# ASASPXL: New Clother for Analysing ARBAC Policies

Anh Truong[1(✉)] and Silvio Ranise[2]

[1] Faculty of Computer Science and Engineering,
Ho Chi Minh City University of Technology, Ho Chi Minh City, Vietnam
anhtt@hcmut.edu.vn
[2] Security and Trust Unit, FBK-Irst, Trento, Italy
ranise@fbk.eu

**Abstract.** Access Control is becoming increasingly important for today's ubiquitous systems. In access control models, the administration of access control policies is an important task that raises a crucial analysis problem: if a set of administrators can give a user an unauthorized access permission. In this paper, we consider the analysis problem in the context of the Administrative Role-Based Access Control (ARBAC), one of the most widespread administrative models. We describe how we design heuristics to enable an analysis tool, called ASASPXL, to scale up to handle large and complex ARBAC policies. An extensive experimentation shows that the proposed heuristics play a key role in the success of the analysis tool over the state-of-the-art analysis tools.

**Keywords:** User-role reachability problem · Administration · Safety analysis · Access control · Model checking · Heuristics · Security

## 1 Introduction

Modern information systems contain sensitive information and resources that need to be protected against unauthorized users who want to steal it. The most important mechanism to prevent this is Access Control [7] which is thus becoming increasingly important for today's ubiquitous systems. In general, access control policies protect the resources of the systems by controlling who has permission to access what objects/resources.

Role-Based Access Control (RBAC) [14] is one of the most widely adopted access control models in the real world. In RBAC, access control policies specify which users can be assigned to roles which, in turn, are granted permissions to perform certain operations in the system. Usually, RBAC policies need to be evolved according to the rapidly changing environments and thus, it is demanded to have some mechanisms to control the modification of the policies. Administrative RBAC [6] is the corresponding widely used administrative model for RBAC policies. The main idea of ARBAC is to provide certain specific users, called administrators, some permissions to execute operations, called administrative


T.K. Dang et al. (Eds.): FDSE 2016, LNCS 10018, pp. 267–284, 2016.
DOI: 10.1007/978-3-319-48057-2_19

actions, to modify the RBAC policies. In fact, permissions to perform administrative actions must be restricted since administrators can only be partially trusted. For instances, some of them may collude to, inadvertently or maliciously, modify the policies (by sequences of administrative actions) so that untrusted users can get sensitive permissions. Thus, automated analysis techniques taking into consideration the effect of all possible sequences of administrative actions to identify the safety issues, i.e. administrative actions generating policies by which a user can acquire permissions that may compromise some security goals, are needed.

Several automated analysis techniques (see, e.g., [4,8,11,12,16,17]) have been developed for solving the user-role reachability problem, an instance of the safety issues, in the ARBAC model. Recently, a tool called ASASPXL [13] has been shown to perform better than the state-of-the-art tools on sets of benchmark problems in [10,16]. The main advantage of the analysis technique inside ASASPXL over the state-of-the-art techniques is that the tool can solve the user-role reachability problem with respect to a finite but unknown number of users in the policies manipulated by the administrative actions. However, ASASPXL does not scale to solve problems in some recently proposed benchmarks in [17]. This is because the so-called state explosion problem has not been handled carefully and thus, prevent ASASPXL to tackle such benchmarks.

In this paper, we study how to design heuristics to enable ASASPXL to analyze large and complex instances of user-role reachability problems. The main idea is to try to alleviate the state explosion problem, which is well-known problem in model checking techniques, in the analysis of ARBAC policies. We also perform an exhaustive experiment to conduct the effectiveness of proposed heuristics and compare ASASPXL's performance with the state-of-the-art analysis tools.

The paper is organized as follows. Section 2 introduces the RBAC, ARBAC models, and the related analysis problem. Section 3 briefly introduces the automated analysis tool ASASPXL and the model checking technique underlying it. The proposed heuristics to enable ASASPXL to scale to solve user-role reachability problem are described in Sect. 4. Section 5 summarizes our experiments and Sect. 6 concludes the paper.

## 2 RBAC, ARBAC, and the Reachability Problem

In *Role-Based Access Control (RBAC)*, access decisions are based on the roles that individual users have as part of an organization. The process of defining roles is based on a careful analysis of how an organization operates. Permissions are grouped by role name and correspond to various uses of a resource. A permission is restricted to individuals authorized to assume the associated role and represents a unit of control, subject to regulatory constraints within the RBAC model. For example, within a hospital, the role of doctor can include operations to perform diagnosis, prescribe medication, and order laboratory tests; the role of nurse can be limited to a strict subset of the permissions assigned to a doctor such as order laboratory tests.

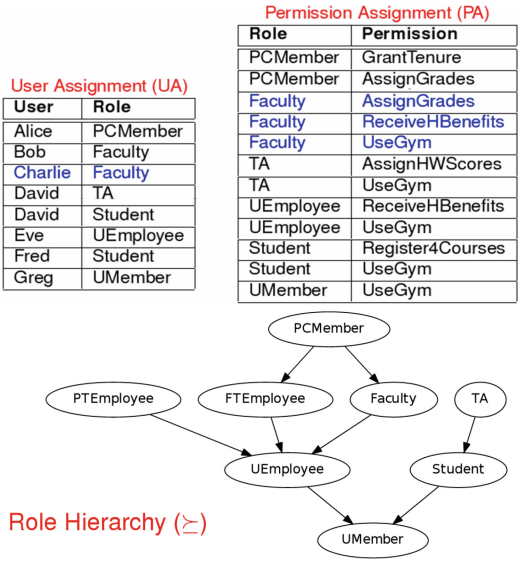


**Fig. 1.** User and Permission Assignments; and Role Hierarchies

We formalize a *RBAC policy* as a tuple $(U, R, P, UA, PA, \succeq)$ where $U$ is a set of users, $R$ a set of roles, and $P$ a set of permissions. A binary relation $UA \subseteq U \times R$ represents a user-role assignment and a binary relation $PA \subseteq R \times P$ represents a role-permission assignment. A user-role assignment specifies the roles to which the user has been assigned while a role-permission assignment specifies the permissions that have been granted to a role. A partial order $\succeq$ on $R$ is a role hierarchy of the policy, where $r_1 \succeq r_2$ means that $r_1$ is *more senior than* $r_2$ for $r_1, r_2 \in R$, i.e., every permission assigned to $r_2$ is also available to $r_1$.

A user $u$ is an *explicit member* of role $r$ when $(u, r) \in UA$ while the user $u$ is an *implicit member* of role $r$ if there exists $r' \in R$ such that $r' \succeq r$ and $(u, r') \in UA$. A user $u$ *has permission* $p$ if there exists a role $r \in R$ such that $(r, p) \in PA$ and $u$ is a (explicit or implicit) member of $r$.

*Example 1.* Consider an RBAC policy describing a department in a university as depicted in Fig. 1. The top-left table is the user-role assignment, the top-right is the role-permission assignment, and the bottom is an example of role hierarchies (The role at the tail of an arrow is more senior than the one at the head).

Let us consider the user *Charlie*: he is an *explicit* member of role *Faculty* because the tuple $(Charlie, Faculty)$ is in the user-role assignment $UA$. Additionally, role *Faculty* has been assigned to permissions *AssignGrades*, *ReceiveHBenefits*, and *UseGym*. Thus, *Charlie* can assign grades, receive benefits and use the gym through the role *Faculty*.

Let us consider the role hierarchy: role *Faculty* is more *senior* than role *UEmployee* (i.e., $Faculty \succeq UEmployee$). Therefore, *Charlie* is an *implicit* member of the role *UEmployee*, and thus he can also use all permissions assigned to the role *UEmployee*. □

### 2.1 Administrative RBAC (ARBAC)

Access control policies need to be maintained according to the evolving needs of the organization. For flexibility and scalability in large distributed systems, several administrators are usually required and there is a need not only to have a consistent policy but also to ensure that the policy is modified by administrators who are allowed to do so.

Several administrative frameworks have been proposed on top of the RBAC model to address these issues. One of the most popular administrative frameworks is Administrative RBAC (ARBAC) [6] that controls how RBAC policies may evolve through administrative actions that update the $UA$ and $PA$ relations (e.g., actions that update $UA$ include assigning or revoking user memberships into roles).

**Formalization.** Usually, administrators may only update the relation $UA$ while $PA$ and $\succeq$ are assumed constant. This is because a change in $PA$ and/or $\succeq$ implies a change in the organization (see [16] for more detail). From now on, we focus on situations where $U$ and $R$ are finite, $P$ plays no role, and $\succeq$ can be ignored[1] (and then, we only need to process the explicit members of a role when considering the role member relations). Thus, a RBAC policy is a tuple $(U, R, UA)$ or for short $UA$ if $U$ and $R$ are clear from the context.

Since administrators can be only partially trusted, administration privileges must be limited to selected parts of the RBAC policies, called *administrative domains*. An administrative domain is specified by a *pre-condition* defined as follows:

**Definition 1.** *A* pre-condition $C$ *is a finite set of expressions of the forms* $r$ *or* $\overline{r}$ *where* $r \in R$.

A user $u \in U$ *satisfies* a pre-condition $C$ if, for each $\ell \in C$, $u$ is a member of $r$ when $\ell$ is $r$ or $u$ is not a member of $r$ when $\ell$ is $\overline{r}$ for $r \in R$. We also say that $r$ is a positive role and $\overline{r}$ is a negative role in $C$.

Permission to assign users to roles is specified by a ternary relation *can_assign* containing tuples of the form $(C_a, C, r)$ where $C_a$ and $C$ are pre-conditions, and $r$ a role. Permission to revoke users from roles is specified by a binary relation *can_revoke* containing tuples of the form $(C_a, r)$ where $C_a$ is a pre-condition and $r$ a role. In both cases, we say that $C_a$ is the *administrative pre-condition*, $C$ is a *(simple) pre-condition*, $r$ is the *target role*, and a user $u_a$ satisfying $C_a$ is the *administrator*. The relation *can_revoke* is only binary because simple pre-conditions are useless when revoking roles (see, e.g., [16]). When there exist users satisfying the administrative and the simple (if the case) pre-conditions of an administrative action, the action is *enabled*.

The semantics of the administrative actions in the ARBAC policy $\psi := (\mathit{can\_assign}, \mathit{can\_revoke})$ is given by the binary relation $\rightarrow_\psi$ defined as follows:

[1] We can transform a policy with role hierarchies to a policy without them by pre-processing away the role hierarchies as shown in [15].

**Definition 2.** *$UA \rightarrow_\psi UA'$ iff there exist users $u_a$ and $u$ in $U$ such that either:*

- *there exists $(C_a, C, r) \in can\_assign$, $u_a$ satisfies $C_a$, $u$ satisfies $C$ (i.e. $(C_a, C, r)$ is enabled), and $UA' = UA \cup \{(u, r)\}$ or*
- *there exists $(C_a, r) \in can\_revoke$, $u_a$ satisfies $C_a$ (i.e. $(C_a, r)$ is enabled), and $UA' = UA \backslash \{(u, r)\}$.*

A *run* of the administrative actions in $\psi := (can\_assign, can\_revoke)$ is a possibly infinite sequence $UA_0, UA_1, ..., UA_n, ...$ such that $UA_i \rightarrow_\psi UA_{i+1}$ for $i \geq 0$.

*Example 2.* Consider the RBAC policy with the $UA$ relation depicted in Fig. 1 and an administrative action $(\{PCMember\}, \{Student, \overline{TA}\}, PTEmpl) \in can\_assign$, i.e., the administrative pre-condition is $C_a = \{PCMember\}$, the simple pre-condition is $C = \{Student, \overline{TA}\}$, and the target role is $PTEmpl$.

User *Alice* satisfies the pre-condition $C_a$ because $(Alice, PCMember) \in UA$. User *Fred* satisfies the pre-condition $C$ because he is a *Student* but not a *TA* (e.g., $(Fred, Student) \in UA$ and $(Fred, TA) \notin UA$). As a sequence, the administrative action is enabled.

We can update the current $UA$ to $UA' = UA \cup \{(Fred, PTEmpl)\}$ by executing the following instance of the administrative action specified above: administrator *Alice* (who has role *PCMember*) assigns role *PTEmpl* to user *Fred*.

Notice that *Alice* cannot assign role *PTEmpl* to *David* because he is not only a *Student* but also a *TA* (i.e., *David* does not satisfy the pre-condition $C$). □

### 2.2 The User-Role Reachability Problem

Normally, policy designers and administrators want to foresee if the interactions among administrative actions, as seen in the Example 2, can lead the system to conflict states violating the security requirements of the organization (e.g., the security requirements forbid a user to be assigned to some sensitive roles). Thus, they need to analyze access control policies in order to discover such violation. This problem is called as the user-role reachability problem and is defined as follows.

**Definition 3.** *A pair $(u_g, R_g)$ is called a* (RBAC) goal *for $u_g \in U$ and $R_g$ a finite set of roles. The cardinality $|R_g|$ of $R_g$ is the* size *of the goal.*

**Definition 4.** *Given an initial RBAC policy $UA_0$, a goal $(u_g, R_g)$, and administrative actions $\psi = (can\_assign, can\_revoke)$; (an instance of) the* **user-role reachability problem***, identified by the tuple $\langle UA, \psi, (u_g, R_g)\rangle$, consists of checking if there exists a finite sequence $UA_0, UA_1, ..., UA_n$ (for $n \geq 0$) where (i) $UA_i \rightarrow_\psi UA_{i+1}$ for each $i = 0, ..., n-1$ and (ii) $u_g$ is a member of each role of $R_g$ in $UA_n$.*

In real scenario, subtle interactions between administrative actions in real policies may arise that are difficult to be foreseen by policy designers and administrators. Thus, automated analysis techniques are thus of paramount importance to analyze such policies and answer the user-role reachability problem.

The analysis techniques we will present in the following will be able to establish this automatically for the problem in ARBAC.

## 3 Model Checking Modulo Theories and the Reachability Problem

**Model Checking Modulo Theories (MCMT).** MCMT [9] is a framework to solve reachability problems for infinite state systems that can be represented by transition systems whose set of states and transitions are encoded as constraints in first-order logic. Several systems have been abstracted using such symbolic transition system such as parametrised protocols, sequential programs manipulating arrays, timed system, etc. (see again [9] for an overview).

MCMT framework uses a backward reachability procedure that repeatedly computes the so-called pre-images of the set of *goal* states, that is usually obtained by complementing a certain safety property that the system should satisfy. Then, the set of backward reachable states of the system is obtained by taking the union of the pre-images. At each iteration of the procedure, the procedure checks whether the intersection between the set of backward reachable states and the initial set of states is non-empty (i.e., *safety* test) or not (i.e., the *unsafety* of the system: there exists a (finite) sequence of transitions that leads the system from an initial state to one satisfying the goal). Otherwise, when the intersection is empty, the procedure checks if the set of backward reachable states is contained in the set computed at the previous iteration (*fix-point* test) and, if yes, the *safety* of the system (i.e. no (finite) sequence of transitions leads the system from an initial state to one satisfying the goal) is returned. Since sets of states and transitions are represented by first-order constraints, the computation of pre-images reduces to simple symbolic manipulations and testing safety and fix-point to solving a particular class of constraint satisfiability problems, called Satisfiability Modulo Theories (SMT) problems, for which scalable and efficient SMT solvers are currently available (e.g., Z3 [2]).

**ASASPXL.** In [3,5], it is studied how the MCMT approach can be used to solve (variants of) the user-role reachability problem. On the theoretical side, it is shown that the backward reachability procedure described above decides (variants of) the user-role reachability problem. On the practical side, extensive experiments have shown that an automated tool, called ASASP [4] implementing (a refinement of) the backward reachability procedure, has a good *trade-off* between *scalability* and *expressiveness*. Immediately after ASASP, a set of much larger instances of the user-role reachability problem has been considered in [10]. Unfortunately, ASASP does not scale to solve the set of problem. This is in line with the following observation of [10]: "model checking does not scale adequately for verifying policies of very large sizes." Then, in [13], a new tool based on the MCMT approach, called ASASPXL, has been proposed to efficiently solve much larger instances of the user-role reachability problem. The new analysis tool ASASPXL is build on top of MCMT, the first implementation of the MCMT approach. The choice of building a new analysis tool instead of modifying ASASP

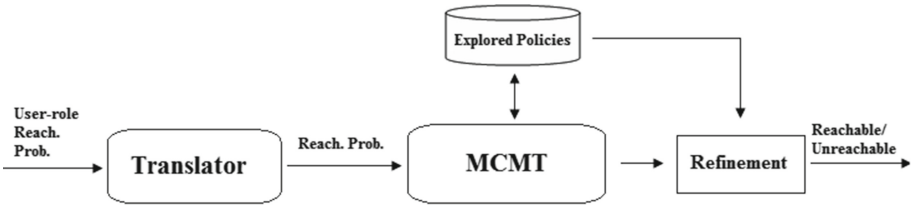


**Fig. 2.** ASASPXL architecture

gives some advantage. First, we only need to write a translator from instances of the user-role reachability problem to reachability problems in MCMT input language, a routine programming task. Second, MCMT has been developed and extensively used for the past years. It is thus more robust and offers a high degree of confidence. Third, we can re-use some features of a better engineered incarnation of the MCMT approach that can be exploited to significantly improve performances, as shown in [13].

The structure of ASASPXL is depicted in Fig. 2. It takes as input an instance of the user-role reachability problem and returns `reachable`, when there exists a finite sequence of administrative operations that leads from the initial RBAC policy to one satisfying the goal, and `unreachable` otherwise. To give such results, ASASPXL firstly translates the user-role reachability problem to the reachability problem in MCMT input language (module **Translator**). Then, it calls the model checker MCMT to verify the reachability of the problem. Finally, according to the answer returned by the model checker (in the data storage **Explored Policies**), ASASPXL refines it and returns `reachable` or `unreachable` as its output (module **Refinement**).

To keep technicalities to a minimum, we illustrate the translation on an instance of the user-role reachability problem as follows.

*Example 3.* Let $U = \{u_1, u_2, u_3, u_4, u_5\}$, $R = \{r_1, ..., r_8\}$, initially $UA := \{(u_1, r_1), (u_2, r_2), (u_5, r_5)\}$, and

$$(\{r_1\}, \{r_2\}, r_3) \in can\_assign \quad (1)$$
$$(\{r_3\}, \{r_4, \overline{r_5}\}, r_6) \in can\_assign \quad (2)$$
$$(\{r_4\}, \{r_5\}, r_7) \in can\_assign \quad (3)$$
$$(\{r_2\}, \{r_7\}, r_8) \in can\_assign \quad (4)$$
$$(\{r_2\}, r_3) \in can\_revoke \quad (5)$$
$$(\{r_5\}, r_4) \in can\_revoke \quad (6)$$

The goal of the problem is $(u_5, \{r_8\})$.

To formalize this problem instance in MCMT, ASASPXL firstly generates an unary relation $u_r$ per role $r \in R$. The initial relation $UA$ can thus be expressed as

$$\forall x. \left[ \begin{array}{l} (u_{r_1}(x) \leftrightarrow x = u_1) \wedge (u_{r_2}(x) \leftrightarrow x = u_2) \wedge (u_{r_5}(x) \leftrightarrow x = u_5) \wedge \neg u_{r_2}(x) \wedge \neg u_{r_3}(x) \wedge \\ \neg u_{r_4}(x) \wedge \neg u_{r_6}(x) \wedge \neg u_{r_7}(x) \wedge \neg u_{r_8}(x) \end{array} \right].$$

A tuple, for instance, $(\{r_3\}, \{r_4, \overline{r_5}\}, r_6)$ in *can_assign* is formalized as

$$\exists x \exists y. \left[ u_{r_3}(x) \wedge u_{r_4}(y) \wedge \neg u_{r_5}(y) \wedge \forall \lambda. (u'_{r_6}(\lambda) \leftrightarrow (\lambda = y \vee u_{r_6}(\lambda))) \right]$$

and a tuple, for example, $(\{r_2\}, r_3)$ in *can_revoke* can be expressed as

$$\exists x \exists y. \left[ u_{r_2}(x) \wedge u_{r_3}(y) \wedge \forall \lambda. (u'_{r_3}(\lambda) \leftrightarrow (\lambda \neq y \wedge u_{r_3}(\lambda))) \right]$$

where $u_r$ and $u'_r$ indicate the value of $U_r$ immediately before and after, respectively, the execution of the administrative action (we also have omitted—for the sake of compactness—identical updates, i.e. a conjunct $\forall \lambda. (u'_r(\lambda) \leftrightarrow u_r(\lambda))$ for each role $r$ distinct from the target role in the tuple of *can_assign* or *can_revoke*). The other administrative actions are translated in a similar way.

The goal $(u_5, \{r_8\})$ can be represented as:

$$\exists x. u_{r_8}(x) \wedge x = u_5$$

The pre-image of the goal, that is computed by the model checker MCMT, with respect to $(\{r_2\}, \{r_7\}, r_8)$ is the set of states from which it is possible to reach the goal by using the administrative action $(\{r_2\}, \{r_7\}, r_8)$. This is formalized as the formula (see [5] for details)

$$\exists x \exists y. ((u_{r_7}(y) \wedge y = u_5) \wedge u_{r_2}(x)),$$

On this problem, MCMT returns `unreachable` (i.e., there does not exist a finite sequence of administrative operations that lead from the initial policy $UA$ to one satisfying the goal). □

Recently, a tool named VAC has been proposed in [8] for solving the user-role reachability problem of ARBAC policies. In [8], it is shown that VAC outperforms RBAC-PAT [16], MOHAWK [10], and ASASPXL on the problems in [16] and on a new set of complex instances of the user-role reachability problem. It was natural to run ASASPXL on these new benchmark problems: rather disappointingly, it could tackle such problem instances, however, its performance cannot be comparable with the new tool VAC (e.g., ASASPXL returns time-out in some problems). The reason of the bad scalability of ASASPXL is that ASASPXL does not work well on the user-role reachability problems with some specific features such as the problem containing some sub-problems having same structure of administrative actions; and the problems in which no state can be reached from the initial state. These and other problems have lead us to design new heuristics to make ASASPXL more scalable, as we will see in the next sections.

## 4 ASASPXL with new Heuristics

To enable ASASPXL to scale up to analyze the complex instances of the user-role reachability problem as shown in the previous section, our main idea is to design heuristics that help to alleviate the so-called state explosion problem, one of the commonly known problems in model checking techniques that must be addressed to solve most real-world problems. One of the main source of complexity is the large number of administrative actions; thus, for scalability, the original set of actions must be refined by using heuristics that tries to eliminate administrative actions that do not contribute to the analysis of RBAC policy. This and other techniques to control the state explosion problem will be detailed in the following (sub-)sections. Before going to the details of heuristics, we emphasize that all heuristics in the following will be implemented in a module named **Heuristics** and will be put before module **Translator** in the architecture of ASASPXL in Fig. 2. The ASASPXL's input, a user-role reachability problem, will be processed by module **Heuristics** before being forwarded to module **Translator** and then to module MCMT as described in Sect. 3.

### 4.1 Backward Useful Actions

The main idea to alleviate the state explosion problem is to eliminate as much as possible administrative actions that is useless to the analysis of ARBAC policy. This is done by extracting increasingly larger sub-sets of the tuples in the original set of administrative actions $\psi$ so as to generate a sequence of increasingly more precise approximations of the original instance of the user-role reachability problem. The heuristics to do this is based on the following notion of an administrative action being useful.

**Definition 5.** Let $\psi$ be the set of administrative actions and $R_g$ a set of roles in an ARBAC policy:

- A tuple in $\psi$ is 0-*useful* iff its target role is in $R_g$.
- A tuple in $\psi$ is $k$-*useful* (for $k > 0$) iff it is $(k-1)$-useful or its target role occurs (possibly negated) in **either** the simple pre-condition **or** the administrative pre-condition of a $(k-1)$-useful transition.

A tuple $t$ in $\psi$ is *useful* iff there exists $k \geq 0$ such that $t$ is $k$-useful.

Let $\psi^{\leq k} = (can\_assign^{\leq k}, can\_revoke^{\leq k})$ denote the set of all $k$-useful tuples in $\psi = (can\_assign, can\_revoke)$. It is easy to see that $can\_assign^{\leq k} \subseteq can\_assign^{\leq k+1}$ and $can\_revoke^{\leq k} \subseteq can\_revoke^{\leq k+1}$ (abbreviated by $\psi^{\leq k} \subseteq \psi^{\leq k+1}$) for $k \geq 0$. Since the sets $can\_assign$ and $can\_revoke$ in $\psi$ are bounded, there must exist a value $\tilde{k} \geq 0$ such that $\psi^{\leq \tilde{k}} = \psi^{\leq \tilde{k}+1}$ (that abbreviates $\psi^{\leq \tilde{k}} \subseteq \psi^{\leq \tilde{k}+1}$ and $\psi^{\leq \tilde{k}+1} \subseteq \psi^{\leq \tilde{k}}$) or, equivalently, $\psi^{\leq \tilde{k}}$ is the (least) fix-point, also denoted with $lfp(\psi)$, of useful tuples in $\psi$. Indeed, a tuple in $\psi$ is useful iff it is in $lfp(\psi)$.

*Example 4.* Let $\psi$ be the administrative actions in Example 3 and $R_g := \{r_8\}$. The sets of $k$-useful tuples for $k \geq 0$ are the following:

$$\begin{aligned}
\psi^{\leq 0} &:= (\{(\{r_2\}, \{r_7\}, r_8)\}, \emptyset) \\
\psi^{\leq 1} &:= \psi^{\leq 0} \cup (\{(\{r_4\}, \{r_5\}, r_7)\}, \emptyset) \\
\psi^{\leq 2} &:= \psi^{\leq 1} \cup (\emptyset, \{(\{r_5\}, r_4)\}) \\
\psi^{\leq k} &:= \psi^{\leq 2} \text{ for } k > 2
\end{aligned}$$

Now, we run the user-role reachability problem: $\langle UA, \psi^{\leq 2}, (u_5, \{r_8\})\rangle$. ASASPXL returns `unreachable` on this problem instance. We obtain the same result if we run the tool on the translation of the following problem instance: $\langle UA, \psi, (u_5, \{r_8\})\rangle$. This leads to the following proposition. □

**Proposition 1.** A goal $(u_g, R_g)$ is unreachable from an initial user-role assignment relation $UA$ by using the administrative operations in $\psi$ iff $(u_g, R_g)$ is unreachable from $UA$ by using the administrative operations in $lfp(\psi)$.

The proof of this fact can be obtained by slightly adapting the proof for the proposition in [13] and is thus omitted here.

### 4.2 Forward Useful Actions

In Sect. 4.1, we have introduced a heuristics identifying the set of useful actions (that is a subset of the original set of administrative actions) that is enough for solving the user-role reachability. We start using the roles in the goal to identify 0-useful actions and then using roles in the pre-conditions of $k$-useful actions to decide $(k+1)$-useful actions. Dually, we can start from the roles in the initial states and *forwardly* compute the set of useful actions. This is captured by the notion of *forward* useful action as follows:

**Definition 6.** Let $\psi$ be the set of administrative actions, $R$ be the set of roles, and $R_i := \{r | (u, r) \in UA_0\} \cup \{\overline{r} | r \in R\}$ be a set of roles occurring in the initial policy $UA_0$. A tuple $\tau \in \psi$ :

- is forward 0-*useful* iff its pre-condition is a subset of $R_i$
- is forward $k$-*useful* (for $k > 0$) iff it is:
  - $(k-1)$-useful or,
  - its pre-condition is a subset of $R_i = R_i \cup \{r|$ r is the target role of a $(k-1)$-useful action$\}$

$\tau$ is *forward useful* iff there exists $k \geq 0$ such that $\tau$ is forward $k$-useful.

Let $\psi_F^{\leq k} = (can\_assign^{\leq k}, can\_revoke^{\leq k})$ denote the set of forward $k$-useful actions in $\psi = (can\_assign, can\_revoke)$, it is easy to see that $\psi_F^{\leq k} \subseteq \psi_F^{\leq k+1}$ for $k \geq 0$ and there exists a value $\tilde{k} \geq 0$ such that $\psi_F^{\leq \tilde{k}} = \psi_F^{\leq \tilde{k}+1}$ (i.e., $lfp_F(\psi) = \psi_F^{\leq \tilde{k}}$). Similar to the heuristic for backward useful actions above, we conclude the following proposition.

**Proposition 2.** A goal $(u_g, R_g)$ is unreachable from an initial user-role assignment relation $UA$ by using the administrative operations in $\psi$ iff $(u_g, R_g)$ is unreachable from $UA$ by using the administrative operations in $lfp_F(\psi)$.

*Example 5.* Let consider again Example 3. The set $R_i$ of roles in $UA_0$ is $\{r_1, r_2, r_5, \overline{r_1}, \overline{r_2}, ..., \overline{r_7}, \overline{r_8}\}$.

The sets of forward $k$-useful tuples for $k \geq 0$ are the following:

$$\psi_F^{\leq 0} := \{(\{r_1\}, \{r_2\}, r_3), (\{r_2\}, r_3), (\{r_5\}, r_4),$$
$$\psi_F^{\leq k} := \psi_F^{\leq 0} \text{ for } k > 0,$$

ASASPXL returns `unreachable` on the user-role reachability problem $\langle UA, \psi_F^{\leq 0}, (u_1, \{r_8\})\rangle$ that confirms the results in Examples 3 and 4. □

**The Combination of Backward and Forward Useful Actions.** The module **Heuristics** in Sect. 4 works as follows to take into consideration the forward and backward useful actions. First, the module computes $\psi^k$ and $\psi_F^k$ that are the set of backward $k$-useful and forward $k$-useful actions, respectively. Then, the module will compute the intersection $\psi_U$ of the sets $\psi^k$ and $\psi_F^k$ that is expected to be much smaller than $\psi^k$, $\psi_F^k$, and the original set $\psi$. Finally, the set of useful actions $\psi_U$ is used to replace the original set $\psi$ in solving the user-role reachability problem. The correctness and completeness of taking into consideration the intersection instead of the set of forward or backward useful actions is guaranteed by Proposition 3 that is simply a corollary of Propositions 1 and 2.

**Proposition 3.** A goal $(u_g, R_g)$ is unreachable from an initial user-role assignment relation $UA$ by using the administrative actions in $\psi$ iff $(u_g, R_g)$ is unreachable from $UA$ by using the administrative operations in $lfp(\psi) \cap lfp_F(\psi)$.

### 4.3 Ordering Administrative Actions

We recall that the module MCMT implements the backward reachability procedure that computes the sets of backward reachable states from the goal. Basically, at each iteration, the procedure takes the first administrative action in the set $\psi$, computes its backward reachable states (pre-image) and then checks the intersection between the initial state and the backward states (by using an SMT solver to check the satisfiability). If the intersection is not empty (i.e., the goal is reachable from the initial state), the procedure returns `reachable` and stops. Otherwise, it selects the second action and repeats the process until all actions have been considered. This idea gives two advantages: first, the procedure can stop as soon as possible when it decides that the goal is reachable by checking an action and thus, not necessary to check the remaining actions; second, the fix-point formula can be divided into a set of smaller formulae, namely *local fix-points*, that is easier to be checked by SMT solvers. The original fix-point is reached when all the local fix-points are reached.

Clearly, the selection of the next action for computing the pre-images should be handled carefully since this will cause some redundant in the analysis that may negatively affect the performances of the procedure. In fact, if the goal is reachable and the administrative action, let us say $\tau$, that helps the procedure in deciding the reachability of the goal is at the end of the action list, the current version of the backward reachability procedure must computes the pre-images for all actions before $\tau$ that are actually redundant computations. It is thus desirable to design a heuristics to select the next action to maximize the possibility of picking up an action that is important to show the reachability of the goal.

Our heuristics is based on the idea of how "close" between the set of states produced by computing the pre-image with respect to a given action and the set of initial states. This is because for each iteration, the procedure checks if the intersection between the pre-image generated by the given action and the set of initial states is empty, and then uses this check to decide the reachability of the goal. To illustrate how an action is "closer" than another, let us consider the following example:

*Example 6.* Let $U = \{u_1, u_2\}$, $R = \{r_a, r_1, ..., r_7\}$ initially $UA := \{(u_1, r_a), (u_1, r_1), (u_1, r_2), (u_1, r_5)\}$, and the set $\psi$ contains:

$$(\{r_a\}, \{r_1, r_2, \overline{r_4}\}, r_7) \in \mathit{can\_assign} \tag{7}$$

$$(\{r_a\}, \{r_1, r_3\}, r_7) \in \mathit{can\_assign} \tag{8}$$

The pre-images of the two actions (7) and (8) (computed by the backward reachability procedure) are represented by formulae $\exists x, y.(r_a(x) \wedge r_1(y) \wedge r_2(y) \wedge \neg r_4(y))$ and $\exists x, y.(r_a(x) \wedge r_1(y) \wedge r_3(y))$, respectively. It is easy to see that the set of reachable states of action (7) is contained in the initial state $UA$ (i.e., their intersection is not empty). We also notice how all the roles in the precondition of action (7) appear in $UA$ while role $r_3$ in the precondition of action (8) does not. In this case, we say that action (7) is closer (to the initial state) than action (8). Then, action (7) should be selected before action (8) in the backward reachability procedure. □

We define the function *Diff* calculating how "close" two sets of roles are as follows:

**Definition 7.** Let $C_1$ and $C_2$ be pre-conditions, the difference between $C_1$ and $C_2$ is:

$$\mathit{Diff}(C_1, C_2) := (C_1^+ \backslash C_2^+) \cup (C_1^- \backslash C_2^-)$$

where $C_1^+$ and $C_2^+$ are sets of positive roles in $C_1$ and $C_2$, respectively; $C_1^-$ and $C_2^-$ are sets of negative roles in $C_1$ and $C_2$, respectively.

We illustrate how the function *Diff* is used in the heuristic by the following example:

*Example 7.* Let us consider again Example 6. First, the heuristic will calculate $R_i = \{r_a, r_1, r_2, r_5, \overline{r_a}, \overline{r_1}, ..., \overline{r_7}\}$ that represents all roles occurring in the initial $UA$ as defined in Definition 6.

Let consider action (7) with its precondition $C_1 = \{r_a, r_1, r_2, \overline{r_4}\}$, the heuristic then computes $\mathit{Diff}(C_1, R_i) = \emptyset$. Similarly, the precondition of action (8) is $C_2 = \{r_a, r_1, r_3\}$ and $\mathit{Diff}(C_2, R_i) = \{r_3\}$.

Since $|\mathit{Diff}(C_2, R_i)| > |\mathit{Diff}(C_1, R_i)|$, we say that action (7) is closer (to the initial state) than action (8). In other words, the precondition $C_1$ can be easily satisfied by the initial $UA$ while $C_2$ requires more tuples, for instance $(u_1, r_3) \in UA$, to be satisfied. Thus, the heuristic will select the actions (7) to compute its pre-image before (8) □

We add this heuristic to the tool ASASPXL by adding a sub-module, namely **Ordering the Actions**, to module **Heuristics** mentioned above. After computing the set of useful actions as in Sects. 4.1 and 4.2, **Heuristics** will invoke the sub-module **Ordering the Actions** with the set $\psi_U$ of useful actions as the parameter. The sub-module then orders the administrative actions in $\psi_U$ and returns the ordered set as workflow below:

1. Let $\psi_U$ be the set of actions and $R_i$ containing all roles occurring in the initial state $UA_0$.
2. For each $\tau = (C_a, C, r) \in \psi_U$:
   (a) If $C_a = \emptyset$ and $C = \emptyset$:
      i. set $\tau$ be the first order in $\psi_U$ (for several actions with $C_a = C = \emptyset$, we do not care the order between them)
   (b) Else:
      i. Calculate $\mathit{Diff}_\tau := \mathit{Diff}(C_a \cup C, R_i)$ for $\tau$
3. Order the actions in $\psi_U$ by their $|\mathit{Diff}_\tau|$ (from lower value to higher one)
   (a) If $|\mathit{Diff}_{\tau_1}| = |\mathit{Diff}_{\tau 2}|$ where $\tau_1 = (C_{a1}, C_1, r_1)$ and $\tau_2 = (C_{a2}, C_2, r_2)$:
      i. $\tau_1$ has higher order if $|C_{a1} \cup C_1| < |C_{a2} \cup C_2|$ and vice versa

Initially, the procedure computes the set $R_i$ containing all roles in the initial $UA_0$. Then, it calculates the set $\mathit{Diff}$ for each administrative action in $\psi_U$ (Step 2). Administrative actions of the form $(\emptyset, \emptyset, r)$ are set highest order in Step 2(a) since its pre-conditions are alway satisfied. For several actions of the form $(\emptyset, \emptyset, r)$, we do not care about the order between them. The procedure then classifies the actions in $\psi_U$ based on their $\mathit{Diff}$ (Step 3). Notice how the procedure prioritizes the action containing smaller set of pre-conditions (Step 3(a)) for the actions having the same $|\mathit{Diff}|$. This is because the formula representing the set of backward reachable states generated by the action (see, e.g., Example 6) may be smaller (i.e., containing less literals) than the others and thus easier for the SMT solver to check the satisfiability.

## 5 Experiments

We have implemented ASASPXL and heuristics in Python and used the MCMT model checker [1] for computing the pre-images. We have also conducted an

experimental evaluation to show the scalibility of ASASPXL and compare it with state-of-the-art analysis tools such as MOHAWK [10], VAC [8], and PMS [17] on two benchmark sets from [10] and [8]. Note that PMS contains 2 versions, namely *Prl* and *Fwd* that implement the analysis with/without applying their parallel algorithm [17].

**Remark.** Sometimes, to simplify the analysis of ARBAC policies, *separate administration assumption* (for short, SA) has been applied (see, e.g. [16]) which amounts to requiring that administrative roles (i.e., roles occurring in the administrative precondition $C_a$) and regular roles (i.e., roles occurring in the simple precondition $C$) are disjoint. This permits to consider just one user, omit administrative users and roles so that the tuples in *can_assign* are pairs composed of a simple precondition and a target role (i.e., $(C, r)$) and the pairs in *can_revoke* reduce to target roles only (i.e., $(r)$). In the state-of-the-art analysis tools mentioned above, MOHAWK requires this assumption while the other two and ASASPXL do not need it. The benchmarks are thus classified as either SA benchmarks (that require SA assumption) or non-SA benchmarks (that do not the the assumption) as in the following.

**Description of Benchmarks.** The first benchmark set is a SA benchmark taken from [10]. It contains three synthetic test suites: **Test suite 1** contains policies in which roles occur only positively in the (simple) pre-conditions of *can_assign* rules and the set of *can_revoke* rules is non-empty. **Test suite 2** contains policies in which roles occur both positively and negatively in *can_assign* rules and the set of *can_revoke* rules is empty. **Test suite 3** contains policies in which roles occur both positively and negatively in *can_assign* rules and the set of *can_revoke* rules is non-empty. The second benchmark set is a non-SA benchmark from [17]. It contains 10 instances of the user-role reachability problem inspired by a university.

**Evaluation.** We perform all the experiments on an Intel Core I5 (2.6 GHz) CPU with 4 GB Ram running Ubuntu 11.10

Table 1 reports the results of running ASASPXL, PMS, VAC and MOHAWK on the first benchmark set. Notice that all problems in this benchmark are unsafe (i.e., analysis tools returns "reachable"). Column 1 shows the name of the test suite, column 2 contains the number of roles and administrative operations in the policy. Columns 3, 4, 6 and 7, and 8 show the average times (in seconds) taken by MOHAWK, VAC, PMS (with two versions), and ASASPXL, respectively, to solve the instances of the user-role reachability problem associated to an ARBAC policy. For MOHAWK and VAC, the average time also include the time spent in the slicing phase (a technique for eliminating irrelevant users, roles, and administrative operations that are non relevant to solve a certain instance of the user-role reachability problem, see [8,10] for more details) and the verification phase. Column 6 and 10 represent the number of actions remaining after the slicing phase of VAC and the useful actions obtained by ASASPXL, respectively.

Experiments for the benchmark that does not adopt the separate administration assumption are reported in Tables 2; their columns have the same

**Table 1.** Experimental results on the "complex" benchmarks in [10]

| (Separate administration assumption) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Test suite | # Roles ◇ | MOHAWK | VAC | | PMS | | ASASPXL | |
| | | | | | *Fwd* | *Prll* | | |
| | #Rules | Time | Time | # Rules | Time | Time | Time | # Rules |
| Test suite 1 | 3 ◇ 15 | 0.45 | 0.29 | 1 | 0.38 | 0.45 | **0.09** | 1 |
| | 5 ◇ 25 | 0.53 | 0.35 | 1 | 0.38 | 0.47 | **0.11** | 1 |
| | 20 ◇ 100 | 0.64 | 0.35 | 1 | 0.35 | 0.39 | **0.12** | 1 |
| | 40 ◇ 200 | 0.97 | 0.69 | 1 | 0.49 | 0.57 | **0.31** | 2 |
| | 200 ◇ 1000 | 2.69 | 0.95 | 1 | 0.47 | 0.55 | **0.38** | 1 |
| | 500 ◇ 2500 | 4.88 | 1.59 | 1 | 0.97 | 1.16 | **0.70** | 1 |
| | 4000 ◇ 20000 | 16.99 | 1.88 | 1 | 33.55 | 22.39 | **1.27** | 2 |
| | 20000 ◇ 80000 | 51.57 | 2.72 | 1 | *TO* | *TO* | **1.27** | 2 |
| | 30000 ◇ 120000 | 65.51 | 4.12 | 1 | *TO* | *TO* | **1.69** | 2 |
| | 40000 ◇ 200000 | 131.17 | 9.94 | 1 | *TO* | *TO* | **2.29** | 2 |
| Test suite 2 | 3 ◇ 15 | 0.45 | 0.25 | 1 | 0.36 | 0.37 | **0.15** | 1 |
| | 5 ◇ 25 | 0.55 | 0.39 | 1 | 0.35 | 0.38 | **0.28** | 1 |
| | 20 ◇ 100 | 0.59 | 0.24 | 1 | 0.32 | 0.49 | **0.16** | 1 |
| | 40 ◇ 200 | 1.21 | 0.56 | 1 | 0.54 | 0.59 | **0.15** | 1 |
| | 200 ◇ 1000 | 2.55 | 0.83 | 1 | 0.59 | 0.63 | **0.14** | 1 |
| | 500 ◇ 2500 | 6.12 | 1.52 | 1 | 1.54 | 0.83 | **0.47** | 2 |
| | 4000 ◇ 20000 | 15.51 | 1.63 | 1 | 29.17 | 21.39 | **1.18** | 2 |
| | 20000 ◇ 80000 | 26.12 | 5.25 | 1 | *TO* | *TO* | **1.22** | 2 |
| | 30000 ◇ 120000 | 98.95 | 6.73 | 1 | *TO* | *TO* | **1.28** | 2 |
| | 40000 ◇ 200000 | 146.84 | 11.89 | 1 | *TO* | *TO* | **1.43** | 2 |
| Test suite 3 | 3 ◇ 15 | 0.51 | 0.15 | 1 | 0.37 | 0.35 | **0.08** | 1 |
| | 5 ◇ 25 | 0.45 | 0.19 | 1 | 0.55 | 0.49 | **0.09** | 1 |
| | 20 ◇ 100 | 0.87 | 0.31 | 1 | 0.42 | 0.62 | **0.16** | 1 |
| | 40 ◇ 200 | 0.99 | 0.67 | 1 | 0.46 | 0.57 | **0.19** | 2 |
| | 200 ◇ 1000 | 7.23 | 2.12 | 1 | 0.92 | 1.28 | **0.56** | 2 |
| | 500, 2500 | 4.69 | 1.20 | 1 | 0.74 | 0.97 | **0.10** | 1 |
| | 4000 ◇ 20000 | 15.15 | 4.61 | 1 | 20.49 | 15.13 | **1.17** | 2 |
| | 20000 ◇ 80000 | 32.35 | 3.85 | 1 | *TO* | *TO* | **2.25** | 2 |
| | 30000 ◇ 120000 | 115.11 | 9.65 | 1 | *TO* | *TO* | **1.69** | 2 |
| | 40000 ◇ 200000 | 157.35 | 10.32 | 1 | *TO* | *TO* | **2.55** | 2 |

*TO*: time out *Err*: Error *m*: minute

**Table 2.** Experimental results on the benchmarks in [17]

| (Non separate administration assumption) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Test case | # Roles ⋄ # Rules | Answer | Vac | | Pms | | asaspXL | |
| | | | | | *Fwd* | *Prll* | | |
| | | | Time | # Rules | Time | Time | Time | # Rules |
| Test 1 | 40 ⋄ 487 | Unsafe | 17.25 | 3 | 0.83 | **0.68** | 1.15 | 2 |
| Test 2 | 40 ⋄ 450 | Safe | 0.21 | 0 | 0.91 | 0.75 | **0.19** | 0 |
| Test 3 | 40 ⋄ 462 | Unsafe | 9.33 | 3 | 0.92 | 0.93 | **0.71** | 2 |
| Test 4 | 40 ⋄ 446 | Unsafe | 7.51 | 3 | 0.99 | 45.16 | **0.69** | 2 |
| Test 5 | 40 ⋄ 480 | Unsafe | 48.31 | 47 | 1.25 | **0.91** | 2.12 | 9 |
| Test 6 | 40 ⋄ 479 | Unsafe | 26.62 | 13 | 1.02 | **0.86** | 1.69 | 4 |
| Test 7 | 40 ⋄ 467 | Unsafe | 1 m 12.56 | 101 | 4.22 | 3.26 | **1.85** | 2 |
| Test 8 | 40 ⋄ 484 | Unsafe | 1 m 16.23 | 65 | 5.08 | 2 m 16.21 | **2.04** | 8 |
| Test 9 | 40 ⋄ 463 | Unsafe | 1 m 35.11 | 89 | 5.91 | 6 m 35.24 | **2.91** | 11 |
| Test 10 | 40 ⋄ 481 | Unsafe | 29.94 | 38 | **0.65** | 0.75 | 2.45 | 5 |

semantics as in previous table with additional column "Answer" reports the results returned by analysis tools (*Safe* means the goal is unreachable while *Unsafe* means the goal is reachable). We do not report the experimental result of Mohawk because it cannot handle user-role reachability problems without the separate administration assumption.

The results clearly show that asaspXL performs significantly better than Mohawk, Pms, and Vac in the first benchmark set (Table 1). Notice that Pms throws a time-out (that is set to 10 min) in the biggest test cases. For the second benchmark set, asaspXL outperforms Pms and is much better than Vac. We emphasize that the number of actions after using module **Heuristics** in asaspXL is reduced significantly (column 9).

**Table 3.** Experimental results when turning on/off heuristics in Sect. 4

| Test case | # Roles ⋄ # Rules | Answer | asaspXL | |
|---|---|---|---|---|
| | | | **Without** Heuristic | **With** heuristics |
| Test 1 | 40 ⋄ 487 | Unsafe | 2 m 52.73 | **1.15** |
| Test 2 | 40 ⋄ 450 | Safe | 16.22 | **0.19** |
| Test 3 | 40 ⋄ 462 | Unsafe | 1 m 1.63 | **0.71** |
| Test 4 | 40 ⋄ 446 | Unsafe | 57.15 | **0.69** |
| Test 5 | 40 ⋄ 480 | Unsafe | 2 m 35.87 | **2.12** |
| Test 6 | 40 ⋄ 479 | Unsafe | 2 m 45.71 | **1.69** |
| Test 7 | 40 ⋄ 467 | Unsafe | 3 m 17.33 | **1.85** |
| Test 8 | 40 ⋄ 484 | Unsafe | *TO* | **2.04** |
| Test 9 | 40 ⋄ 463 | Unsafe | *TO* | **2.91** |

*TO*: time out *Err*: Error *m*: minute

Table 3 shows experimental results when we run ASASPXL on the instances of user-role reachability problem in Table 2 with/without heuristics introduced in Sect. 4. Columns 1, 2, and 3 have the same semantic as previous tables. Column 4 reports the analysis time when turning off heuristics while column 5 shows the performance obtained by using heuristics. The results prove the effectiveness of heuristics on the analysis. In many cases, the analysis time is reduced significantly, for example, from 3 min to nearly 2 s.

## 6 Conclusions

We have presented techniques to enable the MCMT approach to solve instances of user-role reachability problem. We have also designed a set of heuristics that help our analysis techniques to be more scalable. The main idea is to reduce as much as possible the number of administrative actions in the original problem. An excerpt of an exhaustive experimental evaluation has been conducted and provided evidence that an implementation of the proposed techniques and heuristics, called ASASPXL, performs significantly better than MOHAWK, VAC, and PMS on a variety of benchmarks from [8,10].

As future work, we plan to design new heuristics based on some functionalities provided by the model checker MCMT such as the capability of tracking the visited states for later use. Another interesting line of research for future work is to consider the combination of backward and forward reachability procedure to speed up the analysis of the model checker.

## References

1. http://homes.di.unimi.it/~ghilardi/mcmt
2. http://research.microsoft.com/en-us/um/redmond/projects/z3
3. Alberti, F., Armando, A., Ranise, S.: Efficient symbolic automated analysis of administrative role-based access control policies. In: Proceeding of ASIACCS, pp. 165–175. ACM Press (2011)
4. Alberti, F., Armando, A., Ranise, S.: ASASP: automated symbolic analysis of security policies. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 26–33. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22438-6_4
5. Armando, A., Ranise, S.: Automated symbolic analysis of ARBAC policies. In: Cuellar, J., Lopez, J., Barthe, G., Pretschner, A. (eds.) STM 2010. LNCS, vol. 6710, pp. 17–34. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22444-7_2
6. Crampton, J.: Understanding and developing role-based administrative models. In: Proceedings of 12th CCS, pp. 158–167. ACM Press (2005)
7. Capitani, D., di Vimercati, S., Foresti, S., Jajodia, S., Samarati, P.: Access control policies and languages. Int. J. Comput. Sci. Eng. (IJCSE) **3**(2), 94–102 (2007)
8. Ferrara, A.L., Madhusudan, P., Nguyen, T.L., Parlato, G.: VAC - verifier of administrative role-based access control policies. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 184–191. Springer, Heidelberg (2014). doi:10.1007/978-3-319-08867-9_12

9. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. Logical Methods Comput. Sci. (LMCS) **6**(4), 1–48 (2010)
10. Jayaraman, K., Ganesh, V., Tripunitara, M., Rinard, M., Chapin, S.: Automatic error finding for access control policies. In: Proceedings of 18th CCS, pp. 163–174. ACM (2011)
11. Jha, S., Li, N., Tripunitara, M.V., Wang, Q., Winsborough, H.: Towards Formal Verification of Role-Based Access Control Policies. IEEE Trans. Dependable Secure Comput. **5**(4), 242–255 (2008). IEEE
12. Li, N., Tripunitara, M.V.: Security analysis in role-based access control. ACM Trans. Inf. Syst. Secur. (TISSEC) **9**(4), 391–420 (2006). ACM Press
13. Ranise, S., Truong, A., Armando, A.: Boosting model checking to analyse large ARBAC policies. In: Jøsang, A., Samarati, P., Petrocchi, M. (eds.) STM 2012. LNCS, vol. 7783, pp. 273–288. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38004-4_18
14. Sandhu, R., Coyne, E., Feinstein, H., Youmann, C.: Role-based access control models. IEEE Comput. **2**(29), 38–47 (1996). IEEE
15. Sasturkar, A., Yang, P., Stoller, S.D., Ramakrishnan, C.: Policy analysis for administrative role-based access control. Theor. Comput. Sci. **412**(44), 6208–6234 (2011). Elsevier
16. Stoller, S.D., Yang, P., Ramakrishnan, C., Gofman, M.I.: Efficient policy analysis for administrative role-based access control. In: Proceedings of 14th CCS, pp. 445–455. ACM Press (2007)
17. Yang, P., Gofman, M., Yang, Z.: Policy analysis for administrative role based access control without separate administration. In: Wang, L., Shafiq, B. (eds.) DBSec 2013. LNCS, vol. 7964, pp. 49–64. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39256-6_4