**

Baumgrass A, Strembeck M (2013) Bridging the gap between role mining and role engineering via migration guides. Inf Secur Tech Rep 17(4):148–72
Benedetti M, Mori M (2018) Parametric RBAC Maintenance via Max-SAT. In: Proceedings of the 23rd ACM on Symposium on Access Control Models and Technologies. ACM, Indianapolis, Indiana. pp 15–25
Colantonio A, Di Pietro R, Ocello A (2008) A cost-driven approach to role engineering. In: Proceedings of the 2008 ACM symposium on Applied computing. ACM, Fortaleza, Ceara. pp 2129–2136
Cook SA (1971) The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing. ACM, Shaker Heights, Ohio. pp 151–158
Ferraiolo DF, Sandhu R, Gavrila S, Kuhn DR, Chandramouli R (2001) Proposed nist standard for role-based access control. ACM Trans Inf Syst Secur (TISSEC) 4(3):224–274
Fisler K, Krishnamurthi S, Meyerovich LA, Tschantz MC (2005) Verification and change-impact analysis of access-control policies. In: Proceedings of the 27th International Conference on Software Engineering, ICSE '05. ACM, New York. pp 196–205
Ghallab M, Nau D, Traverso P (2004) Automated Planning: Theory and Practice. Elsevier
Guo Q, Vaidya J, Atluri V (2008) The role hierarchy mining problem: Discovery of optimal role hierarchies. In: Computer Security Applications Conference, 2008. ACSAC 2008. IEEE, Anaheim, California. pp 237–246
Hu J, Zhang Y, Li R, Lu Z (2010) Role updating for assignments. In: Proceedings of the 15th ACM symposium on Access control models and technologies. ACM, Pittsburgh, Pennsylvania. pp 89–98
Jafarian JH, Takabi H, Touati H, Hesamifard E, Shehab M (2015) Towards a general framework for optimal role mining: A constraint satisfaction approach. In: Proceedings of the 20th ACM Symposium on Access Control Models and Technologies. ACM, Vienna. pp 211–220
Johnson DS (1973) Approximation algorithms for combinatorial problems. In: Proceedings of the fifth annual ACM symposium on Theory of computing. ACM, Austin, Texas. pp 38–49
Kern A, Kuhlmann M, Schaad A, Moffett J (2002) Observations on the role life-cycle in the context of enterprise security management. In: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies, SACMAT '02. ACM, New York. pp 43–51
Luo C, Cai S, Su K, Huang W (2017) Ccehc: An efficient local search algorithm for weighted partial maximum satisfiability. Artif Intell 243:26–44
McDermott D, Ghallab M, Howe A, Knoblock C, Ram A, Veloso M, Weld D, Wilkins D (1998) PDDL - The Planning Domain Definition Language CVC TR-98-003/DCS TR-1165 (Oct.) Yale Center for Computational Vision and Control, Yale University
Mitra B, Sural S, Vaidya J, Atluri V (2016) A survey of role mining. ACM Comput Surv (CSUR) 48(4):50
Molloy I, Chen H, Li T, Wang Q, Li N, Bertino E, Calo S, Lobo J (2008) Mining roles with semantic meanings. In: Proceedings of the 13th ACM symposium on Access control models and technologies. ACM, Estes Park, CO. pp 21–30
Mori M, Benedetti M (2019) Web page with downloadable experiments and datasets. https://onlinerbacfixing.github.io/cybersecurity2019/. Accessed 1 Apr 2019
Ni Q, Lobo J, Calo S, Rohatgi P, Bertino E (2009) Automating role-based provisioning by learning from examples. In: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, SACMAT '09. ACM, New York. pp 75–84
Saenko I, Kotenko I (2016) Reconfiguration of rbac schemes by genetic algorithms
Seipp J, Röger G (2018) Fast Downward Stone Soup 2018 (planner abstract). In Ninth International Planning Competition (IPC 2018), Deterministic Part:72–74
Takabi H, Joshi JB (2010) Stateminer: An efficient similarity-based approach for optimal mining of role hierarchy. In: Proceedings of the 15th ACM symposium on Access control models and technologies. ACM, Pittsburgh, Pennsylvania. pp 55–64
Tseitin GS (1983) On the complexity of derivation in propositional calculus. In: Automation of reasoning. Springer. pp 466–483
Vaidya J, Atluri V, Guo Q, Adam N (2008) Migrating to optimal rbac with minimal perturbation. In: Proceedings of the 13th ACM symposium on Access control models and technologies. ACM, Estes Park, CO. pp 11–20
Wachs HL (2014) How to succeed with role management and avoid common pitfalls. Research Document G00262708. Gartner, Inc., Stamford, CT 06902-7700, U.S.
Wachs HL (2015) Take control of enterprise role management. Research Document G00262285. Gartner, Inc., Stamford, CT 06902-7700, U.S.
Xia H, Dawande M, Mookerjee V (2014) Role refinement in access control: Model and analysis. INFORMS J Comput 26(4):866–884
Zhang W, Chen Y, Gunter C, Liebovitz D, Malin B (2013) Evolving role definitions through permission invocation patterns. In: Proceedings of the 18th ACM symposium on Access control models and technologies. ACM, Amsterdam. pp 37–48